A Comparative Analysis of Dynamic Software Update Methods in regard to Safety-critical Systems

Dennis Karlberg*, Max Enelund†, Niklas le Comte‡
Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
* guskarlbde@student.gu.se † gusenelma@student.gu.se ‡ guslecni@student.gu.se

***Abstract*—Software is an ever evolving product that is updated to extend the functionality and to reduce bugs within a system. Many systems are required to maintain a high availability to provide their services. Dynamic software update is a mechanism which allows the software to be updated during run-time. As a result, applying this technique to systems increases their overall availability. Systems that could benefit from this technique e.g. air-control systems, banking systems and other safety-critical systems, require minimal downtime.**

In this study, we compared two dynamic software update methods in regards to safety and efficiency in performing an update. The two methods were *code relinking* and *reference indirection*. This was done through model-checking using the model-checking tool UPPAAL. We started with a literature review to understand the fundamentals of the mechanism, before creating our models and conducting the experiment. The experiment simulated 2000 executions of each model.

The experiment showed that using the method of *code relinking* is both faster and more consistent in terms of update-time. *Reference indirection*, due to its need to update a shared indirection table, requires a safer overall system-state in order to successfully perform an update, thereby increasing both the update-time itself, as well as the inconsistency of it.

Although inferior in the experiment, *reference indirection* is still a suitable method for safety critical-systems. As long as the system does not need to push an update within a certain amount of time, the two methods are more or less equally fitted to work in a safety-critical environment.

The mechanism that causes the slowness and inconsistency off reference indirection is the method's need to require a safer state before performing an update, which could positively benefit the safety of the system. This study is the first study to compare *code relinking* and *reference indirection* via model-checking.

***Index Terms*—Dynamic software update, Safety-critical, Code relinking, Reference indirection, Experiment, Model-checking**

I. Introduction

Today - software systems are developed through numbers of iterations where new functionality, bug fixes etc. are performed to evolve the software. The traditional way of delivering updates to software systems is usually done in the fashion of freezing the execution, saving the current state, loading in the new version, fetching the saved state and finally unfreezing the execution. Therefore the availability of the software is decreased while applying an update [1]. The recent trend of continuous integration/deployment [2], [3], shows that there are more and more frequent software updates than previously. Safety-critical systems are systems that require high availability and reliability e.g., air-control systems, autonomous vehicles [4]. For an update to be executed the system has to be off-line, and therefore be unavailable.

Throughout the years dynamic software update (DSU) has been a widely studied field. The purpose of it is to increase availability in terms of not having to shutdown the software as it is being updated; instead the update is applied during runtime [1], [5], [6], [7], [8]. Systems like banking-systems, traffic control systems are systems that would benefit from this technology. The goal of DSU functionality is to have flexibility, robustness, ease of use and low overhead which according to Hicks et al. [9] are the four evaluation criteria for DSU. According to Chen et al. [10] DSU is not applicable for systems with a large amount of existing binaries, furthermore they require specific compilers to compile the updated code. Multi-threaded software needs an update mechanism that is safe and timely [11]. In order for it to be safe, conditions have to be fulfilled in order for the update to be performed i.e. safe-points that have to be reached for all the nodes [1].

Some current solutions applied to high availability systems use *hot standbys* [9] i.e. redundant machines, to be able to update their systems, but this increases both the cost and complexity of the systems. With the help of DSU mechanisms there is no need for the redundant hardware to perform updates and still have an available system. The DSU mechanism could help safety-critical systems that need to have high availability with the ability to perform updates during runtime.

In this study we will refer to the DSU technique of *code relinking*, an *active* DSU method. We will also refer to the *reference indirection* DSU technique, a *passive* DSU method. These two terms are used by Hicks et al. [12] when describing the concepts of these two mechanisms. Both of these will be covered more in section IV-B.

The structure of the thesis is as follows: We start in section II by presenting the purpose of this study and why it is important, we also declare our research questions. This follows by discussing our methodology which includes our experiment variables and instruments, hypothesis and preparation in section III. In section IV we discuss the fundamentals of a DSU mechanism and describe two existing methods to perform an update and how they differ in regards to safety. After we learn the fundamentals we continue to our model design where we explain our developed models in the model-checking tool UPPAAL [13]. Before we show the result of the experiment in section VII we explain how the experiment was executed in section VI. After this we will discuss the findings in the study in section VIII and end the thesis with a conclusion in section X.

II. Purpose Of The Study

The purpose of this research study is to compare two different DSU methods in respect to properties from safety-critical systems, to determine how well suited these methods are for such a system. An experiment was to be conducted using the model-checking software UPPAAL [13], where the DSU methods were tested.

RQ1. Which DSU methods currently exist?

RQ1.1. What kind of attributes do current DSU methods have?

RQ1.2. How do active DSU methods differ from passive DSU methods?

RQ2. How do active DSU methods differ from a passive DSU methods in terms of safety?

RQ2.1. Which safety benefits and respective drawbacks do active DSU methods contribute to safety-critical systems?

RQ2.2. Which safety benefits and respective drawbacks do passive DSU methods contribute to safety-critical systems?

III. Methodology

We use two methods of data collection, a qualitative literature review and an experiment done through model checking of two DSU methods. The methods are modeled based on their presented algorithms and behaviour.

A. Experiment

In section III-A4 we describe how we prepared the model designs and our work in UPPAAL. In section V we describe how the two DSU methods are modelled in UPPAAL. Below we describe which variables we use, our hypothesis and a description of UPPAAL.

1) Variables and Instruments:

Independent variable: The experiment has one independent variable with two levels: *code relinking* (active) and *reference indirection* (passive).

Dependent variable: (1) *availability* which will be measured by *downtime*. (2) *consistency* which will be measured by *update time and downtime* in terms of its consistency i.e. if the downtime/update time always stays more or less the same.

Parameter: The experiment has one parameter which is the model-checking software UPPAAL.

2) Hypothesis: The null hypothesis for the experiment is that the performance in terms of safety is the same for the active- and passive methods. The alternative hypothesis is that the active method has a better performance than the passive method in terms of safety.

H0: Active == Passive

H1: Active > Passive

3) UPPAAL: The model-checking tool UPPAAL was used for the experiment in the study. We created a model which was model-checked for properties such as *consistency*, *availability* and *reliability* when applying DSU behaviour of the two methods mentioned above. “UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.).” [13]. The tool is a collaborative development effort by the Department of Information Technology at Uppsala University in Sweden and the Department of Computer Science at Aalborg University in Denmark. The UPPAAL extension SMC (statistical model-checker) [14] collaboratively developed by Aalborg University and INRIA Rennes is used in the experiment. The latest build of the tool was released in 2014.

4) Experimental Preparation: Before we started to develop our models in UPPAAL we analyzed the findings in the literature review in section IV and viewed existing models and documentation of UPPAAL provided by the developers, to learn how to create models with it. Furthermore, meetings with our supervisor that had prior knowledge of UPPAAL helped us understand it better and showed functionalities within it.

We analyzed the data generated by UPPAAL when exporting the CSV-file from the tool. The structure of the CSV files did not suit our needs and therefore had to be restructured to ease the process of making statistical tests. Because of the massive lines of text in the CSV-file from the 2000 simulations that we planned to do, this task would be too time consuming to do manually. A Python script was written to manage this task instead.[1]

RStudio [15] was used for the statistical parts of the thesis. The normality of the sample will first be analyzed to choose the proper statistical test. Shapiro-Wilk-Test [16] will be used for this. The sample will also be analyzed by generating both Box-plots and Q-Q plots which show outliers and if the data is normally distributed.

