Modeling and Verification of Software (MOVES)
Department of Computer Science
RWTH Aachen University

Master Thesis

Model-Based Criticality Analysis by Impact Isolation

Bernhard Ern

Supervised by
Priv.-Doz. Dr. Thomas Noll
Prof. Dr. Ir Joost-Pieter Katoen

Aachen, January 10, 2012

Declaration

I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such.

Hiermit versichere ich, diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht zu haben.

Aachen, January 10, 2012

(Bernhard Ern)

Contents

Chapter 1

Introduction

1.1 Motivation

Increasing the quality of software in aerospace systems has a high priority during the development. Not only the crash of the Ariane 5 in 1996 shows that software faults are a serious problem, also the loss of the Eutelsat W3B satellite in 2010 that crashed because of "a major anomaly" [28] shows that aerospace systems are even today sensitive to every kind of faults. Considering the development and building costs of an aerospace system, the safety, and consequently the life time, of the system has economically the highest priority.

In this thesis an error is every kind of decision during the development process which needs to be changed later. In normal software projects, the costs of correcting errors grow exponentially with the progress of the project. This is visualized in figure 1.1 which is based on the results of [4]. This leads to the strategy that errors have to be detected and corrected as early as possible in the development process. Therefore, in complex systems, testing developed functionalities should not only be done in the testing phase after the implementation. Also in the design phase it is necessary to create a coarse model of the system and start developing tests. In this thesis we will work with the modeling language SLIM which is used for modeling and specifying hardware/software systems. We see in figure 1.1 that the relative costs for correcting an error during the design phase are quite low compared to the costs in the deployment.

For aerospace systems we cannot only consider the errors during the development process. Also hardware dependent failures like the loss of a signal can appear. Additionally, in safety critical parts of the system, we need a criticality

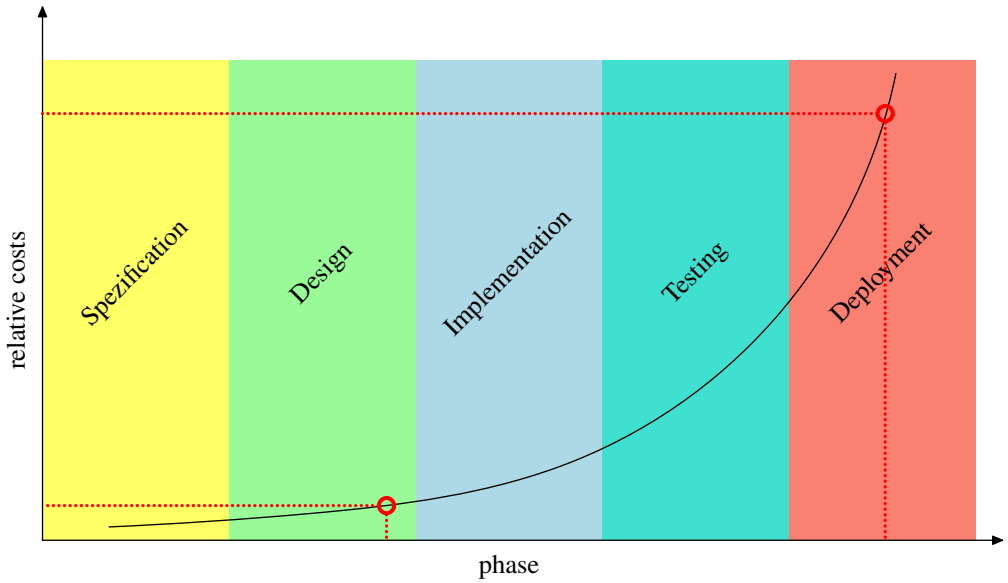


Figure 1.1: Cost of change curve depending on the development phase.

categorization of such failures, because during the deployment some failures lead to fatal situations like losing the whole system. This leads to the question: How do we calculate a criticality analysis? A categorization of failures is done in “Failure modes, effects and criticality analysis (FMECA)” [13] which is standardized for the European space industry in ECSS [12]. In general, the criticality is defined as the “combined measure of the severity of a failure mode and its probability of occurrence” [13]. For calculating the probability of occurrence there exist system depending measurements and we will not go into this point in this thesis. We will concentrate on the calculation of the severity of a failure. Therefore we only focus on the question: How do we calculate the severity of a component?

The severity categories are presented in the table 1.2 which is directly taken from [13]. In SLIM the appearance of a fault will be handled by FDIR (Fault Detection, Identification and Recovery) components. Now an FDIR component is classified by the criticality of the corresponding failures. Depending on this criticality, sufficient test cases need to be developed. However, classifying an FDIR component according to the failure is not exact enough. The idea of this thesis is to use the isolated effect of an FDIR component on the system as basic information for the criticality analysis. In detail, an FDIR component which only locally impacts parts of the system is categorized with a low criticality. On the

SYSTEM LEVEL FMEA/FMECA	
Severity category	Failure effect
Catastrophic 1S	• Loss of life, life threatening or permanently disabling injury or occupational illness, loss of an element of an interfacing manned flight system. • Loss of launch site facilities. • Long-term detrimental environmental effects.
Catastrophic 1	• Loss of system.
Critical 2S	• Temporary disabling but not life threatening injury, or temporary occupational illness. • Loss of, or major damage to other flight systems, major flight elements, or ground facilities. • Loss of, or major damage to public or private property. • Short-term detrimental environmental effects.
Critical 2	• Loss of mission.
Major 3	• Mission degradation.
Negligible 4	• Any other effect.

Figure 1.2: ECSS categorization of FMECA severity.

contrary, FDIR components which affect many different components or even high criticality components receive a high criticality. Hence, the criticality analysis of FDIR components is based on detecting and understanding the impact on other components. Also an FDIR reaction on a fault is not only limited to the reaction of one component. It is possible that several FDIR components react on a fault and influence each other. The detection and understanding of the impact is still done by time consuming and error-prone manual code reviews. Therefore, we are searching for a full automatic model-based criticality analysis by isolating the impact of FDIR components.

> "It is also extremely difficult to visualize how a change in the system (monitor, response, or parameter) will affect the rest of the system."

This quotation comes from a spacecraft fault management workshop result of the NASA in 2009. It shows that in the environment of spacecraft systems it is difficult to detect and analyze the effect of one part of the system on another part. How to analyze and visualize this effect is the motivating task of this master thesis. For detecting the effect of a change, the most model checking approaches are not appropriate. The reason for this is that model checker are checking the current

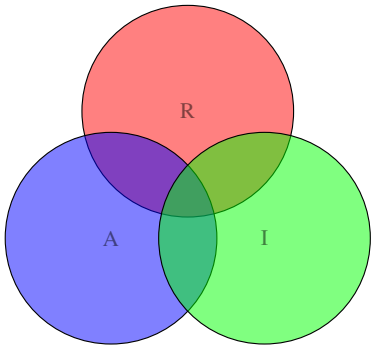


Figure 1.3: R: Real impact, A: Assumed impact, I: Isolated impact.

model for a given property, but if the property (or the effect of a change) is not known preliminarily, other concepts of analyzing need to be taken into consideration. The impact isolation is an exploratory analysis technique which exactly deals with these problems. In general, the problem of detecting the effect of a change to the system is called "change impact analysis". Change impact analysis deals with the measurement of the impact of a single static software change during the development. In software engineering, this field has been analyzed for many years. For example, one of the first publications on this area has been done in 1978 [30], but still today change impact analysis is an interesting, complex and important technique to understand software and to increase its quality. The calculation of the impact in the change impact analysis is similar to the calculation of the dynamic impact of a component to the rest of the system. To separate these two impact analysis techniques, we call the analysis of the dynamic impact of a component in a running system "impact isolation".

Figure 1.3 depicts three impact sets, which are part of every impact isolation. The first set is the real impact (R) of a component. The second set is the impact which is assumed from the developer (A). The last set is the set of impact which is found by the automatic calculated impact isolation (I). With these three sets we will formalize the important facts and aims of the model-based criticality analysis.

- The criticality of a component is only based on A, because R is unknown and I is maybe not reliable.
- The wrong overestimation $A \setminus R$ leads to an overestimation of the criticality.

- The wrong underestimation $R \setminus A$ leads to an underestimation of the criticality.
- IF $R \setminus I \neq \emptyset$ then the impact isolation is called not safe.
- IF $I \setminus R \neq \emptyset$ then the impact isolation is called an over-approximation.

The aim of the impact isolation is to reduce the sets $A \setminus R$ and $R \setminus A$. This means that the assumed impact and the real impact should be as similar as possible and in the perfect situation $A = R$ holds. For reaching this aim, we are using the impact isolation to convince the developer to reduce $A \setminus R$ and $R \setminus A$. For this, let us assume in the first step that $R = I$ which means that the impact isolation is no over-approximation and safe. Then the developer only needs to calculate I to discard $A \setminus I$ and $I \setminus A$ as wrong estimation. Unfortunately, the impact isolation can be unsafe or an over-approximation.

If the impact isolation is unsafe, then it is difficult to convince the developer to reduce $A \setminus R$ by using $A \setminus I$, because the developer cannot remove the impacts which are not found by the isolation. Reducing $R \setminus A$ by looking at $I \setminus A$ is still applicable, because $I \setminus A \subseteq R$ and therefore every element from $I \setminus A$ can be added to A.

On the other hand if the impact isolation is an over-approximation, convincing the developer to reduce $R \setminus A$ is difficult. The developer cannot add every impact which is found by the isolation. Analogously, reducing $A \setminus R$ is still possible if the over-approximation is not too coarse such that $A \subset I$ holds.

In both cases, the developer cannot trust the isolation completely. Nevertheless, even if the isolation is unsafe and an over-approximation, then the developer can still use the results as an orientation point for searching impacts or discarding overestimations.

This reasoning refines the question of the calculation of the severity to the question: How do we calculate the impact of a component?

1.2 Motivating examples

In this section three examples will be presented. Every example describes coarsely a different system which needs to be modeled. Each of these examples will be used and described more detailed during the thesis. The first two examples are academic which are used to explain the concepts of this thesis. The last example

is a practically relevant model which will be used to test the concepts of this thesis under realistic conditions. The modeling process in SLIM is not trivial and the rise of unexpected behavior can occur in every model. Each of these examples describes a concurrency system. One problem with concurrency systems is that the different components will influence each other in ways unintended by the developer. This leads to the question: "How can the influence of one component on another component be detected?" This is the main problem of each of the following examples and should be kept in mind during the whole thesis. Now let us take a short view of the problem description of the different examples.

1.2.1 Sensorfilter Example

The first example is the sensorfilter of an aerospace system. It is part of the COMPASS tool set [27]. The basic idea of this system is that the sensor is grabbing some data, e.g. noises or the infrared picture of an observed object. Then this data is sent to a filter which manipulates the data of the filter, for example by removing background noises or detecting anomalies in the picture. Let us assume that the correctness of the generated values has a high priority in a larger system. Additionally there exists the probability that the sensor or the filter suffer a defect. In this situation a monitor detects this defect by observing an impossible behavior of the components and tries to resolve it by interchanging the defect device with a backup component. Additionally if also in the backup component a fault is detected then the monitor alerts the system, that the data of the sensorfilter system may be corrupt.

The main question which motivates this example is: How does the monitor influence the running sensorfilter system? This question can be divided into two subquestions:

> If the monitor switches the sensor into the backup modus, does this influence the behavior of the filters?
>
> If the monitor switches the filter into the backup modus, does this influence the behavior of the sensors?

If the system is correctly designed, we assume that the change of the filters component has no effect on the sensor. On the contrary, a change of the sensors will influence the behavior of the filter, because the filter uses the data of the sensor. To ensure that this assumption is correct, we need a proper analysis technique.

Another question is the negation of the impact question. Does the component not impact another component? Also this information needs to be detected in the impact isolation.

1.2.2 Life Support System Example

The second example is the modeling of a life support system for a space shuttle. This example is developed during this thesis. This full automatic system always guarantees that the shuttle is supplied by enough oxygen and the temperature is inside of a livable interval, such that humans will survive. For this functionality an oxygen generator and a heater are responsible. These two components only work if they are adequately supplied with energy. Therefore, the oxygen generator and the heater have their own battery. Unfortunately, it is possible that the battery discharges over time. In this case a monitor for the oxygen generator and a monitor for the heater have to detect the under voltage of the device and activate an additional backup support. This will be modeled by a backup battery which can be switched to support the voltage either of the oxygen generator or of the heater.

This example also raises questions on the effects of two different faults which affect each other. This leads to the general question: Can we detect the impact of components which have a changed behavior, because of fault propagation? Or in this special case: If both FDIR monitors are active at the same time, does this lead to an unexpected behavior?

1.2.3 Satellite Case Study

The last example is more complex than the previous one and describes the possible modeling criteria for a satellite. This case study will be submitted on the ICSE 2012 [18]. Every satellite will eventually run through 4 different phases.

- **Launch:** In the first phase, the satellite is dropped by a carrier rocket into the orbit and will adjust to the sun to guarantee the energy supply.
- **Transfer:** In the second phase, the satellite will reach the correct position in the orbit and will follow a given path for circling around the world.
- **Service:** At this point the satellite will start the service operations and communicate with the earth.

- **Post Disposal:** At the end of the life cycle of the satellite, it stops the service operation and waits for leaving its orbit for disposal.

For realizing all these phases, the satellite contains different components with different tasks. For the adjustment after the launch, a sun sensor is necessary to detect the position of the sun. For charging the battery, a solar system is an essential part of the system. For reaching the orbit, a propulsion system will manage the thrusters and the tanks for the fuel. As soon as the satellite starts with the service operation, the adjustment to the earth needs earth sensors and for staying in the correct orbit a gyroscope is used. To guarantee an unobstructed functionality, the satellite needs the correct temperature. Therefore, thermometer sensors are needed and a heater for the regulation. All these components are running in parallel and are managed by several control units. The essential devices are observed by monitors, such that in the case of a fault it will be detected, isolated and solved. This is implemented by different reactions like reseting devices or switching to redundant backup devices. All these functionalities will be realized by a concurrency system.

Of course such a complex system does not only raise one question, but the main problem at this point is: how can we calculate the impact between components on such a huge model and how can we display it in such a way that the developer of the system collects useful information, without becoming dumped by verbose information? Also it is clear that in such a huge system, the effect of one component will propagate through the system and not just affects other components directly. How can we detect indirect impacts? We will substantiate these questions in the case study in chapter 5.4 and analyze a small part of the satellite.

1.3 Structure of the Thesis

The structure of this thesis is designed by a problem driven approach. Therefore, the three introduced examples and the questions raised in the motivation are the corner points of this thesis. The three main questions are:

- How do we calculate a criticality analysis?
- How do we calculate the severity of a component?
- How can the influence of one component on another component be detected?

This are questions which could be asked from the developer which applies the criticality analysis. The next section is structured by questions which are raised from these elemental questions.

Chapter 2: Preliminaries

In chapter 2, the necessary preliminaries and the background knowledge for the thesis will be presented. As mentioned in the introduction, change impact analysis is an important part in software engineering. The outcome of this is that there are many different analysis approaches and frameworks available. Therefore, the first part of the introduction will deal with an introduction to change impact analysis and categorize the impact isolation approach of this thesis. The second part will be an introduction to the related and applicable work of the impact isolation. Also the COMPASS project of the RWTH Aachen and the modeling language SLIM will be presented.

Used Example: 1.2.1 Sensorfilter

Chapter 3: Concepts for the Impact Isolation

The main part of the thesis is chapter 3. It presents the concepts of the impact relation, the counting matrix, the comparing paths and the transitive closure of the counting matrix. These four concepts are motivated through the questions which are founded in the life support system example and the satellite case study.

Used Example: 1.2.2 Life Support System, 1.2.3 Satellite Case Study

Main Questions:

Q1 "What is an impact in a SLIM model?"

Q2 "How can we calculate the impact of a component"

Q3 "How can the impact be visualized?"

Q4 "How can deviating impacts be analyzed?"

Q5 "How can indirect impacts be detected?"

Chapter 4: State Space Optimization

Chapter 4 will deal with the exploration of possible optimizations of the concepts which were presented in the previous chapter. The satellite and the sensorfilter are the main examples for this section and show the advantage of the more detailed analysis. In this connection, the basic strategy is to reduce the size of the analyzed state space by dividing the state space into subspaces. Also the application of the state space reduction techniques partial-order reduction, bisimulation and slicing is analyzed.

Used Example: 1.2.3 Satellite Case Study, 1.2.1 Sensorfilter

Main Questions:

Q6 "How can we make the impact isolation more accurate?"

Q7 "How can we make the impact isolation faster?"

Chapter 5: Experimental Evaluation

In chapter 5, the results of the experimental evaluation of the thesis are presented. The wrapper for generating the state space will be presented in greater detail, because the wrapper is the foundation for the following benchmarks in this chapter. Also the concepts of the created impact tool will be presented. The satellite case study will be the main example for this chapter, because it shows the current limitations for the benchmarks and also the application of the impact isolation on a realistic model.

Used Example: 1.2.3 Satellite Case Study

Main Questions:

Q8 "How to apply the impact isolation on realistic models?"

Chapter 6: Future Work

The future work in chapter 6 will discuss how the results in this thesis could be improved or expanded. At this point, we separate between conceptional expansions and performance improvements.

Chapter 7: Summary

Last but not least, in chapter 7 the results and conclusions of this thesis will be summarized and discussed.

Chapter 2

Preliminaries

In this chapter we will focus on the necessary background information for the impact isolation. At first we present the COMPASS project and particularly the modeling language SLIM which will be used as the environment for the impact isolation. After this, we introduce two traditional strategies for change impact analysis. Last but not least, we will merge these points to motivate the impact isolation for SLIM.

2.1 Presentation of SLIM

Spacecraft systems are highly complex and difficult to model. Especially for the autonomy of satellites in the aerospace, a fail safe design is essential. Just assume the communication to a satellite or other independent spacecraft system fails and the reconnect fails. Then the satellite is literally dead. Additional the interaction between hardware and software components makes the impact isolation interesting for aerospace systems. The isolation of a block of code which affects a certain hardware behavior is another interesting part of the analysis. In the next subsection, the COMPASS project will be presented as a tool to verify and validate the correctness and performance of aerospace systems.

2.1.1 The COMPASS Project

The COMPASS project (**Co**rrectness, **M**odeling and **P**erformance of **A**ero**s**pace **S**ystems) is a project of the RWTH Aachen University together with Fondazione Bruno Kessler and Thales Alenia Space to improve the correctness, safety and

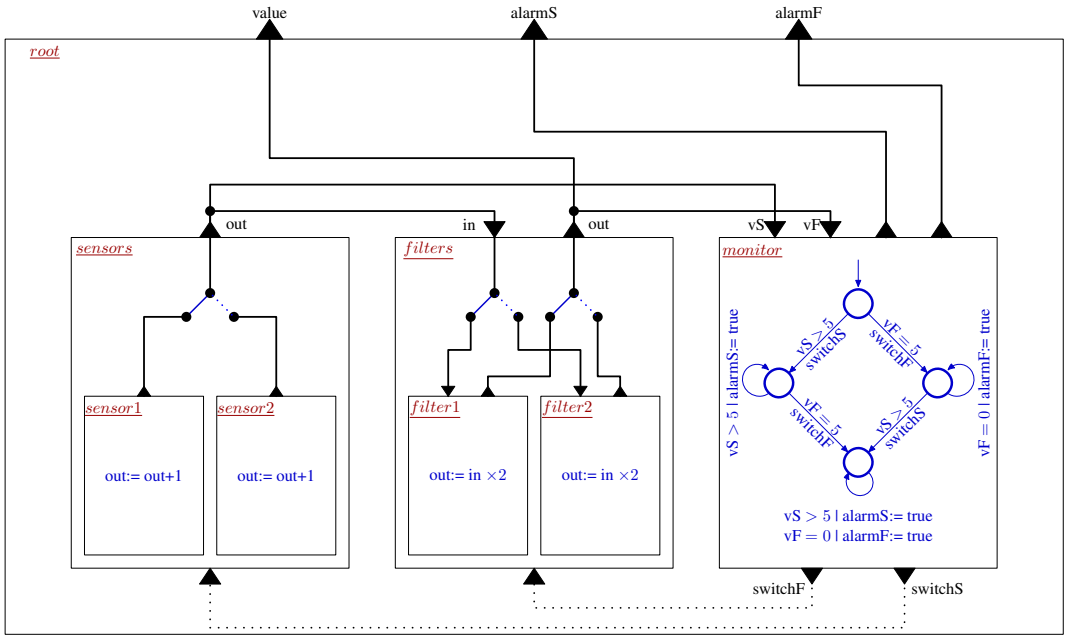


Figure 2.1: Abstract sketch of a the SLIM model sensorfilter.

performability of aerospace systems. The details of the COMPASS project are presented in [6, 27]. The main fact of the COMPASS project for this thesis is that aerospace systems can be modeled in SLIM (**S**ystem-**L**evel **I**ntegrated **M**odeling) and checked by the COMPASS tool kit. This thesis will focus on doing an impact isolation directly on SLIM models. The other functionalities of the COMPASS tool like property checking with NuSMV, probabilistic property checking with marcov rewarded model checking or the generation of fault trees, are not used in the impact isolation techniques of this thesis. Nevertheless, at present the COMPASS project does not contain an appropriate analyzing concept for impact isolation. After all, this is the reason for developing an impact isolation for SLIM. Before we will do this, we need to introduce the SLIM language.

2.1.2 The SLIM Language

The complete syntax and semantics of the SLIM language are presented in [10,11] and another detailed definition is found in [23]. For understanding the basic functionality of SLIM, it is not necessary to present the complete syntax and semantics at this point, but the basic behavior of a SLIM model is essential for understanding

the idea behind the impact isolation of this thesis. For this consider the example SLIM model in the appendix A.1.1 which models the first example (1.2.1) about the sensorfilter system.

An abstract visualization of the model is presented in figure 2.1. A box represents a component of the model. If a box is inside an outer box, it means that the inner box is a subcomponent of the outer box. The black triangles on every box are the data/event ports of the component. If the triangle points towards the box, it is an "in data port" or an "in event port", otherwise it is an "out data port" or an "out event port". The lines connecting two or more triangles are the connections or flows of the component. If the lines are drawn continuously, then this is a data connection between data ports. If the line is dashed, then we visualize an event connection between two event ports. If the connection contains a dot and outgoing from this dot are two arrows to different components, then this means that the data value is sent to both destination data ports. If two or more connections are crossed without a dot, then both data values do not influence each other. The switches in the boxes represent the mode or data port depending connections or flows. The blue text inside a box informally describes a transition of the corresponding component. The exact description of the notation is visualized in appendix A.3.

