Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Design and Implementation of a Service Recommendation System for Clams

Matthias Weilinger

Course of Study: Informatik

Examiner: Prof. Dr. Kurt Rothermel

Supervisor: Otto Bibartiu, M.Sc.

Commenced: November 20, 2020

Completed: May 20, 2021

Abstract

Cloud computing has been on the rise for years and will not stop following this trend in the future. Developers must have an environment where they can design cloud applications and receive feedback on the Quality of Service their design provides. This thesis aims to provide a framework that calculates the availability of a software design based on a given reliability model. First, this work describes the essential principles needed to understand the model and the calculations. This includes scenarios, Message Sequence Charts, and Labelled Transition Systems. Then it describes the implementation of a scenario-based reliability model in detail. Additionally, I propose an algorithm that maximizes the availability value by recommending suitable services based on the availability model. The performance and precision of the implementation are then evaluated. We will see that the precision is accurate, and the number and density of transactions between cloud services influence the runtime the most. Finally, I summarize the found results and look at the future developments of the topic.

Kurzfassung

Cloud Computing ist seit Jahren auf dem Vormarsch und das wird sich in Zukunft auch nicht ändern. Entwickler brauchen eine Umgebung, in der sie ihre Cloud Anwendungen entwickeln können und Feedback über den Quality of Service von ihren Designs erhalten. Das Ziel dieser Thesis ist es ein Framework zu entwickeln, das die Verfügbarkeit eines Softwaredesigns auf Basis eines gegeben Verfügbarkeitsmodell berechnet. Zunächst werden die notwendigen Prinzipien beschrieben die gebraucht werden, um das Model und die Berechnungen zu verstehen. Das umfasst Szenarios, Message Sequence Charts und Labelled Transition Systems. Danach wird die Implementierung eines Szenario basierten Verfügbarkeitsmodells erläutert. Zusätzlich stelle ich einen Algorithmus vor, der den Verfügbarkeitswert maximiert, indem er passende Services basierend auf dem Verfügbarkeitsmodell vorschlägt. Die Genauigkeit und Performanz der Implementierung wird danach evaluiert. Es wird sich zeigen, dass die Genauigkeit akkurat ist und die Anzahl und Dichte der Transaktionen zwischen Services den größten Einfluss auf die Laufzeit hat. Schließlich fasse ich die gewonnen Erkenntnisse zusammen und schaue auf zukünftige Entwicklungen.

Contents

1 Introduction **13**

2 Background **15**
2.1 Scenarios 15
2.2 Message Sequence Charts 16
2.3 Labeled Transition System 17
2.4 The Cheung user-oriented reliability model 18
2.5 Reliability Analysis using Scenarios 19
2.6 OpenClams 20

3 System requirements **23**

4 Implementation **25**
4.1 Labeled Transition System 25
4.2 Component Labelled Transitions System 27
4.3 Minimization 27
4.4 Composition 32
4.5 Calculating Reliability with Chueng 37
4.6 Recommendation of suitable services 37

5 Related Work **39**

6 Results **41**

7 Discussion **45**

8 Conclusion and Outlook **47**
8.1 Conclusion 47
8.2 Outlook 47

Bibliography **49**

List of Figures

2.1 The five scenarios used in the Boiler Control System represented as bMSCs . . . 16
2.2 The hMSC of the Boiler Control System 17
2.3 LTS for component Control and scenario Analysis 18
2.4 The annotated hMSC of the Boiler Control System 20

4.1 Probabilistic component LTS synthesized for component Control 28
4.2 Probabilistic component LTS synthesized for component Sensor 28
4.3 Probabilistic component LTS synthesized for component Database 29
4.4 Probabilistic component LTS synthesized for component Actuator 29
4.5 Minimized component LTS for component Control 31
4.6 Minimized component LTS for component Sensor 31
4.7 Minimized component LTS for component Database 31
4.8 Minimized component LTS for component Actuator 32
4.9 A edge list representation of the parallel composition of the Boiler Control System 36
4.10 The matrix derived from the synthesized Boiler Control System LTS 38

6.1 The time it takes to calculate the reliability of a system compared to the number of scenarios. The used system has six components with ten transitions 42
6.2 The time it takes to calculate the reliability of a system compared to the number of components. The used system has one one scenario and n-1 transitions 42
6.3 The time it takes to calculate the reliability of a system compared to the number of state transitions. The used system has five components and one scenario 43
6.4 The time it takes to replace generalized components of a system compared to the number of replaceable components 43

List of Algorithms

1 Composition breadth first search 34
2 GetNextCompositeStates 35

1 Introduction

Cloud solutions have to offer a well-defined Quality of Service (QoS). Those are specified in a part of the so called Service Level Agreements (SLA) where the cloud customer and the cloud provider set the framework conditions of their contract. In case of failure to deliver the specified QoS the solution owner has to expect significant loss of reputation and ultimately loss of revenue. In order to avoid such negative implications in the first place, an optimal service architecture should be created. Consequently, proper modeling tools are required to design and evaluate architectures. They allow the designer to find suitable cloud services that make it possible to fulfil the requirements as stated in the SLA. A useful way to model cloud systems' specifications is to use scenarios. A scenario depicts how different components interact to achieve a common goal. The Clams project provides a cloud modeling language for cloud applications. It uses message sequence charts (MSC), a commonly used method to design and display those aforementioned scenarios. Additionally, Clams uses transition diagrams to associate scenarios with each other to model and evaluate complex user behaviors. In order to express modeling uncertainty in service selection, the clams project provides the user with cloud computing patterns. These patterns are placeholders for possible services that fit the pattern descriptions. This allows to develop architectures independent of services offerings of particular cloud providers. Nevertheless, there is no way yet to replace patterns with actual services. The web app of Clams provides a simple plugin management system to add new evaluation modules. This thesis aims to implement a plugin that further evaluates the design. An essential QoS that must not be ignored is the availability of a cloud system. The average cost of downtime is 5600 dollars per minute, according to a 2014 study by Gartner [Ler14]. However, this is an average value. Depending on the business and environment, this cost can reach up to 500k per hour. Thus, the designer needs to know in advance which reliability their system can provide. Additionally, the provider always aims to maximize the availability of their product. In case a different component provides overall greater reliability, surely it should replace the inferior one. In this thesis, I implement a given availability model as an evaluation module for the web app and combine it with an iterative search algorithm to recommend services that increase the availability of the application.
In detail, this thesis includes the implementation of an availability model as proposed in the paper "Using Scenarios to Predict the Reliability of Concurrent Component-Based Software Systems"by Rodrigues et al. [RRU05]. With this model, the designer can directly calculate a precise reliability value for their system architecture. Furthermore, a search algorithm is implemented that maximizes the availability value by recommending suitable services based on the availability model. Not only does this method suggest components that increase the reliability, but it finds the global maximum. The result of these implementations is presented and discussed. Finally, I summarise the findings of this thesis and present an outlook and future work that might follow this work.

2 Background

In this chapter, definitions and technical concepts used throughout this thesis will be laid out and explained with examples, where applicable.

2.1 Scenarios

A scenario is a description of a system's actions. It describes how its components, the user, and the environment interact with each other. The result of these interactions is the functionality of the system [UKM04]. In this thesis, we use the Boiler Control System as an example as described by Uchitel et al. [UKM04]. It describes a system where the boiler temperate can be regulated according to the measured pressures inside the boiler. The system consists of four components, which are Control, Actuator, Database and Sensor.

- The **Sensor** is responsible for measuring the pressure.
- The **Actuator** activates the heating of the boiler.
- The **Database** can store the pressures measured by the Sensor component.
- Lastly, the **Control** unit is responsible for coordinating the other components. It tells the Actuator to act accordingly to the measured pressures from the Sensor.

In this system, five scenarios represent the commutation between the components. They are called Initialise, Register, Analyse, Terminate and End.

- In **Initialise** Control tells the Sensor to start with the measuring.
- In **Register** the Sensor reports the measured data to the Database.
- In **Analyse** the Control unit requests the latest data from the Database in the form of a query. The Database then sends the requested data back to the Control component. Then the Control unit tells the Actuator to control the temperature accordingly.
- The **Terminate** scenario represents the Control component telling the Sensor to stop tracking the pressure.
- Lastly, there is the **End** scenario where Control tells Sensor to shut down completely.

These scenarios are represented as basic Message Sequence Charts (bMSC) in Figure 2.1. We are going to take a closer look at bMSCs in the next chapter.

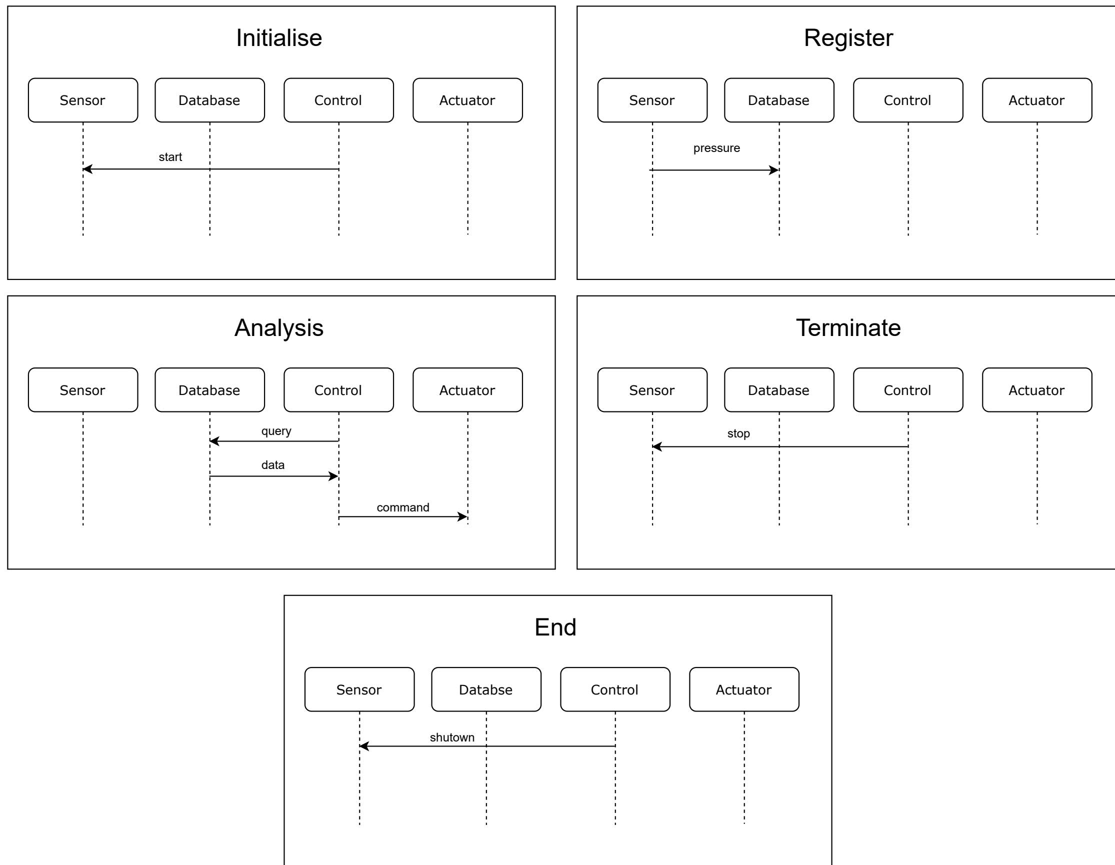

Figure 2.1: The five scenarios used in the Boiler Control System represented as bMSCs