IV. Dynamic Software Update

A. General

DSU is a software mechanism with the purpose of reducing the downtime of long-lived systems that always have to be available. Instead of the traditional way of updating: (1) shutdown the system (2) apply the update and (3) restart it, the update is applied during runtime. A solution that exists for systems that do not support DSU, is through the use of redundant hardware [17], [11], [18], also called *hot standbys* [9]. To perform an update with redundant hardware the system switches to the *hot standby* while the update is applied in the traditional way onto the main hardware. This solution increases the cost and complexity of the system due to maintenance and the redundant hardware. The use of a DSU mechanism would solve this issue and still provide nearly the same availability. To perform a dynamic update, multiple steps

[1]The Python code can be viewed at: https://github.com/salkin91/SEMThesis17/tree/master/code

need to be performed. (1) Producing a replacement module, (2) loading a new replacement module into memory and (3) transferring execution to the new replacement module when appropriate. The correctness of the system should not be impacted by the update [18].

As mentioned in Section I Hicks et al. [9], [12] define four goals that should be part of a DSU mechanism:

> **Flexibility:** The whole system should be able to perform an update without any downtime.
> **Robustness:** Errors and other faults caused by an update should be minimal.
> **Ease of use:** The update process should be simple to prevent any unnecessary errors and/or faults while applying the update.
> **Low overhead:** The DSU mechanism should have a minimal effect on the performance of the system.

It is hard to satisfy all of these four goals, but Hicks et al. [12] implemented a DSU system that achieved all of these. In Meides and Munoz-Esconis [1] additional goals and requirements of a DSU mechanism are identified:

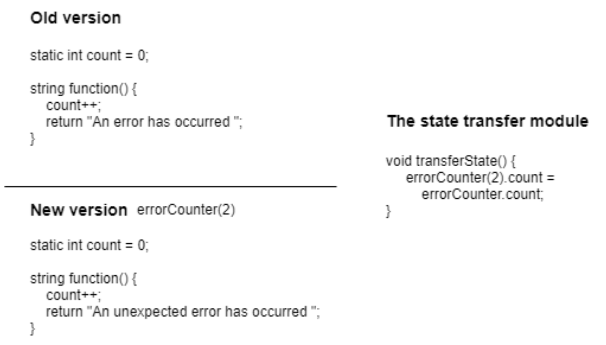
1) Transparency: Transparency for an update mechanism means that the update mechanism should be hidden, not to limit the system during use, development or run-time. Three levels of transparency can be identified [1]. First the (1) *user transparency*, the end user should not be aware that an update is being performed, which is the ideal scenario. The update mechanism should not affect or limit the way the user is using the system. Then there is the (2) *programmer transparency*, where the developer is not limited by the update mechanism in design decisions or development praxis, as well as not being required to have any knowledge of the mechanism. (3) *Application transparency*, the update mechanism does not limit how the system is designed or implemented. The performance of the system is not affected by the update mechanism, and does not impact the behaviour of the system.

2) Generality: Every part of the system should be able to be updated during runtime, but the update mechanism should not impact the modularity of the system. In order for this to increase performance and add new functionality. According to Ajmani et al. [19], the modularity allows the system not to depend on legacy code which makes the system more robust.

3) Consistency and Integrity: As Miedes and Munoz-Esconi [1, pp.3] put it "the update of a component leaves it and the whole application in a consistent or correct state". According to Banno et al. [20] there are two types of consistencies when applying a dynamic update. The *consistency of data* relates to when a component A1 is replaced with component A2, it is required that there exists a state transfer from component A1 to A2 for the new component to be initialized correctly. Otherwise component A2 will not be compatible with the rest of the system. The second type is *consistency of control flow*, when the update has been performed, component A2 will continue running from where A1 stopped. This to ensure that a request is not lost or served twice. Gregersen and Jørgensen [21] argue that the system should have the same behaviour when updated by a dynamic update mechanism as it would have if it was updated statically.

4) State Preservation: Before an update the state of the component has to be stored, to preserve the state for the new component [1]. This is closely related to *consistency and integrity*. To be able to achieve consistency, the state has to be preserved and transferred to the new component. Sridhar et al. [22] present a five-step way of handling the state transfer from an old component to a new one.

> **Initiation:** The module replacement has to be initialized.
> **Module Integrity:** The state of the changing module has to be stored.
> **Module Rebinding:** The new module has to be loaded and linked into the run-time environment, and create the new object which will replace the old one.
> **State Migration:** The state of the old module has to be transferred into the new object to get an equivalent object as the old one.
> **Instance Rebinding:** The old modules have to be linked to the new object.



```
static int count = 0;

string function() {
   count++;
   return "An error has occurred ";
}
```

```
void transferState() {
    errorCounter(2).count =
        errorCounter.count;
    }
```

```
static int count = 0;

string function() {
   count++;
   return "An unexpected error has occurred ";
}
```

Fig. 1. Example of implementation of a state transfer module

Hicks and Nettles [12] provide an example of how the implementation of a state transfer can be done. In Fig. 1 there are two different versions of the file errorCounter, with a function that returns a string and counts the number of times the function has been executed. The value of the variable count has to be transferred from version n to version n + 1 to keep the system consistent. This is done via a separate module where the state transfers are specified. They can be more complex than the provided example.

5) Version Coexistence: This requirement ensures that different versions of a component coexist and run concurrently [1], [19]. This allows a client running an old version to still have it working correctly with an updated server, which runs both the old and new version. When the client has been updated it gets serviced by the new version on the server. If a system using *inter process communication* (IPC) would only update one of the nodes, unexpected behaviour could occur [6]. *Version coexistence* is suited for medium to large-scale systems where it is impossible to update every node at once.

The programming language Erlang has support for this type of mechanism, called *hot loading* [1]. It can replace single modules and run two versions of the module concurrently. Processes using the old version will continue to work while new processes use the new version.

6) Quiescence: Quiescence is a term used to describe a system that can perform an update in a safe manner. The module being updated won't be used by another component during the update [1]. This can be achieved through two different approaches: (1) a *whitelist* of program locations (also called *update points*) where the program enters a safe state. When every thread entered one of these safe points the update is applied. (2) A *blacklist* of functions that must be inactive for an update to be applied [11].

Some argue that by only having one safe point per thread it would delay the application to reach full *quiescence*. Instead by having more safe points in each thread it would reduce the delay. But according to Heyden et al. [23] there is a major drawback in having many safe points for each thread: it will be harder for the developer to reason about the correctness of each potential update point and this might threaten the correctness of the application. According to Neamtiu and Hicks [11] by only using a few (1-2) safe points in each thread *quiescence* could be achieved in less than ten milliseconds.

Banno et al. [20] argue that the use of safe points is not enough to ensure a safe and correct update and they give an example when such a mechanism is invalid. If a module consists of two functions, *A()* and *B()*, and function *A()* is always executed before *B()*, then if the update moves some operation from A() to B() and the update is applied in between these functions, the moved operation will be executed twice, thus resulting in an invalid state. To complement the safe points some *change constraints* should be valid before an update can be applied, to ensure a safe and correct update.