The sensorfilter example starts with the root component Sensorfilter. The root is in every SLIM model on the top of the component hierarchy and particularly not defined as a subcomponent of another component. Normally, a component assigns a name as an identifier to every subcomponent. The root has no upper component. Therefore it receives the standard name root as the identifier. Every component in this example is modeled after the same definition for non-data components. A component is defined by two parts: The system specification and the system implementation. For example the system specification of the root component is:

```
system Sensorfilter
  features
    value: out data port int default 4;
    alarmS: out data port bool default false observable;
    alarmF: out data port bool default false observable;
end Sensorfilter;
```

It describes the communication ports of the root component. Hereby $value$ is the result of the sensorfilter model, an out data port which is initialized with the integer value 4. The other out data ports $alarmS$ and $alarmF$ are booleans which are set to true, if a failure in the system is detected. The keyword observable is only used to observe the data ports with the COMPASS tool. It does not affect

the interpretation of the current model. The exact calculation of these ports and other details of the component are described in the system implementation of the Sensorfilter component:

```
system implementation Sensorfilter.Impl
  subcomponents
    sensors: system Sensors accesses mybus;
    filters: system Filters accesses mybus;
    monitor: system Monitor accesses mybus;
    mybus: bus MyBus;
  connections
    data port sensors.output −> filters.input;
    data port filters.output −> value;
    data port sensors.output −> monitor.valueS;
    data port filters.output −> monitor.valueF;
    data port monitor.alarmS −> alarmS;
    data port monitor.alarmF −> alarmF;
    event port monitor.switchS −> sensors.switch;
    event port monitor.switchF −> filters.switch;
end Sensorfilter.Impl;
```

Under the keyword subcomponents, the subcomponents are defined. This is the basic of the hierarchical structure of SLIM. In figure 2.1, the three subcomponents sensors, filters and monitor are modeled as inner box of the root component. The subcomponent $mybus$ is only used for describing how the other subcomponents communicate with each other. The bus has no other effect on the model and will be ignored in this example. Every subcomponent is again defined on the same definition structure as the component root. In the next part of the definitions follows the keyword connections. Connections are one of the main elements of SLIM which make it possible to model complex systems in SLIM. Connections use the values of a component and transmit them to another component. For example, the value of the data port $output$ of the sensors subcomponent will be transmitted to the data port $input$ of the subcomponent filters. In the figure 2.1 every connection is modeled as an arrow from the component which transmits the value to the component which receives the value. With connections generated values can be transmitted over arbitrarily many hierarchy levels and spread the value of a device in a subcomponent through the complete model. One important limitation of the connections is that it is forbidden to model cyclic dependencies. The transmission of the values through connections does not need any time such that spreading of the values is an atomic step with every taken transition of the model. This leads

us to the next point of system implementation the transitions. In the implementation of the root components there are no transitions and no modes, therefore let us consider the model of the sensors component:

```
device Sensor
  features
    output: out data port int default 1;
end Sensor;
device implementation Sensor.Impl
  modes
    Cycle: activation mode;
  transitions
    Cycle -[when output<5 then output:=output+1]-> Cycle;
end Sensor.Impl;
```

The sensor is a device which creates data for the system. This is modeled by a transition which increments the value of the data port $output$. The transitions which are defined under the keyword transitions are modeled with the structure m -[e **when** g **then** f]-> m', where m, m' are modes of the same component which are defined under the keyword modes. e is an optional event, which will be explained in the next component. g is an optional guard which needs to be true to activate the transition. f is an optional function which possibly affects one or more out data ports (and non-data subcomponents). As we see here, the sensor only increments the out data port $output$ by one if its value is smaller than five. This out data port is used from the component sensors which is defined as following:

```
system Sensors
  features
    output: out data port int default 1;
    switch: in event port;
end Sensors;
system implementation Sensors.Impl
  subcomponents
    sensor1: device Sensor in modes (Primary);
    sensor2: device Sensor in modes (Backup);
  connections
    data port sensor1.output -> output in modes (Primary);
    data port sensor2.output -> output in modes (Backup);
  modes
    Primary: activation mode;
```

```
    Backup: mode;
  transitions
    Primary −[switch]−> Backup;
end Sensors.Impl;
```

This component reveals the complexity of SLIM models. The sensor contains two subcomponents, the devices sensor1 and sensor2. By taking the transition $Primary - [switch] - > Backup$, the mode of the component switches from $Primary$ to $Backup$. This mode change affects the connections of the device such that initially taking the data from the sensor1 is changed to using the data of the backup sensor sensor2. This transition is only active if another transition triggers also the event $switch$ and the guards of both transitions hold. This synchronized taking of two transitions is called handshake. The values of event ports are transmitted by connections similar to the connections of data ports and are identical to booleans. This means that events are available or not. The event connections are drawn in figure 2.1 as the dashed arrows between the sensors/filters and the monitor. The corresponding event to the $switch$ event in this component is defined in the component monitor:

```
fdir system Monitor
  features
    valueS: in data port int default 0;
    switchS: out event port;
    ...
end Monitor;
fdir system implementation Monitor.Impl
  modes
    OK: activation mode;
    FailS: mode;
  ...
  transitions
    OK −[switchS when valueS>5]−> FailS;
    ...
end Monitor.Impl;
```

We see that the keyword fdir is written in front of the specification and implementation definition. This keyword does not directly change the behavior of the component. It is only used to mark it as FDIR (Fault Detection, Identification and Recovery) component. FDIR components are part of the nominal SLIM model, but normally they only react if a fault occurs. They are used to monitor the corresponding components, in this example the values of the sensors and filters com-

ponents, and are used to detect and isolate faults. After the detection, the FDIR component gives an instruction to lead the system back to a safe state. In this example, the transition $OK - [switchSwhenvalueS > 5] - > FailS$ switches the sensor to the backup mode by sending the $switch$ event to the sensors component.

As mentioned in the beginning, the full model is presented in the appendix A.1.1. In combination with figure 2.1, all information are available to understand the introducing example of SLIM. The important remark of this example is that the effect of a single transition possible affects the data ports of many different components over several hierarchical levels. Because of the connections and the possibility to make them dependent on modes of components, it is difficult to understand which transition affects which components by just analyzing the source code. This emphasizes the statement from the introduction that manual code reviews are time consuming and error-prone. Therefore, we need an automatic model-based approach.

Covering Erroneous Behavior: Extended Model

Another feature of the SLIM model is to expand the nominal models with probabilistic functionalities. This is also described in detail in [10, 11]. At this point, a short example will be enough to show the ideas behind the error models of SLIM. In appendix A.1.2 there is a possible error model for the Sensorfilter example. The basic idea behind the error model is to expand the nominal model by building the cross product between the modes of the component in the nominal model and the modes of the error model. In the error model, error events occur with a given probability and change the error modes. For example, in the error model of the sensor:

```
error model implementation SensorFailures.Impl
  events
    drift: error event occurrence poisson 0.083;
    die: error event occurrence poisson 0.00001;
    dieByDrift: error event occurrence poisson 0.00015;
  transitions
    OK -[ die ]-> Dead;
    OK -[ drift ]-> Drifted;
    Drifted -[ dieByDrift ]-> Dead;
end SensorFailures.Impl;
```

The error mode is changed by the transition $OK - [die] - > Dead$ from OK to $Dead$. The used error event die occurs with a probability of 0.083. The error

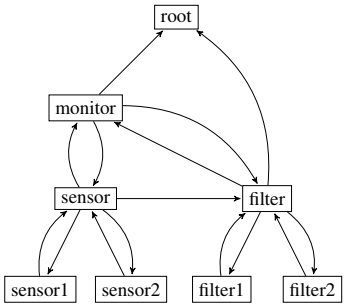
model does not directly affect the nominal model. To model an influence, the fault injections are needed.

With the fault injection, it is possible to give the change of an error mode a semantic. In the appendix A.1.3 there is a possible fault injection for the sensor and filter components of the error model. In this fault injection, the data port $output$ of component sensor1 and sensor2 will be set to 15 and the $output$ of the filter1 and filter2 component will be set to 0 if the corresponding error mode $Dead$ will be reached. These three models together are the basis for an extended model in SLIM which compares the properties of the nominal model, the error model and the fault injection to a single model of the system. This is also described in greater detail in [10, 11].

2.2 Motivation for Impact Isolation

Before we will introduce the change impact analysis, we need to clarify the difference between the requirements of the change impact analysis of the software engineering and the requirements of the impact isolation in this thesis. In the change impact analysis, a static change is executed in one or more components. Hereupon the impact of this single static change needs to be analyzed. In our case, a not static change is executed. Moreover, we have a dynamic change during the execution of the model. This change is invoked by a fault in a component of the system. We will concentrate on the impact of the FDIR monitors which observe the system. The reason is that these components have in general a complex impact on the system and the criticality of these components is difficult to assign. However, the analysis of the propagation of the change is similar to the analysis in software engineering. Therefore, let us consider the change impact analysis in greater detail. From this point on we will call the change impact analysis only impact analysis and if we reference to the analysis of this thesis we will call it impact isolation.

As mentioned in the beginning of chapter 1, impact analysis is highly researched in software engineering. The reason why impact analysis is so important for software engineering is the maintenance of software projects. If one part of a system is changed by a maintenance operation then this change causes unpredictable effects to other parts of the system. Therefore, for all these affected components test cases need to be developed to test the maybe new behavior. For calculating the affected components many different approaches and categories of impact analysis techniques are developed. Generally, impact analysis is separated



	root	sensors	filters	monitor	sensor1	sensor2	filter1	filter2
root								
sensors			X	X	X	X		
filters	X			X			X	X
monitor	X	X	X					
sensor1		X						
sensor2		X						
filter1			X					
filter2			X					

Figure 2.2: Callgraph and connectivity matrix.

into two main categories, the static and dynamic analysis. The main difference between the two techniques is that the static analysis only calculates the dependencies for the analysis from the source code of the software. On the other hand, the dynamic analysis uses the information which is gained from the execution of the software. In the introduction of [19], the difference between dynamic and static impact analysis is described and gives more detailed insight into the different techniques. A more accurate and detailed framework of comparison of impact analysis approaches is presented in [2].

In the next two subsections, we will present two possible approaches for impact analysis. Additionally, we will present and compare the differences between the static and dynamic impact analysis.

2.2.1 Static Impact Analysis

The traditional technique for calculating a dependency-based static impact analysis is to calculate a call graph [1]. The call graph visualizes the dependencies of the components in a software system. The difficult part is to calculate these dependencies, particularly in the case of a huge software project with millions of

lines of code. These dependencies are calculated on source code level or on architectural level [31] and arise for example by function calls or return values after a structural analysis [5]. More SLIM related, we assume that the example call graph and the corresponding connectivity matrix of figure 2.2 are the result of the dependencies which are generated through the connection of data and event ports. Independent from the method how the dependencies of the call graph are calculated, there are again several ways to continue the impact analysis. This reaches from the naive approach of calculating the reachability matrix (e.g. using Floyd-Warshall algorithm on the connectivity matrix) to using semantics from the source code to reduce the complexity [5]. Most of the static impact analysis techniques are an over-approximation of the impact. An example for this is in figure 2.2 when we assume that in component filter1 is a change. Then, the resulting set of impacted components is {root,sensors,filters,monitor,sensor1,sensor2,filter1,filter2} by doing a reachability search starting in component filter1. We see that the full system is in the impact set of filter1. Additionally, we see that the transitions are not used to calculate the impact. Hence, we have not used all information for calculating the impact set. This is a problem surfacing in most cases of applying a static impact analysis. The result is often an over-approximation, because the dependencies between the components are simplified. This over-approximation leads to the possibility that resources are wasted for analyzing the effect on unaffected parts of the software system.

2.2.2 Dynamic Impact Analysis

Also for dynamic impact analysis there are several techniques possible, for example, path-based and coverage-based impact analysis [7, 25]. In this subsection the pros and cons of dynamic impact will be analyzed exemplarily for the path-based dynamic impact analysis [19]. The basic idea of this technique is to calculate the impact of a program depending on a set of generated executions of the program. The problem with this dynamic impact analysis is that for SLIM it is difficult to interpret a single execution, because the components of a concurrency system are running in parallel and are more difficult to analyze than a sequential execution. For this example, we assume to have a program language which uses function calls and receives return values to communicate with other parts of the system. Nevertheless, from this example, we receive much information about the dynamic impact analysis.

For this, let us consider figure 2.3. The call graph on this picture is the result of a hypothetical static impact analysis of the Sensorfilter which is written

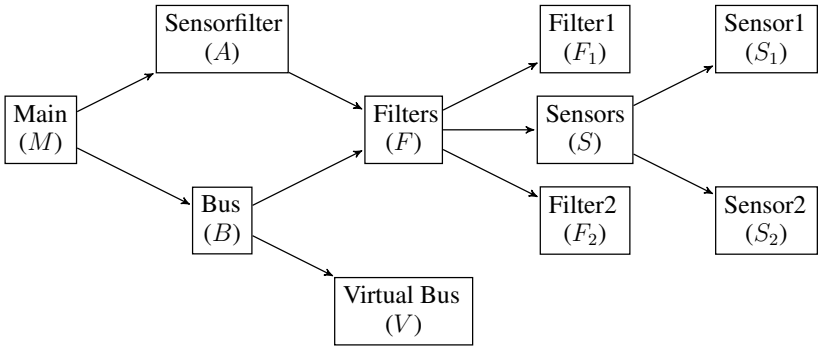


Figure 2.3: Call graph of the a sequential program.

in a sequential programming language. Main (M) is the main function of the program and manages the system. For example, M uses the data from the Sensorfilter (A) or the Bus (B) by a function call. A possible implementation is that M uses the function call $Sensorfilter.getValueInMode(SensorBackup)$. In this example the parameter and the return values are important for the impact of a function. Therefore, the impact is independent of the direction of the edges in both directions. Let us assume that there is a change in function S. Then with static impact analysis the whole call graph is affected by this change, because a change of function S expands forward on an edge (if the parameters of called functions are affected) and backwards (if the returned value is affected by the change). Now let us consider the following executions for a dynamic impact analysis:

1: $M\ B\ r\ A\ F\ F_1\ r\ S\ S_1\ r\ r\ r\ r\ r$

2: $M\ B\ r\ A\ F\ S\ S_2\ r\ r\ r\ r\ r$

3: $M\ A\ F\ F_1\ r\ S\ S_1\ r\ r\ F_2\ r\ r\ r\ r$

In which r means the return of a function. In the first execution, the function B returns before function S is reached, such that a change in S does not affect B. This also is the case for function F_1. The resulting impact set after the first execution is {M,A,F,S,S_1}, because M,A and F are affected by the return value of S. S_1 is added because of affected parameters in the function call. After the second execution, the impact set will be expanded by S_2, and after the third execution, F_2 will be added.

We see on this example that the dynamic impact analysis delivers a more accurate impact analysis than the static analysis, but the dynamic analysis does not guarantee to be safe. This means that the set of executions is in most cases a strict subset of all possible executions and we assume in this subset is it the case that impacts are missing. For example, the new execution $M\ B\ A\ F\ F_1\ r\ S\ S_1$ r r r r r r adds component B to the impact set. Of course, there are lots of improvements for both analysis techniques, but these two examples are sufficient to show the difference between static and dynamic impact analysis.

2.2.3 Impact Isolation of SLIM Models

The general approach for modeling a system in SLIM is to create the nominal SLIM model of the system. After this, the error model and the fault injections which describe an error case for a component will be added. Depending on the nominal model and the error case, an FDIR component will be added to the nominal model (in the example of the sensorfilter the monitor) which detects and solves the faults in the system and leads the system back to a normal behavior. As mentioned before, the safety in aerospace systems is very important, therefore it is useful to group FDIR components into criticality classes. This means if an FDIR component impact lots of different or important parts of the system, then it will receive a high critical level. Depending on these critical levels the different FDIR components will be tested or analyzed in a more detailed manner. So we search for a technique to isolate the effect of a component. This leads to the practical question which motivates this thesis:

> If a fault appears in a component and an FDIR component reacts to solve this fault, then which components are influenced by the FDIR component?

In this situation, the change impact analysis seems to be a proper analysis technique to find the relations between the single components in the system. In the following chapter, a transition based impact isolation for SLIM models will be presented. The isolation techniques are theoretically based on the impact analysis, but also adapted to SLIM such that we have an approach to solve this problem of identifying the impact of one SLIM component on another one.

Chapter 3

Concepts for the Impact Isolation

As mentioned in the end of the previous chapter, developing SLIM models is not trivial. In particular, the FDIR components which lead the system back to a safe state after a fault, are problematical. In order to understand the background of the problem, let us consider example 1.2.2 about the life support system in greater detail.

In the appendix A.2.1 there is the full model of a life support system and figure 3.1 is a visualization of the system. The idea behind this model is that a space shuttle is automatically supplied with oxygen by the component oxygen and with an accurate temperature by a heater. Both components only work if they have more than 3 units of energy. For the energy supply three batteries are in the model. Every battery produces 6 units of energy. The batteries battery1, battery2 and batteryB send the energy to a distributor which routes the energy of battery1 to the oxygen component and the energy of battery2 to the heater. The FDIR oxygen and the FDIR heater are the FDIR monitors of the system and observe the energy support of the heater and oxygen component. If one of these components have not enough energy to work correctly, then the responsible monitor switches the backup battery batteryB to the under supplied component where the energies of the two batteries are summed, such that the component receives enough energy for staying active. The events for switching to the backup mode of the battery are drawn in figure 3.1 as the dotted arrows. For the oxygen component, the green dotted arrow leads from oxygen to the distributor. The red dotted arrow, which represents the switch event of the heater component, leads from the heater to the distributor .

The error model in appendix A.2.2 and the fault injection in appendix A.2.3 extend the life support system by a small probability that battery1 or battery2 can

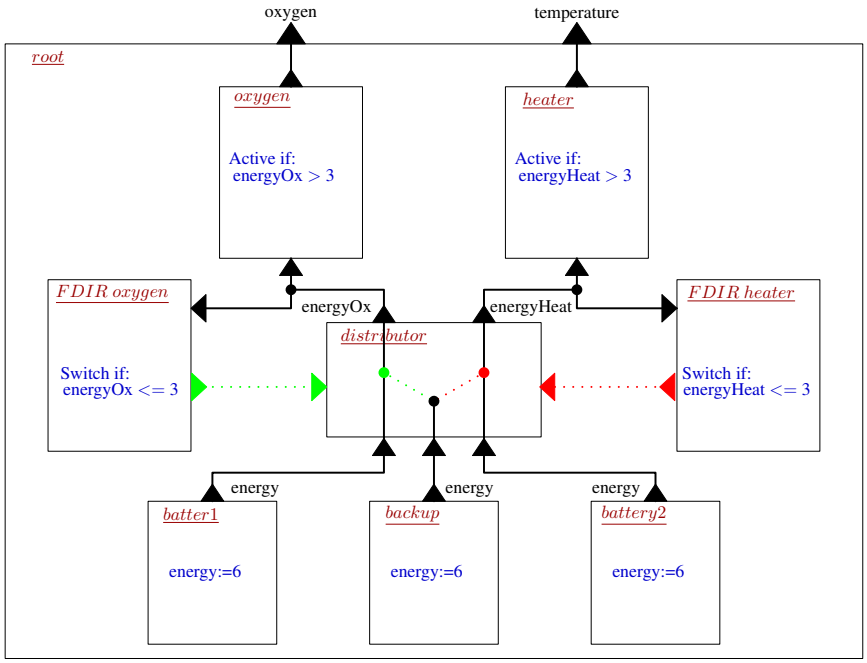


Figure 3.1: Visualization of the life support system model.

discharge and produce only 4 units of energy or after a defect even only 2 units. If one of the batteries only produces 2 units of energy, then the corresponding FDIR monitor activates the backup battery.

Let us assume we modeled this example and after adding the FDIR components we are not sure whether they are doing exactly what we expected. Does the FDIR heater really only affect the heater and the FDIR oxygen only affect the oxygen component? This leads us to the more general question **Q1** "What is an impact in a SLIM model?". The impact relation of the next chapter gives an intuitive approach to analyze the behavior of single components of the system and answers this question.

3.1 The Relations of Transition Impact

Before we define a technique to measure the impact of a component, we need to formalize the environment of the isolation. Therefore, the following definitions are the fundamentals for the impact isolation.

3.1.1 Definitions of the Impact Relation

For this conceptual chapter we need to reduce a SLIM model to the state space of the model. For this, we consider the slightly adapted definition of a transition system of [3]:

Definition 1 (Transition System)
A transition system TS is a tuple (S,T,$\rightarrow$,I) where

- S a set of states.
- T a set of transitions.
- $\rightarrow \subseteq S \times T \times (T \cup \epsilon) \times S$, the transition relation.
- I the initial state.

■

The difference of the definition of a standard transition system is at first, that every state is uniquely defined by the variables of every state. The variables of the states are the data ports, the modes and the data subcomponents of the (extended) SLIM model, in short $var(S)$. The transitions T are the transitions which are defined in the nominal SLIM model under the keyword transitions together with the automatically generated transitions of the error model and fault injection. They are uniquely identified by the label of the SLIM transition (start mode, target mode, event, guard, effect) and the component path of the component which implements the transition. The transition relation $\rightarrow$ is expanded, because SLIM models allow to handshake two transitions. Therefore, the transition relation only contains handshake transitions. Not synchronized transitions are expanded to handshake transitions by adding the empty transition ϵ. For example, if we have states $s_1, s_2 \in S$ and the handshake transitions $t_1, t_2 \in T$ then $(s_1, t_1, t_2, s_2) \in \rightarrow$ if t_1 and t_2 leads from s_1 to s_2. Hereby the order of t_1 and t_2 is not important, meaning that $(s_1, t_1, t_2, s_2) \in T$ if and only if $(s_1, t_2, t_1, s_2) \in T$. If t_1 is not a handshake transition, then we write $(s_1, t_1, \epsilon, s_2) \in \rightarrow$. The probabilities from the error models are ignored in the state space and are always 1, meaning that the probabilistic transitions of the error model are indistinguishable from nominal transitions.