2.2 Message Sequence Charts

A Message Sequence Chart (MSC) can be used to represent a scenario. It is a simple and intuitive graphical representation and therefore widely used [ITU11]. In Figure 2.1 MSCs are used to represent the interactions between the components. One can explicitly see these interactions in the form of arrows that point from one component to another. The MSCs displayed in Figure 2.1 are called a basic Message Sequence Charts (bMSC), which describe finite interactions between a set of components [UKM04]. For example, in the Register bMSC, the Sensor instance sends a message with the measured pressure value to the Database instance. Generally, bMSCs do not imply an order in which messages should be sent, but in our case, there are four scenarios where only one action is happening. Consequently, no other execution order is possible. For the Analysis scenario, we have three messages that could result in different execution orders[ITU11]. From a logical point of view, though, the Database component can only send data to Control if a query was already received. Furthermore, the Control instance can not give the Actuator any command as long as Control has not received the requested data. Hence there is only one possible execution order in this scenario as well. In the implementation part, the order of arrows in the bMSCs does matter and implies the execution order. An additional restriction is that self-referencing is not allowed. In other words, there can not be an arrow from a component to itself.
The second type of MSC is high-level Message Sequence Charts (hMSC), which provide a way to

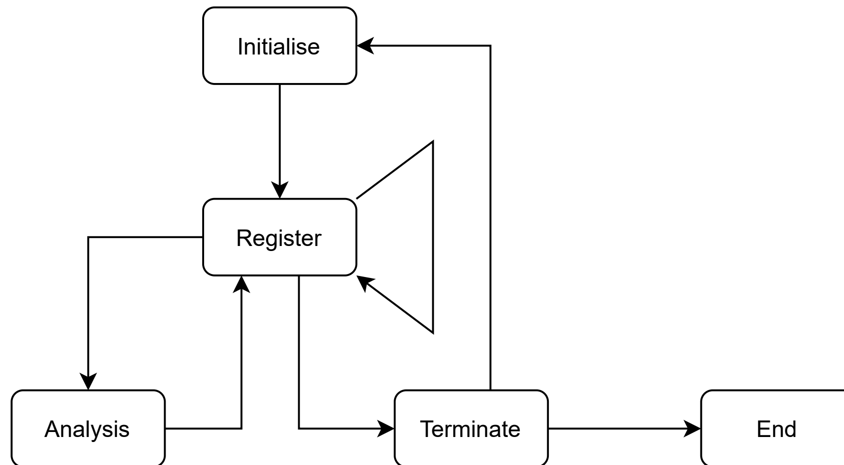

Figure 2.2: The hMSC of the Boiler Control System

compose bMSCs [UKM04]. These are used to show possible paths of execution of a system. It shows a possible continuation after the execution of each bMSC. An hMSC is a directed graph, with bMSCs as nodes and directed edges that connect the nodes. In Figure 2.2, we see the hMSC representation of our Boiler Control System. For instance, if we look at the Register node, we have three possible continuations after the scenario is executed. We can continue with the Analysis or Terminate scenario or perform Register again since these are adjacent nodes. When combining multiple bMSCs into one, the components that have identical names are the same. That means the Control component in Initialise is the same as in Register. In order to have different components that have the same type but are not identical, for example, two different MongoDB databases, the developer just names them differently (e.g., MongoDB1 and MongoDB2).

2.3 Labeled Transition System

A Labelled Transition System (LTS) is a finite state machine. It represents the message exchange between components of a distributed system [UKM04]. An LTS is a directed graph with nodes representing the state of a system and edges representing a transition from one state to another. Those edges are labeled with the message exchanged in the transition process, hence the name *Labelled* Transition System. Since it is a finite state machine, an LTS has an initial starting state and an end state with no outgoing transitions. Reaching the end state means the execution process was successful. An LTS can also have an error state which also has no outgoing transition. Reaching this state means the execution process failed [UKM04]. In Figure 2.3 we have an example LTS that describes the execution process of the Control Component in the Analysis scenario. How we generate an LTS from a bMSC will be discussed later. What we can see in Figure 2.3 is the starting state 0, from which query can be performed. After that, we reach state 1, where the data transaction happens. Now from state 2, we can perform the command transaction and reach the end of state 3. These labels are consistent with the messages sent or received from the Control instance in the Analysis scenario. The order of transactions is also consistent with the scenario because as long as we have to start with state 0, there is no other traversal order through the LTS.

Figure 2.3: LTS for component Control and scenario Analysis

2.4 The Cheung user-oriented reliability model

To calculate the reliability of a software system, Rodrigues needs a reliability model that expresses reliability as a function. This function should only include the properties that are given by scenarios. There are two properties that we can derive from scenarios. First, the reliability values of the different components in the system, and second the frequency of the utilization of these components. With this in mind, Rodrigues chose the user-oriented reliability model proposed by Cheung [Che80] since it uses exactly those two things. Cheung uses a directed graph to represent the structure of the system. A node N_i represents a program module and a directed edge(N_i, N_j) represents a possible transfer of control from module N_i to N_j. A probability P_{ij} is attached to every edge. R_i is the reliability of module N_i. Two additional terminal states (C and F) are added to the graph. C represents the correct execution, while F represents the failed execution. For every node N_i an additional edge (N_i, N_j) with probability $1 - R_i$ is added that stands for an error that could occur during the execution of moduleN_i. The original transition probability of every (N_i, N_j) is modified so that it is $R_i P_{ij}$. These transitions represent the correct execution of a module N_i and the transition of control to module N_i. For the last module N_n a transition (N_n, C) with probability R_n is added that represent correct termination of the system. Let $\{N_1, N_2, ..., N_n\}$ be the states of the model with N_1 as the starting state and N_n as the exit state. Let $P_{ij} = 0$ if there is no transition (N_i, N_j). Let M' be the transition matrix, where M'_{ij} represents the probability of transition from state i to state j:

$$M' = \begin{array}{c} \\ C \\ F \\ N_1 \\ N_2 \\ \vdots \\ N_n \end{array} \begin{array}{c} \begin{array}{cccccc} C & F & N_1 & N_2 & \dots & N_n \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 1 - R_1 & 0 & R_1 P_{12} & \dots & R_1 P_{1n} \\ 0 & 1 - R_2 & 0 & R_2 P_{22} & \dots & R_2 P_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ R_n & 1 - R_n & 0 & 0 & \ddots & 0 \end{pmatrix} \end{array}$$

Let M be the matrix obtained from M' when deleting the rows and columns that correspond to C and F:

$$M = \begin{array}{c} \\ N_1 \\ N_2 \\ \vdots \\ N_n \end{array} \begin{array}{c} \begin{array}{cccc} N_1 & N_2 & \dots & N_n \end{array} \\ \begin{pmatrix} 0 & R_1 P_{12} & \dots & R_1 P_{1n} \\ 0 & R_2 P_{22} & \dots & R_2 P_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ddots & 0 \end{pmatrix} \end{array}$$

Cheung shows that the system reliability can be calculated as $Rel = S(1,n) * R_n$, which is the probability of successfully transitioning from N_i to N_n in any execution order multiplied with the probability of transitioning from N_n to C. The calculation of $S(1,n)$ is done by

$$S(1,n) = (-1)^{n+1} \frac{|M|}{|I - M|}$$

with I standing for the identity matrix with the dimensions of M and $|M|$ and $|I - M|$ being the determinant of M and I-M, respectively.

2.5 Reliability Analysis using Scenarios

Rodrigues shows a way to start at scenarios and transform them to represent the transition matrix M that we need for the Cheung model. This transition can be divided into two steps. The first one is annotating the scenarios with probabilities. There are two different kinds of probabilities that we have to add to our scenarios. One is the probability of transition between scenarios PTS_{ij}. These probability values are added to the edges of the hMSC. The other kind of probability we call R_c and they represent the reliability of a component and are added to the bMSCs [RRU05]. In the implementation part, we do not have to worry about annotating the scenarios with any probabilities. As we will see in the next section, the clams framework already provides these values. For our working example, the Boiler Control System, we need those values. In Figure 2.4 one can see the values we use for every PTS_{ij}. The edge pointing to Initialize with the probability of 1 is not of interest for now. By the way, the representation of the hMSC one can see here was directly taken from the web interface provided by clams.
The reliability values R_c we choose as follows:

- $R_{Control} = 0.95$
- $R_{Sensor} = 0.99$
- $R_{Database} = 0.999$
- $R_{Actuator} = 0.99$

We do not go into detail about how we derive the values for PTS_{ij} and R_c because it is not part of this thesis. We refer the reader to Musa [Mus93] for further information on how to derive these values.
The second step of transforming scenarios into the transition matrix M is the synthesis of the probabilistic LTS. Rodrigues uses the synthesis approach proposed by Utchitel et al. [UKM04] and adds extensions. This step can be divided into four smaller steps. In this chapter, we look at them on a theoretical level. In the implementation part, we will take a very in-depth look at every one of them and explain how we modified them to work with the calms framework. The four steps are:

1. For each component C_i and each bMSC S_j an LTS C_iS_j is created that represents the components local behaviour in S_j. For every incoming and outgoing arrow that C_i has in S_j, a corresponding transition in C_iS_j is added.