7) Rollbackness: The mechanism of a *rollback* makes it possible to revert an update back to a given version of the system. This mechanism is important if a new version of the system consists of faults and other flaws, then the ability to go back to a previous version is necessary to still have a stable system [1].

The requirements mentioned above suit different kinds of systems. Depending on if the system is single-threaded, multi-threaded or distributed different requirements are needed to be able to ensure that a DSU mechanism is safe and correct. For example if a system is distributed and communication between nodes is necessary for the system to operate correctly then a requirement like *quiescence* or *version coexistence* are suited, to not have miscommunication between nodes or other failures. While this might not be as important for single-threaded systems.

B. Active- and Passive mechanisms

Having established the core principles of DSU, we still need to investigate how the mechanism work practically. Hicks et al. [12] discuss the idea of dynamically implementing *patches* into a running system. They mention two fundamental ways of implementing a dynamic patch: *State transfer-based* updating and *dynamic linking*.

State transfer-based updating is based on the idea of recompiling a new version of the system, notifying the old version that a new version is ready and transferring the state of the old version to the new. However, there are clear drawbacks mentioned by Hicks et al. [12], most important of which - from a safety and distributed system perspective - being the inability to make old-code and new-code coexist, as well as updates always affecting the entirety of the system.

The other approach - and the focus of our study - Hicks et al. [12] calls *dynamic linking*. Instead of executing a new version of the program, patches are applied directly into the current execution and the state is transferred locally i.e. on a functional level, as opposed to the system wide state transferred using the state transfer-based updating approach. Dynamic linking can be done in two fashions, *active* and *passive*.

Once the update or patch has been linked into the program, all existing function calls and so forth must be directed to use the new definitions and stubs from the update. This is done either by *code relinking (Active)* or *reference indirection (Passive)* [12].

Code relinking works as follows: after applying a patch or update to the code, the entire program is re-linked. Thus all references to the old functions or definitions will be redirected to the new ones. This is considered to be active as the dynamic linker needs to analyze the entire program and "re-link" all the older code to point to the newer code. An example of such a change is shown in Fig. 2.

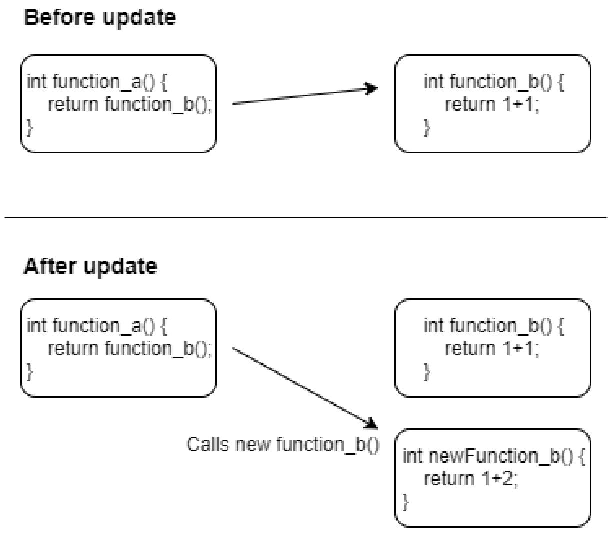


Fig. 2. Example of the active DSU mechanism

In contrast, a way to achieve the same results is via *reference indirection*. Using this method means one would need to compile the affected modules so that the global indirection table is updated accordingly. The work flow of using this method would be to load the patch and then alter the effected entries in the table to point to the new version. This method is considered to be *passive* as the existing code is compiled in order to notice new changes. The result of this is that the dynamic linker only needs to update the table instead of keeping track of the entire existing code. An example of such a change is shown in Fig. 3.

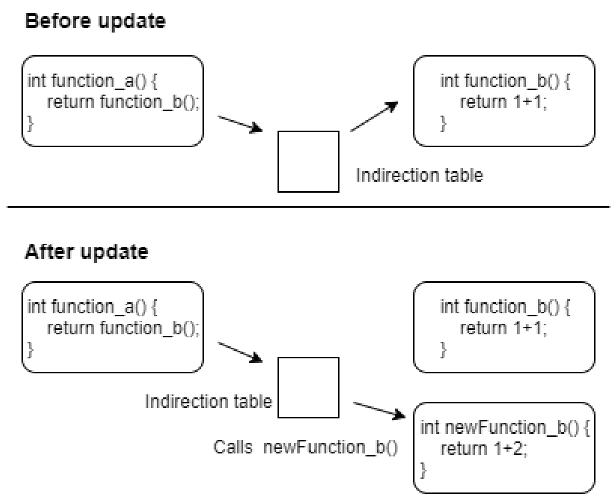


Fig. 3. Example of the passive DSU mechanism

Both methods will achieve the same results, but each has drawbacks of its own. Using reference indirection means that all function calls etc. need to go through an extra layer, the indirection table, in order to access definitions and functions. This increases the overhead of the program while also increasing the complexity of the program, making this method more difficult to implement. In contrast to this, code relinking is simpler to implement but it may also increase update time since all the callers of functions need to be re-linked [12]. For a larger safety-critical project this could render this method unsuitable.

C. Differences in regards to Safety

When talking about safety in the scope of DSU it is important to understand that patching or updating a system using these methods is not without risk. A system can run correctly after being patched but could still have been incorrectly updated depending on what state the program was in during the patch. This is usually countered by constraining, e.g. through a timing restriction when a patch may be applied [24]. Timing restriction however is not completely suitable for DSU methods as it could theoretically preclude an update to a degree that is unacceptable. Hayden et al. [23] identified three common approaches to counteract this problem, *Activeness safety (AS), Con-freeness safety (CFS)* and *manual identification*.

Activeness safety is according to Hayden et al. the most popular approach and also the approach we used in our model-testing, in unison with timing restrictions. Activeness safety prevents updates from being applied to functions that are currently *active*.

As *code relinking* and *reference indirection* are both suggested methods of DSU, as a result, we assume these approaches are viable and suitable in this scenario, which is tested in the experiment. Knowing this we anticipate no significant differences in either benefits or drawbacks between *active*- and *passive*-methods in terms of safety as the result of the experiment.

V. Model Design

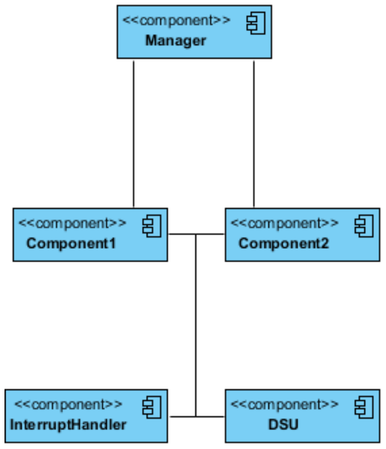


Fig. 4. Structure of the base system used in both methods.

Both of our systems, active and passive, work through a chance based approach. All actions in the system, whether that action is sending an update notification (c.f. Section V-B) or a component becoming active (c.f. Sections V-C, V-D), it is all based on chance. To ensure consistency in the data we achieve from the simulations both systems are built using the same base structure (see Fig. 4) and values. The two systems only differ where needed to simulate their active and passive mechanisms. Both systems are also built with the *safety first* principle. If a component is ever actively needed by the system, the component drops any potential updates and resumes normal behaviour. This is simulated through the use of an *InterruptHandler*.[2]

A. InterruptHandler

We designed the *InterruptHandler* as a way to simulate the urgency of a safety-critical system. Every now and then, the InterruptHandler broadcasts a message to all components that will abort any potential update in progress. This message is handled differently in the passive and active model, but the essential purpose of the automaton is to simulate that the immediate need of any component can interrupt the update.