In SLIM it is possible to model the system such that more than one root component is available, but at this point we assume that the root component is chosen before the state space is calculated and therefore the initial state is unique. To simplify the notation, the following short cuts are used in the transition systems:

Definition 2 (Short Cuts)

- We write $s \xrightarrow{t_1,t_2} s'$ for $(s, t_1, t_2, s') \in \rightarrow$.
- We write $s \xrightarrow{t_1} s'$ for $(s, t_1, \epsilon, s') \in \rightarrow$.
- Given $t \in T$ and $s \in S \cup \{\perp\}$ then $t(s) = \begin{cases} s', & \text{if } s \xrightarrow{t} s' \\ \perp, & \text{otherwise} \end{cases}$.
- Given handshake $t_1, t_2 \in T$ and $s \in S \cup \{\perp\}$ then $|t_1 t_2|(s) = \begin{cases} s', & \text{if } s \xrightarrow{t_1,t_2} s' \\ \perp, & \text{otherwise} \end{cases}$.
- We write $active(s)$ for $\{t_1 \in T \mid \exists s' \in S. \exists t_2 \in T \cup \{\epsilon\}. s \xrightarrow{t_1,t_2} s'\}$.

■

Now we will define the impact relation between transitions. The aim of this relation is to detect concurrencies between transitions. Therefore the impact relation is founded in the partial order semantic and especially in Mazurkiewicz's traces [21].

Definition 3 (Impact Relation)
Given are $s \in S$ and $t_1, t_2 \in T$ then t_1, t_2 can be in s in relation Independent ($\|$), Dependent ($\nparallel$), Conflict($\#$) or Enable ($\lhd$). The relations are defined as following:

- $t_1 \parallel t_2$ if $t_1(t_2(s)) = t_2(t_1(s))$ and $t_1(t_2(s)) \neq \perp$.
- $t_1 \nparallel t_2$ if $t_1(t_2(s)) \neq t_2(t_1(s))$ and $t_1(t_2(s)) \neq \perp$ and $t_2(t_1(s)) \neq \perp$.
- $t_1 \# t_2$ if $t_1, t_2 \in active(s)$ and $t_2 \notin active(t_1(s))$.
- $t_1 \lhd t_2$ if $t_2 \notin active(s)$ and $t_2 \in active(t_1(s))$.

Let $\rho \in \{\|, \nparallel, \#, \lhd\}$ then $s \vDash t_1 \; \rho \; t_2$ means that in state s the transition t_1 and t_2 are in relation ρ. ■

A visualization of the definition of the impact relation is shown in the figure 3.2. We see that with the relations Independent ($\|$) and Dependent ($\nparallel$) an important part of the detection of concurrency is covered. Also with Conflict ($\#$) and

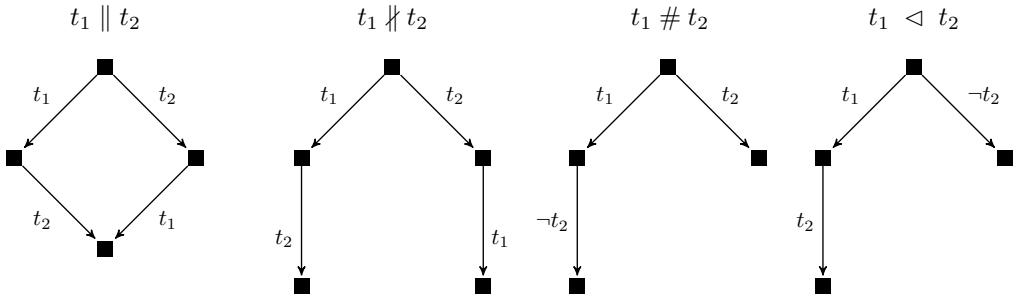


Figure 3.2: The four impact relations $\parallel, \nparallel, \#$ and $\lhd$.

Enable ($\lhd$) it is easy to check whether a transition disables or enables another transition. Another interesting information which follows directly from the definition is the symmetrical behavior. For every state $s \in S$ it holds that $t_1 \parallel t_2$ if and only if $t_2 \parallel t_1$. The same holds for the $\nparallel$ relation. For $\#$ and $\lhd$ does this commutative behavior not hold.

The idea behind this relations is based on the Mazurkiewicz's traces. Therefore let us recapitulate the definition of Mazurkiewicz trace equivalence.

Definition 4 Mazurkiewicz Trace Equivalence
Given the traces $Traces$ of a transition system TS, an alphabet A which are the transitions of TS and an independence relation I. Let $x, y \in Traces$ then we define x and y as Mazurkiewicz trace equivalence ($x \equiv_I y$) if:

$$\exists u, v \in A^* \text{ and } \exists (a, b) \in I \text{ such that } x = uabv \text{ and } y = ubav$$

■

We see that $\equiv_I$ is based on an independent set I which is the equivalent to the impact relation $\parallel$. Now we assume that (a, b) are not in an independent relation. Then the order of the appearance of a and b influences the rest of the trace. For example, if $a \nparallel b$ then it appears that $x = uabv \in Traces$, but $y = ubav \notin Traces$. Hence the relation of the transitions a, b significantly affects the trace. To analyze this relation, it is not enough to consider single traces. We need to find patterns in the transition system. Therefore, we will calculate the state space for searching impact relations.

At this point the question is raised whether the four relations $\parallel, \nparallel, \#$ and $\lhd$ cover every possible combination of transitions in a transition system. This will be shown in the following lemma.

Lemma 1 (Impact Relation Covering)
The impact relation covers every direct combination of two transitions, which both occur in a state and its successors. ■

Proof. For proofing the lemma every possible combination of the activeness of two transitions in a state and its successors will be analyzed by a case-by-case analysis. An overview over the cases gives figure 3.3.

Case: 0 $t_1 \notin active(s) \wedge t_2 \notin active(s)$
Obvious if neither t_1 nor t_2 is active in s. Then this case gives no useful information about the relations of t_1 and t_2, independent from the activeness in the successors. Therefore here is no relation derivable.

Case: 1.1 $t_1 \notin active(s) \wedge t_2 \in active(s) \wedge t_1 \notin active(t_2(s)) \wedge t_2(t_1(s)) = \perp$
In this case only t_2 is active in s and t_1 is neither active in s nor after taking t_2 therefore in this state t_2 does not influence t_1. Therefore also here no relation between t_1 and t_2 is derived.

Case: 1.2 $t_1 \notin active(s) \wedge t_2 \in active(s) \wedge t_1 \in active(t_2(s)) \wedge t_2(t_1(s)) = \perp$
This is the typical case for $t_2 \lhd t_1$. t_1 is not active in s but after taking t_2, t_1 becomes active in $t_2(s)$.

Case: 2.1 $t_1 \in active(s) \wedge t_2 \notin active(s) \wedge t_1(t_2(s)) = \perp \wedge t_2 \notin active(t_1(s))$
This case is symmetric to **Case: 1.1** and delivers no new information for the relation between t_1 and t_2.

Case: 2.2 $t_1 \in active(s) \wedge t_2 \notin active(s) \wedge t_1(t_2(s)) = \perp \wedge t_2 \in active(t_1(s))$
This case is symmetric to **Case: 1.2** and the typical combination for $t_1 \lhd t_2$.

Case: 3.1 $t_1 \in active(s) \wedge t_2 \in active(s) \wedge t_1 \notin active(t_2(s)) \wedge t_2 \notin active(t_1(s))$
In this case even two relations between t_1 and t_2 are derived, because t_1 disables t_2 and t_2 disables t_1, therefore here $t_1 \# t_2$ and $t_2 \# t_1$ are derived.

Case: 3.2 $t_1 \in active(s) \wedge t_2 \in active(s) \wedge t_1 \notin active(t_2(s)) \wedge t_2 \in active(t_1(s))$
This case is the typical combination for $t_2 \# t_1$, because t_1 is active in s but taking t_2 disables t_1.

Case: 3.3 $t_1 \in active(s) \wedge t_2 \in active(s) \wedge t_1 \in active(t_2(s)) \wedge t_2 \notin active(t_1(s))$
As expected this combination is symmetric to **Case: 3.2** and derives $t_1 \# t_2$.

Case	$t_1 \in active(s)$	$t_2 \in active(s)$	$t_1 \in active(t_2(s))$	$t_2 \in active(t_1(s))$	relation
0	0	0	{0,1}	{0,1}	N/A
1.1	0	1	0	N/A	N/A
1.2	0	1	1	N/A	$t_2 \lhd t_1$
2.1	1	0	N/A	0	N/A
2.2	1	0	N/A	1	$t_1 \lhd t_2$
3.1	1	1	0	0	$t_1 \# t_2, t_2 \# t_1$
3.2	1	1	0	1	$t_2 \# t_1$
3.3	1	1	1	0	$t_1 \# t_2$
3.4	1	1	1	1	$t_1 \parallel t_2, t_2 \parallel t_1$ $t_1 \nparallel t_2, t_2 \nparallel t_1$

Figure 3.3: Overview over the cases of the lemma showing which transition is active in which state. 1 means yes, 0 means no and N/A means that this is not possible.

Case: 3.4 $t_1 \in active(s) \wedge t_2 \in active(s) \wedge t_1 \in active(t_2(s)) \wedge t_2 \in active(t_1(s))$
The last case is also the most interesting, because four possible relations are derived. If it holds that $t_1(t_2(s)) = t_2(t_1(s))$ then $t_1 \parallel t_2$ and its symmetric counterpart $t_2 \parallel t_1$ are derived. In the other case that $t_1(t_2(s)) \neq t_2(t_1(s))$ then the symmetric relations $t_1 \nparallel t_2$ and $t_2 \nparallel t_1$ are derived.

We see that every combination in which both transitions occur is covered by at least one relation. Hence no direct impact gets missing during the isolation. □

3.1.2 Handshake Relations

Until now the impact relation is defined for unsynchronized transitions, but how does this definition match with handshake transitions? The basic idea is that in the case of handshake, the relation will be calculated for every handshake pair. In SLIM, it is possible to use multi-way synchronization. This means that one transition performs a handshake with more than one other transition. Depending on this, it follows that a transition can be in one state in multiple impact relations, depending on the handshake partner. With a different handshake partner, a transition differently affects other transitions. This extension is done by extending definition 3 as following:

Definition 5 (Impact Relation for Handshake Transitions)
Given $s \in S$ and $t_1, t_2 \in T$, then t_1, t_2 can be in this state in multiple relations which are defined as following

- $t_1 \parallel t_2$ if $\exists t_{h_1}, t_{h_2} \in T \cup \{\epsilon\}$ such that
  $|t_1 t_{h_1}|(|t_2 t_{h_2}|(s)) = |t_2 t_{h_2}|(|t_1 t_{h_1}|(s))$ and $|t_1 t_{h_1}|(|t_2 t_{h_2}|(s)) \neq \perp$.
- $t_1 \nparallel t_2$ if $\exists t_{h_1}, t_{h_2} \in T \cup \{\epsilon\}$ such that
  $|t_1 t_{h_1}|(|t_2 t_{h_2}|(s)) \neq |t_2 t_{h_2}|(|t_1 t_{h_1}|(s))$ and
  $|t_1 t_{h_1}|(|t_2 t_{h_2}|(s)) \neq \perp$ and $|t_2 t_{h_2}|(|t_1 t_{h_1}|(s)) \neq \perp$.
- $t_1 \# t_2$ if $\exists t_{h_1} \in T \cup \{\epsilon\}$ holds that $t_2 \in active(s)$ and $t_2 \notin active(|t_1 t_{h_1}|(s))$.
- $t_1 \lhd t_2$ if $\exists t_{h_1} \in T \cup \{\epsilon\}$ $t_2 \notin active(s)$ and $t_2 \in active(|t_1 t_{h_1}|(s))$.

■

With this definition we ignore the handshake case, because it is just an extension of the old definition by finding the fitting handshake pairs. From now on we concentrate only on the impact relation for unsynchronized transitions. This will reduce the complexity of the following concepts. We will keep in mind that the handshake transitions are treated the same way by just lifting the definitions.

Summary of the Impact Relation

In this section the impact relation for transitions was defined and gives us the necessary basics to develop an impact isolation for SLIM. The fundamental question **Q1** "What is an impact in a SLIM model?" is answered. Impact in a SLIM model is from now on the impact relation between two different transitions. With this fundamentals we will develop other concepts which expand the impact isolation.

3.2 Counting Matrix Representation

Just analyzing the relation between two transitions by looking at one state is not significant enough, in particular if as seen in the section 2.1 the effect of a transition depends on the value of the variables of the current state. This leads to the questions **Q2** "How can we calculate the impact of a component" and **Q3** "How can the impact be visualized?". The solution to these questions is inspired by the representation of concurrencies by an ordering matrix [29]. The basic idea is still

to detect the impact of a component to other components by comparing two transitions. Therefore now we consider two sets of transitions and comparing pairwise the transitions of the two sets. These transitions are compared by summing over the appearance of the relation in every state of the state space. For this, we need to define a counting matrix.

3.2.1 Definitions of the Counting Matrix

Definition 6 (Counting Matrix)
Given the transition system $Space$ = (S,T,$\rightarrow$,I) and $T_1, T_2 \subseteq T$ with $T_1 = \{t_1^1, \ldots, t_n^1\}, T_2 = \{t_1^2, \ldots, t_m^2\}$ two ordered sets of transitions, then a counting matrix M^ρ for relation $\rho \in \{\|, \nparallel, \#, \lhd\}$ is defined as:

$$M^\rho := M^{|T_1|\times|T_2|} \text{ with } a_{ij} = |\{s \in S \mid t_i^1 \in T_1 \wedge t_j^2 \in T_2 \wedge s \vDash t_i^1 \ \rho \ t_j^2\}|$$

■

A simple algorithm for calculating the counting matrix is shown in figure 3.4. The function pos gives the position of the transition in the ordered sets $pos(t_j) = j$ for $t_j \in \{t_1, \ldots, t_n\}$. Important in the definition of the counting matrix and the application of the algorithm is that only relations of the structure $t_1 \ \rho \ t_2$ for $t_1 \in T_1$ and $t_2 \in T_2$ are calculated. In particular, the non-commutative relations $t_2 \ \# \ t_1$ and $t_2 \lhd t_1$ are not calculated. This is a desired behavior, because we want to know the effect of the first component on the second. Therefore, we want to know whether a transition in the first component disables or enables a transition of the second component and not the other way round. To make the counting matrix more readable we do a normalization transformation of the absolute counted appearances of the relations to its discrete occurrence probabilities. This means for all $i, j \in \{1, \ldots, n\}$ and $\rho_1, \rho_2, \rho_3, \rho_4 \in \{\|, \nparallel, \#, \lhd\}$ pairwise distinct we do relative$(a_{i,j}^{\rho_1}) := \frac{a_{i,j}^{\rho_1}}{\sum_{i=1,\ldots,4} a_{i,j}^{\rho_i}}$.

To make the concept of the counting matrix more understandable, we now apply the counting matrix on the life support system example.

3.2.2 Applying the Counting Matrix

We saw the life support system in the beginning of this chapter. For creating this SLIM model the techniques of this chapter were applied during the full modeling

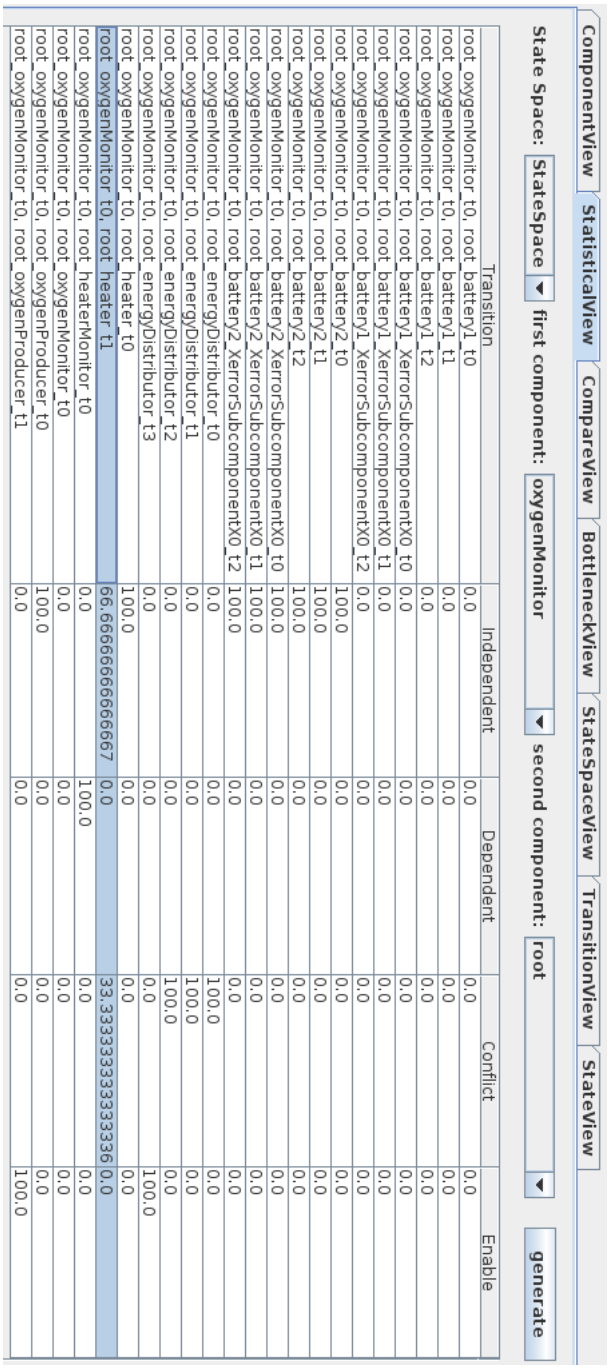
Algorithm Counting matrix M^{ρ} for T_1, T_2.

$a_{pos(t_1),pos(t_2)} := 0 \;\; \forall t_1 \in T_1, \forall t_2 \in T_2$
for all $s \in S$ **do**
  for all $t_1 \in |T_1| \cap active(s)$ **do**
    for all $t_2 \in |T_2| \cap (actice(s) \cup active(t_1(s))$ **do**
      if $s \vDash t_1 \; \rho \; t_2$ **then**
        $a_{pos(t_1),pos(t_2)} := a_{pos(t_1),pos(t_2)} + 1$

Figure 3.4: Algorithm for calculating a counting matrix.

process. The first interesting result was found during the creation of the nominal SLIM model in appendix A.2.1. Just the possibility to calculate the full state space and to visualize it for the developer is a useful support. Minor mistakes like the mix up of “oxygen>3” with “oxygen<3” in the guard of a transition is conspicuous in small state spaces like the life support system. Nevertheless for subsystems with less than 100 states, calculating the state space is useful. But for bigger or strongly connected models, it is nearly useless. If we add the error model from appendix A.2.2 and the fault injection from appendix A.2.3, the model has 425 states and is highly connected, therefore observing the pure state space is not profitable anymore.

Therefore, we will analyze the impact of the FDIR oxygen monitor to the rest of the system. For this, we calculate the counting matrix for the transitions of the FDIR oxygen monitor in the first set. The second set contains all transitions of the system. The picture of the counting matrices is displayed in figure 3.5. The picture was created by the impact tool (which will be presented in section 5.3) and needs the translation of the transition short names of figure 3.6. Both together reveal a possible unexpected impact for the developer. The transition of the FDIR oxygen is in $33,\bar{3}\%$ with a heater transition in the conflict relation. This behavior was not expected during the modeling process, because the switch of the backup battery to the oxygen component should not disable a transition in the heater. We see that the counting matrix delivers the answers for the questions **Q2** “How can we calculate the impact of a component?” and **Q3** “How can the impact be visualized?”. We have a technique to detect and display the impact of different components. However, we also found a new problem. The counting matrix does not contain enough information to find a reason for the two different



Transition	Independent	Dependent	Conflict	Enable
root_oxygenMonitor_t0, root_battery1_t0	0.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_battery1_t1	0.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_battery1_t2	0.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_battery1_XerrorSubcomponentX0_t0	0.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_battery1_XerrorSubcomponentX0_t1	0.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_battery1_XerrorSubcomponentX0_t2	0.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_battery2_t0	100.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_battery2_t1	100.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_battery2_t2	100.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_battery2_XerrorSubcomponentX0_t0	100.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_battery2_XerrorSubcomponentX0_t1	100.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_battery2_XerrorSubcomponentX0_t2	100.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_energyDistributor_t0	0.0	0.0	100.0	0.0
root_oxygenMonitor_t0, root_energyDistributor_t1	0.0	0.0	100.0	0.0
root_oxygenMonitor_t0, root_energyDistributor_t2	0.0	0.0	100.0	0.0
root_oxygenMonitor_t0, root_energyDistributor_t3	0.0	0.0	0.0	100.0
root_oxygenMonitor_t0, root_heater_t0	100.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_heater_t1	66.66666666666667	0.0	33.33333333333336	0.0
root_oxygenMonitor_t0, root_heaterMonitor_t0	0.0	100.0	0.0	0.0
root_oxygenMonitor_t0, root_oxygenMonitor_t0	0.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_oxygenProducer_t0	100.0	0.0	0.0	0.0
root_oxygenMonitor_t0, root_oxygenProducer_t1	0.0	0.0	0.0	100.0

Figure 3.5: Counting matrix of the life support system

ComponentView | StatisticalView | CompareView | BottleneckView | StateSpaceView | TransitionView | StateView

Name	Component	Label
root_battery1_t0	root_battery1	DefaultInitialMode-[XerrorSubcomponentX0.#discharge when (XerrorStateX0) = (_OK) then (energy) := (4);]-> DefaultInitialMode
root_battery1_t1	root_battery1	DefaultInitialMode-[XerrorSubcomponentX0.#deffect when (XerrorStateX0) = (_Discharged) then (energy) := (2);]-> DefaultInitialMode
root_battery1_t2	root_battery1	DefaultInitialMode-[XerrorSubcomponentX0.#deffect when (XerrorStateX0) = (_OK) then (energy) := (2);]-> DefaultInitialMode
root_battery1_XerrorSubcomponentX0_t0	root_battery1_XerrorSubcomponentX0	OK-[#discharge then (XerrorStateX0) := (_Discharged);]->Discharged
root_battery1_XerrorSubcomponentX0_t1	root_battery1_XerrorSubcomponentX0	Discharged-[#deffect then (XerrorStateX0) := (_Defect);]->Defect
root_battery1_XerrorSubcomponentX0_t2	root_battery1_XerrorSubcomponentX0	OK-[#deffect then (XerrorStateX0) := (_Defect);]->Defect
root_battery2_t0	root_battery2	DefaultInitialMode-[XerrorSubcomponentX0.#discharge when (XerrorStateX0) = (_OK) then (energy) := ((energy) - (2));]-> DefaultInitialMode
root_battery2_t1	root_battery2	DefaultInitialMode-[XerrorSubcomponentX0.#deffect when (XerrorStateX0) = (_Discharged) then (energy) := (2);]-> DefaultInitialMode
root_battery2_t2	root_battery2	DefaultInitialMode-[XerrorSubcomponentX0.#deffect when (XerrorStateX0) = (_OK) then (energy) := (2);]-> DefaultInitialMode
root_battery2_XerrorSubcomponentX0_t0	root_battery2_XerrorSubcomponentX0	OK-[#discharge then (XerrorStateX0) := (_Discharged);]->Discharged
root_battery2_XerrorSubcomponentX0_t1	root_battery2_XerrorSubcomponentX0	Discharged-[#deffect then (XerrorStateX0) := (_Defect);]->Defect
root_battery2_XerrorSubcomponentX0_t2	root_battery2_XerrorSubcomponentX0	OK-[#deffect then (XerrorStateX0) := (_Defect);]->Defect
root_energyDistributor_t0	root_energyDistributor	noBackup-[switchHeat]->heatSupport
root_energyDistributor_t1	root_energyDistributor	heatSupport-[switchOx]->oxySupport
root_energyDistributor_t2	root_energyDistributor	noBackup-[switchOx]->oxySupport
root_energyDistributor_t3	root_energyDistributor	oxySupport-[switchHeat]->heatSupport
root_heater_t0	root_heater	initHeat -[when (temperature) > (1) then (temperature) := ((temperature) - (1));]-> initHeat
root_heater_t1	root_heater	initHeat -[when ((energy) > (3)) and ((temperature) < (5)) then (temperature) := ((temperature) + (1));]-> initHeat
root_heaterMonitor_t0	root_heaterMonitor	initHeatMo-[switchToHeat when (heaterEnergy) <= (3)]->switchedHeatMo
root_oxygenMonitor_t0	root_oxygenMonitor	initOxMo-[switchToOx when (oxyEnergy) <= (3)]->switchedOxMo
root_oxygenProducer_t0	root_oxygenProducer	initOx -[when (oxygen) > (1) then (oxygen) := ((oxygen) - (1));]-> initOx
root_oxygenProducer_t1	root_oxygenProducer	initOx -[when ((energy) > (3)) and ((oxygen) < (5)) then (oxygen) := ((oxygen) + (1));]-> initOx

Figure 3.6: Transition translation of the life support system

relations. This leads to the next general question **Q4** "How can deviating impacts be analyzed?". The answer for this question will be presented in the next section.