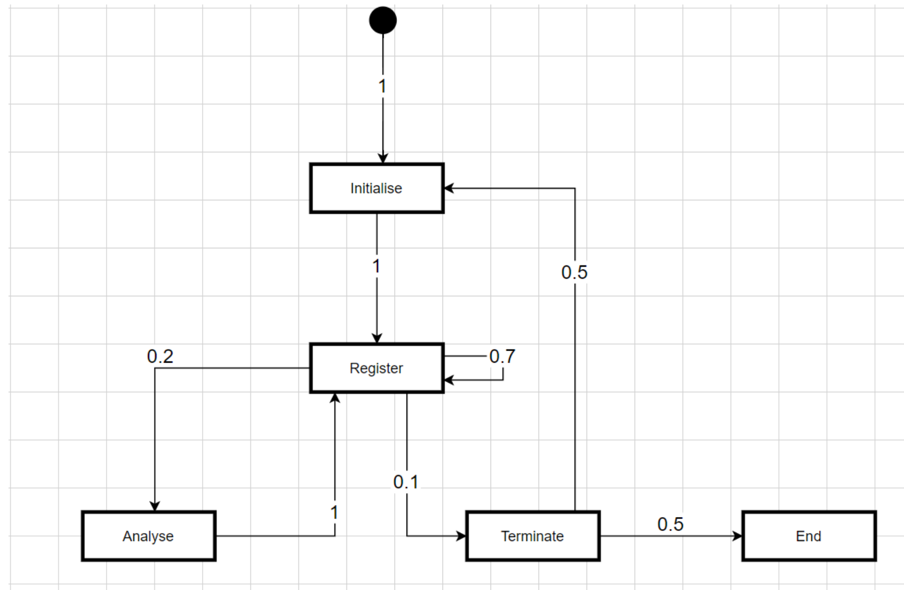

Figure 2.4: The annotated hMSC of the Boiler Control System

2. For every component C_i all its corresponding LTSs $C_i S_j$ are combined into a single component Labelled Transitions System. According to the hMSC the final state of $C_i S_j$ is connected with the starting state in $C_i S_k$ if there is a transition from scenario S_j to S_k
3. All generated cLTSs are reduced into a trace-equivalent, deterministic, minimal LTS.
4. All minimal LTSs are combined into a parallel compositing that represents the whole system's behavior.

The result of these four steps can be directly translated into the transition Matrix M on which we can perform the reliability Model by Cheung.

2.6 OpenClams

The clams project is a cloud application modeling solution that is scenario-based. By combining MSC with cloud computing patterns is describes scenarios at a system level. Additionally, it offers the functionality to set placeholders for components and can replace those with refined concrete services. The two main parts of the clams project that are important for this thesis are the webserver and the clamsml framework at which we are taking a closer look at now[1].

[1] https://github.com/openclams

2.6.1 Webserver

The webserver component provides a graphical user interface that allows the user to design their cloud system. The scenarios are called sequence diagrams and look very similar to the representation in Figure 2.1. The hMSC is called a user graph. Here we can take all sequence graphs that we already designed and combine them in the same way as we discussed in Section 2.2. Notice that in this user graph, the transition probabilities can be added just how it is necessary for the approach by Rodrigues as al. as seen in Section 2.5. We can choose if we want to use a specific component or take a more generalized one that should be replaced with a specific one by the program that we implement in this thesis. Notice how the labels the transitions in the scenarios are missing. In this application, it is not necessary to name the transitions. The reliability calculation must identify the exact transitions. This is why we have to come up with a method for that in our implementation. We will see how this is solved in chapter Section 4.1. By clicking the evaluation button on the top, a post request is sent to a server. The body of this request includes the JSON representation of the model that was serialized by the clamsml framework. Since we have four components that are the same across the scenarios in our example, we have to make sure that is also the case in the webserver representation.

2.6.2 ClamsMl

As you just saw, the clamsml framework provides a method to serialize the cloud system designed in the web interface into JSON format. It also provides methods for demoralizing the JSON file back to a clams model. Additionally, it contains the class structure for the object representation of clams models. We use this class structure in our implementation to work with the model and convert it into the data structure needed for the reliability calculation as seen in Section 2.5.

3 System requirements

There are four assumption that the system has to fulfill for the reliability calculation to work:

1. The reliability values of components are independent of each other. This means that the usage of a component does not affect the reliability of another one. Especially if a component replaces another one, the reliability of no other component in the system gets changed.
2. Failures are independent of each other. That means an error occurring in the system can be corrected by another error. If the system runs into a fail state, it can no longer be recovered.
3. The transfer of control between components is a Markov process, meaning the transition from one state to another is independent of the history and only dependent on the source state.
4. The system has only and exactly one initial and one final scenario in the hMSC. In case multiple initial and final scenarios are mandatory, super-initial and super-final scenarios can be introduced according to the super-initial state and super-final state as proposed by Wang et al. [WWC].

4 Implementation

In the following chapter, we are going to talk about the implementation part of the reliability model. Rodrigues proposed two steps to get from the bMSC and hMSC specification of a system to the probabilistic LTS that we need to apply the Cheung Model to the system. The first step is annotating the scenarios with component probabilities R_c and probabilities for transitions between scenarios PTS_{ij}. When designing a cloud system in the web interface of Clams, both of these annotations are already done. For this reason, we can skip the step in the implementation part and go directly to the next step that describes how to transform annotated scenarios into a probabilistic LTS. This step was further divided into four smaller steps, as we have seen in Section 2.5. We will take a close look at every one of them, see how an actual implementation looks, show which modifications are needed to make the approach work with the clams framework, and supplement the Boiler Control System process as an example.

4.1 Labeled Transition System

Rodrigues proposed four steps to get from an annotated scenario specification to the probabilistic LTS. Now we are going to look at the first one. The first step is to go over all components C_i and all scenarios S_j and create an LTS C_iS_j for every combination of the two. For example, if the system has five scenarios and seven different components, one ends up with five times seven, which equals 35, different LTSs. As we have seen in Section 2.3 an LTS is a directed graph with nodes and directed edges that connect the nodes. An LTS is created by projecting the behavior of a component in a particular scenario. That means that for every arrow in S_j that either points from C_i to another component C_k or point from a component C_k to C_i a edge for C_iS_j is synthesized. In an LTS, we add labels to the edges. The labels we add here are the ones we can see in the bMSC representation of the scenario. Let us consider an example to demonstrate this. In Figure 2.1 we take the Analysis scenario on the bottom left and the Control component. As we can see, Control has an outgoing arrow, an incoming arrow, and then another outgoing arrow. For the LTS $C_{Control}S_{Analysis}$ this means there have to be three edges, one labeled *query*, *data* and *command* respectively. Since edges connect nodes, we also need to synthesis nodes for the LTS C_iS_j. For n edges in the LTS, we generate $n + 1$ states. These nodes are connected by the edges so that the LTS becomes a path graph. The order in which the edges connect the nodes is identical to the order in which they appear in S_j. Let us consider the example LTS $C_{Control}S_{Analysis}$ once more. Here we have three edges which means we need four nodes that we call N_0 through N_3. The first arrow connected to Control is *query*, which means a edge labeled connects N_0 and N_1 directed towards N_1. The next edge, labeled *data*, connects node N_1 with N_2. Finally, *command* connects N_2 with N_3. This result can be seen in Figure 2.3. The fact that we have $n + 1$ nodes also means that if a component C_i has no incoming or outgoing arrows in S_j, or is not present in S_j, C_iS_j consists of a single node only. This is for example the case for $C_{Actuator}S_{End}$ or $C_{Database}S_{Initialise}$.

Additional work is needed if the arrow in a scenario is an incoming arrow to the component. For every incoming arrow C_i has in S_j we need to synthesise an additional edge in C_iS_j. Again we label these edges according to the labels in the scenario. The source state of these new edges is the same as the one that has the identical label. However, the target state is different. The target node of these new edges has to be created first. We call it error state N_{err}. Now we look at the example $C_{Control}S_{Analysis}$ one more. $C_{Control}$ has one incoming arrow in $S_{Analysis}$ which means an additional edge, labeled *command*, and the additional error node N_{err} have to be synthesised. The new *command* has node N_2 as the source, since that is the source of the old *command* and N_{err} as its target.
In the next part, the annotations come into place. Every transition in each C_iS_j that we generated now receives a probability value. For every C_iS_j we take a look at all its edges and annotate them as follow:
In the case that the edge points towards the N_{err}, we add the probability value $1 - R_{C_i}$. Remember that R_{C_i} is the reliability value of the component C_i. If the edge does not have N_{err} as the target, but a transition that does point towards N_{err} has the same label, we annotate it with R_{C_i}. Every other edge in C_iS_j receives the value 1.
As a result the sum of probability values of all outgoing edges of every node is equal to one, except for N_{err}, since it has no outgoing edges. If we look at our example LTS $C_{Control}S_{Analysis}$ we can see the result of the whole process, with the annotation in Figure 4.1. The grey box with the name *Control_Analysis* with its filled in edges, represents the resulting LTS $C_{Control}S_{Analysis}$. The reliability value is 0.05 for the *data* edge leading to the error state, here called -1, since $1 - R_{Command}$ is $1 - 0.95$. Accordingly the other *data* edge gets 0.95 as its value. *Data* and *command* both receive 1. All 25 resulting LTSs from the Boiler Control System can be seen in Figure 4.1 through Figure 4.4 within the grey boxes.
Now let us also look at which additional challenges emerged in the concrete implementation with clams. In the final implementation, we first generated the nodes and then the edges. In order to globally identify the states, they received four properties:

id a number that is unique in C_iS_j. If there are n states in an LTS, they are just enumerated from 0 to n-1

type an enum that indicates if the state is the first, last, error, or just a normal state in the LTS. If there is only one state in the LTS, we only mark it as the first state.

componentIDX a number that indicates at which index the component can be found in the components array of clamsml

graphID clamsml gives every scenario in the system a unique name. GraphID is this name.