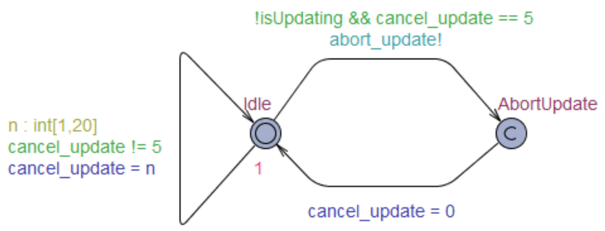


Fig. 5. InterruptHandler automaton from UPPAAL.

[2]The UPPAAL xml file can be found at: https://github.com/salkin91/SEMThesis17/tree/master/model

B. *Manager*

The purpose of the *Manager* automaton is to notify all components of incoming updates. If an update has not already been executed, the Manager sends an update request to all components. After sending the request, it enters the *UpdatingAll* state where it waits until it has received confirmation from all components that they have been updated. If the update is aborted at any point, the manager resumes the *Idle* state and will retry the update process. When all components are done updating and the Manager has received all confirmations, the Manager transitions to the *Done* state, which is the end point of the simulation.

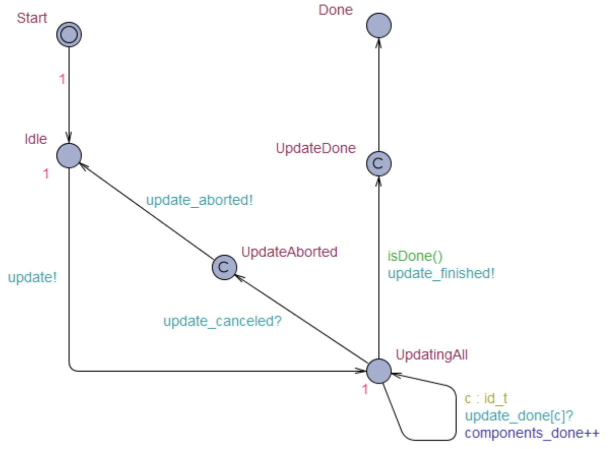


Fig. 6. Manager automaton from UPPAAL.

C. *Active - Code Relinking*

This model consists of four different automata, *Manager, ActiveDSU, ActiveComponent* and *InterruptHandler*. The update is requested via the Manager automaton which relays request to all components within the system. Within the Component automaton we decided on having a boolean flag to keep track of any pending update. This boolean is changed with the event sent out by the Manager to indicate that there is an update waiting to be applied.

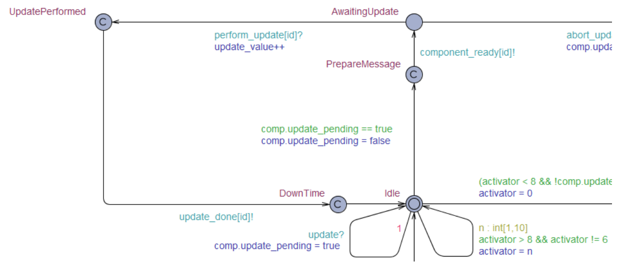


Fig. 7. A cutout of the ActiveComponent automaton from UPPAAL that simulates the components update mechanism.

Whenever the boolean is flagging, there is a pending update and the component can enter a specific state considered a safe state to receive an update. Upon entering this state an event is fired towards the ActiveDSU automaton which enqueues the component and then applies the update. The ActiveDSU keeps track of all components throughout this process and once all components have indicated successful update an event is fired towards the Manager automaton and the update is considered to be completed.

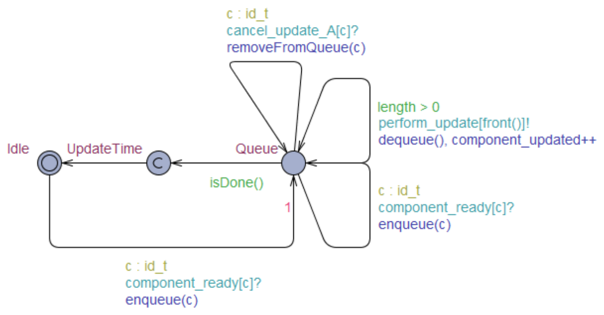


Fig. 8. ActiveDSU automaton from UPPAAL.

At any time the InterruptHandler automaton can interrupt the update. If a component is currently in the safe state it will receive an event that it is immediately needed for the safe continuation of the program. The component will then flag itself in need of an update and continue with the normal path of the program with no delay. Once it is complete and is idle once again, it enters the safe state and notifies the ActiveDSU that it is ready for the update to be performed.

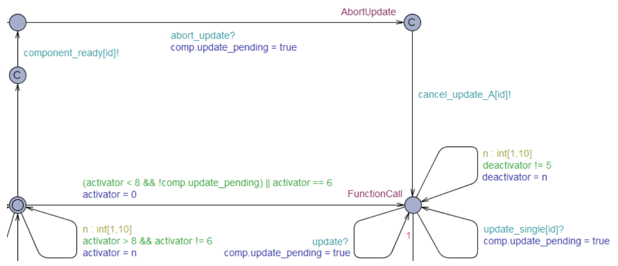


Fig. 9. A cutout of the ActiveComponent automaton from UPPAAL that simulates the components abort mechanism as well as its working behaviour.

D. *Passive - Reference Indirection*

The passive model consist of the same base automata as the active model; *Manager, PassiveDSU, PassiveComponent* as well as the *InterruptHandler*. There are two primary differences between *Code Relinking* (active) and *Reference Indirection* (passive), the first one being its runtime behaviour. A system built for reference indirection needs to pass through an indirection table for each and every function call as previously discussed in IV-B.

The other fundamental difference between the two methods is how the system ensures Quiescence (c.f. Section IV-A) i.e. how the system ensures that an update is safe to perform. Although both active and passive use the same *blacklist* approach, the nature of the way it is handled differentiates between the two methods. While code relinking uses the blacklist only to ensure that the component is not needed while

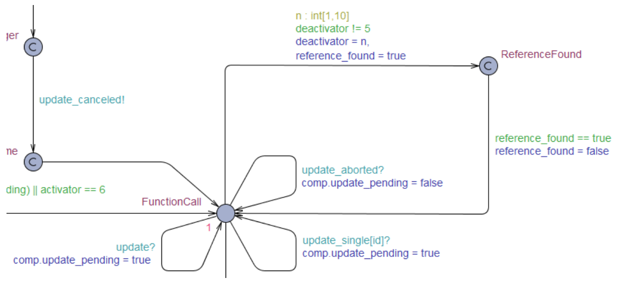


Fig. 10. A cutout of the PassiveComponent automaton from UPPAAL that simulates the component being active.

the update is in progress. The reference indirection mechanism uses the blacklist approach to ensure that no component affected by the update currently uses the indirection table, the mechanism of which will be explained further below.