3.3 Comparing Different Related States

The basic idea of this section is to use the information of the counting matrix to create an example path. This path shows the difference between two states with two identical transitions, but in different relations. This example path will then contain enough information to show the developer why the unexpected behavior occurs. The definition of these comparing paths is as following.

3.3.1 Definitions of the Comparing Paths

Definition 7 (Comparing Paths)
Given $t_1, t_2 \in T$ and two states $s^{\rho_1}, s^{\rho_2} \in S$ with $s^{\rho_1} \vDash t_1\ \rho_1\ t_2$ and $s^{\rho_2} \vDash t_1\ \rho_2\ t_2$ with $\rho_1, \rho_2 \in \{\|, \nparallel, \#, \lhd\}$ and $\rho_1 \neq \rho_2$. Additionally let s_0 be the (not necessarily unique) common nearest predecessor of s^{ρ_1} and s^{ρ_2}. Then the comparing paths p_1, p_2 are defined as:

- p_1 is the shortest path between s_0 and s^{ρ_1}.
- p_2 is the shortest path between s_0 and s^{ρ_2}.

If s_0 does not exist then also the p_1 and p_2 do not exist. ■

The algorithm for calculating the comparing path is sketched in figure 3.7 and the simple idea is that in s^{ρ_1} and in s^{ρ_2} starts a backward breadth-first search. These two searches are synchronized, which means that in every iteration only the next level of the predecessors are added to the visited states. In figure 3.7 these levels are visualized by the differently colored layers. At some point a (not necessarily unique) state is visited by both searches and will be defined as the state s_0. Now from this state the shortest path back to the start states is easily calculated. It fulfills exactly the definition of the comparing paths.

The next problem is how we can use the information on these paths for understanding the different behaviors of the two transitions. For analyzing the paths we need more definitions. Keep in mind that these definitions are only for non-handshake transitions, the expansions to handshake transitions are analogous to section 3.1.2.

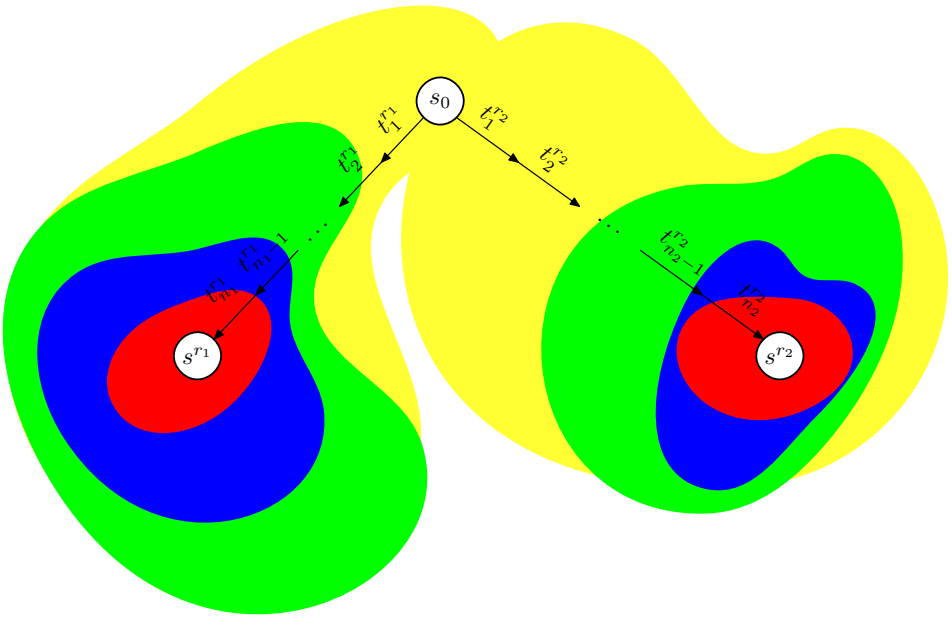


Figure 3.7: Sketch of a synchronized backward breadth-first search for calculating the comparing paths.

Definition 8 (Effect of a Transition)
Let $s, s' \in S$ then the difference of two states is defined as:

$$\text{diff}(s, s') := \{v \in var(S) \mid val(v[s_1]) \neq val(v[s_2])\}$$

With $val(v[s])$ is the value of the variable v in state s. Let t be the transition of the state change $s \xrightarrow{t} s'$ then the effect of t is defined as:

$$e_t := \text{diff}(s, s')$$

■

Definition 9 (Important Effect of a Transition)
Let t be the transition of the state change $s \xrightarrow{t} s'$ and $s_1, s_2 \in S$ the states of interest, then the important effect of a transition is

$$ie_t(s_1, s_2) := e_t \cap \text{diff}(s_1, s_2)$$

■

Then the output of the analysis of the comparing paths is defined as following.

Definition 10 (Output of Comparing Paths)
Let $p_1 = s_0^1, s_1^1, \ldots, s_n^1$ and $p_2 = s_0^2, s_1^2, \ldots, s_m^2$ the two comparing paths with $s_0^1 = s_0^2$ the starting state, then the output sets $\text{set}_1, \text{set}_2$ are defined as:

$$\text{set}_{\rho_1} := \{(s_i^1, t_i^1, s_{i+1}^1, ie_{t_i^1}(s_0^1, s_n^1)) \mid s_i^1 \xrightarrow{t_i^1} s_{i+1}^1 \wedge i \in \{1, \ldots, n\}\}$$

and

$$\text{set}_{\rho_2} := \{(s_i^2, t_i^2, s_{i+1}^2, ie_{t_i^2}(s_0^2, s_m^2)) \mid s_i^2 \xrightarrow{t_i^2} s_{i+1}^2 \wedge i \in \{1, \ldots, n\}\}$$

■

In other words, we take for every transition on both comparing paths the start and target states and output the important effect of the transition. To see the usefulness of these comparing paths let us continue with our example of the life support system.

3.3.2 Applying the Comparing Paths

We saw that in the end of the first part of the life support system example the FDIR oxygen transition is in conflict with the heating transition of heater. Let us from now on call these transitions t_m for the monitor transition and t_h for the heater transition. Just to remind, these two transitions are in $66, \bar{6}\%$ in the relation $t_m \parallel t_h$ and in $33, \bar{3}\%$ in relation $t_m \# t_h$.

The first task is to find an adequate $s^{\parallel}$ and a $s^{\#}$ for the comparing paths algorithm. A good heuristic for choosing these states is taking two states which are pretty close together in the state space, such that the comparing paths are also short and significant. One result of the comparing paths is then:

$$\text{set}_{\parallel} = \emptyset$$

$$\text{set}_{\#} = \{(254, root.battery2\_t1/root\_battery2\_errorSubcomponent\_t1, 155, ie_1),$$
$$(155, root\_energyDistributor\_t0/root\_heaterMonitor\_t0, 31, ie_2)\}$$

$$ie_1 = \{root.energyDistributor.inputHeat : 4 \rightarrow 2,$$
$$root.battery2.errorState : 2 \rightarrow 1,$$
$$root.heater.energy : 4 \rightarrow 2,$$

$root.battery2.energy : 4 \rightarrow 2,$
$root.energyDistributor.outputHeat : 4 \rightarrow 2,$
$root.battery2.errorSubcomponent.mode : Discharged \rightarrow Defect,$
$root.battery2.errorSubcomponent.errorState : 2 \rightarrow 1,$
$root.heaterMonitor.heaterEnergy : 4 \rightarrow 2\}$

$ie_2 = \{root.heater.energy : 2 \rightarrow 8,$
$root.heaterMonitor.mode : 12 \rightarrow 11,$
$root.energyDistributor.mode : noBackup \rightarrow heatSupport,$
$root.energyDistributor.outputHeat : 2 \rightarrow 8,$
$root.heaterMonitor.heaterEnergy : 2 \rightarrow 8\}$

To make the important effect more expressive, the values of the variables before and after taking the transition are added in this example. Remember that in appendix 3.6 there is a translation for the transition short cuts, which are necessary to understand this example. At first, we consider the $\text{set}_{\|}$, which is empty so we know that the state for creating the $\|$ relation is the starting point for the analysis and the path is only this state itself.

Let us continue with the $\#$ relation. We see in the ie_1 of the handshake transition $root.battery2\_t1/root\_battery2\_errorSubcomponent\_t1$ that the energy of the heater drops to 2 energy units, because the battery2 error state changed to defect. Now we know that battery1 is defect, because the t_m transition is still active. Additionally the battery2 is defect, because the handshake transition $root\_energyDistributor\_t0/root\_heaterMonitor\_t0$ is in the next step of the path activated. This transition switches the backup battery to the heater. Hence now if t_m is taken, the oxygen component is resupplied with energy, but the heater is disabled by this transition. Now the developer knows that if both batteries are defect, only the heater or the oxygen component is active at one time. Based of this knowledge the developer can remodel the system and will find a better solution like splitting the energy of the backup battery batteryB to both components to avoid the mutual disturbing. So we saw that comparing paths will find an answer for the question **Q4** "How can deviating impacts be analyzed?".

3.4 Transitive Closure over the Counting Matrix

We saw that the impact relation detects the direct impact of one component to another component. However, normally the aim of an impact isolation is to detect

the impact over several steps. This is difficult to detect with normal analyzing techniques. Assuming we are in the satellite case study, for example, if we have a failure in the earth sensor and this will be fixed by the responsible FDIR component. Then this change affects the control unit which manages the whole system. Followed by another step in this unit the propulsion system will be involved. Then the change in the propulsion system will cause a new effect in another sensor device. Is there a possibility to detect this impacts over several steps? Or more generally **Q5** "How can indirect impacts be detected?".

We saw in the chapter 2.2.1, about the static impact analysis, that we used a dependency matrix to create an impact analysis. The weakness of this strategy is an over-approximation over the impact of the components. If we accept this weakness, then we can augment the impact isolation by building the transitive closure over the counting matrices. With the counting matrices we have not only the advantage to detect impacts. We also identify whether there is an enabling ($\lhd$), disabling ($\#$), or ordering ($\nparallel$) effect. This advantage has to be preserved. Therefore, the transitive closure over the counting matrices will take the relations into account. The first step to reach this aim is to define the concatenation of the impact relation.

Definition 11 (Concatenation of the Impact Relation)
Given are the transitions t_i,t_j and t_k which are in relation $\nparallel, \lhd$ or $\#$. Then the concatenation of the impact relation is defined as following:

$$\frac{t_i \lhd t_k \wedge t_k \lhd t_j}{t_i \lhd t_j} \qquad \frac{t_i \# t_k \wedge t_k \lhd t_j}{t_i \# t_j} \qquad \frac{t_i \lhd t_k \wedge t_k \# t_j}{t_i \# t_j}$$

$$\forall \rho \in \{\nparallel, \#, \lhd\} : \frac{t_i \; \rho \; t_k \wedge t_k \nparallel t_j}{t_i \nparallel t_j} \qquad \frac{t_i \nparallel t_k \wedge t_k \; \rho \; t_j}{t_i \nparallel t_j}$$

■

This concatenation rules cover every combination between $\lhd$, $\nparallel$ and $\#$ as first relation and $\lhd$, $\nparallel$ and $\#$ as second relation with the exception of $t_i \# t_k \wedge t_k \# t_j$. The reason for ignoring this rule is that in this situation t_i, t_k and t_k are all active in the same state. This means that t_i and t_j are already in a relation through this state. Therefore, creating a new impact with $t_i \# t_k \wedge t_k \# t_j$ gives in the best case no information gain and else just an over-approximation.

Also, some of the rules are more obvious than other rules. For example, let us analyze the rule $\frac{t_i \lhd t_k \wedge t_k \# t_j}{t_i \# t_j}$ in greater detail. Considering the left example in figure 3.8, we see that t_i needs to be taken to activate t_k. Then t_k disables t_j. So

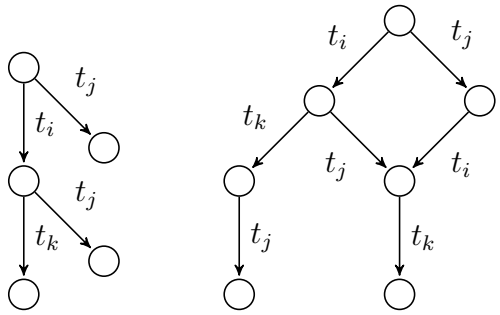


Figure 3.8: Example of $\frac{t_i \lhd t_k \wedge t_k \# t_j}{t_i \# t_k}$ (left) and $\frac{t_i \lhd t_k \wedge t_k \nparallel t_j}{t_i \# t_k}$ (right).

we say through the enabling effect from t_i to t_k it is possible in the next step to disable t_j. Therefore we conclude that $t_i \# t_j$.

On the other hand, let us consider the rule $\frac{t_i \lhd t_k \wedge t_k \nparallel t_j}{t_i \nparallel t_j}$ and the right example in figure 3.8. We see that $t_i \parallel t_j$ holds in this example, but also we see that t_i enables t_k which is in relation $t_k \nparallel t_j$. Now we conclude that the appearance of t_i performs an indirect ordering impact on t_j, because it enables t_k which has an ordering impact. We see in this example that it is not easily apparent what exactly is an indirect impact. Nevertheless, for the following all of the rules create an indirect impact which will be used to expand the impact isolation.

To use this concatenation of the impact relation as transitive closure, we apply an adapted variant of the Floyd-Warshall algorithm [15] which is presented in figure 3.9. To initialize the algorithm, let $\forall_{1 \leq i \leq n} \forall_{\rho \in \{\nparallel, \lhd, \#\}} : a^{\rho}_{i,i} := 0$. The correctness of this adapted algorithm follows directly from the correctness of the Floyd-Warshall algorithm, because the three assignments only affect each other with values of previous iterations. E.g. $a^{\lhd}_{i,j}$ does not depend on $a^{\#}_{i,j}$, which was calculated in the same iteration. This is only the case if $k = i$ or $k = j$, but in both cases $a^{\rho}_{i,k} = 0$ or $a^{\rho}_{k,j} = 0$ for all ρ.

The $\odot$ and $\oplus$ of figure 3.9 are place holder operations, where $\odot$ has a higher precedence than $\oplus$. We propose two different assignments. The first proposal is to use for $\oplus$ the addition and for $\odot$ the multiplication. With this, we are able to use the values of the counting matrix without any modifications. Then the result of the closure is derived from the relative values of the relations. However the interpretation of these values is difficult. It gives an approximation how likely this transitive relation is in the state space. Another more intuitive proposal is the use of the minimum for $\oplus$ and the addition for $\odot$. With this assignment, the Floyd-

Algorithm Transitive closure of counting matrices.

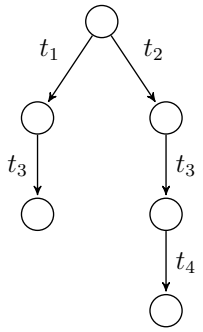
for all $k = 1 \dots n$ **do**
  for all $i = 1 \dots n$ **do**
    for all $j = 1 \dots n$ **do**
      $a^{\#}_{i,j} = a^{\#}_{i,j} \oplus a^{\triangleleft}_{i,k} \odot a^{\#}_{k,j} \oplus a^{\#}_{i,k} \odot a^{\triangleleft}_{k,j}$;
      $a^{\triangleleft}_{i,j} = a^{\triangleleft}_{i,j} \oplus a^{\triangleleft}_{i,k} \odot a^{\triangleleft}_{k,j}$;
      $a^{\nparallel}_{i,j} = a^{\nparallel}_{i,j} \oplus a^{\triangleleft}_{i,k} \odot a^{\nparallel}_{k,j} \oplus a^{\#}_{i,k} \odot a^{\nparallel}_{k,j} \oplus a^{\nparallel}_{i,k} \odot a^{\nparallel}_{k,j} \oplus a^{\nparallel}_{i,k} \odot a^{\triangleleft}_{k,j} \oplus a^{\nparallel}_{i,k} \odot a^{\#}_{k,j}$;

Figure 3.9: Algorithm for the transitive closure of the counting matrices.

Warshall algorithm calculates the shortest path between these relations. This is useful, because we also have besides the impact and the relation of the transitions the minimal number of steps which are needed to generate this impact.

We see that the introducing problem of this section is solved with the transitive closure over the counting matrix. Also we are able to detect dependencies over several components. Nevertheless, also the mentioned over-approximation is obvious. Figure 3.10 shows a cutout of a state space where the relation between t_1 and t_4 will be calculated. The left reasoning drawn in red shows the over-approximation which leads to the result that t_1 enables t_4, which is obviously false. The correct reasoning drawn in green on the right side, shows the correct example that t_1 disables t_4. If we accept the over-approximation of the transitive closure of the counting matrix, then in this section, we found an answer to the question **Q5** "How can indirect impacts be detected?".

Unfortunately, depending on the number of transitions, the table of the transitive closure is not well-arranged. A clearer visualization is to choose one transition which needs to be analyzed and then build an impact graph starting from this transition. This is just the unfolding of the transitive closure starting on the given transition. Another interesting feature is that in this visualization it is easily possible to cut branches of impacts by ignoring some relations. This is useful if a few relations invoke an impact explosion or the user is sure that these impacts are an over-approximation.



$$\frac{t_1 \lhd t_3 \wedge t_3 \lhd t_4}{t_1 \lhd t_4} \qquad \frac{\dfrac{t_1 \# t_2 \wedge t_2 \lhd t_3}{t_1 \# t_3} \quad \wedge \quad t_3 \lhd t_4}{t_1 \# t_4}$$

Figure 3.10: Example for over-approximation of the transitive closure.

3.5 Decisions Which Lead to the Current Analyzing Approach

The basic idea of the transition based impact isolation of this chapter was to calculate the full state space of the SLIM model and to search for patterns in the transitions which gives the developer the possibility to detect the influence of the created FDIR components on other components in the model. Three decisions are mainly responsible for this concept.

- **The first decision: Develop a dynamic impact isolation**
  In the beginning, there was the choice between a static or dynamic isolation approach. The static analysis of SLIM models is with the slicing of SLIM models [22] already a part of the COMPASS project and we also know that the static impact is in many cases a strong over-approximation. Additionally, the effect of a transition on the data ports of the system is not obvious, since through mode depending flows or connections the effect of a transition can affect different parts of the system depending on the current state. All this leads to the decision to use a dynamic approach to develop the impact isolation.

- **The second decision: Calculating the full state space for the isolation**
  The calculation of the full state space is founded in the whole path-based dynamic impact analysis [19]. The weakness (or the strength depending on the view point) of this analysis is that not every path is covered in sufficiently large models. Therefore, the resulting impact set is not assured to be complete. If we use the whole state space as input then we have all

information to calculate all impacts. Of course, the calculation of the full state space is not adequate for high performance analysis, because of the state space explosion problem. But to demonstrate the concepts in this thesis it will be sufficient. Reducing the input of the impact isolation is still an interesting task for future work on this topic.

- **The third decision: Only use transitions to detect the impact**
  Also this decision is not finite. Of course, impacts can also be detected by effects on variables, but for this thesis, the impact isolation has been limited to transitions. Expanding the concepts to values of variables is part of the future work on the impact isolation. In particular for making the transitive closure more accurate.

Summary of the Impact Isolation

We saw in the example of the life support system that even in small models the combination of error cases can lead to an unexpected behavior which was not considered during the modeling process. Finding such behavior through model checking is not easy, because the properties of unexpected behavior are of course unknown. Therefore the impact isolation approach in this thesis gives the developer the possibility to find such behavior just by comparing the transitions of components. In this chapter we found the answers to the questions **Q1** till **Q5**. However calculating a full state space and executing the impact algorithms on the state space is time and memory consuming. Therefore, the next chapter will deal with optimization approaches to reduce the state space or to execute the impact isolation more accurately by reducing the search from the full state space to only a subspace.