Additionally, every state has an array with all its incoming and an array with all its outgoing transitions.
After the states are created, the transitions get created. In clams, it is not necessary to label the arrows in the scenarios. For this reason, we came up with another method to identify them correctly globally. Instead of a label, the following properties are used:

id a number that represents how many arrows from the same component to the same other component are present in this scenario. For the Boiler Control System, this id is 0 for all edges since there is no scenario in which a component sends two messages to the same other component.

graphID the same as for the nodes

componentIDX the same as for the nodes

sourceComponentIDX the componentIDX of the edges source node. This property might seem redundant since we just read this property from the edges source node, but it is essential for identifying the edge in later steps.

targetComponentIDX the componentIDX of the edges target node. Just as for the sourceComponnetIDX this becomes important later.

From now on, if we talk about labels, what is actually meant are those four properties. If the labels of two edges are compared, what really happens is that all of those properties are compared. Additionally, the edge has the source and target node, and the probability as properties, but these are not part of the label.

4.2 Component Labelled Transitions System

Now, with all the generated LTSs, the algorithm can go on with the second step. For each component C_i the algorithm combines all associated LTSs $\{C_iS_1, ..., C_iS_n\}$ into one, according to the transitions in the hMSC. The resulting LTS is called a Component Labeled Transition System (cLTS). This is done by connecting the final state of C_iS_j with the initial state of C_iS_k if there is a transition from S_j to S_k in the hMSC. Now that multiple edges represent different aspects in a cLTS let the edges created in step one be state-transitions and the edges created in this step be τ-transitions. The probability value of a τ-transition is the same as in the hMSC. In Figure 4.1 for example there is a transition from the final state in *Control_Analysis* to the initial state in *Control_Register* with probability 1, since there is a transition from *Analysis* to *Register* in the hHSC, which can be seen in Figure 2.4. If there is a self-loop like the one at the *Register* scenario, the initial and the end state of one LTS are connected like it can be seen in Figure 4.2 in the *Sensor_Register* LTS. Of course, if an LTS consists of only a single state, it is the initial and final state. The cLTSs that result from combining all $C_{Database}S_j$ and $C_{Actuator}S_j$ can bee seen in Figure 4.3 and Figure 4.4 respectively.
As for the implementation with clams, the following challenges emerged:
An additional class that represents τ-transitions has to be added. These edges do not need labels; the probability, source, and target state are all necessary properties. Additionally, identical as for state-transitions the states receive two additional arrays where one holds all incoming and the other one all outgoing τ-transitions. There is one additional edge in the figures which connects a red point with a state. It is just for indicating which state is the initial state of the whole cLTS. In the implementation, this transition is not needed. The algorithm marks this state with a flag.

4.3 Minimization

In the next step, the algorithm minimizes the probabilistic component LTSs to their deterministic minimal form. This is done by eliminating the τ transitions. Rodriguez describes this as an intuitive process where the transition's source and target state are merged into one, with the merged state getting all the incoming transitions of the source state and the outgoing transitions of the target state. The probabilities of the outgoing transitions are multiplied with the probability of the eliminated τ

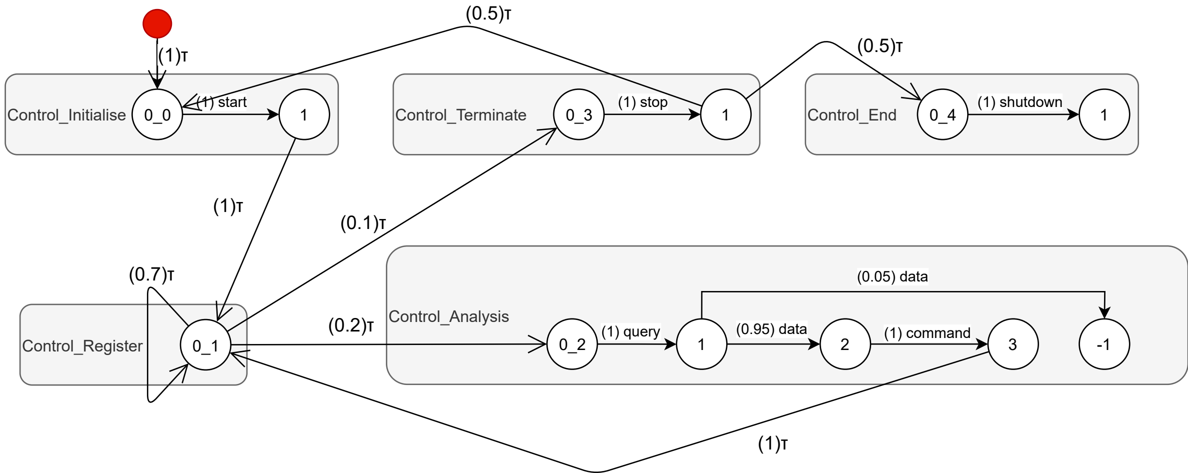

Figure 4.1: Probabilistic component LTS synthesized for component Control

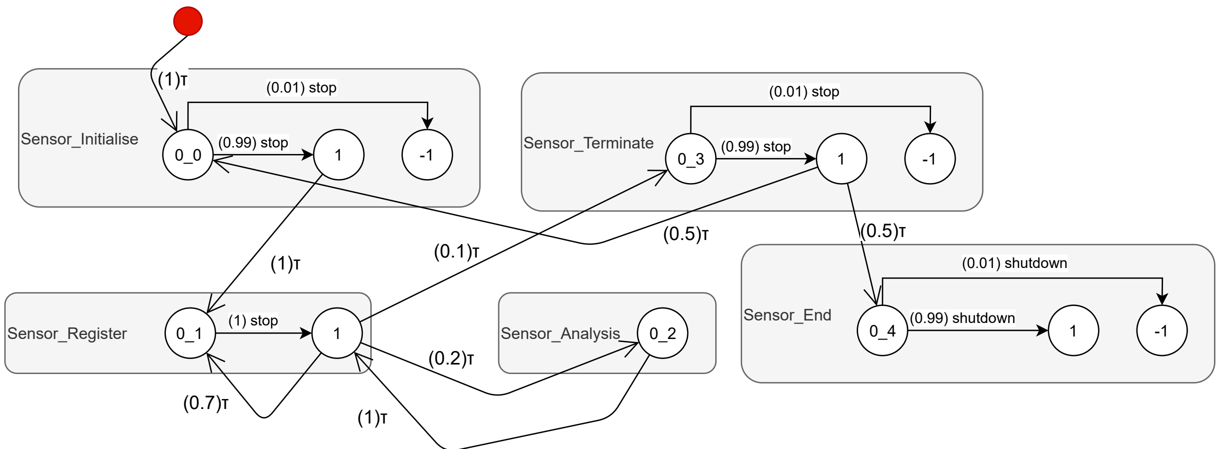

Figure 4.2: Probabilistic component LTS synthesized for component Sensor

transition. Rodrigues also shows that τ self-loops as they can bee seen in Figure 4.1 and Figure 4.4 at state 0_1 can simply be ignored. Finally, the probabilities of all transitions must be normalized [RRU05]. Now let us take a closer look at how this minimization process was implemented into clams.
At first, we combine all error states of a cLTS into one. This means that all error states of a cLTS should be deleted except for one, and all error transitions should point to the one that was not deleted. It is easy to filter out the error transitions of an LTS because they were tagged as such with an enum, as described in Section 4.1. In Figure 4.3 for instance, we can see the cLTS of the Database component. Here there are two different error transitions, one in the Database_Register and one in the Database_Analysis LTS. Let us say the algorithm deletes the one in Database_Analysis. Finally, we direct the (0.001)query transition to the error state of Database_Register. The result can be seen in the minimized LTS for Database, represented in Figure 4.7. Here only the single error state -1 is present, to which both the pressure and the query error transitions point.
As you might have noticed, there is a new state in the minimized LTSs with the label E. This state is