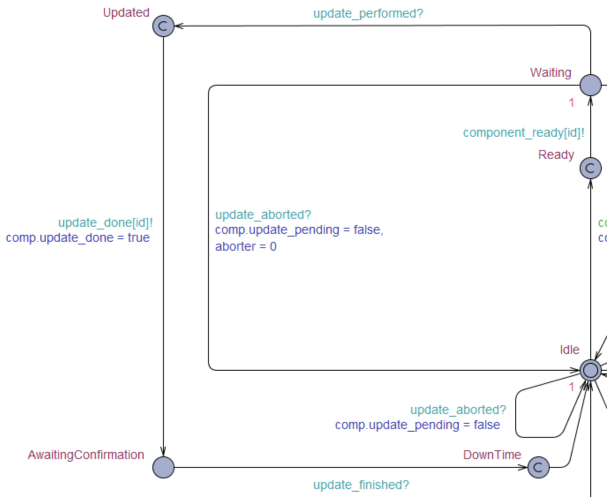


Fig. 11. A cutout of the PassiveComponent automaton from UPPAAL that simulates the components update mechanism.

The passive model, similarly to the active model, keeps track of pending updates through the use of a boolean. Depending on the state the Component automaton is in when the Manager automaton signals the update, the Component either proceeds directly into the update process, or the flag is set and the Component knows that an update is pending. Whenever a component is in the *Idle* state and an update is either incoming or pending, it proceeds with the update process. However, as previously mentioned; to ensure Quiescence, the component must wait for all other components to be inactive before proceeding with the update. Hence, the component enters a *Waiting* state. While in the waiting state, at any time, the update may be interrupted by the InterruptHandler.

If a component's update process is aborted by the InterruptHandler, all components resume normal behaviour until the Manager automaton eventually re-initializes the update process.

As a component enters the waiting state, a message is sent to the PassiveDSU automaton, notifying it that an update

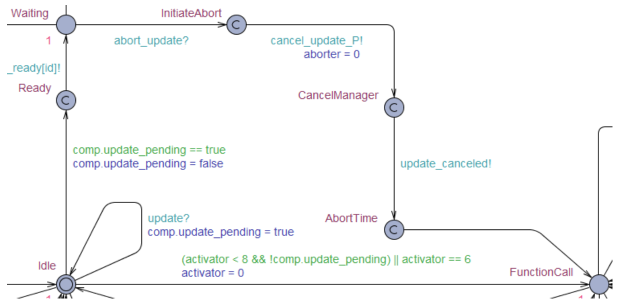


Fig. 12. A cutout of the PassiveComponent automaton from UPPAAL that simulates the components abort mechanism.

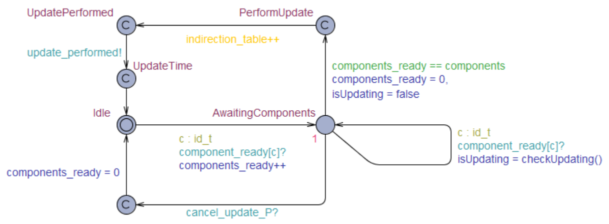


Fig. 13. PassiveDSU automaton from UPPAAL.

is in progress. At this point the PassiveDSU enters a state called *AwaitingComponents*, the purpose of which is to listen for incoming ready-messages from other components. When all components are ready, the update is executed by the PassiveDSU. To simulate the updating of the indirection table, a global variable is updated, as opposed to the active model where local variables (local references) inside the components are updated.

After an update is complete, a message is sent from the PassiveDSU to all components. The components exit the update process and resume normal behaviour.

VI. Experiment Execution

Before the execution of the experiment the modelled systems were model-checked for the property *deadlock-free*, which means that the system wont get stuck in any state. Both modelled DSU methods passed this test, therefore we can assume they are not going to deadlock during the experiment. For the execution of the experiment we used the SMC extension of UPPAAL to simulate 2000 executions of both methods. The tool extracted the time the models reached different states within every execution, which will be used as a criterion to find out the superior method. We made two different executions for each of the two methods where we increased the number of components within the model, starting with two and ended with four to measure any significant increase in time depending on the number of components.

The data received from the experiment had to be managed first by the written Python script[3]. When imported into RStudio the data had to be handled once again due to redundant

[3]The Python code can be viewed at: https://github.com/salkin91/SEMThesis17/tree/master/code

and extra values inside each of the simulations which was not relevant to include in the sample.

VII. Result

A. Literature Study Results

1) Attributes of a DSU method: There are many characteristics that a DSU method can have. Depending on the system the DSU method is applied to, different design decisions have to be made. The most general attributes such a mechanism should include are: *generality* - every part of the system can be updated, *consistency* - the behavior of the system is not affected by the update, and *low overhead* - the performance of the system is not negatively affected by the update. Error and faults caused by an update should be minimal, therefore even more attributes need to be considered. *Quiescence* and extra defined constrains can be used to make sure that the update is performed in a safe state. *Version coexistence* can be used to run two versions of the system at the same time, thus a client running the old version will be serviced by the server running the old version, while clients with the new version get serviced by the new server version. *Rollbackness* can be used to roll back to the old version of the system if the new version contains faults or if the update could not be finished.

This summarizes what kinds of attributes a DSU method need to consider, providing us with an answer to **RQ1.1**.

2) Difference between Active and Passive DSU methods: Essentially there are two fundamental differences between an active and passive DSU method: the way Quiescence is handled and the way references are updated. An active method handles the reference updating locally, inside the functions themselves. As a result, the references in one component can be changed whenever the component allows it (i.e. when it is inactive and not needed by the system), without affecting any other components. This makes for a simpler system to implement, but a more complex update process due to the fact that every caller of functions needs to be re-linked locally.

A passive method on the other hand, is designed to notice new changes, as opposed to the active method where the *dynamic linker* (the mechanism that handles the re-linking) has to actively analyze where re-linking is needed. In the method of *reference indirection* this is done through the use of an *indirection table*. Whenever an update is pushed, the indirection table is updated with new references accordingly. Using this method increases the implementation complexity of the system as well as the overhead due to the need to pass through the indirection table each function call. It does however simplify the update process by gathering all references into a table and simply updating that table, not all references locally.

This summarizes the differences between active and passive methods, providing us with an answer to **RQ1.2**.

B. Experiment Results

The data from the experiment was tested in several steps. The data will first be analyzed to investigate if it is normally distributed. The statistical test will then be chosen from the analysis result. The result from the simulations with two components will after that be compared with the simulations with four components respectively. We will use the confidence level of 0.99 for all the statistical tests.

1) Analyzing the sample: When analyzing the boxplot in Fig. 14 the data there have many outliers in the chart. These can not be removed from the sample because we do not fully know the reason behind them, but what we can suspect is that they show the absolute worst case scenario in the update time. A possible reason for the outliers could be that the update was aborted several times before finishing it due to the randomized safety mechanism within the models.

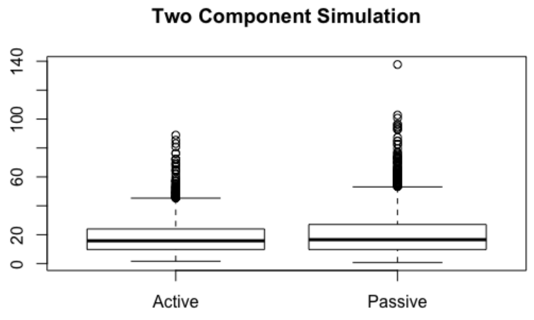


Fig. 14. Box-plot of sample with two components

2) Testing Normality: In order for the data to provide us with with useful information, the null hypothesis that the data is normally distributed should first be rejected. This is done through the use of a normality test, in this case, the *Shapiro Wilk's* [16] test for normality was used. The result of executing the Shapiro Wilk's test on the data provided us with the following;

Active: W = 0.89114, p-value < 2.2e-16

Passive: W = 0.85599, p-value < 2.2e-16

According to the *p-value* achieved from running the normality test - with a confidence level set at 0.99 - the null hypothesis that the data is normally distributed can be rejected on both the *active* and *passive* data.