Chapter 4

State Space Optimizations

This chapter is motivated by the two important questions **Q6** "How can we make the impact isolation more accurate?" and **Q7** "How can we make the impact isolation faster?". Let us start with question **Q6** and consider the small part of the satellite case study in figure 4.1. This is only a very small cutout of the case study which is also strongly simplified to explain the intuition for this part of the model. A more detailed description of the same part of the system is given in the case study of section 5.4.

In figure 4.1 there are three components. At first, there is the AOCS component which is the global mode manager of the satellite. It is modeled like a finite state machine with the states containing the current mode. The states are the normal mode (NOM) which represents the standard behavior of the system. The planet acquisition mode (PAM) is used if the satellite is using the earth sensors to determine the current position. The sun acquisition mode (SAM) is used if the satellite is using the sun sensors alternatively. The orbit change mode (OCM) is used to readjust the path of the satellite. These are the standard modes of the satellite. The advanced modes are the degraded mode (DM), which is used if special operations are executed like the FDIR component. If during this execution a fault is detected (for example a sensor and its backup component are defect) the system will change to the safety mode (SM). The state change in the AOCS is synchronized by the corresponding events. These events are multi-way synchronized with different components of the system which need to change the current mode. E.g. the FDIR components which only executes in the DM. Also many different functionalities depend on the current mode and receive this information over the out data port $current\_mode$. For example, in SAM, the earth sensors are not supplied with energy.

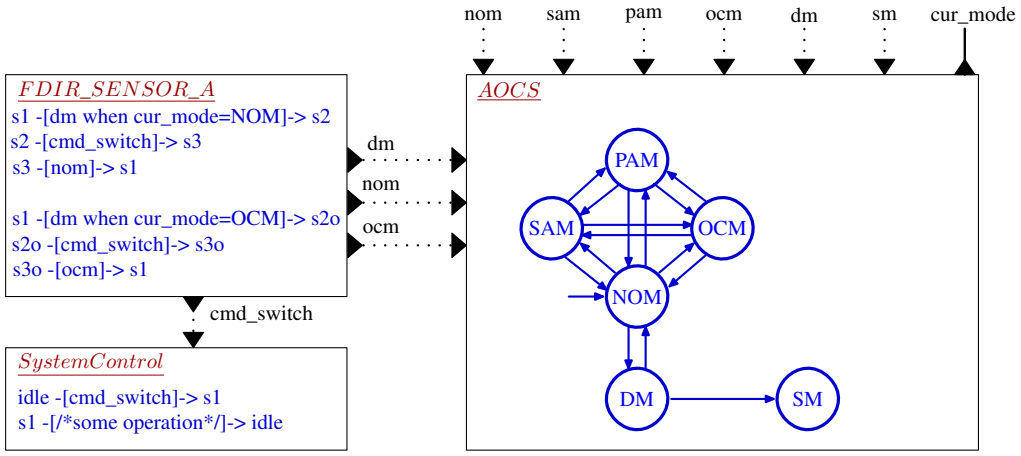


Figure 4.1: Visualization of the mode synchronization in the satellite case study.

The second component is the FDIR_SENSOR_A which reacts on a fault of the system, which is not specified in this example. The sensor contains two different FDIR routines which depend on the current mode of the AOCS component. Both routines react on the same fault, but which is invoked depends on the current mode of the AOCS. The first routine reacts if the current mode is PAM and the second mode reacts if the current mode is OCM.

Every FDIR routine is executed in the DM mode. Therefore, both components use a transition to send the dm event to the AOCS component. After this the FDIR operations will be executed. Finally, after the execution depending on the previous mode the event nom or ocm will be send to the AOCS component. Both routines react in this simplified example equally. The second transition of both routines is synchronized with the SystemControl and is used to execute the necessary changes in the system. This is again not described in detail at this point.

After the development of this part of the satellite, the developer will be confronted with the question whether the defect solving routine of the FDIR component during the DM mode can affect different parts of the system. With the counting matrix we only calculate the direct impact depending on the local operation of the FDIR component. On the other hand, if we are using the transitive closure for calculating the full impact we have two problems. The first problem is that through the transition s3 -[nom]-> done, the FDIR routine impacts the AOCS component. From this point an impact explosion takes place, because nearly every part of the system depends on the current mode of the AOCS. How do we

avoid this explosion? The second problem is that both routines are related to the SystemControl and this leads to unexpected side effects. This leads to the question: How can we limit our impact isolation only to the FDIR routine and ignore the impacts to the AOCS? The proposed technique separates the state space and focuses on the impact isolation on a single subspace.

4.1 Partition of the State Space with Bottlenecks

The first idea of creating a partition of the state space is quite simple. We take the full state space as an undirected graph, choose a transition and remove it from the state space. Then we start a search from the initial node and calculate the connected components of the state space. The result is a set of subspaces which are only connected by the removed transition. To make this algorithm more formal we will use the definition of the transition system of the state space to define the partitioned transition system.

Definition 12 (Partitioned Transition System)
Given a transition system $TS = (S, T, \rightarrow, I)$ which represents the state space for the partition and a bottleneck transition $t_B \in T$ (and $t_{B_2} \in T$ if the transition is handshaked) then the partitioned transition system $PTS = \{TS_1, \ldots, TS_n\}$ with $TS_i = (S_i, T_i, \rightarrow_i, I_i)$ and

- State complete: $\bigcup_{i=1\ldots n} S_i = S$
- State distinct: $\forall_{i \neq j} : S_i \cap S_j = \emptyset$
- Bottleneck partition: $\forall_{(s,t_1,t_2,s') \in \rightarrow}((s \in S_i \wedge s' \in S_j \wedge i \neq j) \rightarrow (t_1 = t_B \wedge t_2 = t_{B_2}))$
- Transition relation complete: $\bigcup_{i=1,\ldots,n} \rightarrow_i \; = \; \rightarrow \setminus \{(s, t_1, t_2, s') \in \rightarrow \mid s \in S_i \wedge s' \in S_j \wedge i \neq j\}$
- Initial states: $I_i = \{s' \in S_i \mid (s, t_B, t_{B_2}, s') \in \rightarrow \wedge s \in S_j \wedge i \neq j\} \cup (S_i \cap I)$

■

With this definition we are able to directly apply the algorithms of chapter 3 on a subspace, because every subspace is again a transition system and therefore all necessary preconditions are satisfied. Also, this algorithm can recursively be applied to the subspaces to continue the separation. Now we have an idea how

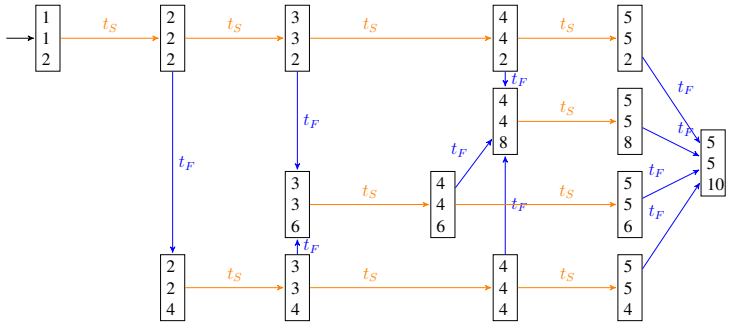


Figure 4.2: Sensorfilter without fault injection state space.

to generate the partition so let us again consider the problems of the satellite case study.

Applying the State Space Partition

The concrete state space separation of the satellite case study is done in section 5.4. At this point we will discuss the expected result. Therefore we use the transition s3 -[nom]-> s1 as the bottleneck, because we assume that this transition separates the state space and we can analyze the subspace which contains the FDIR routine. The result of this choice of the bottleneck transition leads to the problem that the state space will not be separated. This transition is not enough to create two different components. Therefore we are not able to apply the bottleneck partition. The reason for this and the result will be explained in the next subsection.

4.1.1 Improvements of the Bottleneck

We saw that the technique of the partition of the state space is not suitable for each model. The reason is that the most realistic SLIM models are highly cyclic and strongly connected, in particular spacecraft systems. Therefore, only removing one transition is not enough to create a separation in the state space. For example, in case study the FDIR transition which was used for the bottleneck is not enough to create a partition. If we use the s2 -[cmd_switch]-> s3 together with the s1 -[dm when cur_mode=NOM]-> s2 as the bottleneck, then we receive a separation of the FDIR routine from the rest of the system.

We will analyze this separation in greater detail, but the satellite case study is too large to describe it in detail at this point. Therefore, let us consider the full state space of the sensorfilter example. At first we concentrate at the nominal model without fault injection in figure 4.2. Every state contains a vector

$$\begin{pmatrix} s_{out} \\ f_{in} \\ f_{out} \end{pmatrix}$$

where s_{out} is the current value of the sensors out data port. Additionally f_{in} and f_{out} are the in and out data port of the filters component. If there are no fault injections these three variables are sufficient to separate every state in the state space. The transition t_S is the transition of the sensor1 and transition t_F the transition of the filter1. We see in figure 4.2 that a separation of the state space is easily done by choosing t_S or t_F.

Now let us assume we augment the model by the fault injections of the example. Then suddenly no separation of the state space is possible anymore. The reason for this lies in the transition structure of the monitor component. As we see in appendix A.1.1, the monitor has the basic structure that the mode will change from OK to $FailF$ after a filter error and continuously to $FailSF$ after a sensor error. Additionally it will change from OK to $FailS$ with a sensor error and then to $FailSF$ with a filter error. This is again a structure which is not apportionable into two parts, because this diamond consists of four different transitions.

Motivated by these problems, a small adaption of the bottleneck calculation will solve all these difficulties. The simple idea is that a bottleneck is not a single transition, but rather a set of transitions. This solves the problem that cyclic or highly connected models are not apportionable. Especially complex FDIR components can now be separated. The adaption in the definition of the partitioned transition system is as following:

Definition 13 (Set Partitioned Transition System)
Given a transition system $TS = (S, T, \rightarrow, I)$ which represents the state space for the partition and a set $B = \{t_{B_1}, \ldots, t_{B_n}\}$ with $t_{B_i} \in T$ (and a set $B^H = \{t^h_{B_1}, \ldots, t^h_{B_1}\}$ with t_{B_i} is handshaked with $t^h_{B_i} \in T$) then the set partitioned transition system $SPTS = \{TS_1, \ldots, TS_n\}$ with $TS_i = (S_i, T_i, \rightarrow_i, I_i)$ and

- State complete: $\bigcup_{i=1\ldots n} S_i = S$
- State distinct: $S_i \cap S_j = \emptyset$

- Bottleneck partition: $\forall_{(s,t_1,t_2,s')\in\rightarrow}((s \in S_i \wedge s' \in S_j \wedge i \neq j) \rightarrow (\exists k\ .t_1 = t_{B_k} \wedge t_2 = t^H_{B_k}))$
- Transition relation complete: $\bigcup_{i=1,\ldots,n} \rightarrow_i \ = \ \rightarrow \backslash \{(s,t_1,t_2,s') \in\rightarrow \mid s \in S_i \wedge s' \in S_j \wedge i \neq j\}$
- Initial states: $I_i = \{s' \in S_i \mid \exists t_B \in B.\exists t^H_B \in B^H.(s,t_B,t^H_B,s') \in\rightarrow \wedge s \in S_j \wedge i \neq j\} \cup (S_i \cap I)$

■

Now the developer can take a set of transitions to partition the state space. With this extension highly connected and complex systems can be separated if the developer knows the transitions which will be used for creating the bottleneck, like the transitions of the FDIR_SENSOR_A. We saw that the separation is possible. Also we see that this separation is useful for controlling the impact isolation. The concrete application of the bottleneck improved isolation will be presented in section 5.4. One problem of the multi transition bottleneck is that an automatic search for bottlenecks with more than one unspecified transition will test many different combinations. This leads to a high number of different partitions which are at most useless for the analysis of the relations between transitions. Therefore, finding such a bottleneck still needs to be done by the intuition of the developer.

Summary of the Bottleneck Partition

In this section we saw a possibility to segment the state space to make an analysis more meaningful. So we found one of many answers to the question **Q6** "How can we make the impact isolation more accurate?". Another important problem is still that the state space of many concurrency models explodes and is too time and memory consuming to generate a full state space. Therefore, the next section deals with the question of how to handle the state space explosion problem.

4.2 State Space Minimization

If we take a look on the satellite case study in the benchmark section 5.2, we see that there are many concurrency devices which manage a relatively high amount of data. The SLIM model of the complete satellite case study has approximatively 100,000,000 states and is even with high performance symbolic model checkers

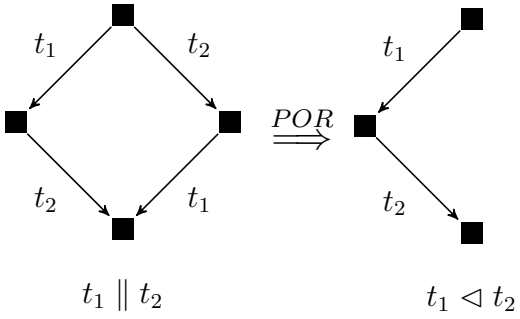


Figure 4.3: Partial-order reduction changes relations.

like NuSMV [8] only with high memory resources feasible. The state space explosion is a well known problem and there exist a lot of techniques to reduce the state space. In this section we examine three reduction techniques and the consequences on the impact isolation of this thesis.

4.2.1 Partial-Order Reduction

The first reduction technique is the partial-order reduction [9] for reducing the state space. The construction of the ample-set to cut outgoing transitions without violating the stutter trace equivalents of the state space is only one of many different techniques for partial-order reduction, but all techniques have the aim to reduce the number of diamonds which are created through the interleaving of independent processes. This is described in detail in [3]. The partial-order reduction is one of the most common techniques for reducing the state space and is applied in many model checkers, for example in SPIN [17].

But unfortunately, partial-order reduction is not an adequate reduction if we analyze the impact relations. The reason is that the impact relation uses exactly this diamond structure to classify the relations between the transitions. Figure 4.3 shows a simple example how partial-order reduction changes the impact relation. In the left transition system t_1 and t_2 are interleaving transitions and therefore the relation $t_1 \parallel t_2$ holds. After a partial-order reduction which exactly reduces this interleaving to a single path, t_1 and t_2 are in the relation $t_1 \lhd t_2$. Obviously, we are not able to apply partial-order reduction. Therefore, let us continue with the bisimulation to reduce the state space.

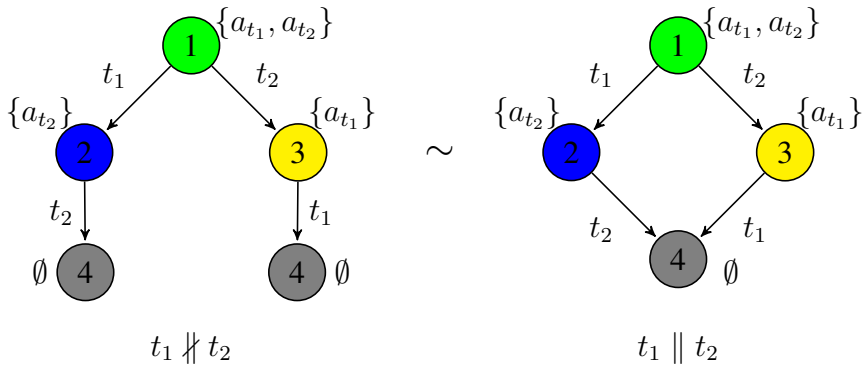


Figure 4.4: Bisimulation with outgoing transitions as atomic propositions.

4.2.2 Bisimulation

The bisimulation is also a common technique for state space reduction of concurrency systems. The basic idea of bisimulation is to collapse all states with the same atomic proposition and refine this collection again if they cannot simulate each other. Some algorithms are for example described in [3, 14]. The important part for the bisimulation in this thesis is the choice of the atomic proposition. The impact relation totally depends on the transitions of the system. Therefore, it seems appropriate to use the outgoing transitions of the state as the atomic proposition.

Unfortunately, this choice of the atomic proposition does not preserve the impact relation. Let us consider figure 4.4. In the left transition system the atomic propositions are exactly the outgoing transitions of the state and the transitions t_1 and t_2 are in relation $t_1 \nparallel t_2$. The numbers in the states are presenting the equivalent classes for this property. After the bisimulation reduction, we see that in the reduced transition system these two transitions are in relation $t_1 \parallel t_2$. Therefore, the bisimulation with such atomic propositions is not impact relation closed. If we want to preserve every relation between transitions we have to choose other atomic propositions.

Another strategy is to choose the incoming transitions instead of outgoing ones, but even if we take incoming and outgoing transitions as atomic propositions, the impact relation will not be preserved in every transitions system. An example for this is shown in figure 4.5. The set of atomic propositions of every

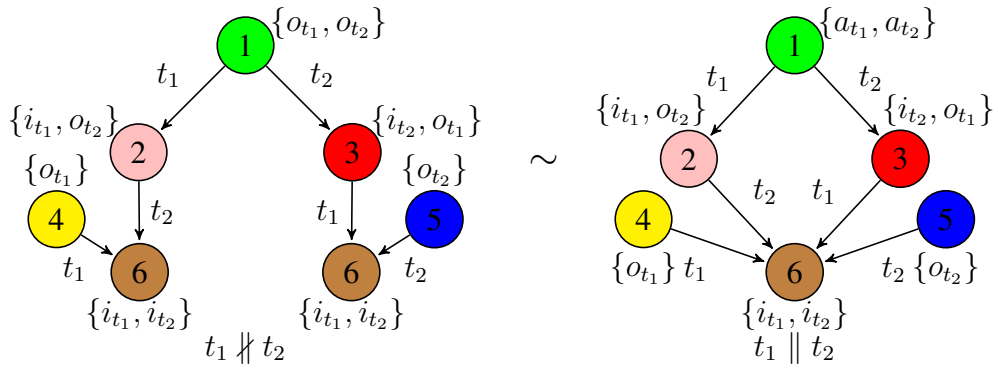


Figure 4.5: Outgoing and incoming transition as atomic propositions.

state contains o_{t_i} if the state has outgoing transition t_i and contains i_{t_i} if the state has the incoming transition t_i. Again we see that after a bisimulation reduction the relation of t_1 and t_2 is changed from $t_1 \nparallel t_2$ to $t_1 \parallel t_2$.

Is the change of the relation $\parallel$ and $\nparallel$ so important that we disclaim of the reduction? Yes, it is. $\parallel$ is the relation which expresses that the transitions do not influence each other and $\nparallel$ means that the order of the transitions is important for maybe safety critical variables. Therefore, with this atomic proposition found in this thesis a bisimulation is not applicable for a state space reduction. Maybe there exists a relation preserving labeling function, but so far, bisimulation is no solution for the state space explosion.

4.2.3 Slicing

The last technique for state space minimization is the static slicing for SLIM [22]. The basic idea is that for a given CTL* formula without next and a given SLIM model a reduction on the source code is applied. This means that parts of the source code are removed, because they have no effect on the CTL* formula. The algorithm uses three sets: D the data elements, E the events and M the modes which are initially filled with the data elements and modes of the formula. Then a fixpoint iteration adds step by step every data element, event or mode which possibly play a role in the calculation of every element in the three sets. This incrementally collecting of data is the same as a static impact analysis. In our

situation we do not have a formula to initialize the three sets. How can we solve this problem?

The idea is to choose a set of transitions to initialize the sets D,E and M. For this, we need to choose a start component and a target component. The aim is to reduce as much as possible from the model without changing the impact of the start component to the target component.

Definition 14 (Initialization of the Slicing Algorithm)
Given T_S the set of all transitions of the start component and T_T the set of all transitions in the target component. Also let $set(g) = \{d \in Dat \mid \text{guard } g \text{ contains } d\}$ and $set(f) = \{d \in Dat \mid \text{ assignment } f \text{ contains } d\}$. Then the initialization of the slicing sets is defined as:

- $D := \{d \in set(g) \cup set(f) \mid m \xrightarrow{e,g,f} m' \in T_S \cup T_T\}$
- $E := \{e \in events \mid m \xrightarrow{e,g,f} m' \in T_S \cup T_T\}$
- $M := \{m \in modes \mid m \xrightarrow{e,g,f} m' \in T_S \cup T_T\}$

■

If we now start the fixpoint iteration and slice the model we get a smaller model. At this point it is not clear whether this model preserves the impact relations for every transition. Before we will check this property we will concentrate on the properties of slicing.

Lemma 2 (Slicing Preserves All Valuations)
For all transitions $t_S = m_S \xrightarrow{e_S,g_S,f_S} m'_S$ and $t_T = m_T \xrightarrow{e_T,g_T,f_T} m'_T$, which are used in the initial set, slicing preserves all valuations of m_S, g_S, m_T and g_T.

Proof. Every possible value combination of the variables of t_S and t_T in the original model is also possible in the sliced model. Assuming this is not the case, then it is easy to create a CTL* formula which exactly covers the missing value combination by assigning m_S, g_S, m_T and g_T with the values. Then this formula is true in the original model and false in the sliced model. This is a contradiction, because slicing preserves CTL*. □

Now we will analyze whether slicing preserves the impact relation.

Lemma 3 (Slicing Preserves Impact Relation)
Given T_S,$T_T \subseteq T$ and the initialization is chosen like in definition 14 then the impact relation for $\nparallel$,$\#$ and $\lhd$ will be preserved in the sliced model.

Proof. Given $t_S \in T_S$ and $t_T \in T_T$ with $t_S = m_S \xrightarrow{e_S,g_S,f_S} m'_S$ and $t_T = m_T \xrightarrow{e_T,g_T,f_T} m'_T$. Let us assume the transitions are in the original model in the following relations:

Case 1: $t_S \lhd t_T$ in the original model.

We know that t_S enables the transition t_T in the original model. Therefore at some point in the sliced model t_S will be active and t_T will not be active. The mode, the event and the guard of t_S and t_T are in the set M,E or D. This guarantees that eventually in the sliced model t_S is active and t_T is not active and eventually t_T becomes active. Else the CTL* without next formula

$$\exists\Diamond(\underbrace{(\neg m_T \vee \neg g_T)}_{t_T \text{ is not active}} \wedge [(\underbrace{(m_S \wedge g_S)}_{t_S \text{ is active}} \wedge \underbrace{(\neg m_T \vee \neg g_T)}_{t_T \text{ is not active}}) \,\mathrm{U}\, (\underbrace{m_T \wedge g_T}_{t_T \text{ is active}})])$$

which only uses the mode and data elements of the initial sets will be true in the original model, but not in the sliced model. It remains to show that for the activation of t_T the transition t_S was taken. This is not expressible by CTL*, but we know that t_S influences in the non-reduced model the mode or data elements of t_S and therefore also in the sliced model. Together with the property that slicing preserves all valuations of both transitions, it follows directly that:
$\Rightarrow t_S \lhd t_T$ is contained at least once in the sliced model.