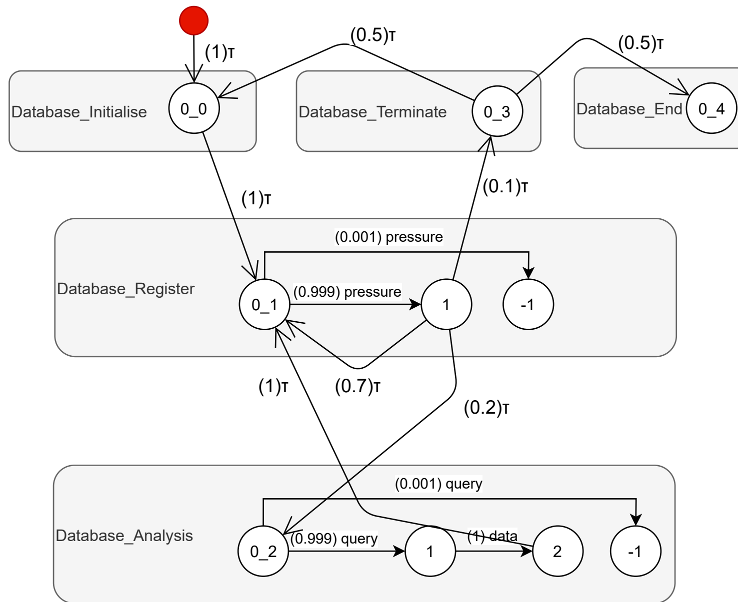

Figure 4.3: Probabilistic component LTS synthesized for component Database

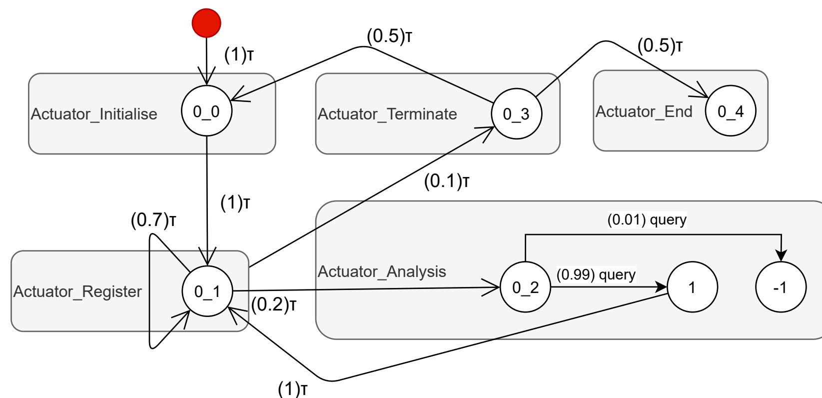

Figure 4.4: Probabilistic component LTS synthesized for component Actuator

introduced as the new final state of the LTS, which indicates the end of the scenario. For this to work, the implementation needs a new enum that represents this new final state. Next, a new state is generated that has the new enum as its type property. Additionally, we create a new transition called endAction that points from the prior final state to the new final state with probability 1. In the Boiler Control System, four new final states are created, one for each cLTS. The prior final states are present in the LTSs representing the *End* scenario since this is the final scenario in the hMSC. For the *Actuator* component the prior final state, as can bee seen in Figure 4.4, is 0_4 in *Actuator_End*. A new τ transition with 0_4 as the source and the E as the target is added.
Now the algorithm starts with the removal of the τ transitions as described above. This is implemented by iterating over all the initial states of the cLTS. Those can easily be filtered out since we marked those, as shown in Section 4.2.
But why are the initial states sufficient? A τ transition always connects a final state of an LTS with an initial state of an LTS. It is not possible that another kind of state has an incoming or outgoing τ

transition. Before we start handling an initial state, we remove all τ self-loops by simply deleting them. In the Boiler Control system, this is done for $(0.7)\tau$ at state 0_1 in Figure 4.1 and for $(0.7)\tau$ at state 0_1 in Figure 4.4. This is done before every handling of a state since new loops can emerge during the process.
Before the iteration starts, it is essential to note that initial states can not have incoming state transitions. For that reason, they are not considered in the next part.
Now the merging of the initial state, is implemented as following. For every possible combination of incoming τ transitions τin_i and outgoing τ transitions τout_j we generate a new τ transition $\tau new_{i,j}$. The probability $\tau new_{i,j}$ is the probability of τin_i times the probability of τout_j. The source of $\tau new_{i,j}$ is source state of τin_i. The destination of $\tau new_{i,j}$ is the destination of τout_j. Next up for every possible combination of incoming τ transitions τin_i and outgoing state transitions $stout_j$ we generate a new state transition $stnew_{i,j}$. The probability of $stnew_{i,j}$ is again the multiplication of probabilities of τin_i and $stout_j$. The source of $stnew_{i,j}$ is the source state of τin_i, the destination is the destination state of $stout_j$. We add all new transitions to the cLTS. In the final step of the iteration we have to consider if the initial state is the initial state of the whole cLTS. In our example this state would be the state 0_0 for all cLTSs from Figure 4.1 to Figure 4.4. Now if this is the case for our state, we have to delete all τin_i. If this is not the case for our state, we also have to delete all τout_j, $stout_j$ and the state itself.

Lets see how this works for the Boiler Control System example. In Figure 4.3 there is the initial state 0_1. This state has three incoming τ-transitions. Let τin_0 be the one from *Database_Initialise*, τin_1 be the one from *Database_Analysis* and τin_2 be the one from *Database_Register*. The state also has two outgoing state transitions. Let (0.001)pressure be $stout_0$ and (0.999)pressure be $stout_1$. The state has no outgoing τ-transitions. The algorithm creates six new transitions $stnew_{i,j}$:

- $stnew_{0,0}$ with probability $1 * 0.001 = 0.001$ from state 0_0 to -1
- $stnew_{1,0}$ with probability $1 * 0.001 = 0.001$ from state 2 to -1
- $stnew_{2,0}$ with probability $0.7 * 0.001 = 0.0007$ from state 1 to -1
- $stnew_{0,1}$ with probability $1 * 0.999 = 0.999$ from state 0_0 to 1
- $stnew_{1,1}$ with probability $1 * 0.999 = 0.999$ from state 2 to 1
- $stnew_{2,1}$ with probability $0.7 * 0.999 = 0.6993$ from state 1 to 1

After that the state 0_1 and all its transitions τin_i and $stout_j$ get deleted.
Lets consider another example, where the initial state is the initial state of the whole cLTS and has outgoing τ-transitions. Let us look at state 0_0 in Figure 4.4. Let the incoming edge from *Actuator_Termiante* be τin_0 and the edge to *Actuator_Register* be τout_0. The edge from the red dot is not considered. The algorithm generates one new transition $\tau new_{0,0}$ with probability $0.5 * 1 = 0.5$ from state 0_3 to 0_1. Since 0_0 is the initial state of the whole cLTS, only τin_0 is deleted.
After the merging process, it might be the case that the probability of all outgoing transitions from a state does not sum up to 1. Hence we must normalize those. In order to do that, the algorithm iterates over all remaining states. For each state, the algorithm sums up the probabilities of its outgoing state transitions. Now the probability of every $stout_j$ is updated to the old probability divided by the sum of probabilities.
After the merging process, it is possible for some states to be identical. Therefore we have to process them as well. Two states are identical if they have the same outgoing transitions with the same label

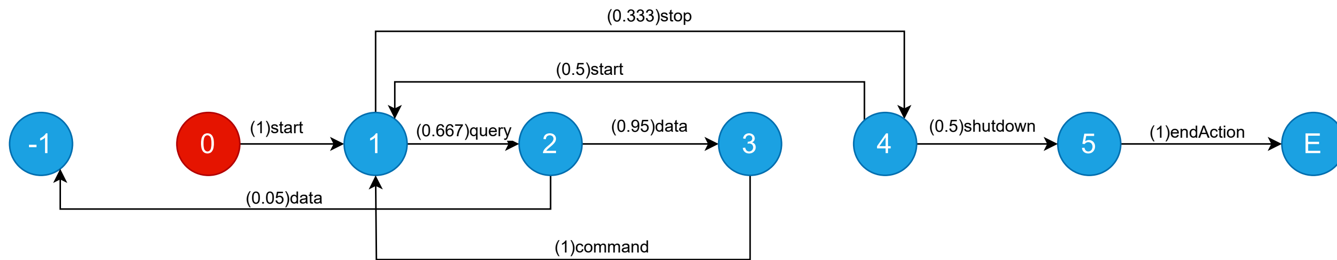

Figure 4.5: Minimized component LTS for component Control

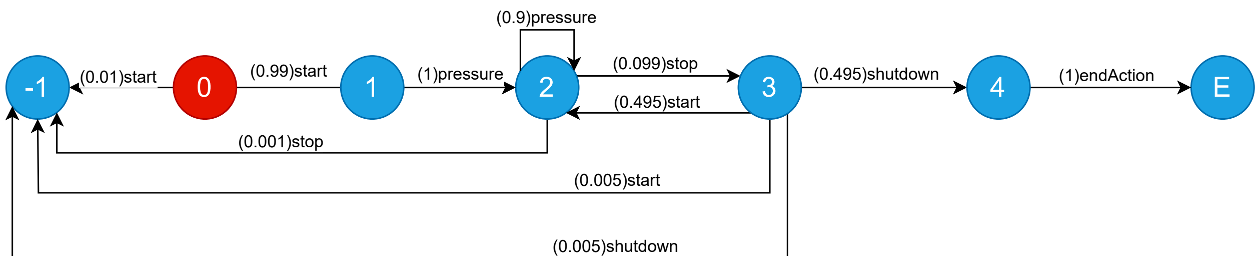

Figure 4.6: Minimized component LTS for component Sensor

to the same state. We combine those by giving one of them the incoming transitions of the other one. Additionally, we delete the other state and all its outgoing transitions. Of course, the error and final state have identical outgoing edges, none to be exact, but they are ignored in this step.
There could also be identical transitions that have to be merged. For this process, one must differentiate between error and non-error transitions. At first, the algorithm filters out all error transitions. Two transitions are identical if they have the same source and destination state. They are combined by deleting one of them and updating the other one's probability to the sum of both probabilities. For error transitions, the process is more complicated. A state may have more than one transition to an error state. Therefore the destination state alone is not sufficient as the deciding factor. Two error transitions are identical if they have the same source state and have the same source component, destination component, the same component index, and the same graph id. Then we can combine those the same way as seen above.
The minimized LTSs for the Boiler Control system can be seen in Figure 4.5, Figure 4.6, Figure 4.7, and Figure 4.8.