However, due to the fact that the p-value was as low as it was, we decided to construct a *Q-Q plot*[25] to visualize the distribution of the data.

By observing Fig. 15 and Fig. 16, we can conclude that the majority of the data is normally distributed, but as a result of our large sample size, the significant number of outliers makes the data deviate from normality giving us a skewed normality line.

3) Statistical Test: Due to the data not being normally distributed and that we had one independent variable with two levels, for the large sample size we choose the Paired T-test as our statistical test, which is robust enough to handle the large sample size. With the null hypothesis **Active == Passive** we want to investigate if we can reject it. The Paired T-test gave the following values:

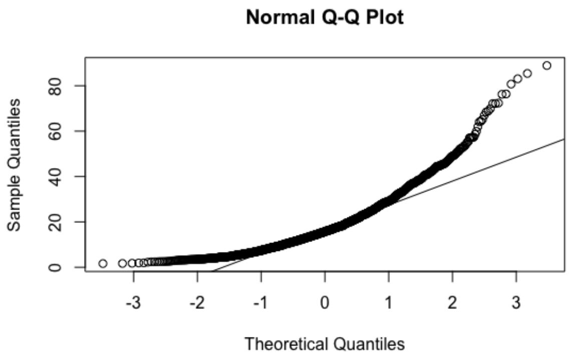


Fig. 15. Q-Q plot of the active two component data.

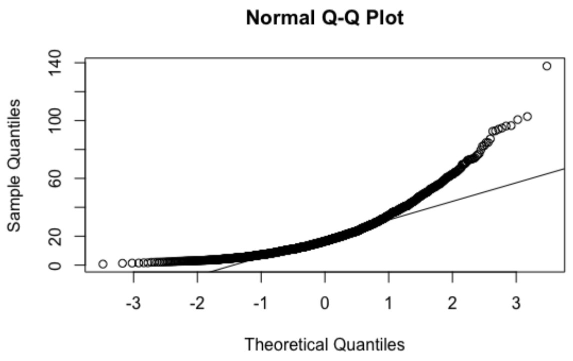


Fig. 16. Q-Q plot of the passive two component data.

t = -4.987, df = 3757.4, p-value = 6.41e-07

The *t-value* stands for *value of the t-statistic*. The *df* value provides the *degrees of freedom* for the t-statistics and *p-value* is the *probability value* of the t-statistics [26]. In this case, due to the p-value being very low, lower than our significance level of 0.01 (provided by our 0.99 confidence level), we can reject the null hypothesis. The alternative hypothesis **Active > Passive** however, can not be rejected. Therefore, we can conclude that the Active method is performing better in terms of efficiency for the test with two components.

4) Difference between two and four components.: The data from the four components in Fig. 17 seems similar to the data derived from the two components in Fig. 14, but the data from the four components have more outliers than what is shown in Fig. 14. Similar to the boxplots for both two and four components in Fig. 14 and Fig. 17 there is a relation between the Q-Q plots from the two component data in Fig. 15 and Fig. 16 with the four component data in Fig. 18 and Fig. 19. They follow the same shape as the previous two and we assume the result of the statistical test will be similar to the one with the two components.

The statistical test used for the test with four components was conducted the same as the one with the two components above. The result of it was:

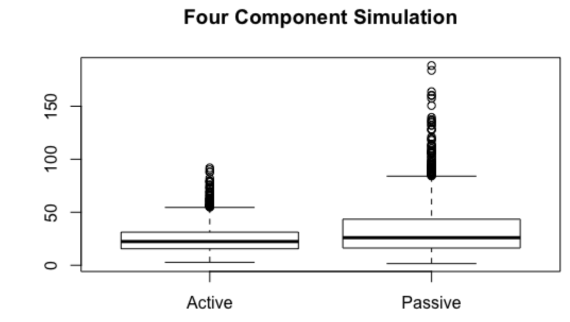


Fig. 17. Box-plot of sample with four components.

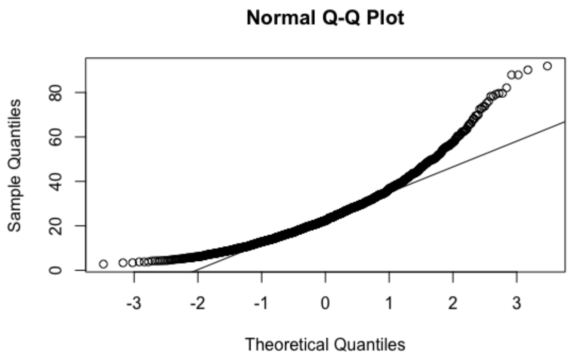


Fig. 18. Q-Q plot of the passive four components data.

t = -13.333, df = 3067.5, p-value < 2.2e-16

As above we use the confidence level of 0.99 and therefore we can reject the null hypothesis.

The difference in the means between the data with two components and four components is shown in Fig. 20. What can be concluded is that the update time is increasing with the number of components in the system.

We can after conducting our experiment conclude that in regards to safety, there are no significant differences between the two methods subjected to test. Both the active and the passive methods are based on the same presumption of implementing one or several safety constraints such as *activeness safety, con-freeness safety* or *manual identification*. Failing to do so could result in safety issues such as incomplete updates, incorrectly applied patches or unexpected behaviours. Knowing this and with the results from our experiment, being unable to measure safety variances using UPPAAL, we can reason towards **RQ2** and its sub-questions that there are no significant differences between active and passive methods of DSU. The results however, show that the active method is more efficient time-wise (shown in Fig. 15 and Fig. 16), so if the safety restriction is time-based this could have an impact on the choice of which method to implement.

What could be observed by the results in the difference

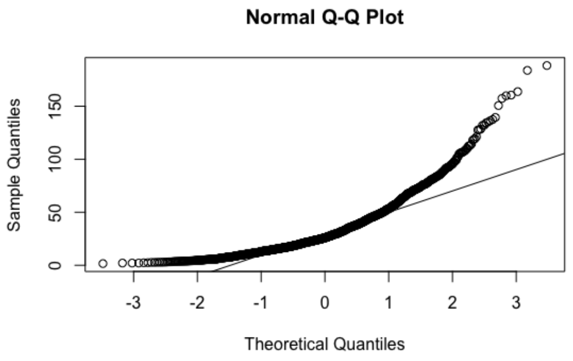


Fig. 19. Q-Q plot of the passive four components data.