Case 2: $t_S \# t_T$ in the original model.

The proof of this case is analogous to the proof of **Case 1**: The difference lies in the used CTL* formula which shows that t_S and t_T are active in one state and eventually t_T becomes inactive. The CTL* formula

$$\exists\Diamond(\underbrace{(m_T \wedge g_T)}_{t_T \text{ is active}} \wedge [(\underbrace{(m_S \wedge g_S)}_{t_S \text{ is active}} \wedge \underbrace{(m_T \wedge g_T)}_{t_T \text{ is active}}) \,\mathrm{U}\, (\underbrace{\neg m_T \vee \neg g_T}_{t_T \text{ is not active}})])$$

shows that the conditions for t_S disables t_T hold in at least one state. Then the same argumentation like in **Case 1** holds that t_S influences directly t_T and all combinations of the variables in these transitions of the original model also appear in the sliced model.
$\Rightarrow t_S \# t_T$ is contained at least once in the sliced model.

Case 3: $t_S \nparallel t_T$ in the original model.

This proof is slightly more complex. We have the case that the state $t_1(t_2(s)) \neq$

$t_2(t_1(s))$. At first we need to show that there is a state s in the sliced model with t_1 and t_2 are active in s. This is quite simple, because if this state would not exist, the CTL* formula

$$\exists\Diamond(\underbrace{m_S \wedge g_S}_{t_S \text{ is active}} \wedge \underbrace{m_T \wedge g_T}_{t_T \text{ is active}})$$

will lead to a different result in the sliced model. Now there are three possible cases how the the order of t_S and t_T lead to two different states.

Case 3.1: t_S affects the assignment f_T of t_T.
All data elements of f_T are in the initial D. Therefore, after the fixpoint iteration every f_T affecting data element is in D; particularly the connection between t_S and t_F with all intermediate connections and flows are in the sliced model. Together with the argument that all possible variable combinations of the data elements in f_T are in the sliced model, the sliced model also contains $t_S \nparallel t_T$.

Case 3.2: t_T affects the assignment f_S of t_S.
Analogous to **Case 3.1**, just switching the t_S with t_T and use t_S instead f_T as analyzed assignment.

Case 3.3: The order of t_S and t_T affects another data element which is not in the guard of t_S or t_T.
This case is not really a case of the proof, because this is not possible in SLIM. If $t_S \nparallel t_T$ is in the original model, then in this case there is no transition between t_S and t_T. Therefore the effect of the first applied transition needs to be saved by a flow or connection until the second transition is applied and can distinguish which transition was first used. However, this is not realizable in SLIM, because connection and flows do not contain cycles or have the possibility to save a data value for one step. This is only possible with transitions, but if another transition is applied between t_S and t_T in the original model, then we are leaving the case that we analyze $t_S \nparallel t_T$.

$\Rightarrow t_S \nparallel t_T$ is contained at least once in the sliced model.

$\Rightarrow$ The impact relations $\nparallel$, $\lhd$ and $\#$ are closed under slicing. $\square$

But what is with the $\|$ relation? This relation is not necessarily closed under slicing, because both transitions have no effect on each other. This does not play an important role, because $\|$ is not changed into another relation. It is just not guaranteed to be part of the system after the slicing. And so we do not lose information about the impact or create new impacts. The only change in the impact matrix are that the counted appearances of the relations will be changed, because of the new structure of the state space.

Problems of Slicing and State Space Reduction in General

The problem of the state space minimization through slicing is that we need to specify which components are used to invoke an impact. This is not such a limitation for the start component, because this component is mostly given, like a newly implemented FDIR component. The necessary choice of a target component for the analysis is the obstructive constraint, because to find out which component is affected by the start component is the target of the impact isolation. Therefore, for every possible target component the slicing reduction has to be applied. So there is only a separation of the state space into a set of smaller state spaces which also must be analyzed.

Why is it so hard to find an applicable state space reduction? The answer lies in the structure of the impact isolation. If we start the isolation, it is unknown which parts of the system are affected. That is what we want to find out. We do not have a property which is used for reduction like in model checking. So what else can we choose as criteria? This is an interesting topic for future work, because we will see in the next chapter how important the size of the state space will be for the impact isolation. In summary, we have not found an answer to **Q7** "How can we make the impact isolation faster?" in this thesis. Nevertheless, we analyzed three reduction techniques and found a lot of insights to increase our understanding of a proper reduction for impact isolation. It is difficult to reduce parts of the system while preserving the impacts, because this is itself nothing else than an impact isolation.

Chapter 5

Experimental Evaluation

This section will deal with the experimental evaluation of the concepts of the previous chapters. At first we deal with the problem of generating the state space of a SLIM model. Then we will analyze the performance of the tool which generates the impact isolation, followed by a short look at the impact tool itself. Last but not least, we will apply a realistic impact isolation on the satellite case study.

5.1 The Wrapper

One of the essential parts of this thesis is to calculate the full state space of a SLIM model, because the concepts of the previous chapters are based on the state space. Normally the calculation of the full state space is the most naive approach for starting a model checking algorithm. Therefore, in the COMPASS project, the highly optimized symbolic model checker NuSMV [8] is used, which avoids to calculate the full state space. At this point we had the choice between two strategies. Firstly, to force NuSMV to calculate the full state space and find a possibility to use this state space. Alternatively, the second possibility was to translate the SLIM model to a language of a full state model checker.

We chose the translation approach for this thesis, because it seemed to be more effective to use a full state space model checker which was designed for the state space generation. At this point, we used the model checker SPIN [17]. In figure 5.1 is a sketch of the process for the calculation of the state space. The state space generation is fragmented into the generation of three documents: The **model**, the **state space generation files** and the **state space**. The design step of every of these documents is as following:

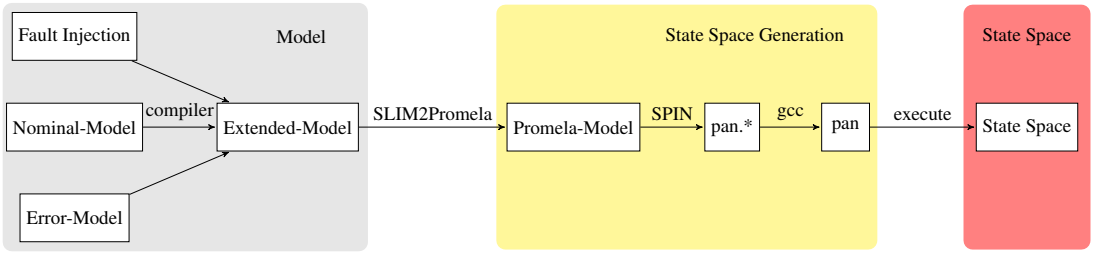


Figure 5.1: Visualization of the process for the state space calculation.

Document 1: The Model

As we saw in the examples, every non-trivial SLIM model is build up by a nominal SLIM model, the error models and the fault injection. These three files are compiled to the extended SLIM model. This step is already implemented in the COMPASS project. For the usage in SPIN, the SLIM model needs to be translated into Promela. This step is implemented in [23], where also the correctness of this step is shown.

Document 2: The State Space Generation Files

The equivalent Promela model will be translated with SPIN into a few pan files, which contain the model as C source code. These files need to be compiled with the gcc compiler [16]. The result is the executable version of the Promela model.

Document 3: The State Space

The last step only contains the execution of the created pan files. At this step the full model will be scanned by a depth first search and it is possible to output every visited state during the search and to calculate the transitions corresponding to the state change.

How is it possible to force SPIN to output the states and the transitions of the SLIM model? Before we will answer this question, we will demonstrate that this approach for a wrapper raises some problems. At first, with the current implementation of the SLIM to Promela translation it is not possible to model the SLIM state space in a one-to-one translated Promela state space. This means that for the atomic step in SLIM which changes one state to another state by using a (maybe handshake) transition, the Promela state space needs more than one step. In detail, the diffusion of the data port values through the connections and flows needs to be calculated through a fixpoint iteration between all involved components.

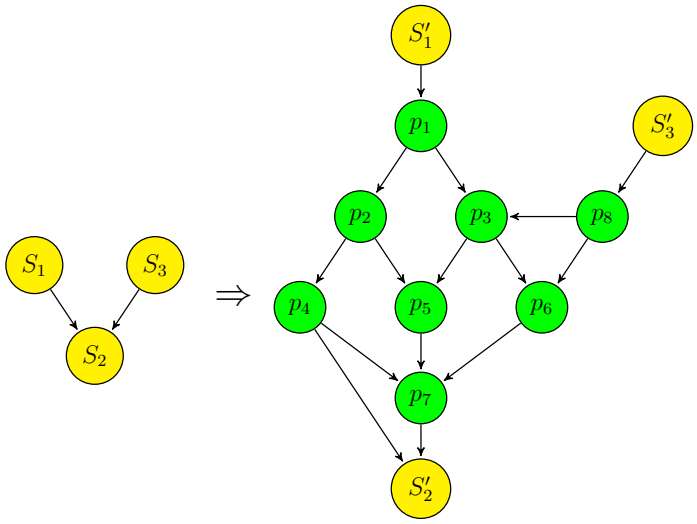


Figure 5.2: Cutout of a SLIM model versus an equivalent Promela model.

The solution is to identify the SLIM states in the Promela state space and use the embedded c-code of Promela. This c-code will be ignored by the translation from Promela to the executable pan files. When the pan files will be executed, then also the embedded code will be executed. With an embedded **printf** command we will output the information directly to the stdout. Through the backtracking during the depth first search in the execution and the possible handshake between transitions, it is not trivial to find the correct position in the Promela model for the embedded print code.

The basic idea is to print the SLIM state (e.g. the data elements and the current values) and the next transition (e.g. the label and the component of the transition) directly before this transition is taken. Then we calculate the fixpoint iteration and print the reached state. So we receive in every step the tuple (start state, transitions, target state). This information is sufficient to calculate the full state space. Figure 5.2 visualizes this step by a cutout of a SLIM state space on the left side. The corresponding Promela state space is on the right side. The yellow states which are labeled with S' are the states which output the SLIM state if they will be visited during the search and the green states which are labeled with p are the Promela states which are created by the fixpoint iteration. If we consider the right model more detailed, it will be clear that during the depth first search a problem occurs. Assume the search through the state space follows the path

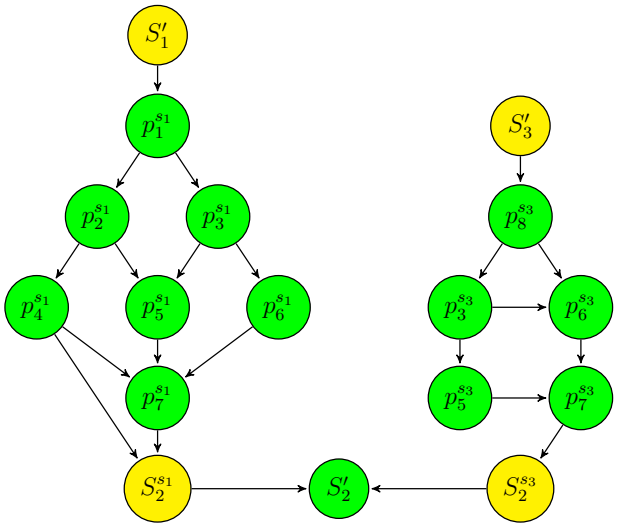


Figure 5.3: Avoiding incomplete depth first search.

$S'_1 \rightarrow p_1 \rightarrow p_3 \rightarrow p_6 \rightarrow p_7 \rightarrow S'_2$. In state S'_1, the current SLIM state and the taken transition will be printed. After the execution in S'_2, the target state will be printed. Let us assume the search comes at some point to $S'_3 \rightarrow p_8$, then in S'_3 the state and the transition will be printed, but the depth first search will stop in p_8, because p_3 and p_6 are already visited. Therefore S'_2 will never be reached again and the tuple will never be complete.

The best solution for this problem is to force SPIN to continue the search until S'_2, but this would need major changes in the SPIN translation. An easier, but also more memory consuming, way is to give every Promela state of the fixpoint iteration a flag which identifies it with the SLIM state which starts the iteration. In figure 5.3 is shown how this flag separates the two calculations and makes it possible to print every tuple correctly.

This was only one of several small problems which occur in the wrapper. The rest of the solutions are details of the implementation and will not be explained at this point. With the correct embedded c-code in the fitting position in the SLIM to Promela translation it is possible to pipe the full state space of the SLIM model to an XML file and use it as the basis for the impact isolation algorithms. The performance of the state space calculation and the rest of the impact algorithm will be presented in the next section.

5.2 Benchmarks

This chapter will present the benchmarks which are applied on the examples and the case study of this thesis. The benchmarks of the satellite case study are more detailed than the benchmarks of the two examples. These examples are academic and created to motivate the problems and help to understand the algorithms. In contrast, the satellite is a realistic model and big enough to be meaningful for a benchmark. The used environment for the benchmarks is a $48 \times 2{,}1$ GHz 192 GB RAM blade. Unfortunately the algorithms are not optimized for parallel computing.

A summary of all results is presented in the table 5.4. In this table the **S2P** row shows the time which the translation from SPIN to Promela needs. The **SPIN+GCC** row presents the summed time of the translation from SPIN and the compile time of GCC. Together with the **Execution** row, which contains the time of creating the XML state space file, these four rows offer the state space generation. The last four rows are the Java [24] algorithms which implement the concepts of the thesis. **Load XML** is the time which is needed to load the state space from the XML-file to the Java classes. **Bottlenecks** is the algorithm for separating the state space with a given set of transitions into subspaces. And the **Counting** row shows how long it needs to calculate the counting matrix with all transitions of the model. Last but not least, the **Compare** row displays the time for calculating comparing paths for two given states. The transitive closure is not in the benchmark, because the calculation of the transitive closure is independent of the size of the state space and is only dependent on the number of the transitions of the SLIM model. The calculation takes only a few seconds even in the largest model.

Previous analyses of the satellite case study with the high performance model checker NuSMV result that the satellite model has more than 100.000.000 states. Furthermore, we see that we are not able to generate the full state space of this model. Satellite S1 shows the results if we slice the model with the property of a given component. The problem is that the model is so highly connected such that even the sliced model is too big for the state space generation. This does not mean, that the slicing reduction is useless. The number of data elements of the sliced model is strongly reduced (Booleans from 537 to 186, Integers from 248 to 108 and Modes from 120 to 80). The variation of the slicing property leads in the most cases to this core model. There are also properties which lead to trivial models like satellite S2. The manually reduced (by removing redundant devices or limit data values) models satellite 10k, 20k, 30k show that the wrapper is effec-

Example	States	state space generation			Java algorithms			
		S2P	SPIN+GCC	Execute	Load XML	Bottleneck	Counting	Compare
Sensorfilter	501	0.586s	5.276s	7.104s	1.251s	0.106s	0.592s	0.043s
Lifesupport	425	0.431s	2.978s	4.936s	0.933s	0.174s	0.558s	0.078s
Satellite	∼100,000,000	3m53s	SegFault	N/A	N/A	N/A	N/A	N/A
Satellite S1	?	4m47s	SegFault	N/A	N/A	N/A	N/A	N/A
Satellite S2	2	5.490s	1.567s	4.362s	0.025s	0.05s	0.002s	0.017s
Satellite10k	10,792	3.661s	16m41s	29m34s	2m12s	3.546s	5.919s	0.479s
Satellite20k	21,584	3.526s	17m11s	69m28s	4m55s	12.53s	14.555s	0.556s
Satellite30k	31,044	3.567s	25m04s	181m45s	8m01s	25.715s	30.252s	0.631s
Satellite120k	∼120,000	4.261s	25m41s	OOM	N/A	N/A	N/A	N/A

Figure 5.4: Results of the benchmarks.

tive enough to generate state spaces with thousands of states. The satellite 120k model exhausts the 192 GB memory with generating 33.232 states. However, for making a feasibility test of the algorithms in this thesis, the generation rate is sufficient. We see that even in the 30k model, the Java algorithms all need less then one minute, even without high performance optimization and using a slow programming language like Java.

Weaknesses of the Wrapper

We saw in the benchmarks that 192GB RAM suffice to generate realistic models with round about 33.000 states. This leads to the fact that the limiting factor in the isolation is the wrapper for generating the state space. The explanation for this is quite easy. We saw in section 5.1 that we must make a fixpoint iteration after every taken transition to calculate the resulting SLIM state. In this fixpoint iteration, hundreds of Promela states will lie between two SLIM states. This blow up is increased by the flagging technique which is needed for the correctness of the state space generation. Even if we will find a high performance wrapper which will generate millions of states, loading this state space into another program is still problematical. For example, the 30.000 states XML-file of the satellite 30k model needs 847MB RAM. Making the concepts of this thesis practically applicable means to find a solution for these two problems. This will be an important part for the future work which will deal with the optimization of the state space generation of SLIM models.

5.3 Presentation of the Impact Tool

At this point we will present a short overview of the impact tool, which implements the concepts of chapter 3 and 4. The program is written in Java and is split in different views, each view implements one of the presented concepts or displays other useful information for the analysis of the state space.

Loading View:

This is the initial screen of the tool. The user can chose between two possibilities to start a project. The first one is to create a new project by choosing a project name and adding a set of SLIM models, error models and a fault injection file. Additionally, the user can activate the slicing option and assign slicing parameters. After this the tool generates a new project by using the python [26] wrapper which is described in section 5.1. The XML state space will now be loaded by the tool. The second possibility to start a project is to load an existing state space. This will omit the wrapper and directly load the XML file from the given project name.

Refinement View:

Input: State space S, set of bottleneck transitions.
Output: Subspaces $S_1, \ldots, S_m$, drawing of the subspaces.
This screen implements the state space partition of section 4.1. The user can separate the full state space into smaller parts by a given set of bottleneck transitions. This newly created partition is available in every other view which receives the state space as input. The partition can be reversed again and can be redone by other transitions.

Statistical View:

Input: State space, start and target component for impact.
Output: Counting matrix as table, closure over counting matrix as table, impact graph as drawing or table.
The statistical view implements the concepts of the counting matrix of section 3.2. The user can choose the start component and the target component, in which every transition of the component and every subcomponent will be added to the counting matrix. So if the user wants to compare all transitions he just needs to choose the root component in the start and target component. Also, the limitation to a subspace for the calculation can be selected. Additionally the closure can be calculated and the impact graph for a given transition can be drawn or presented as a table.

Compare View:

Input: State space, first and second transition, first and second relation.
Output: Comparing paths as table.
The compare view implements the concepts of the comparing paths of section 3.3. The user can chose the used space, the compared transitions, and the used relations. This calculates the set of states in the different relations. In an intermediate step, the user can choose one state of each set, which is the basis of the comparison paths algorithm. The result will be presented in two tables where the direct compare between the paths is possible.

Bottleneck View:

Input: Start and target state space, used bottlenecks.
Output: Graph of the path.
The now following views are used for collecting additional information and are not parts of the concepts in the thesis. In the bottleneck view a subspace and an outgoing bottleneck can be chosen. This bottleneck leads to another subspace, where also a bottleneck will be chosen. With this information a shortest path through the second subspace is calculated and displayed.

State Space View:

Input: State space, variables for collapsing states.
Output: Drawing of the given state space.
In this view, the full state space or a given subspace is drawn. The user can select data elements which can be used to collapse a set of states. Also it is possible to select a transition which will be highlighted in the drawn state space.

Transition View:

Output: All transition as table.
The transition view contains the translation of the short name of every transition to the SLIM label and the component where the transition is implemented.

State View:

Input: Id or variable values.
Output: Found state with variable values and enabled transitions.
In the last view the user can assign limits on the data elements or the id of the state to select a searched state. For these states a data filter selects

the variables which are displayed. Also the active transitions and the target states of the selected state are displayed.

All these views are designed to make the results of the impact isolation of a component more understandable. The results which are collected by applying the tool on the satellite case study will be presented in the next section.

5.4 Satellite Case Study

In previous parts of the thesis we saw the application of impact isolation on academic examples. In this section we will deal with the question **Q8:** "How to apply the impact isolation on realistic models?". Therefore at this point we will apply the impact concepts on a reduced model of the satellite case study. This model is more involved than the other examples, making the results directly practical. The model is with 16008 states more complex and describing it involves much detail. Therefore we will describe only a part of the model for later analysis.

Description

We will analyze the defect of an earth sensor and the subsequent reaction on the system. A cutout of the model is presented in figure 5.5. The main component in this example is the Attitude and Orbit Control System (AOCS) of the satellite. It manages all components that control the satellite. Three subcomponents of the AOCS are of particular importance: The two infrared earth sensors EARTH_SENSOR_A, EARTH_SENSOR_B and the Control and Data Unit (CDU). The CDU contains a Processor Module (PM) which includes the SoftWare On-Board Data Handling component (SW_OBDH). The SW_OBDH controls all software operations of the system. The AOCS_FDIR manages the FDIR components of the AOCS and in particular the IRES_A_FDIR, which handles fault detection and isolation in the corresponding earth sensor. If the software needs to change the active hardware topology, then this goes through SystemControl .