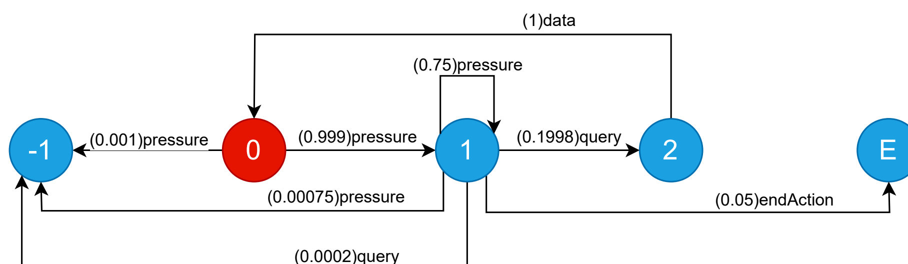

Figure 4.7: Minimized component LTS for component Database

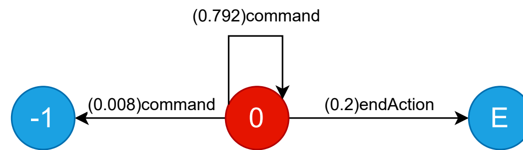

Figure 4.8: Minimized component LTS for component Actuator

4.4 Composition

In this part, we combine the previously minimized cLTSs into one. The result is a so-called parallel composition of the minimized component LTSs [RRU05]. Again, this result is an LTS that represents the actions of the whole system and shows in which order and with which probabilities actions across all components can be performed. So how do we get that? In the implementation part, two new structures have been added that help with that. One represents the nodes in the new LTS and is called CompositeState. This new class has two attributes; one is called states which is an array the size of number of components. In the Boiler Control System example, it has the size four. The other attribute, called outgoingTransitions, is an array where all edges that have this node as the source saved. Let us call the states present in the minimal LTSs ComponentStates to distinguish between them. The second data structure is called CompTransistion. This class represents the edges between the new CompositeState nodes. This class has a source and a target attribute of type CompositeState and a probability attribute of type number.
A CompositeState describes at which state in every minimal LTS the system currently is. For example, it could be at state 1 in the Control and Sensor LTSs and at state 0 in the Database and Actuator component. Let that be <1,1,0,0>. The numbers stand for the current state, each minimal LTS beginning with the state from the Control component followed by the states from Sensor, Database, and Actuator. Intuitively the initial CompositeState is <0,0,0,0>. A CompositeState has an outgoing edge if two ComponentStates have an outgoing edge with the same label. For <0,0,0,0> this is the case for the start transition, since the Control as well as the Sensor component have this outgoing edge from state 0 (Figure 4.5 and Figure 4.6). To generate all other possible CompositeStates, we apply an exhaustive breadth-first search (BFS) starting at <0,0,0,0>. A pseudo-code implementation of this search algorithm can be seen in 1. A BFS needs a frontier list, where it saves the states that are yet to expand. In the case of BFS, this list should be implemented as a FIFO queue. Its initialization can be seen in line 2. It also needs an explored set, where the states that were already expanded get saved (line 4). The algorithm terminates if the frontier has no more entries. The explored set is initialized as an empty set, and the frontier has the initial state as its single element. In the loop part, the search algorithm starts by popping the first element of the frontier and saving it as the current node (line 5). In our example, this would be the state <0,0,0,0>. This state is added to the explored set (line 6). If this state is the final state <E,E,E,E> there can not be any outgoing transitions; therefore, the algorithm stops the current iteration and continues with the next one (line 7). In the other case, the algorithm invokes a method called GetNextStates. The way this function works is described in **??** 2. We will take a closer look at it later on. For now, it is enough to know that this invocation returns a list of CompTransitions that are possible from the current node. The array can hold error, as well as non-error transitions (line 8). Each of those transitions is handled. At first, the transition is added to the outgoingTransitions array of the current state (line 10). If the transition is an error transition, the algorithm continues with the next one (line 11). If it is not an error transition, though, the algorithm checks if the

target state of the transition is already in the explored set. If it is, we are done since all outgoing edges of this state are already found. If it is not, the state is pushed to the frontier queue (line 13). This loop is executed until the algorithm tries to pop a new current node from the frontier, but the frontier is empty. This means that it has visited all possible states, and no more transitions are possible.

Now, let's take a closer look at the GetNextCompositeStates method. As seen before, this method receives the current node as an argument and returns an array with all possible outgoing CompTransitions from this node. Let the current node be $< s_{c_1,i}, s_{c_2,j}, ..., s_{c_n,k} >$. A CompTransition is possible if two states $s_{c_i,j}$ and $s_{c_k,l}$ of the current node have an outgoing transition with the same label. The first thing this method does is to combine all outgoing edges of all states in the current CompositeState into one array. Then it filters out all error transitions, since for now, we are only interested in non error transitions. First, the special case of the endAction transition is handled. If the number of transitions with label endAction in the array is the same as there are components, the algorithm returns a single CombTransition. Its source state is the current node, the target is <E,E,...,E> and the probability is 1. If this is not the case the algorithm continues.
In the next step the method searches for transitions that have the same label and saves them as tuples. For each found tuple a new CombTransition, let it be ct_{new}, is generated. The source state of ct_{new} is the current node. The target state is a modification of the current state. Let the transitions in the tuple be $t_{c_i,j}$ and $t_{c_k,l}$, while $t_{c_i,j}$ connects state $s_{c_i,m}$ with $s_{c_i,n}$ and $t_{c_k,l}$ connects state $s_{c_k,o}$ with $s_{c_k,p}$. The target CompositeState would be identical to the current node, except $s_{c_i,m}$ is replaced with $s_{c_i,n}$ and $s_{c_k,o}$ with $s_{c_k,p}$. The label of ct_{new} is the same as for $t_{c_i,j}$. The probability of ct_{new} is the multiplication of the probability values of $t_{c_i,j}$ and $t_{c_k,l}$. For every ct_{new}, except for the endAction case, an error CompTransition ct_err_{new} is generated. The source state is again the current node. The target is <-1,-1,...,-1>. For the probability value, the algorithm selects the error transition of all outgoing state transitions that has the same label as ct_{new}. Its probability value is multiplied with the probability value of the non error transition in the tuple, that belongs to the minimal LTS of the component that initialises the transition. An example for this could be the pressure transition. The error transition would be in the Database LTS. Its value is multiplied with the pressure transition from the Sensor LTS, since in the scenario, Sensor is the one that sends the pressure arrow.

Lets say the current node is <1,1,0,0>. This node is added to the explored set. This node is not the final CompositeState. Next all possible outgoing transitions are computed. All outgoing state transitions at node <1,1,0,0> are:

- (0.333)stop in Figure 4.5
- (0.667)query in Figure 4.5
- (1)pressure in Figure 4.6
- (0.999)pressure in Figure 4.7
- (0.001)pressure in Figure 4.7
- (0.792)command in Figure 4.8
- (0.008)command in Figure 4.8
- (0.2)endAction in Figure 4.8

At first, the algorithm only considers non error transitions, which excludes (0.001)pressure and (0.008)command. With the remaining transitions, tuples with the same labels are generated. In this case, the only possible tuple is ((0.999)pressure, (1)pressure). For this tuple, a new CompTransition is generated. The source state is <1,1,0,0>. The target state is <1,2,1,0>. The probability value is $0.999 * 1 = 0.999$. For the new CompTransition, a new error CompTransition has to be generated. Its source state is <1,1,0,0>. The target state is <-1,-1,-1,-1>. The error state transition with the label pressure is (0.001)pressure. Therefore the probability is $0.001 * 1 = 0.001$.
Just as in the minimization step, the transition probabilities have to be normalized to 1. After that, some additional work is needed. During the composition process, duplicate states and transitions might have been produced. Those have to be removed.
The resulting parallel composition of the Boiler Control System can be seen in Figure 4.9. This representation is in the form of an edge list for readability reasons. For each state, the outgoing edges are listed. Each row stands for an edge with its probability, label, and the target state. State Q11 has no outgoing transitions and represents the final state; for that reason, it is marked with stop.

Algorithm 1: Composition breadth first search

input : The initial CompositeState
output : An array with all coposite states

```
1  init ← the initial CompositeState
2  frontier ← a FIFO with init as the only element
3  explored ← an empty set
4  while not Empty?(frontier) do
5      node ← Pop(frontier)
6      add node to explored
7      if FinalCompState?(node) then continue
8      transitions ← GetNextCompStates(node)
9      foreach t in transitions do
10         add t to node.transitionsOut
11         if ErrorTransition?(t) then continue
12         if not AlreadyExplored?(t.to) then
13             add t.to to frontier
14         end
15     end
16 end
```

Algorithm 2: GetNextCompositeStates

```
input  : an CompositeState
output : An array with all possible CompTransitions out from the input CompositeState
1  node ← the CompositeState
2  compTrans ← an emtpy set
3  transPairs ← an emtpy set of tuples
4  allStateTrans ← node.getAllStateTransitionsOut()
5
6  /* check if the endAction transtion can be performed                          */
7  if allStateTrans.filter(t => t.label == endAction).length == node.states.length then
8  |   newState ← <E,E,...,E>
9  |   newTrans = new CompositeTransition(from = node, to = newState, prob = 1)
10 |   compTrans.add(newTrans)
11 |   return compTrans
12 end
13
14 allErrorTrans ← node.getAllStateErrorTranstionsout()
15 filteredTrans ← allStateTrans ∪ allErrorTrans
16
17 /* find pairs of transtiions that have the same label                          */
18 foreach t in filteredTrans do
19 |   index ← filteredTrans.indexOf(t)
20 |   transPartner ← filteredTrans.slice(index + 1).find(tp => tp.label == t.label)
21 |   if transPartner != null then
22 |   |   transParis.add([t,transPartner])
23 |   end
24 end
25
26 /* create a new CompositeTransition for each pair                              */
27 foreach pair in transPair do
28 |   newState ← node.updateStates(pair)
29 |   newProp ← pair[0].prob * pair[1].prob
30 |   newTrans ← new CompositeTransition(from = node, to = newState, prob = newProb)
31 |   compTrans.add(newTrans)
32 |
33 |   /* for each transition create a error transition                           */
34 |   sender ← pair.find(t => t.componentIDX === t.sourceComponentIDX)
35 |   errorTrans ← allErrorTrans.find(et => et.label == pair[0].label)
36 |   errorProb ← sender.prob * errorTrans.prob
37 |   newErrorTrans ← new CompositeTransition(from = node, to = undefined, prob = errorProb)
38 |   compTrans.add(newErrorTrans)
39 end
40 return compTrans
```

Q0 (0.01) start → **Error**
(0.99) start → **Q1**

Q1 (0.001) pressure → **Error**
(0.999) pressure → **Q2**

Q2 (0.001) pressure → **Error**
(0.801) pressure → **Q2**
(0.0004) query → **Error**
(0.039) query → **Q3**
(0.00002) stop → **Error**
(0.158) stop → **Q4**

Q3 (0.05) data → **Error**
(0.95) data → **Q5**

Q4 (0.005) start → **Error**
(0.495) start → **Q6**
(0.005) shutdown → **Error**
(0.495) shutdown → **Q7**

Q5 (0.005) command → **Error**
(0.466) command → **Q8**
(0.0005) pressure → **Error**
(0.523) pressure → **Q9**