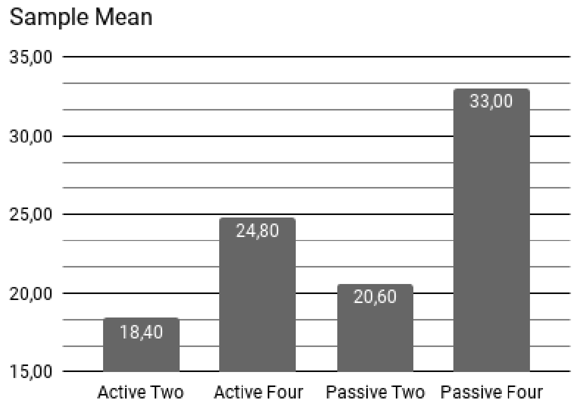


Fig. 20. Chart of sample mean values

between the statistical test with two and four components was that the active method was significantly better in both tests. By examining Fig. 20 which shows the means of the different tests, it is shown that the overhead increases by the number of components inside the system for both methods. While the mean slightly increases for the active method the passive method's mean increases even more, due to a higher number of outliers in the passive method. Our passive model was developed so that an update could only be applied when every component did not use the indirection table, which means that the more components inside the model, the longer time it took for it to finish. As for the active method which does not have the extra indirection and can update as soon as it has no caller, the active component does not have to wait for a fully quiescence system. If we were to add even more components to the experiment both of the method's means would increase but there would be a higher rate for the passive method.

VIII. Discussion

A. Threats to Validity

Below, the validity threats of the study are analyzed and discussed. The most critical validity threats to the study identified are threats to internal, construct and external validity. All of these will be covered below as well as the conclusion validity and how they are mitigated.

One of the most critical validity threats recognized is the *construct validity*. Due to our inexperience of the model-checking tool UPPAAL being limited to one assignment from one course within the Software Engineering and Management program, the knowledge of the tool's possibilities and limitations was not known. Therefore, we could not be certain that what we wanted to measure was possible with the tool. The desired dependent variables were therefore in jeopardy. Through discussions with our supervisor whom had more experience using UPPAAL we came to the conclusion that it was possible to measure time, which could then be used to get the desired dependent variables. But later research of the tool and its capacities showed that the SMC [14] extension gave an even better result. One dependent variable that was not possible to retrieve from the tool was the *number of faults* to measure the reliability of the modeled systems.

To mitigate the *conclusion validity* the normality of the sample from the experiment was checked with the Shapiro-Wilk test [16] in order to choose the correct statistical test. The sample size consisted of 2000 simulated executions from each model. Such a large sample size gave us a good representation of the results from our experiment.

The largest risk considered for the *internal validity* of the study was our inexperience of conducting research, not having done any prior research at an university level, therefore the execution might not be flawless. As mentioned above, the our knowledge in UPPAAL will also affect the design of the modeled systems and how to properly illustrate the desired systems. The risk of us being biased has also to be considered, especially when two different methods are compared, if we prefer one of the two. This risk has been mitigated by modeling the two methods as equal as possible with the only difference being their unique mechanisms.

Concerning the *external validity* the possibility not to generalize was the largest threat. The experiment was built of model-checking performed on the two DSU methods, but to fully get an answer of which method of the two were the superior one, more research has to be conducted where implementations of the two methods are compared. The result of this experiment can give a valuable impression on which the superior method is, but without actual implementation of the two methods we can not be certain which of the two works better in a real-life scenario.

B. Modeling in UPPAAL

As mentioned in the section above, our knowledge of UPPAAL were limited to one university assignment where we built a small system and verified a number of properties. We did not fully know what could be extracted from the tool at the start of the thesis. We created three different UPPAAL models during the thesis work.

We started to build something small and simplistic, trying to understand the fundamentals of the tool once again. The tool consists of two different ways of communicating: sending a message to a specific automaton or send a broadcast to every automaton listening for the message. While building the first automata we did not yet know how to measure time within the tool, but when finished, we learned about *clocks*. Clocks could be used to measure time within the tool, they had their own time unit inside of UPPAAL which did not correspond to seconds or milliseconds. Trying to integrate the clocks into the model was complicated due to our model's design. At this point we built two different new models to have the clocks integrated from the start[4]. We discovered that the clocks did not give a satisfactory result, first: they only returned one value, second: the result from using the clocks were only the sum of the clocks constraints set by us. We did not fully understand the clocks and we were not pleased with the result, as it did not provide any useful measurement for the experiment.

We started to investigate the SMC extension of UPPAAL, example models provided by the extension showed a way to simulate x number of executions of the automata. The time when every simulation entered a specific state in the model could be extracted from the tool. This was what we wanted to have. We went back to the first automata we created to integrate it with the SMC extension. There was limited information available about how the extension worked, therefore we used trial and error. There were several minor changes that had to be made for it to work properly. When fully working we made the final touches to the automata before conducting the experiment.

Due to limited resources of how to use UPPAAL many hours went into learning the fundamentals and how to use different functionalities within the tool.

C. Experiment Results

In regards to downtime; at the time we started this thesis, measuring safety in terms of *availability* seemed obvious. Updating a system during run-time, provides that the system can still be functional while the update is in progress. Discovering more about the mechanisms of DSU we realized that downtime is not really a factor to consider, due to the fact that the downtime of any system using a DSU mechanism is incredibly low, borderline non-existent. The two methods we chose to investigate in our experiment - *code relinking* and *reference indirection* - are both different variants of *dynamic linking*, meaning that their purpose is to re-link the system with new references, to a new version of the system that is already loaded and ready to go. The act of this re-linking does not provide the system of any substantial downtime that would affect it negatively in terms of safety.

Due to the fact that downtime turned out to be an irrelevant value to measure in terms of safety, we did not measure the downtime consistency. What we could measure was the consistency of the update-time.

Looking at the data achieved through the experiment we can see that the number and significance of the outliers is greater in the passive method than the active. In practicality this means that the worst case as well as the overall consistency of the passive method is worse than the active. Consistency in terms of downtime essentially boils down to reliability. When applying an update, an expected time as to when the update is finished is beneficial, not least in terms of safety. The active method - according to our data - will yield a faster, more consistent and reliable update-time.

Although inferior based on the experiment data, the passive method is still a viable method in a safety-critical system environment. The cause of the slower and less consistent update-time is the method's need to have a safer overall system state before applying an update. Ensuring a safer system state will increase the overall update-time and decrease the update-time consistency, due to the urgency of a safety-critical system. It could however also decrease the risk of possible errors or faults due to the applied update. Updating only one indirection table as opposed to the references directly in the functions, does seem a lot less error prone. Especially due to the fact that the safe point at which an update can be applied - using the active method - only ensures the component itself is inactive and not needed. As opposed to the the passive method where all components affected by the update is part of the blacklist. Bare in mind that this is only speculation and was unfortunately something we could not test through model-checking. It is merely based on the information and knowledge retrieved from the literature study and modelling of the two methods.

D. Future Work

To get a better understanding of the difference in efficiency of these two DSU methods they have to be implemented and tested on real-life systems. The model-checking is not enough to determine how well they would perform against one another. There is also the opportunity to apply them to safety-critical systems and observe how well they perform on such a system to get an even better result than we could discover in this thesis. For this mechanism to be used on such systems in the future it has first to be tested on them and we believe that this technique could be very beneficial, both in cost and easy to use where e.g. a car could be updated while driving instead of going to the mechanic to make an offline update.

IX. Related Work

DSU systems have been a widely studied field for many years. Most of the papers produced describe how a DSU system can be implemented, what requirements/characteristics such a system should have. Many studies implement their own DSU system or extend an existing one where the system's performance is usually measured. There exists studies that evaluate *timing restrictions* for a DSU system, but evaluating and model-checking a comparison between *reference indirection* and *code relinking* has not yet been done which makes this study unique. Below, papers which are related to our study will be presented.

[4]These two models can be viewed at: https://github.com/salkin91/SEMThesis17/tree/master/model/other_models

Heyden et al. [24] made the first empirical evaluation of three popular *timing restrictions*. These three were: (1) *activeness safety*, the update can not be applied to active functions. (2) *con-freeness safety*, allows type-safe modifications to active functions and (3) *manual identification*, there exists update points where an update can be applied. These three techniques were tested on three systems: *OpenSSH*, *vsftpd* and *nglRCd*. What they found was that the *manual identification* prevented all of the failures while the other two prevented most of them. They also found that activeness safety and con-freeness safety allowed more update points than manual identification but the delay of the manual identification was minimal, which is supported by [23], [11].

Chen et al. [6] implement a DSU prototype system called MUC (Multi-version for Updating of Cloud) which supports multi-version execution. This prototype is verified by applying it to the cloud applications *Redis* and *Icecast*. They found that the applications which used MUC had performance loss while an update was performed but it is a sacrifice to get a reliable and continuous service. Another implementation of a DSU system is *Ginseng* [17] which is a DSU system for C single-threaded systems which has been used to update long-lived software with the result of low overhead and its extension *STUMP* [11] (Safe and Timely Updates to Multi-threaded Programs), supports dynamic updates for multi-threaded systems. The extension is using safe points to perform safely updates. *Javelus* [27] is a DSU system for Java programs that have a lazy approach in which it only updates objects on-demand instead of extensive tracing of the heap. It also uses safe points for a safely updates. *POLUS* [28] supports updates of multi-threaded systems and it also supports roll-backs to reverse an update. It is using coexistence of the old version and the new version of the system where an update can be applied at any time. To maintain the coherence of the two versions a state synchronization function is called.

There are many different requirements and goals in which a DSU system can consist of to create a DSU functionality that is both safe, consistent and with low overhead. Almost every reference in this paper refers to at least one of them, either by the framework or implementation that is created. Meides and Munoz-Esconis [1] highlight most of the theory behind DSU mechanisms covered in the literature.

X. Conclusion

In this study, we compared two different methods of dynamic software update: *code relinking* and *reference indirection* in regards to safety and their efficiency in performing an update. The two methods were compared via model-checking with the model-checking tool UPPAAL. We discovered that it was hard to measure *code relinking* and *reference indirection* in terms of safety due to the fact that they are two different approaches to dynamic linking. We expected that downtime would be one of the primary factors in discovering whether a method is more or less safe. However, what became apparent when studying the subject was that downtime is barely a factor to consider due to the fact that there barely is any.

What could be measured in the model-checking tool UPPAAL was the efficiency of these two methods where *code relinking* was significantly faster than *reference indirection* in terms of finishing the update. To fully understand how these two methods could be applied to a safety-critical system more research has to be conducted. This study is the first study to compare these two methods via model-checking.

Acknowledgement

We would like to thank our supervisor Hang Yin for all his support and help throughout this thesis work. His knowledge and guiding has helped us a lot.

Additionally we would like to thank Jonas Kahler for proofreading and providing us useful feedback for further improvement of the thesis.

References

[1] E. Miedes and F. D. Munoz-Escoi, "Dynamic software update," *Instituto Universitario Mixto Tecnológico de Informática, Universitat Politècnica de València, Technical Report ITI-SIDI-2012/004*, 2012.

[2] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 235–245.

[3] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, 2017.

[4] J. Bowen and V. Stavridou, "Safety-critical systems, formal methods and standards," *Software Engineering Journal*, vol. 8, no. 4, pp. 189–209, 1993.

[5] V. P. La Manna, J. Greenyer, C. Ghezzi, and C. Brenner, "Formalizing correctness criteria of dynamic updates derived from specification changes," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2013 ICSE Workshop on*. IEEE, 2013, pp. 63–72.

[6] F. Chen, W. Qiang, H. Jin, D. Zou, and D. Wang, "Multi-version execution for the dynamic updating of cloud applications," in *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, vol. 2. IEEE, 2015, pp. 185–190.

[7] M. Zhang, K. Ogata, and K. Futatsugi, "An algebraic approach to formal analysis of dynamic software updating mechanisms," in *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, vol. 1. IEEE, 2012, pp. 664–673.

[8] M. Felser, R. Kapitza, J. Kleinöder, and W. Schröder-Preikschat, "Dynamic software update of resource-constrained distributed embedded systems," *Embedded System Design: Topics, Techniques and Trends*, pp. 387–400, 2007.

[9] M. Hicks, J. T. Moore, and S. Nettles, *Dynamic software updating*. ACM, 2001, vol. 36, no. 5.

[10] G. Chen, H. Jin, D. Zou, Z. Liang, B. B. Zhou, and H. Wang, "A framework for practical dynamic software updating," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 4, pp. 941–950, 2016.

[11] I. Neamtiu and M. Hicks, "Safe and timely updates to multi-threaded programs," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 13–24.

[12] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 6, pp. 1049–1096, 2005.

[13] Uppaal. [Online]. Available: http://www.uppaal.org/

[14] Uppaal statistical model-checker. [Online]. Available: http://people.cs.aau.dk/~adavid/smc/

[15] Rstudio. [Online]. Available: https://www.rstudio.com/

[16] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, No. 3/4, pp. 591 – 611, Dec. 1965.

[17] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, *Practical dynamic software updating for C*. ACM, 2006, vol. 41, no. 6.

[18] J. Montgomery, "A model for updating real-time applications," *Real-Time Systems*, vol. 27, no. 2, pp. 169–189, 2004.

[19] S. Ajmani, B. Liskov, and L. Shrira, “Modular software upgrades for distributed systems,” *ECOOP 2006–Object-Oriented Programming*, pp. 452–476, 2006.

[20] F. Banno, D. Marletta, G. Pappalardo, and E. Tramontana, “Handling consistent dynamic updates on distributed systems,” in *Computers and Communications (ISCC), 2010 IEEE Symposium on*. IEEE, 2010, pp. 471–476.

[21] A. R. Gregersen and B. N. Jørgensen, “Dynamic update of java applications—balancing change flexibility vs programming transparency,” *Journal of Software: Evolution and Process*, vol. 21, no. 2, pp. 81–112, 2009.

[22] N. Sridhar, S. M. Pike, and B. W. Weide, “Dynamic module replacement in distributed protocols,” in *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*. IEEE, 2003, pp. 620–627.

[23] C. M. Hayden, K. Saur, M. Hicks, and J. S. Foster, “A study of dynamic software update quiescence for multithreaded programs,” in *Hot Topics in Software Upgrades (HotSWUp), 2012 Fourth Workshop on*. IEEE, 2012, pp. 6–10.

[24] C. M. Hayden, E. K. Smith, E. A. Hardisty, M. Hicks, and J. S. Foster, “Evaluating dynamic software update safety using systematic testing,” *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1340–1354, 2012.

[25] Understanding q-q plots. [Online]. Available: http://data.library.virginia.edu/understanding-q-q-plots/

[26] Student’s t-test. [Online]. Available: https://stat.ethz.ch/R-manual/R-devel/library/stats/html/t.test.html

[27] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lu, “Javelus: A low disruptive approach to dynamic software updates,” in *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, vol. 1. IEEE, 2012, pp. 527–536.

[28] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, “Polus: A powerful live updating system,” in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 271–281.