In this model it is possible that the earth sensors lose the lock on to the earth. This fault is captured in the earth sensors's error model and in the corresponding fault injection. This is modeled analog to the faults of the previous examples. Therefore we omit the description of the error model and fault injection. If the lock is lost in EARTH_SENSOR_A then this data is propagated over port connections to the IRES_A_FDIR component. This is this visualized in figure 5.5 by the

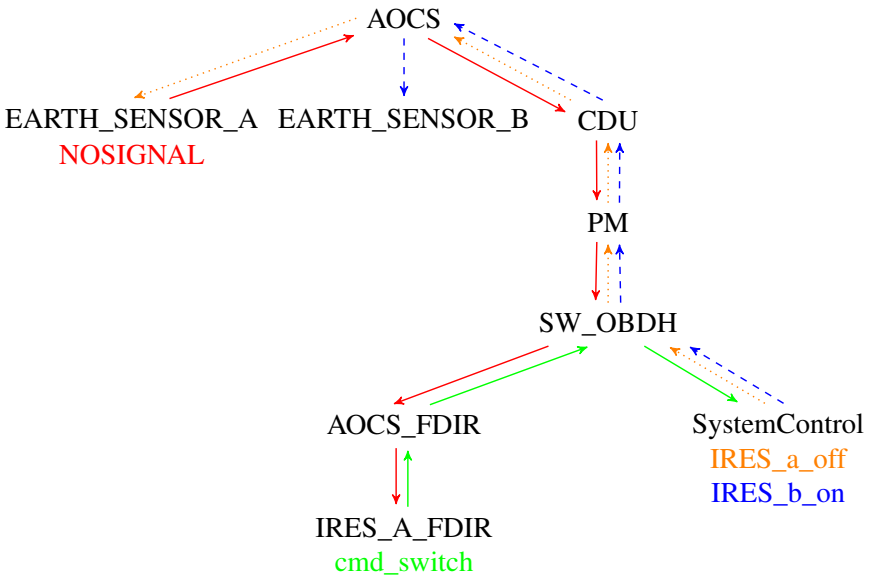


Figure 5.5: Data and event propagation in the satellite case study.

red flow (EARTH_SENSOR_A, AOCS, CDU, PM, SW_OBDH, AOCS_FDIR, IRES_A_FDIR). The reaction of the IRES_A_FDIR is more complex compared to the previous examples, as several transitions are executed to react to the fault. These transitions are the following:

t_I: idle -[require_proc when status=NOSIGNAL]-> s1

t_0^N: s1 -[cmd_changemode_DM when AOCS_current_mode=NOM]-> s2;

t_1^N: s2 -[cmd_changemode_SM when active = REDUNDANT]-> release;

t_2^N: s2 -[cmd_switch when active=NOMINAL then active:=REDUNDANT]-> s3;

t_3^N: s3 -[cmd_changemode_NOM]-> release;

t_0^O: s1 -[cmd_changemode_DM when AOCS_current_mode=OCM]->o2;

t_1^O: o2 -[cmd_changemode_SM when active=REDUNDANT]-> release;

t_2^O: o2 -[cmd_switch when active=NOMINAL then active:=REDUNDANT]-> o3;

t_3^O: o3 -[cmd_changemode_OCM]-> release;

t_R: release -[release_proc]-> idle;

Idle is the initial state of the FDIR component. If the lock is lost then the in data port $status$ is changed to NOSIGNAL. The FDIR component wants to be the only executing software component. In that case this is implemented using the $require\_proc$ event. The reduced model contains three possible FDIR routines depending on the current mode of the system and the activeness of the earth sensors. The original model contains more routines for other mode configurations. For this case study we focus on the four routines: R^N_{NOM}, R^R_{NOM}, R^N_{OCM} and R^R_{NOM}.

$R^N_{NOM} := (t_I, t^N_0, t^N_2, t^N_3, t_R)$ is executed if the system is in the normal mode (NOM) and the earth sensor are not running in backup mode. The FDIR component sends the command for changing into degraded mode (DM). The reaction in this routine is to switch the activity of earth sensors, such that IRES_B_FDIR becomes active. Afterwards the system mode is switched back to NOM mode.

$R^R_{NOM} := (t_I, t^N_1, t_R)$ is executed if the system is in the normal mode (NOM) and the earth sensor is running in the backup mode. The FDIR component sends the command for changing in the degraded mode (DM). The only reaction here is the transformation to the safe mode (SM).

$R^N_{OCM} := (t_I, t^O_0, t^O_2, t^O_3, t_R)$ is executed if the system is in the orbital control mode (OCM) and the earth sensor is not running in the backup mode. The FDIR component sends the command for changing into degraded mode (DM). The reaction at this point is to switch the earth sensors, such that IRES_B_FDIR becomes active. Then the system mode is switched back to OCM mode.

$R^R_{NOM} := (t_I, t^O_1, t_R)$ is executed if the system is in the orbital control mode (OCM) and the earth sensor is running in the backup mode. The FDIR component sends the command for changing into degraded mode (DM). Here, the only reaction is to transmit the system into the safe mode (SM).

Independent of which routine takes place, after the execution the FDIR component releases the process control with $release\_proc$ and changes to the $idle$ mode again.

Let us focus our analysis on R^N_{NOM} and especially on the transition t^N_2 which switches from the primary to the backup sensor. We see in figure 5.5 that the

cmd_switch event will be send to SystemControl. This is visualized by the green flow (IRES_A_FDIR, AOCS_FDIR, SW_OBDH, SystemControl). The SystemControl will react with the following transitions:

t_0^C: idle -[fdir_switch_ires]-> s1;

t_1^C: s1 -[ires_a_cmd_off when active_ires]-> s2;

t_2^C: s1 -[ires_b_cmd_off when not active_ires]-> s2;

t_3^C: s2 -[ires_b_cmd_on when active_ires then active_ires:=not active_ires]-> idle;

t_4^C: s2 -[ires_a_cmd_on when not active_ires then active_ires:=not active_ires]-> idle;

The ControlSystem reacts on the received cmd_switch event which is renamed to $fdir\_switch\_ires$. The active earth sensor will be deactivated and the previously deactivated earth sensor will be activated. In the presented example, EARTH_SENSOR_A will be deactivated and EARTH_SENSOR_B will be activated. This is seen in figure 5.5 by the dotted orange flow (SystemControl, SW_OBDH, PM, CDU, AOCS, EARTH_SENSOR_A) for the deactivation. The propagation of the activation event is drawn as dashed blue flow (SystemControl, SW_OBDH, PM, CDU, AOCS, EARTH_SENSOR_B).

For completeness we also look at the AOCS component, which handshakes with t_0^N, t_1^N, t_3^N. They are used to synchronize the current mode of the system. In total there are 51 transitions to manage the current mode, therefore we will focus only on the three important transitions:

t_0^A: DM -[CDU.changemode_NOM then current_mode := (NOM);]-> NOM

t_1^A: OCM -[CDU.changemode_NOM then current_mode := (NOM);]-> NOM

t_2^A: PAM -[CDU.changemode_NOM then current_mode := (NOM);]-> NOM

Also in the earth sensors are two related transitions. The disabling transition of EARTH_SENSOR_A and the enabling transition of EARTH_SENSOR_B in the following named as:

t_0^{SA}: on -[cmd_off when commandable then status := OFF]-> off;

t_0^{SB}: off -[cmd_on when commandable and power then status := ON]-> on;

Expected Impacts

Before we start the impact isolation let us define the expected impact such that we are able to check after the isolation whether each impact is detected. Let us concentrate on the analysis to the t_N^2 transition which invokes the switch of the sensor components. The direct impacts of t_2^N are:

I_0**:** The most obvious impact is that t_2^N enables t_3^N as the successor transition in R_{NOM}^N.

I_1**:** Then t_3^N should be synchronized with a transition in AOCS . Because during R_{NOM}^N the transition t_0^N changed the global mode to DM we expect that the handshake transition will be t_0^A.

I_2**:** The activation of t_1^C in the SystemControl, which takes place through the handshake of t_2^N with t_0^C.

I_3**:** Because t_0^C is handshaked with transition t_0^{SA} in EARTH_SENSOR_A and t_0^C is impacted in I_2 we assume that also t_0^{SA} is impacted.

I_4**:** We should also consider the indirect impact which we expect as the enabling of the EARTH_SENSOR_B with the transition t_0^{SB}.

Analysis of the Satellite Case Study

Now we will start analyzing this with the techniques of this thesis. At first we compute the direct impact relations. The resulting counting matrix is too big to fully present it here. Therefore only the interesting effects will be described. The following list shows the direct enable impacts (the independent impact are ignored):

t_3^N: s3 -[cmd_changemode_NOM]->release

t_1^C: s1 -[ires_a_cmd_off when not(active_ires)]-> s2

t_0^{SA}: on -[cmd_off when commandable then status := OFF]-> off;

t_0^A: DM -[CDU.changemode_NOM then current_mode := (NOM);]-> NOM

t_1^A: OCM -[CDU.changemode_NOM then current_mode := (NOM);]-> NOM

t_2^A: PAM -[CDU.changemode_NOM then current_mode := (NOM);]-> NOM

Let us first concentrate on the direct impacts. The impact $t_2^N \lhd t_3^N$ is trivial, because t_3^N is the successor of t_2^N which changes the system back to the NOM mode. This is the impact which we forecasted with I_0. Also that $t_2^N \lhd t_0^A$ is in the counting matrix is expected in I_1, because t_0^A is the handshake partner of t_3^N and therefore also responsible for changing the mode to NOM. The next impact in the counting matrix is $t_2^N \lhd t_1^C$.

This impact is more challenging and needs a more detailed explanation. t_2^N is handshaked with t_0^C the first transition of the SystemControl. t_0^C activates the routine for switching the activities of the earth sensors. t_1^C is the successor of t_0^C and responsible for deactivating EARTH_SENSOR_A. Hence t_2^N activates t_1^C which was expected in impact I_2. Directly on this impact follows that $t_2^N \lhd t_0^{SA}$. t_0^{SA} is the transition of EARTH_SENSOR_A and deactivates this sensor. t_0^{SA} is handshaked with the $ires\_a\_cmd\_off$ event to t_1^C. Hence the activation of t_1^C through t_2^N leads directly to the activation of t_0^{SA}. This was forecasted by I_3. With the indirect impact I_4 we will deal later.

By analyzing the other impact relations we see that $t_2^N \lhd t_1^A$ and $t_2^N \lhd t_2^A$ are also in the counting matrix. Let us call this unexpected impact I_U. $t_2^N \lhd t_1^A$ and $t_2^N \lhd t_2^A$ means that during R_{NOM}^N, transitions are enabled that change the system mode from OCM to NOM or from PAM to NOM. This was unexpected. Through the execution of the FDIR components the only permissible transition should be t_0^A from DM to NOM. Based on the impact I_U we search for other components which change the AOCS modes via a handshake. The only other component is the Tracking, Telemetry and Command component (TT&C) which is outside of the AOCS component and therefore the TT&C was initially out of scope.

The TT&C is responsible for the current state of the mission (Launch, Transfer, Service, Disposal). This component influences the mode of the AOCS system. Though in the description of the TT&C component is a comment which stated that to the FDIR components priority is given. We saw through our isolation that this property is not satisfied, because the TT&C component disables the DM which should be the active mode through the whole FDIR execution. The reason is that the $require\_proc$ and $release\_proc$ event does not control the behavior of the TT&C component. They only synchronize the executions of the different FDIR components of the AOCS system. Now thus is clear, it is up to the designer to clarify whether the comment in the TT&C or the behavior of the TT&C is wrong.

Until now we analyzed only direct impacts. What remains to be studied is transitive impact I_4. Through the sequential execution of the transitions of the

Start Transition	Target Transition	Relation
root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t2	root_AOCS_t3	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t2	root_AOCS_t16	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t2	root_AOCS_t32	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t2	root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t3	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t2	root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t0	Conflict
root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t2	root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t2	root_AOCS_earthXsensorXaX0_t4	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t2	root_AOCS_earthXsensorXaX0_t7	Enable
root_AOCS_t3	root_AOCS_t8	Conflict
root_AOCS_t3	root_AOCS_t10	Enable
root_AOCS_t3	root_AOCS_t13	Conflict
root_AOCS_t3	root_AOCS_t14	Enable
root_AOCS_t3	root_AOCS_t22	Enable
root_AOCS_t3	root_AOCS_t25	Conflict
root_AOCS_t3	root_AOCS_t37	Enable
root_AOCS_t3	root_AOCS_t38	Conflict
root_AOCS_t3	root_AOCS_t41	Enable
root_AOCS_t3	root_AOCS_t51	Conflict
root_AOCS_t3	root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_t1	Enable
root_AOCS_t3	root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_t3	Enable
root_AOCS_t3	root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t4	Enable
root_AOCS_t3	root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXBXfdirX0_t3	Conflict
root_AOCS_t3	root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXBXfdirX0_t4	Enable
root_AOCS_t3	root_AOCS_earthXsensorXaX0_t0	Conflict
root_AOCS_t3	root_AOCS_earthXsensorXaX0_t3	Conflict

Figure 5.6: Data and event propagation in the satellite case study.

SystemControl component the direct impact does not show information whether the switch of the sensors works correctly. Therefore we continue with the transitive closure of the counting matrix. The result of the transitive closure is that the activation of the EARTH_SENSOR_B and hence the correct execution of the switch is found in the transitive closure. So far the evidence seems to be convincing, but there is also a problem with the closure. Nearly every transition is transitively influenced through the transition t_2^N in less than 4 steps. The reason for this is the impact of t_2^N on the AOCS transitions. Most transitions of the system depend on the mode of the AOCS . Therefore a change of one of its transitions causes a huge set of subsequent impacts. This leads to the insight that the FDIR component is far away from having a small impact on a small set components. A cutout of this impact is presented in figure 5.6. Here the transition root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t2 is the tool representation of t_2^N and root_AOCS_t3 / t16 / t32 equates t_0^A / t_1^A / t_2^A. We conclude from the figure on the impact explosion starting from root_AOCS_t3. However this is also an interesting observation, which shows that the transitive closure is useful to detect such impact explosions.

One possibility to reduce such impact explosion is the bottleneck partition of the state space. For example an useful bottleneck is $|t_0^N, t_3^N|$. This equates the example of the bottlenecks in section 4.1. The figure 5.7 shows the impact of t_2^N in the resulting subspace. We see that the impact explosion from root_AOCS_3 does not appear.

Start Transition	Target Transition	Relation
root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t2	root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t0	Conflict
root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t2	root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t2	root_AOCS_earthXsensorXaX0_t4	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_AOCSXFDIRX0_IRESXAXfdirX0_t2	root_AOCS_earthXsensorXaX0_t7	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_AOCS_t14	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_AOCS_t15	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_AOCS_t31	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t2	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_AOCS_earthXsensorXaX0_t0	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_AOCS_earthXsensorXaX0_t2	Conflict
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_AOCS_earthXsensorXaX0_t3	Conflict
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_AOCS_earthXsensorXaX0_t5	Conflict
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_AOCS_earthXsensorXaX0_XerrorSubcomponentX0_t0	Conflict
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_AOCS_earthXsensorXbX0_t0	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_TTC_t3	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_TTC_t5	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_TTC_t6	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_TTC_t7	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_TTC_t8	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_TTC_t11	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_TTC_t12	Enable
root_AOCS_CDU_PMXAX0_SWXOBDHX0_SystemControl_t1	root_TTC_t13	Enable
root_AOCS_t14	root_AOCS_t10	Enable
root_AOCS_t14	root_AOCS_t22	Conflict
root_AOCS_t14	root_AOCS_t37	Conflict

Figure 5.7: Data and event propagation in the satellite case study.

Conclusion

During the isolation we saw that the expected direct impacts $I_0, \ldots, I_3$ are in the counting matrix. Also the indirect impact I_4 is found in the transitive closure of the counting matrix. We saw that an unexpected impact I_U is found with the impact relation. Hence there exists mistakes that are found fast and without searching them sequentially. Also the isolation of impacted code is done by the impact isolation, we found enough evidence to isolate the TT&C as the disturbing component. The disadvantage of the impact relation is that the usability of the results strongly depend of structure of the model. In a highly connected model, the impact explosion in the transitive closure makes the analysis more involved.

From the results we conclude that the reduction of the analysis to subspaces is useful. For example if we want to analyze the impact of the FDIR component only in the PAM or NOM mode. This leads to a finer view on the model and reduce the explosion in the transitive closure. Comparing paths is not applicable here, because in this example t_2^N has only enabling effects on other transitions. Nevertheless especially in larger models the impact relation is useful to give a hint for possible mistakes.

In this chapter we saw how we apply the concepts of this thesis to analyze a realistic model. We see that question **Q8:** "How to apply the impact isolation on realistic models?" is answered during this section. Nevertheless there are many different possibilities to improve and expand the concepts of this thesis. The next

chapter deals with the possible future work which is based on the fundamentals which were created during this thesis.

Chapter 6

Future Work

In this chapter we will discuss how we can improve the concepts which were developed during the thesis and categorize the future work on the impact isolation. Generally this chapter is split into two parts. The first part deals with the conceptual improvements of chapter 3 and discusses how we augment the impact relation to increase the usefulness for the developer. The second part will discuss the problems of chapter 4 and 5 for increasing the performance of the impact isolation.

6.1 Conceptual Improvements

Data Elements for Rarefy the Impacts

One point which was not considered during this thesis is the impact of components on data elements. Which component contains a connection or flow depending influence on the data ports of other components? How does the change of the data port of one component propagate through the system? The transition based impact isolation does not deliver an answer on this question and does not consider these effects. Nevertheless it is another interesting approach for analyzing the impact. A possible approach is to simulate the system into a set of states with a certain property and give a set of observed data ports. Then starting from these states, a breadth first search over the available transitions is invoked. In every step the effect of the observed data ports on other data ports needs to be detected and so the observed set grows step by step until a fixpoint is reached. The set of observed data ports is divided by the number of steps which are needed to affect this data port.

The idea behind identifying impact over the effect on data elements is an interesting area of research itself. Nevertheless a combination between this technique and the transition based technique delivers a lot of information for the impact isolation and should be analyzed in future works on this topic.

Probabilistic Evaluation of the Impacts

Another important information which was not analyzed during the thesis is the probabilistic part of the thesis. If we analyze a relation between two transitions, but one (or both) of these transitions are triggered by a fault injection, then the probability that this relation is affecting the system is normally quite low. By using the probabilities, given from the error model, the impact isolation in this thesis will be more meaningful.

Relative Values in the Counting Matrix

We know that a probabilistic measurement of the impact relation would be useful. Hence in the counting matrix there is already a relative value of the appearance of the relation in the state space. Also, the transitive closure over the counting matrix seems to propagate these values intuitively. This can we use for a probabilistic analysis. But how meaningful is the relative value of the counting matrix? Assuming we have one relation between two transitions on a cycle or in a bottleneck. Is this relation more important than other appearances? Also, these values can be used to bound the impact isolation, similar to time-bounded model checking.

Time Dependent Impact Isolation

In SLIM it is also possible to model time dependent models. How can we make the impact isolation more meaningful for time analysis? If one transition triggers another transition, how much time passes between these transitions? Is there a finite time interval? This leads to the more general question: Which components are influenced by another component during a given time interval? This is also an interesting and complex topic to augment the impact isolation.

Over-Approximation of Transitive Closure

The last point of the conceptual future work is to reduce the over-approximation in the transitive closure of the counting matrices. We saw in figure 3.10 that the transitive closure can lead to an over-approximation. One way to reduce this

over-approximation is to create better rules for the closure by searching for more information. Another way is to find a witness for the impacts which cannot be found for over-approximated impacts. A short example: $t_1 \lhd t_2$ and $t_2 \lhd t_3$ holds and we found a path $\xrightarrow{t_1}\xrightarrow{t_2}\xrightarrow{t_3}$ then we get a witness that the transitive relation $t_1 \lhd t_3$ holds in at least one position in the state space. The disadvantage is that to search such paths has a high complexity for longer concatenations. To find a solution to reduce this over-approximation would strongly improve the usability of the impact relation.

6.2 Performance Improvements

We saw during the benchmarks that without a reduction of the state space or a more clever wrapper it is not possible to apply the concepts of this thesis to practically relevant models. Therefore this section will deal with proposals for future work on this area.

Properties for Reduction Techniques

We saw in section 4.2 over the reduction techniques that it is not trivial to find a property which can be used for a state space minimization. However it is not proven that no effective reduction techniques for the impact isolation exist. With the benchmarks results and the knowledge about the state space explosion problem we know that calculating a full state space without reduction is not practically applicable. Therefore, at this point, future work is absolutely necessary.

Wrapper Optimization

We saw in the benchmarks that the current version of the wrapper is able to generate SLIM models with round about 30.000 states. This is not enough for realistic models. The translation from SLIM to Promela to use SPIN for the state space calculation is not effective enough. Also we saw that using a file to save the state space is also not possible for larger state spaces. The only solution which seems to be logical at this point is to rewrite the compiler of the compass project in a high performance language. For example c++ is a candidate for replacing the current python compiler. With this technical basis, it will be possible to create the necessary functions to calculate a full state space. This will be possible without translating SLIM into another language like SMV or Promela. Then the impact

isolation and a lot of other algorithms which use a state space can be founded on the basic compiler system. Additionally using the compiler via inheritance instead as a wrapper to another programming language would solve the state space blow up which is caused by Promela or the problems of a symbolic state space of the SMV model checker NuSMV. Also, piping the state space to an XML file becomes unnecessary.

Full State Space Generation

We saw in the impact isolation of the case study that the transitive closure over the counting matrix is an over-approximation. If we also do not require that the impact isolation is safe, this means that not every impact needs to be detected. Then we could reduce the state space generation only to the search for the relations of the counting matrix. Therefore we would only need to calculate finite paths through the state space with limited length. In every reached state the impact relation of the transitions of this state will be calculated and added to the counting matrix. The numbers in the counting matrix become less meaningful and may not be complete. On the other hand, we could calculate impact of really big models without calculation of the full state space. The COMPASS tool already implements the possibility to calculate random paths of a limited length through the state space, but this implementation in python is far away from having a good performance. The previous proposal of a high performance compiler which calculates the state space could be easily adapted to generate finite paths to search for the impact relations.

This approach contains similarities to bounded model checking. Adapting this strategy to SAT-based symbolic model checking with unsat-cores [20] is a possible future research area. The main problem would be to find a proper encoding for the SAT-problem.

We saw in this chapter that there are a lot of different approaches to augment or improve the presented impact isolation. This shows us that this thesis is just a small part of a complex topic. The results of this small part will be summarized in the next chapter.

Chapter 7

Summary

The motivation for this thesis is to create the fundamentals for a model-based criticality analysis. The idea is to calculate an impact isolation for FDIR components to make it time-saving for the developer to assign severities for FDIR components.

We saw during the thesis that the problem of the impact isolation cannot be solved by standard model checking approaches. The simple reason for this is that questions like "which component is influenced by a change in this component" or even questions like "Does component A influence the calculations of component B?" is untranslatable into a checkable property. This motivates the development of an impact isolation.

With the usage of FDIR components which react on faults, the change impact analysis becomes interesting for SLIM models. Unfortunately, in a concurrency model language like SLIM, a lot of the standard impact analysis strategies are not applicable. This lead to the question **Q1** "What is an impact in a SLIM model?". Motivated through the partial order semantics, we found an impact relation which compares single transitions. This relation gives us the possibility to search the state space for unexpected behavior and display the direct impacts from one component to another component. The advantage of this relation is that the direct impacts are safe and there is no over-approximation. Additionally the impact relation not only says that there exists an impact, it also categorizes this impact into a disabling, enabling or ordering effect.

The next questions **Q2** "How can we calculate the impact of a component?" and **Q3** "How can the impact be visualized?" were answered by the counting matrix. The counting matrix delivers an easy calculation and a clearly arranged overview over the impact relation.

However, we also saw that analyzing only the counting matrix does not contain

enough information to find the reason for the unexpected behavior. This lead us directly to **Q4** "How can deviating impacts be analyzed?". Hence the concept of the comparing paths was introduced. With this concept we had the possibility to find out which conditions are important to cause the unexpected behavior.

After this we saw that analyzing the impacts just by finding direct effects of transitions is not sufficient for a meaningful impact isolation which was formalized by the question **Q5** "How can indirect impacts be detected?". Therefore, the closure over the counting matrices makes it possible to calculate the impact of all transitions through the full system which can be impacted over several steps. The disadvantage of the closure is the over-approximation of the impact relation. This means that in the closure there are impacts which are not part of the system. Nevertheless, the transitive closure of the counting matrices gives an informative overview of the impact of a component.

Another important point was the question for the improvement of the impact isolation **Q6** "How can we make the impact isolation more accurate?". Founded on this question, we developed the separation of the state space through bottlenecks, which can refine the impact relation and make it more reproducible.

Another important part of the thesis was founded in the question **Q7** "How can we make the impact isolation faster?". The state space explosion of concurrency systems is a well known problem which makes it difficult to calculate the full state space of a non trivial model. Therefore, we analyzed the three reduction techniques partial order reduction, bisimulation and slicing. Through this analysis, we gained a lot of information for setting limits to **Q7**. The missing of a certain property in the impact isolation makes it hard to do a proper pre-minimization of the state space. In other words, a pre-minimization which preserves the impact relations is nothing else than a coarse impact isolation. Additionally, we found out that with the slicing approach we move the full state space search to several smaller searches in smaller subspaces. Summarizing, we do not found a perfect solution for **Q7**, but we increased the understanding of the impact isolation and the problems which appear if the model is reduced and how these problems affect the impact isolation.

The benchmarks of the thesis show that the impact isolation techniques of this thesis are feasible. Also the examples show that the impact isolation techniques are useful. Even in the early development where small subsystems of the models are created, the techniques give a hint for mistakes during the modeling process. The question **Q8** "How to apply the impact isolation on realistic models?" was the reason for the satellite case study which lead to the result that we are able to find useful information about the model with the impact isolation. This information

is used to improve the correctness and to increase the understanding of realistic models.

Conclusion

The conclusion of this thesis is that the impact isolation is a useful tool for analyzing concurrency systems. The knowledge whether one component affects another one is extremely important if components are running in parallel. Also the knowledge which components affect safety critical parts of the model is useful. The usage of the impact isolation is not limited on criticality analysis of aerospace systems. There are potential application areas in nearly every safety critical system where the system becomes too confusing for manual code reviews. Regrettably, this thesis only makes first steps in the research of this complex and interesting topic.

Bibliography

[1] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[2] R. S. Arnold and S. A. Bohner. Impact Analysis - Towards a Framework for Comparison. In *Proceedings of the Conference on Software Maintenance*, ICSM '93, pages 292–301, Washington, DC, USA, 1993. IEEE Computer Society.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[4] B. W. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[5] S. A. Bohner. Extending Software Change Impact Analysis into COTS Components. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, SEW '02, pages 175–, Washington, DC, USA, 2002. IEEE Computer Society.

[6] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP '09, pages 173–186, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] B. Breech, M. Tegtmeyer, and L. Pollock. A Comparison of Online and Dynamic Impact Analysis Algorithms. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.

[8] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[9] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State Space Reduction Using Partial Order Techniques. *STTT*, 2(3):279–287, 1999.

[10] COMPASS Project. Semantics of the COMPASS System-Level Integrated Modeling (SLIM) Language. Part of the COMPASS tool download version 20101220, 2010.

[11] COMPASS Project. Specification of the COMPASS System-Level Integrated Modeling (SLIM) Language. Part of the COMPASS tool download version 20101220, 2010.

[12] ECSS. European Cooperation on Space Standardization (ECSS). http://www.ecss.nl/, 2011.

[13] European Cooperation for Space Standardization (ECSS). Failure Modes, Effects and Criticality Analysis (FMECA). *Space Product Assurance*, ECSS–Q–30–02A, 2001.

[14] J.-C. Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, 13:13–219, 1989.

[15] R. W. Floyd. Algorithm 97: Shortest Path. *Commun. ACM*, 5:345–, June 1962.

[16] GCC. The GNU Compiler Collection. http://gcc.gnu.org/, 2011.

[17] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23:279–295, May 1997.

[18] International Conference on Software Engineering (ICSE). ICSE Website. http://www.ifi.uzh.ch/icse2012/, 2012.

[19] J. Law and G. Rothermel. Whole Program Path-Based Dynamic Impact Analysis. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society.

[20] J. Marques-Silva and J. Planes. Algorithms for Maximum Satisfiability Using Unsatisfiable Cores. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 408–413, New York, NY, USA, 2008. ACM.

[21] A. Mazurkiewicz. Trace Theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to other Models of Concurrency*, pages 279–324, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[22] V. Y. Nguyen, T. Noll, and M. Odenbrett. Slicing AADL Specifications for Model Checking. In C. Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 217–221, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[23] M. Odenbrett. Explicit-State Model Checking of an Architectural Design Language using Spin. http://www-i2.informatik.rwth-aachen.de/i2/odenbrett/, 2010.

[24] ORACLE. JAVA Website. http://www.java.com/, 2011.

[25] A. Orso, T. Apiwattanapong, M. J. Harrold, G. Rothermel, and J. B. Law. An Empirical Comparison of Dynamic Impact Analysis Algorithms. In *Proceedings of theInternational Conference on Software Engineering (ICSE 2004)*, pages 47–50, Edinburgh, Scotland, May 2004.

[26] Python. Python Programming Language - Official Website. http://www.python.org/., 2011.

[27] RWTH Aachen University - i2. The COMPASS Project Web Site. http://compass.informatik.rwth-aachen.de/, 2011.

[28] SPACE NEWS. Eutelsat W3B Declared Total Loss; Plans Under Way To Deorbit Craft. http://www.spacenews.com/launch/101029-eutelsat-w3b-declared-total-loss.html, Fri, 29 October, 2010.

[29] G. S. Tjaden and M. J. Flynn. Representation of Concurrency with Ordering Matrices. *IEEE Trans. Comput.*, 22:752–761, August 1973.

[30] S. S. Yau, J. S. Collofello, and T. MacGregor. *Ripple Effect Analysis of Software Maintenance*, pages 60–65. IEEE Press, 1978.

[31] J. Zhao, H. Yang, L. Xiang, and B. Xu. Change Impact Analysis to Support Architectural Evolution. *Journal of Software Maintenance*, 14:317–333, September 2002.

Appendix A

SLIM Models of the Scenarios

A.1 Sensorfilter Model

A.1.1 Sensorfilter SLIM Model

```
system Sensorfilter
  features
    value: out data port int default 4;
    alarmS: out data port bool default false observable;
    alarmF: out data port bool default false observable;
end Sensorfilter;
system implementation Sensorfilter.Impl
  subcomponents
    sensors: system Sensors accesses mybus;
    filters: system Filters accesses mybus;
    monitor: system Monitor accesses mybus;
    mybus: bus MyBus;
  connections
    data port sensors.output -> filters.input;
    data port filters.output -> value;
    data port sensors.output -> monitor.valueS;
    data port filters.output -> monitor.valueF;
    data port monitor.alarmS -> alarmS;
    data port monitor.alarmF -> alarmF;
    event port monitor.switchS -> sensors.switch;
    event port monitor.switchF -> filters.switch;
end Sensorfilter.Impl;
```

```
system Sensors
  features
    output: out data port int default 1;
    switch: in event port;
end Sensors;
system implementation Sensors.Impl
  subcomponents
    sensor1: device Sensor in modes (Primary);
    sensor2: device Sensor in modes (Backup);
  connections
    data port sensor1.output -> output in modes (Primary);
    data port sensor2.output -> output in modes (Backup);
  modes
    Primary: activation mode;
    Backup: mode;
  transitions
    Primary -[switch]-> Backup;
end Sensors.Impl;

device Sensor
  features
    output: out data port int default 1;
end Sensor;
device implementation Sensor.Impl
  modes
    Cycle: activation mode;
  transitions
    Cycle -[when output<5 then output:=output+1]-> Cycle;
end Sensor.Impl;

system Filters
  features
    input: in data port int default 1;
    output: out data port int default 2;
    switch: in event port;
end Filters;
system implementation Filters.Impl
  subcomponents
    filter1: device Filter in modes (Primary);
```

```
    filter2: device Filter in modes (Backup);
  connections
    data port input -> filter1.input in modes (Primary);
    data port input -> filter2.input in modes (Backup);
    data port filter1.output -> output in modes (Primary);
    data port filter2.output -> output in modes (Backup);
  modes
    Primary: activation mode;
    Backup: mode;
  transitions
    Primary -[switch]-> Backup;
end Filters.Impl;

device Filter
  features
    input: in data port int default 1;
    output: out data port int default 2;
end Filter;
device implementation Filter.Impl
  modes
    Loop: activation mode;
  transitions
    Loop -[when output>0 and output!=2*input then output:=2*input]-> Loop;
end Filter.Impl;

fdir system Monitor
  features
    valueS: in data port int default 0;
    valueF: in data port int default 0;
    switchS: out event port;
    switchF: out event port;
    alarmS : out data port bool default false;
    alarmF : out data port bool default false;

end Monitor;
fdir system implementation Monitor.Impl
  modes
    OK: activation mode;
    FailS: mode;
    FailF: mode;
```

```
    FailSF: mode;
  transitions
    OK -[switchF when valueF=0]-> FailF;
    OK -[switchS when valueS>5]-> FailS;
    FailF -[switchS when valueS>5 then alarmF:=valueF=0]-> FailSF;
    FailF -[when valueF = 0 then alarmF:=true]-> FailF;
    FailS -[switchF when valueF=0 then alarmS:=valueS>5]-> FailSF;
    FailS -[when valueS>5 then alarmS:=true]-> FailS;
    FailSF -[when valueF=0 then alarmF:=true; alarmS:=valueS>5]-> FailSF;
    FailSF -[when valueS>5 then alarmS:=true; alarmF:=valueF=0]-> FailSF;
end Monitor.Impl;

bus MyBus
end MyBus;
bus implementation MyBus.Impl
end MyBus.Impl;
```

A.1.2 Sensorfilter Error Model

```
error model SensorFailures
features
    OK: initial state;
    Drifted: error state;
    Dead: error state;
end SensorFailures;

error model implementation SensorFailures.Impl
events
    drift: error event occurrence poisson 0.083;
    die: error event occurrence poisson 0.00001;
    dieByDrift: error event occurrence poisson 0.00015;
transitions
    OK -[ die ]-> Dead;
    OK -[ drift ]-> Drifted;
    Drifted -[ dieByDrift ]-> Dead;
end SensorFailures.Impl;

error model FilterFailures
features
    OK: initial state;
```

```
    Degraded: error state;
    Dead: error state;
end FilterFailures;
error model implementation FilterFailures.Impl
events
    die: error event occurrence poisson 0.007;
    degrade: error event occurrence poisson 0.051;
transitions
    OK -[ die ]-> Dead;
    OK -[ degrade ]-> Degraded;
    Degraded -[ die ]-> Dead;
end FilterFailures.Impl;
```

A.1.3 Sensorfilter Fault Injection

```
<ns0:faultInjection>
  <ns0:association>
    <ns0:componentPath>sensors.sensor1</ns0:componentPath>
    <ns0:errorModel>
      <ns0:packageName>__default__</ns0:packageName>
      <ns0:implementationName>
        SensorFailures.Impl
      </ns0:implementationName>
    </ns0:errorModel>
    <ns0:faultEffectsSpecification>
      <ns0:faultEffects errorState="Dead">
        <ns0:effect>output := 15</ns0:effect>
      </ns0:faultEffects>
    </ns0:faultEffectsSpecification>
  </ns0:association>
  <ns0:association>
    <ns0:componentPath>sensors.sensor2</ns0:componentPath>
    <ns0:errorModel>
      <ns0:packageName>__default__</ns0:packageName>
      <ns0:implementationName>
        SensorFailures.Impl
      </ns0:implementationName>
    </ns0:errorModel>
    <ns0:faultEffectsSpecification>
      <ns0:faultEffects errorState="Dead">
```

```
        <ns0:effect>output := 15</ns0:effect>
      </ns0:faultEffects>
    </ns0:faultEffectsSpecification>
  </ns0:association>
  <ns0:association>
    <ns0:componentPath>filters.filter1</ns0:componentPath>
    <ns0:errorModel>
      <ns0:packageName>__default__</ns0:packageName>
      <ns0:implementationName>
        FilterFailures.Impl
      </ns0:implementationName>
    </ns0:errorModel>
    <ns0:faultEffectsSpecification>
      <ns0:faultEffects errorState="Dead">
        <ns0:effect>output := 0</ns0:effect>
      </ns0:faultEffects>
    </ns0:faultEffectsSpecification>
  </ns0:association>
  <ns0:association>
    <ns0:componentPath>filters.filter2</ns0:componentPath>
    <ns0:errorModel>
      <ns0:packageName>__default__</ns0:packageName>
      <ns0:implementationName>
        FilterFailures.Impl
      </ns0:implementationName>
    </ns0:errorModel>
    <ns0:faultEffectsSpecification>
      <ns0:faultEffects errorState="Dead">
        <ns0:effect>output := 0</ns0:effect>
      </ns0:faultEffects>
    </ns0:faultEffectsSpecification>
  </ns0:association>
</ns0:faultInjection>
```

A.2 Life Support System Model

A.2.1 Lifesystem SLIM Model

```
system Lifesystem
  features
```

```
    oxygen: out data port int default 5;
    temperature: out data port int default 5;
end Lifesystem;
system implementation Lifesystem.Impl
  subcomponents
    battery1: device Battery accesses myBus;
    battery2: device Battery accesses myBus;
    batteryB: device Battery accesses myBus;
    energyDistributor: device EnergyDistributor accesses myBus;
    oxygenProducer: device OxygenProducer accesses myBus;
    heater: device Heater accesses myBus;
    oxygenMonitor: system OxygenMonitor accesses myBus;
    heaterMonitor: system HeaterMonitor accesses myBus;
    myBus: bus MyBus;
  connections
    data port battery1.energy -> energyDistributor.inputOx;
    data port battery2.energy -> energyDistributor.inputHeat;
    data port batteryB.energy -> energyDistributor.inputBackup;
    data port energyDistributor.outputOx -> oxygenProducer.energy;
    data port energyDistributor.outputOx -> oxygenMonitor.oxyEnergy;
    data port energyDistributor.outputHeat -> heater.energy;
    data port energyDistributor.outputHeat -> heaterMonitor.heaterEnergy;
    event port oxygenMonitor.switchToOx -> energyDistributor.switchOx;
    event port heaterMonitor.switchToHeat -> energyDistributor.switchHeat;
    data port oxygenProducer.oxygen -> oxygen;
    data port heater.temperature -> temperature;
end Lifesystem.Impl;

device Battery
  features
    energy: out data port int default 6;
end Battery;
device implementation Battery.Impl
end Battery.Impl;

device EnergyDistributor
  features
    inputOx: in data port int default 6;
    inputBackup: in data port int default 6;
    inputHeat: in data port int default 6;
```

```
    outputOx: out data port int default 6;
    outputHeat: out data port int default 6;
    switchOx: in event port;
    switchHeat: in event port;
end EnergyDistributor;
device implementation EnergyDistributor.Impl
  flows
    outputOx := inputOx in modes(noBackup,heatSupport);
    outputOx := inputOx+inputBackup in modes(oxySupport);
    outputHeat := inputHeat in modes(noBackup,oxySupport);
    outputHeat := inputHeat+inputBackup in modes(heatSupport);
  modes
    noBackup: initial mode;
    oxySupport: mode;
    heatSupport: mode;
  transitions
    * -[switchOx]-> oxySupport;
    * -[switchHeat]-> heatSupport;
end EnergyDistributor.Impl;

device OxygenProducer
  features
    energy: in data port int default 6;
    oxygen: out data port int default 5;
end OxygenProducer;

device implementation OxygenProducer.Impl
  modes
    initOx: initial mode;
  transitions
    initOx -[when energy>3 and oxygen<5 then oxygen:=oxygen+1]-> initOx;
    initOx -[when oxygen>1 then oxygen:=oxygen-1]-> initOx;
end OxygenProducer.Impl;

device Heater
  features
    energy: in data port int default 6;
    temperature: out data port int default 5;
end Heater;
device implementation Heater.Impl
```

```
  modes
    initHeat: initial mode;
  transitions
    initHeat -[when energy>3 and temperature<5 then temperature:=temperature+1]-> initHeat;
    initHeat -[when temperature>1 then temperature:=temperature-1]-> initHeat;
end Heater.Impl;

fdir system OxygenMonitor
  features
    oxyEnergy: in data port int default 6;
    switchToOx: out event port;
end OxygenMonitor;
fdir system implementation OxygenMonitor.Impl
  modes
    initOxMo: initial mode;
    switchedOxMo: mode;
  transitions
    initOxMo -[switchToOx when oxyEnergy<=3]-> switchedOxMo;
end OxygenMonitor.Impl;

fdir system HeaterMonitor
  features
    heaterEnergy: in data port int default 6;
    switchToHeat: out event port;
end HeaterMonitor;
fdir system implementation HeaterMonitor.Impl
  modes
    initHeatMo: initial mode;
    switchedHeatMo: mode;
  transitions
    initHeatMo -[switchToHeat when heaterEnergy<=3]-> switchedHeatMo;
end HeaterMonitor.Impl;

bus MyBus
end MyBus;
bus implementation MyBus.Impl
end MyBus.Impl;
```

A.2.2 Lifesystem Error Model

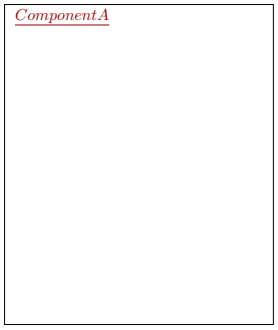
```
error model BatteryFailures
  features
    OK: initial state;
    Discharged: error state;
    Defect: error state;
end BatteryFailures;

error model implementation BatteryFailures.Impl
  events
    discharge: error event occurrence poisson 0.083;
    deffect: error event occurrence poisson 0.020;
  transitions
    OK -[discharge]-> Discharged;
    OK -[deffect]-> Defect;
    Discharged -[deffect]->Defect;
end BatteryFailures.Impl;
```

A.2.3 Lifesystem Fault Injection

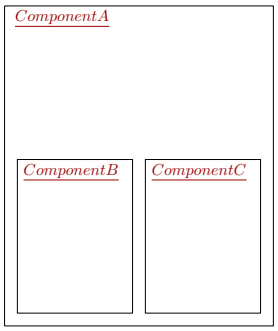
```
<?xml version='1.0' encoding='UTF-8'?>
<ns0:faultInjection xmlns:ns0="http://compass.informatik.rwth-aachen.de/fault">
  <ns0:association>
    <ns0:componentPath>battery1</ns0:componentPath>
    <ns0:errorModel>
      <ns0:packageName>__default__</ns0:packageName>
      <ns0:implementationName>BatteryFailures.Impl</ns0:implementationName>
    </ns0:errorModel>
    <ns0:faultEffectsSpecification>
      <ns0:faultEffects errorState="Discharged">
        <ns0:effect>energy := 4</ns0:effect>
      </ns0:faultEffects>
      <ns0:faultEffects errorState="Defect">
        <ns0:effect>energy := 2</ns0:effect>
      </ns0:faultEffects>
    </ns0:faultEffectsSpecification>
  </ns0:association>
  <ns0:association>
    <ns0:componentPath>battery2</ns0:componentPath>
    <ns0:errorModel>
      <ns0:packageName>__default__</ns0:packageName>
```

```
      <ns0:implementationName>BatteryFailures.Impl</ns0:implementationName>
    </ns0:errorModel>
    <ns0:faultEffectsSpecification>
      <ns0:faultEffects errorState="Discharged">
        <ns0:effect>energy := energy−2</ns0:effect>
      </ns0:faultEffects>
      <ns0:faultEffects errorState="Defect">
        <ns0:effect>energy := 2</ns0:effect>
      </ns0:faultEffects>
    </ns0:faultEffectsSpecification>
  </ns0:association>
</ns0:faultInjection>
```

A.3 Notation of the Visualization of SLIM Models



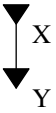
Vizualization of the implementation of component ComponentA



Vizualization of the subcomponents
The ComponentB and ComponentC are subcomponents of ComponentA

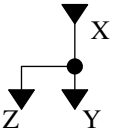
Visualization of out data/event ports
Out data/event port X of a component. Event or data is separated by the corresponding connection.

Visualization of in data/event ports
In data/event port X of a component. Event or data is separated by the corresponding connection.

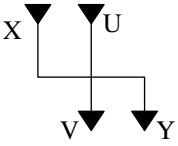


Visualization of data connections
Connection which leads data of data port X to data port Y

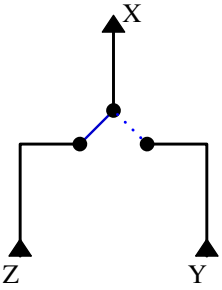
Visualization of event connections
Connection which leads data of data port X to data port Y



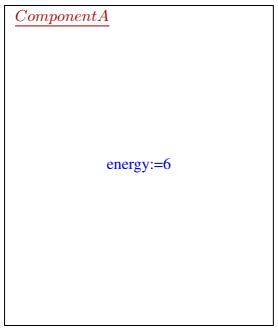
Augmented visualization of data connections
Connection which leads data of data port X to data port Y and data port Z



Crossing of data connections
Crossing of data connections does not influence the behavior of the single connections. Value of data port X leads to data port Y and value of data port U leads to data port V.



Visualization of a conditional connections
Connection which leads data of data port Z to data port X or the data of data port Y to data port X. This can happens through the “in mode” separation of connections or flows. Or the case operator in flows.



Visualization of a transition in a component
Informal description of the effects of transitions of ComponentA