Q6 (0.001) pressure → **Error**
(0.848) pressure → **Q2**
(0.0002) query → **Error**
(0.151) query → **Q10**

Q7 (1) endAction → **Q11**

Q8 (0.001) pressure → **Error**
(0.963) pressure → **Q2**
(0.0004) stop → **Error**
(0.035) stop → **Q12**

Q9 (0.005) command → **Error**
(0.537) command → **Q2**
(0.0005) pressure → **Error**
(0.457) pressure → **Q9**

Q10 (0.05) data → **Error**
(0.95) data → **Q13**

Q11 stop

Q12 (0.005) start → **Error**
(0.495) start → **Q1**
(0.005) shutdown → **Error**
(0.495) shutdown → **Q7**

Q13 (0.005) command → **Error**
(0.44) command → **Q1**
(0.001) pressure → **Error**
(0.555) pressure → **Q9**

Figure 4.9: A edge list representation of the parallel composition of the Boiler Control System

4.5 Calculating Reliability with Chueng

The last step to get from the system model to the availability value for the system is translating the composed LTS from Section 4.4 to a matrix and applying the Cheung model to it as seen in Section 2.5. Cheung states that we need the transition matrix M' with two additional rows and columns for C and F. As seen above, C shows if the execution was successful, and F stands for the error state. In the next step, Cheung deletes those two rows and columns since we do not need them for the calculation. The resulting matrix is called M. In the implementation part, we skip the creation of M'. We can immediately create M with our composed LTS. The translation process works as following.
In M, only the successful transitions are represented. So first, we iterate over all states in the LTS and delete all outgoing error transitions. The algorithm can identify them since it set the target state of an error transition to undefined in the composition step. Now we generate a square zero matrix $M_{i,j}$. The dimension of this matrix corresponds to the number of states in the LTS. In our example, this would be 13. Now for each transition in the LTS, we fill in the corresponding entry in the matrix. If a transition goes from state S_x to state S_y, we save the probability value of this transition to the matrix entry $M_{x,y}$. The resulting matrix is an adjacency matrix for the LTS.

The last thing that has to be done to get to M is putting the row representing the final state to the bottom row. The final row is the one with only zeros. The matrix M for the Boiler Control System can be seen in Figure 4.10. State 11 is the final state. It changed place with state 13. Now we can apply the Cheung model. First, generate an identity matrix the size of M called I. Then, we subtract M from the identity matrix. The determinant of I-M is the denominator in the formula.
To get the nominator, we create a submatrix of I-M, where the first row and the first column are deleted. The determinant of this is the nominator. The result of the division can have a negative or positive sign. Cheung uses the factor -1^n to make the result positive. This algorithm takes the absolute value of the division. The last factor in Cheung's formula is the transition probability from state n-1 to n, where n is the system's final state. In Rodrigues' model, the transition to the final state is labeled endAction and has probability 1. Therefore this factor is left out. The resulting reliability value of the Boiler Control System is 63.25%.

4.6 Recommendation of suitable services

Another task of this thesis was to the design and implementation of a search algorithm that maximizes the availability value by recommending suitable services based on the availability model by Rodrigues. Clams provides the user with the possibility to use abstract components in the design of a system. Parent services are generalized components that can be further refined by choosing one of their child nodes. A leaf node represents a concrete service with a reliability value[1]. Abstraction allows the designer to postpone the decision, which exact service they want to use in the architecture. Additionally, the designer might choose an abstract service and lets the program decide which leaf component to take in order to maximize the reliability of the system. In order to implement a recommendation system for suitable services that maximize the availability, we have to distinguish between reliability models with dependent and interdependent reliability. Component reliability is

[1] https://github.com/openclams

$$
\begin{pmatrix}
0 & 0.99 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.999 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.801 & 0.039 & 0.158 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.95 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.495 & 0.495 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.466 & 0.529 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.848 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.151 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0.963 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.035 & 0 \\
0 & 0 & 0.537 & 0 & 0 & 0 & 0 & 0 & 0 & 0.457 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.95 & 0 & 0 \\
0 & 0.44 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.555 & 0 & 0 & 0 & 0 \\
0 & 0.495 & 0 & 0 & 0 & 0 & 0 & 0.495 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

Figure 4.10: The matrix derived from the synthesized Boiler Control System LTS

depended if the value can change depending on which other components are used in the design. Say two services run on the same hardware, the failure of one would mean the failure of both services. An algorithm that chooses services with dependent reliably values has to solve an optimization problem with multiple dependent variables. A trivial solution for this might be to test all possible combinations of components and choose the one that produces the highest availability. The model described by Rodrigues only considers independent values. That means the use of a component does not impact the reliability of any other component. In that case, an algorithm that solves the recommendation problem only has to replace each abstract service with the leaf component with the highest reliability value. In the final implementation, the algorithm iterates over all abstract components. For each one, it gathers all leaf components in an array with the use of a BFS. It then chooses the array entry with the highest reliability value. If multiple entries have the same value, it just takes the first occurring one.

5 Related Work

This work uses the scenario definition by Uchitel et al. [UKM04]. They introduce a way to synthesize scenarios into Labeled Transition Systems. With this, the developer can design system models in a formal manner. Rodrigues et al. [RRU05] introduce the annotation of scenarios necessary to evaluate a system's availability. In their work, they also present the necessary steps to translate the scenario representation of a system into an annotated LTS, then describes how to minimize and compose. They show how the result is an LTS that can be directly taken to evaluate the reliability using Cheung's model. Cheung's reliability model [Che80] provided the necessary matrix representation of a system. Furthermore, they showed how a formula uses this matrix to calculate the availability of a system.
Different papers show how reliability can be calculated without the use of scenarios. As can be seen in [GLT], state-based models use control flow graphs to represent the system functionality. The underlying assumption is that the transfer of control between components is modeled as Markov chains. Path-based models as in [Sho76] enumerate possible execution paths of the program to calculate the availability.

6 Results

In this part, we look at the performance results of the reliability calculation process. Three variables can be adjusted. One can change the number of components, the number of transitions between the components, and the number of scenarios. I measured the change in execution time while modifying each one of the variables. The measuring begins with the arrival of the request sent from the web interface at the backend. The time was stopped when the reliability value was successfully calculated. To provide a measurement that was not impacted by any exceptional circumstances, like background tasks that slow down the process, each calculation was repeated 100 times, and the average was calculated. The calculations were conducted on an Intel Core i9-7900x with 64GB RAM.
To measure the impact that scenarios have on the calculation, we used a system with six components where each component had one incoming and one outgoing arrow. At first, the system was represented in a single scenario. In order to increase the number of scenarios, the single scenario was split up into multiple. Those were connected in the hMSC. Hence the functionality of the system did not change. The results can be seen inFigure 6.1.
For the components variable, I used a system consisting of one scenario. Each component received a single transition to the component directly to its right neighbor in the bMSC. The result are presented in Figure 6.2.
For the transitions variable, the system used consisted of a single scenario with five different components. The measurements started with zero transitions and were increased to 20 transitions. In the last case, each component had an outgoing edge to all other components. Those results can be seen in Figure 6.3.
Finally, we can measure the time it takes to replace generalized components. The measurement starts when the algorithm gathers possible concrete components and ends when all generalized components are replaced with the best possible alternative. The number of scenarios and transitions is not important here. Three to five concrete components could replace each generalized component. The results can be seen in Figure 6.4.

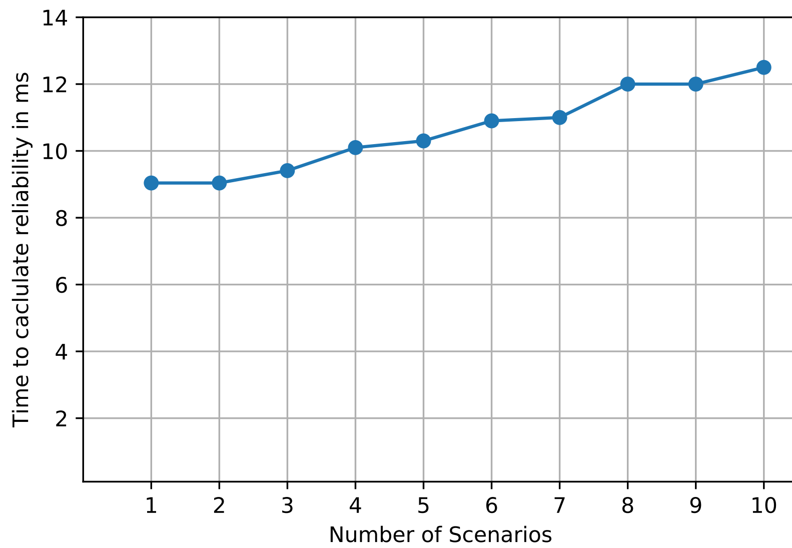

Figure 6.1: The time it takes to calculate the reliability of a system compared to the number of scenarios. The used system has six components with ten transitions

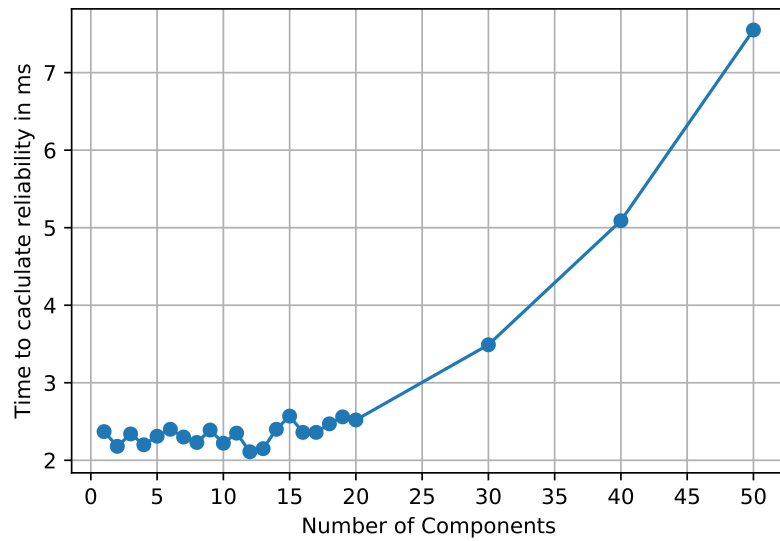

Figure 6.2: The time it takes to calculate the reliability of a system compared to the number of components. The used system has one one scenario and n-1 transitions

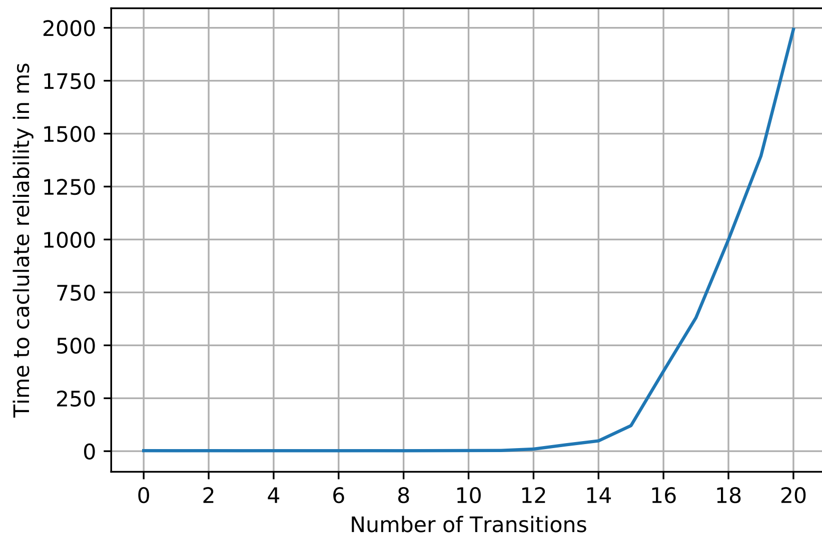

Figure 6.3: The time it takes to calculate the reliability of a system compared to the number of state transitions. The used system has five components and one scenario

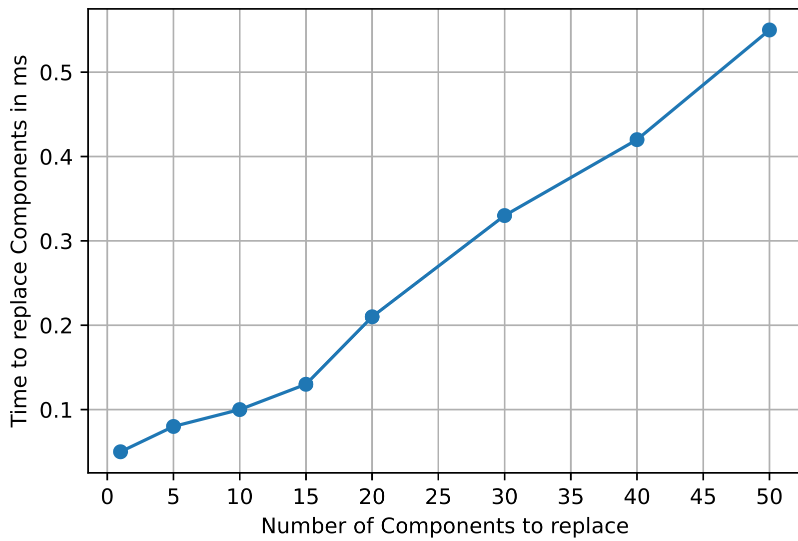

Figure 6.4: The time it takes to replace generalized components of a system compared to the number of replaceable components

7 Discussion

From the results, I can conclude the following remarks. The number of scenarios does not have a significant impact on the runtime of the program. It is undoubtedly visible that an increase in scenarios does indeed increase the time it takes to calculate the system's reliability. However, the impact is minimal. Packing the whole system into a single scenario results in a runtime of 9.04ms. Splitting up the system so that each scenario consist of only one single transition gives a runtime of 12.5ms. Taking the other graphs into account, the designer can choose freely between creating few scenarios with many transitions and many scenarios and with few transitions without thinking about runtime.
When taking a look at Figure 6.2 we see a different result. The number of components does indeed influence the runtime of the program. With only one component, the calculation takes about 2.37ms. Increasing the number of components results in 7.55ms for 50 components. For the first few additions, the runtime is not impacted much but exponentially increases significantly with further increments. This result might be influenced, though, because, with the number of components, the number of transitions increased as well.
For the number of state transitions in the system, we can conclude a significant change in runtime. In Figure 6.3 we can clearly see an exponential increase in calculation time, when adding transitions. I can also conclude that the density of transitions plays a big role since in Figure 6.2 we have a runtime of about 2.56ms while having 20 transitions but 21 components. In Figure 6.3 we can see the result for also 20 transitions, approximates 1993ms, but with only five components.
Now let us compare the probability and reliability values from Rodrigues's paper to the ones that resulted from this implementation. In her paper, Rodrigues also uses the Boiler Control System as an example for her calculations. She states the result of the process as 64.9%. This value is just slightly off the value 63.25%, which my implementation obtained. It is not clear where exactly the differences started as Rodrigues only provides intermediate results for the Control component. As for Control, the results of my and her work are identical throughout. Some differences can be seen in her presentation of the parallel composition. For some states, the probability values of outgoing edges are slightly different from mine. For this reason, I can only assume that this might be due to rounding errors along the process.

8 Conclusion and Outlook

8.1 Conclusion

Cloud computing will have a significant impact on future technologies. Therefore it will also be necessary for developers to approximate the availability of their cloud designs. This paper was able to implement a method to determine an availability value for scenario-based system designs. In this work, I also introduced the concepts necessary to understand how scenario-based system designs have to look. The reader was also intruded on how an actual implementation of Rodrigues's theoretical idea could look. I compared the result of this implementation to Rodrigues's results and could conclude that the differences were minor and are probably due to rounding errors. This work showed the performance impact of design choices and the calculation duration of the reliability value. The number and density of transition are vital factors. Finally, this thesis introduced a method to replace generalized components with services that maximize availability. For this, a simple search for the component with the highest reliability value was sufficient.

8.2 Outlook

For future work, this framework is used to enhance software system reliability using software architecture models. In Section 4.6 I proposed a way to optimize the system's reliability by replacing generalized components. This replacement could be extended to not only include reliability but also cost. When multiple services provide nearly identical reliability promises, the cost of a system might be a crucial deciding factor for the designer. With this in mind, the program's runtime can be further improved when the designer uses it to compare the effect of different services on the system's reliability. In this case, the program can do this without starting the process from the beginning with each component since the structure does not change.

Bibliography

[Che80] R. Cheung. “A User-Oriented Software Reliability Model”. In: *IEEE Transactions on Software Engineering* SE-6.2 (Mar. 1980), pp. 118–125. DOI: 10.1109/tse.1980.234477. URL: https://doi.org/10.1109%2Ftse.1980.234477 (cit. on pp. 18, 39).

[GLT] S. Gokhale, M. Lyu, K. Trivedi. “Reliability simulation of component-based software systems”. In: *Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No.98TB100257)*. IEEE Comput. Soc. DOI: 10.1109/issre.1998.730882. URL: https://doi.org/10.1109%2Fissre.1998.730882 (cit. on p. 39).

[ITU11] ITU-T. “Recommendation Z.120: Message Sequence Chart (MSC) ITU-TS”. In: (Feb. 2011) (cit. on p. 16).

[Ler14] A. Lerner. *The Cost of Downtime*. 2014. URL: https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/ (cit. on p. 13).

[Mus93] J. Musa. “Operational profiles in software-reliability engineering”. In: *IEEE Software* 10.2 (Mar. 1993), pp. 14–32. DOI: 10.1109/52.199724. URL: https://doi.org/10.1109%2F52.199724 (cit. on p. 19).

[RRU05] G. Rodrigues, D. Rosenblum, S. Uchitel. “Using Scenarios to Predict the Reliability of Concurrent Component-Based Software Systems”. In: (2005). Ed. by M. Cerioli, pp. 111–126. DOI: 10.1007/978-3-540-31984-9_9 (cit. on pp. 13, 19, 28, 32, 39, 51).

[Sho76] M. L. Shooman. “Software Reliability: Analysis and Prediction”. In: *Generic Techniques in Systems Reliability Assessment*. Springer Netherlands, 1976, pp. 343–374. DOI: 10.1007/978-94-010-1553-0_28. URL: https://doi.org/10.1007%2F978-94-010-1553-0_28 (cit. on p. 39).

[UKM04] S. Uchitel, J. Kramer, J. Magee. “Incremental elaboration of scenario-based specifications and behavior models using implied scenarios”. In: *ACM Transactions on Software Engineering and Methodology* 13.1 (Jan. 2004), pp. 37–85. DOI: 10.1145/1005561.1005563. URL: https://doi.org/10.1145%2F1005561.1005563 (cit. on pp. 15–17, 19, 39).

[WWC] W.-L. Wang, Y. Wu, M.-H. Chen. “An architecture-based software reliability model”. In: *Proceedings 1999 Pacific Rim International Symposium on Dependable Computing*. IEEE Comput. Soc. DOI: 10.1109/prdc.1999.816223. URL: https://doi.org/10.1109%2Fprdc.1999.816223 (cit. on p. 23).

All links were last followed on May 19, 2021.

Appendix

In this last part, this thesis describes a minor mistake in Rodrigues's paper that results in a small change in the code, if this evaluation server should be used for every other model, except for the Boiler Control System. In this thesis, and also in Rodrigues's paper [RRU05] there is a transition in the final composed LTS from Q6 to Q8 with the transition command, and then from Q8 to Q12 with transition stop (Figure 4.9). This behavior represents a transition in the hMSC from scenario Analysis directly to scenario Terminate without performing the necessary actions at scenario Register at first. This should not be possible, however. Thus, the code in the final implementation has two different versions—one for the Boiler Control System and one for every other system. For the Boiler Control System, an additional line of code is added to get the same results as Rodrigues for comparing purposes. The project's readme file provides the user with the necessary information to choose the correct version.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature