P9

Evaluation of Tools for Formal Verification of OpenTitan Boot Code

Bjarke Hilmer Møller,
Jacob Gosch Søndergaard,
Kristoffer Skagbæk Jensen,
Magnus Winkel Pedersen,
Tobias Worm Bøgedal

Autumn 2020

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg East, DK
Telefon +45 9940 9940
Telefax +45 9940 9798
https://www.cs.aau.dk/

Title:
Evaluation of Tools for Formal Verification of OpenTitan Boot Code

Theme:
Security Verification and Static Analysis

Project Period:
Software 9th Semester
2020/09/02 - 2021/01/11

Project Group:
SV907e20

Participants:
Bjarke Hilmer Møller
Jacob Gosch Søndergaard
Kristoffer Skagbæk Jensen
Magnus Winkel Pedersen
Tobias Worm Bøgedal

Supervisors:
Danny Bøgsted Poulsen
Kim Guldstrand Larsen
René Rydhof Hansen

Number of pages: 94

Date of Completion:
January 8th, 2021

Abstract:

A secure system must be booted with secure boot code. To write boot code is not a trivial task, and bugs and flaws can compromise the security of an entire system. To help prevent such a scenario we look at tools that could be able to formally verify boot code written in C. We use the model checking tools UPPAAL and UPPAAL SMC and the formal verification tool Frama-C and its plugins Eva and WP. To evaluate these tools we apply them to simple examples. The intent of the evaluation is to estimate their applicability in the verification of the boot code of OpenTitan, an open-source Root of Trust. To concretize verification of OpenTitan boot code, we define a number of verifiable security goals as part of our security analysis. The security goals should be verified to give a formal proof that OpenTitan works as intended and that the risk of compromization is very low. The security analysis also includes a threat model that investigates attacks that may circumvent the security mechanisms built into OpenTitan. Based on our investigations and our work with UPPAAL, UPPAAL SMC, and Frama-C we believe that the tools can be used to verify relevant properties concerning the security goals for the OpenTitan boot code.

Contents

Introduction

Today software systems play a large role in our society, since software is present in systems ranging from smartphones to airplanes. Therefore, software has an influence on people's lives, and the demand for software that is safe and behaves correctly is evident. This demand for correct and safe software is not limited to self-driving cars to correctly distinguish humans from roads, or other hyped technologies. Much effort should also be put into software written for more fundamental purposes, such as the booting process. The reason being that incorrect or unsafe fundamental software may compromise an entire system's stack.

A company called MAXSYT [1] has an ongoing business endeavor related to the verification of boot code. MAXSYT is currently in a collaboration with AAU discussing the possibilities within formal verification. This report is written as an investigation into the tools for verifying the correctness and safety of boot code.

Safety in boot code can generically be described as the boot code's ability to fulfill its goals (e.g. do not execute unvalidated code) in the presence of an attacker. The correctness of boot code is necessary to verify in order to address its safety. As software errors such as memory safety bugs may make boot code vulnerable to attacks.

In this project we look into formal verification of code written in C. C is a programming language that is commonly used in boot code. This is despite the fact that C-compilers do not check for common run-time errors (RTEs) like out of bounds access of arrays, reading uninitialized variables, and dangling pointers. The fact that these RTEs are undetected increases the need for formal methods and tools that can verify boot code written in C.

Therefore, we look into the formal verification tools: Frama-C and UPPAAL. Frama-C [2] is a platform that supports a suite of plugins to formally verify code written in C. Additionally, a tool such as UPPAAL [3] is a powerful tool for formally validating and verifying a model of a system built as timed automata. These tools can serve different purposes in terms of verifying a piece of software. We will also look into the underlying implementation and security of OpenTitan which is a new open-source secure chip design project. The reason being that the security goals and verification challenges of OpenTitan are similar to those of MAXSYT's tasked boot code.

This leads to the initial problem statement:

How can code be formally verified, and what needs to be formally verified about the security of OpenTitan?

Chapter 1

Introduction to Boot

In this chapter, we investigate BIOS and UEFI, as both are popular within the realm of boot code. BIOS (Basic Input/Output System) can be described as a form of firmware that is used to initiate hardware during boot. UEFI somewhat overlaps the functionality of BIOS but can be more broadly described as a specification for firmware that interfaces between the kernel and the underlying hardware platform. We will also introduce fundamental concepts within the topic of boot, such as boot loader and secure boot, which will serve as a preliminary for the security analysis of OpenTitan (cf. chapter 5) even though OpenTitan's boot code is not considered BIOS firmware or UEFI compliant.

1.1 BIOS and UEFI

Before a computer can run the Operating System (OS) the OS must be booted. This is often called the boot process and is, in older computer systems, initiated by the Basic Input/Output System (BIOS). The BIOS is the first firmware that is started when a user presses the power button on their machine. The BIOS firmware is located on its own memory chip. The BIOS is responsible for finding and running the software that will boot the OS. To boot the OS, the BIOS looks for a small program, at a certain place in memory, called a boot loader (cf. section 1.2). The boot loader will then boot the OS [4].

The BIOS is an old construct that has been patched and tweaked to keep up with the rapid development in the field of computers. Furthermore, different BIOSs suffer from inconsistencies between vendors. This is because there is no one specification for the BIOS and as such, it can vary in implementation from product to product [5, ch.3].

The BIOS is also built for 8 or 16-bit architectures with patchwork implementations to allow for booting of 32-bit operating systems. However, by around the 2000s, it became clear that the BIOS was becoming obsolete. Booting 64-bit operating systems using a BIOS seemed unfeasible due to the limited size of the boot sector of the BIOS (for the first stage of the OS boot loader). Because of this, people began looking for a standardized replacement to the BIOS [5, ch.3].

As described above, the BIOS is generally deprecated in terms of the bit architectures used in modern systems. Furthermore, the BIOS does not support drivers for devices such as tablets with a stylus, GPUs, and multiple monitors [6]. There is however a successor for the BIOS, called the Unified Extensible Firmware Interface (UEFI). The tasks of the BIOS and UEFI compliant firmware are roughly the same; it is a way of booting the OS. However, the procedure to do so differs.

Extensible Firmware Interface (EFI) is fundamentally made to be able to handle the evolution of technology. A few years after EFI was created it was taken over by an industry forum and named the Unified Extensible Firmware Interface. Members of this forum include Microsoft, Apple, Intel, and AMD [7, ch. 1]. UEFI is not in itself firmware, but rather a specification that firmware can adhere to. If a piece of

firmware adheres to the UEFI specification it is said to be UEFI compliant [8].

UEFI is an interface specification that allows compliant firmware to locate, initiate, and keep track of pre-OS agents (such as OS loaders). The UEFI interface specification is built on different concepts, such as the UEFI System Table, protocols, and the Handle Database [7, ch. 2]. Descriptions of these concepts can be found in appendix A.

1.2 Boot Loader

A boot loader is the first program that is run by the firmware (often BIOS or UEFI compliant) when the system receives power. The main goal of a boot loader is to load another program into memory and initiate its execution. The execution of the program will eventually lead to the system performing its intended task if no system failure occurs.

A boot loader is stored in non-volatile memory at a known address and is trusted to be safe. A UEFI compliant boot loader implements secure boot (cf. section 1.3) thus validating the authenticity and integrity of the loaded program(s). The subsequent loaded program(s) often implicitly trusts that it has been initiated in a safe state. There exist techniques (e.g. measured boot [9]) in which a loaded program may check the integrity and origin of the preceding firmware and boot loader [8, p. 3].

Boot loaders are prone to bugs and security vulnerabilities. This is due to their complexity and their hardware dependence which makes testing and verification difficult [8, p. 2].

1.3 Secure Boot

Secure boot is a feature of the UEFI specification [10, pp. 221–224] developed to make sure a device only boots using software that is verified by the Original Equipment Manufacturer (OEM) [11]. The UEFI compliant firmware is known to be trustworthy by the OEM and acts as the Root of Trust (i.e. the security of the entire system is dependent on the firmware). Secure boot requires that each UEFI application loaded at boot is validated against its signature and a known public key. A UEFI application is signed with a digital signature. The UEFI specification specifies the recognized signature as being, among others, an encrypted SHA-256 hash. The signer has both a (secret) private key for signing and a public key for verification. The digital signature is obtained by hashing the application and then signing the hash using a private key. The UEFI applications are then distributed with the digital signature. The signing is illustrated in Fig. 1.3.1.

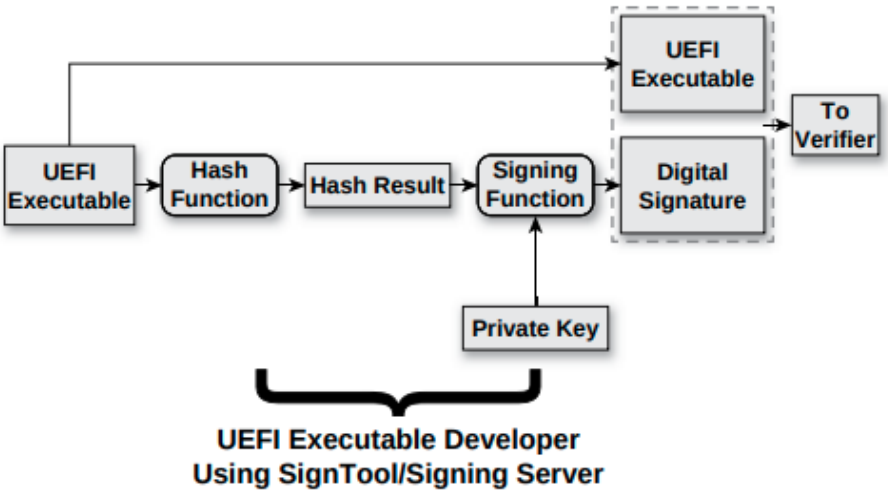


Figure 1.3.1: Signing of UEFI application [10].

The verification of the UEFI applications occurs during boot. Firstly, the hash of the UEFI application is computed. Then the digital signature is decrypted using the public key. The application is trustworthy iff the computed hash is equivalent to the decrypted signature. The verification is illustrated in Fig. 1.3.2.

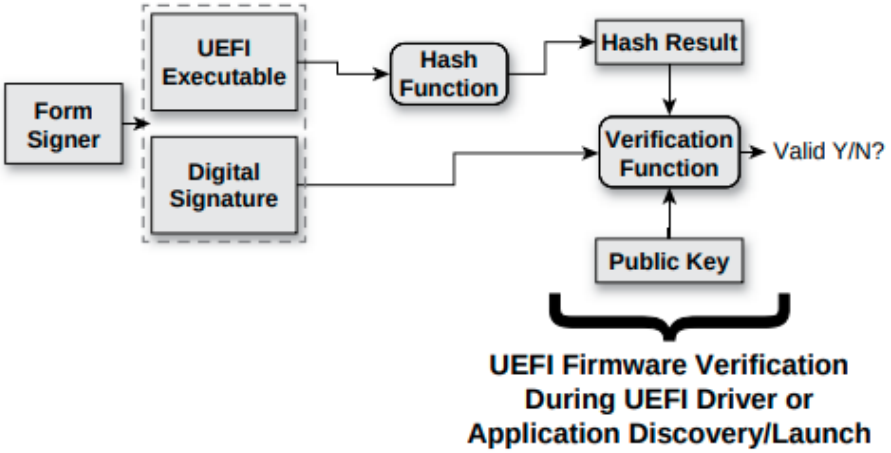


Figure 1.3.2: Verification of UEFI application [10].

Chapter 2

Formal Methods for Verification

In this chapter, we will define some of the sub-fields within formal verification. The purpose is to give a theoretical overview of verification. Testing and verification are two distinguished practices, though they are related. Verification concerns establishing the logical correctness, accuracy, or validity of all execution paths of a system design/implementation. Testing concerns executing a run-through of the system and assessing functional or structural requirements. Testing is an integrated and established part of all software development. Verification is mostly used within the development of safety critical systems, protocols, and embedded systems, and not commonly applied outside of these.

Formal verification techniques do in general have challenges that keep them from widespread use. We believe the most important of these to be the difficulty of verification in practice which is dependent upon technique, tool, and system environment. Verification of a system may require a combination of multiple tools/techniques. Verification tools often have their own language(s) with distinct syntax and semantics which may have a steep learning curve. Furthermore, the "modeling gap" problem is endemic in model checking. This means that there is a gap between the "model" that is verified and the concrete implementation of the system. While this challenge is mostly concerning model checking, static analysis, and theorem proving can be seen as having a problem akin to the "modeling gap". Static analysis takes source code as input, that does not include information about address space and hardware specifics. Additionally, the input is often annotated e.g. with assumptions about function behavior making the model more abstract.

2.1 Model Checking

According to the book "Handbook of Model Checking" [12, p. 1] model checking is defined as "A computer-assisted method for the analysis of dynamical systems that can be modeled by state-transition systems". The goal of model checking is to verify properties or absence of properties in a design or implementation. Multiple model checking tools exist, e.g. UPPAAL [3], TAPAAL [13], and PRISM [14].

A model of a system is formalised according to some specific model semantics. Scientific papers in the *Modelling and Verification* field commonly use timed automata, Petri nets, labeled transition systems (LTS), etc. [12, p. 6]. These model semantics may differ but share common traits. As an example we look at timed automata. The definition of timed automata is taken from [15]. A Timed automaton is defined as a quadruple:

$$(L,\ l_0,\ E,\ I)$$

The components of the quadruple have the following meaning:

- L is a finite set of locations,
- $l_0 \in L$, is the initial location,
- E is a finite set of edges and is defined as: $E \subseteq L \times B(C) \times Act \times 2^C \times L$,

- I is function that assigns invariants to the locations, defined as: $I : L \rightarrow B(C)$,
- C is a set of clocks and B(C) a set of guards over the set of clocks,
- Act is a set of actions.

A state of a timed automaton consists of a specific location, l, and a clock evaluation, v, which is a function $v : C \rightarrow \mathbb{R}_{\geq 0}$ that evaluates the values of clocks. Thus a timed automaton state is a pair (l, v). Progress in a timed automaton happens either through firing an edge to change the current location l and potentially resetting a clock, or by delaying a clock. The transition relation is in [15] defined by:

$$(l, v) \xrightarrow{a} (l', v') \text{ if there is an edge } (l, g, a, r, l') \in E \text{ such that } v \models g,\ v' = v[r], \text{ and } v' \models I(l')$$

$$(l, v) \xrightarrow{d} (l, v + d) \text{ for all } d \in \mathbb{R}_{\geq 0} \text{ such that } v \models I(l) \text{ and } v + d \models I(l).$$

An illustration of a timed automaton can be seen in Fig. 2.1.1

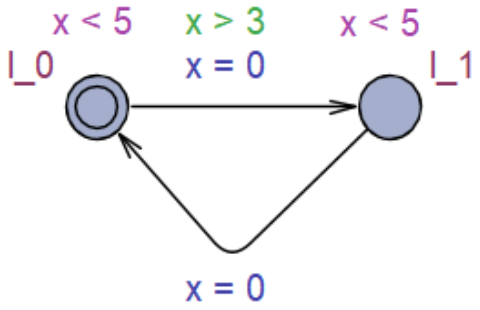


Figure 2.1.1: A timed automaton.

Model checking tools typically implement (subsets of) languages from the logic family (CTL, MITL, LTL, etc.) as the query language to describe system properties as formulae. CTL is commonly used. CTL is interpreted over transition systems (i.e. states and transitions). Transition systems can be seen as a compressed data structure for a tree-like state-space. The temporal operators in CTL allow for specifying safety, liveness, and reachability properties.

Model checking's biggest challenge is arguably the state-space explosion problem. The size of the state-space grows exponentially with the size of the transition system. This presents an issue for verification time and memory usage. In practice, this means that the developer has to model with an appropriate abstraction level to avoid seemingly endless verification time of certain CTL formulae.

There exist several subtypes of model checking. We will describe the probabilistic model checking and statistical model checking subtypes.

2.1.1 Probabilistic Model Checking

Probabilistic model checking (stochastic model checking) extends the model semantics with probabilities. Tools typically implement PCTL, a probabilistic extension of CTL. The result of the verification is a probability for the property to be satisfied. The model checking tool PRISM implements probabilistic model checking. The PRISM tool supports discrete-time Markov chains, Markov decision processes, and continuous-time Markov chains [16].

2.1.2 Statistical Model Checking

Statistical model checking (SMC) extends the model semantics with probability/non-determinism. The output of statistical model checking differs from traditional model checking. The goal is to compute a probability or estimation of a value or that a given property is satisfied. These computations rely on simulation(s) of the model. This has two interesting features. Firstly, although state-space explosion is still prevailing, the execution of the "verification" concerns simulating the model a given number of times. As such it is a more lightweight verification and likely to be considerably faster and practical compared to traditional model checking. Secondly, the result of a query may vary. It is the result of a finite number of executions of models with probabilistic choices. Due to the statistical approach of SMC, it does not provide formal proofs.

The model checking tool UPPAAL SMC implements statistical model checking. UPPAAL SMC models are networks of stochastic timed automata and the implemented query language is an extension of the temporal logic language MITL.

2.2 Static Analysis

This section is based on the first chapter of [17]. Static analysis is a technique used to reason about programs without running them. This method has been used since the 1960s in compilers and IDEs but can also be used to formally verify that a piece of code is correct. As examples, static analyzers could be used for the following:

- Program optimization which is often associated with optimizing compilers. It is used to find out whether there is dead code in a program, whether a variable can be precomputed at compile time, whether an expression inside a loop can be moved outside that loop, among other things.
- Program correctness which is often used to detect errors in code or verify the absence of said errors. The errors could be initialization after reading of variables, dangling references in the program, or an RTE such as division by zero or overflow.
- Program development (often used in modern IDEs for refactoring, debugging, etc.). E.g. where did a given variable get assigned its current value? Can the value of one variable affect another?

It is, however, difficult to reason about any program (which can be gathered from Rice's Theorem). As such, static analyzers try to give approximate results to ascertain useful answers about programs. A program analyzer is considered to be sound if it never gives results that are incorrect. However, a program analyzer is allowed to answer maybe. Thus a trivially sound program analyzer is one that always answers maybe to any question about a program. For verification specifically, we say that a given verification tool is sound if it never misses an error that it is supposed to detect. But such a tool may give false positives. The point of working on static analysis then becomes to create a program analyzer that answers yes as often as possible as opposed to maybe.

2.2.1 CBMC

This section is based on [18]. CBMC [19] is a static analysis tool used for formal verification of code written in ANSI-C using bounded model checking. It lets the user write assertions about any run-time program state. In order to work properly, CBMC needs the program to be in Static Single Assign (SSA) form. CBMC rewrites a copy of the code in five steps:

1. All loops are rewritten to while loops. Then all `break` and `continue` statements are replaced with equivalent `goto` statements.
2. Afterwards, all loops are unrolled. This means that the loops are replaced with a sequence of `if` statements where the branch condition is the same as the loop condition in the original loop. The body of the `if` statements will be the body of the original loop. CBMC inserts its own assertions (one for each original loop) called an "unwinding assertion" to make sure that there are no paths that require more loop iterations than have been performed. [18] does not go into detail about how CBMC handles infinite loops.
3. `goto` statements that point backward are unrolled similarly.
4. All functions are expanded, meaning that instead of calling a function, the body of the function will be there instead. Recursive functions are unrolled similarly to how loops are unrolled.
5. The program is then transformed into Static Single Assign form. This is possible since, at this point, the entire program is one very long main function without loops, backward-pointing `goto` statements, or function calls.

When the code has been rewritten, then for each assertion made in the code, CBMC sets up a propositional formula for whether the execution of the program statements can make that assertion satisfiable. CBMC uses a SAT-solver to find out whether the propositional formulae are satisfiable. If an assertion cannot be proven, CBMC provides a counterexample. The GUI of CBMC allows the user to go backward and forwards in the execution of the code in order to make errors in the code more easily understood.

2.3 Theorem Proving

A mathematical proof is a deducible logical statement, showing that a mathematical assumption logically guarantees a conclusion [20]. There exist two distinct types of mathematical proofs, namely, formal proofs and informal proofs. Theorem proving concerns formal proofs written in formal languages. In formal proofs all steps are axioms or derived from axioms by application of fully stated inference rules [21]. The formalism language used is commonly either (listed in ascending order of expressiveness) propositional logic, first order logic, or high order logic. Formal proofs are used to logically guarantee a system property. A formal proof would then require "modeling" the system and environment in a formalism language.

Typically there is a trade-off between the expressiveness of the formalism and the ease of proof automation. According to [22] simple formalism theorem provers may be so automated that they, to the user, do not appear as theorem provers. [23] states that automatically generated proofs may be hundreds of pages long and difficult to understand. More expressive formalisms can express properties, not in the scope of simpler formalisms, but require extensive manual work [22].

The theorem proving tool Coq [24] uses a combination of high-order logic and a richly-typed functional programming language [25]. Coq provides some theorem proving assistance in the form of interactive theorem proving (i.e. IDE with some automated proof snippets).

The next two chapters concern examples of UPPAAL, UPPAAL SMC, and Frama-C, for formal verification. The examples that we use to show the capabilities of these tools along with how they are used are not founded in boot code however. We have chosen to use more generic examples of systems/code as trying to apply the tools to actual boot code would be outside the scope of the allotted time for this semester. Even without using boot code as our examples we can still show the tools and their potential for this project.

Chapter 3

UPPAAL Modeling

The purpose of this chapter is to gain an insight into the strengths and limitations of formal verification based on model checking and statistical model checking. For doing this, we describe our experience with using UPPAAL and UPPAAL SMC for modeling and verifying a distributed system we developed during our 7th semester. The reason for modeling this system is that we aim to explore the strengths and weaknesses of UPPAAL and UPPAAL SMC, and therefore find it most appropriate to model a known system. Lastly, we will reflect upon the complementary use of UPPAAL and UPPAAL SMC.

3.1 Introduction of P7 System

Our P7 system [26] consists of probes, two APIs (one called SigfoxDB API and another SensorThings API), a database, and a web-based client application. In addition, it uses an external system called Sigfox [27] which allows the probes to send data to a third-party cloud hosted by Sigfox (referred to as Sigfox Cloud), via the Sigfox protocol.

The probes measure different properties of water (i.e. temperature, turbidity, pH) and send this to the Sigfox Cloud via the Sigfox protocol every 15 minutes. When the Sigfox Cloud receives a new measurement from a probe, it initiates a callback to the SigfoxDB API which takes the data and inserts it into the database. This database is a NoSQL database, specifically a MongoDB [28]. Users of the web-based client application can request the data from the probes via the SensorThings API. The SensorThings API then queries the database for the specified data and sends it back to the user. A more detailed description of the system and the communication between the different components can be seen in Fig. 3.1.1 (the arrows indicate flow of information) and the package diagram in Fig. 3.1.2 (the arrows indicates usage).

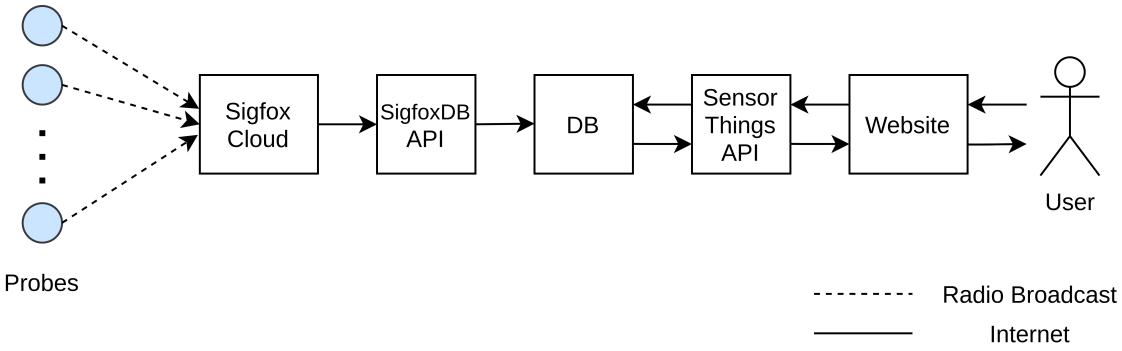


Figure 3.1.1: The components of the probe system and how they communicate.

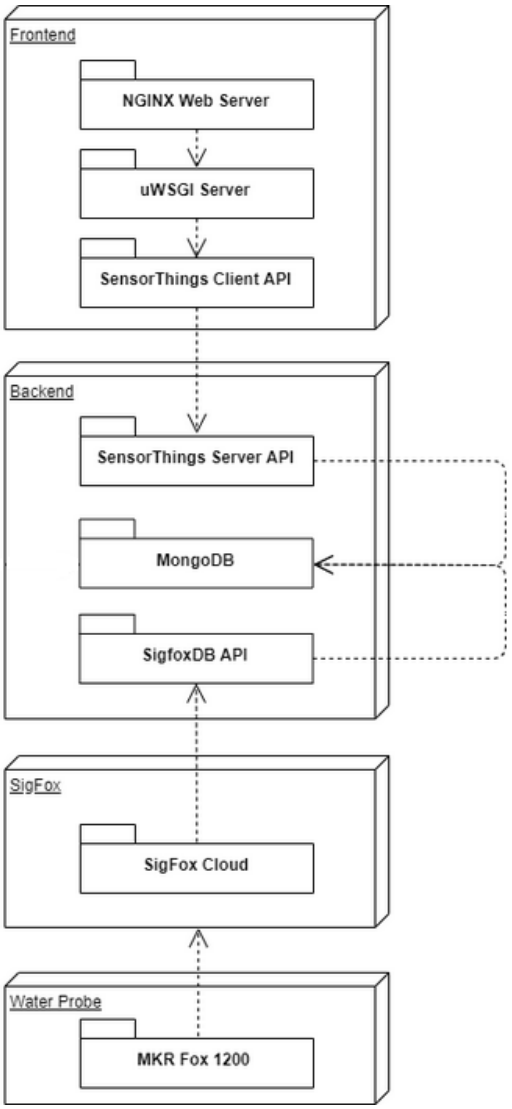


Figure 3.1.2: Package diagram of our P7 system [26].

How the components work and communicate with each other:

- The probes are programmed to store average values for each of the properties they can measure about water.
- The probes take 16 measurements of each property equally distributed over 15 minutes. After that, they send the measurement for each property, that deviates the most from the average, as one message to the Sigfox Cloud via the Sigfox protocol.
- The Sigfox Cloud and the two APIs can always receive new requests and handle incoming requests sequentially.
- When the Sigfox Cloud receives data from a probe it initiates a callback that sends the data to SigfoxDB API.
- SigfoxDB API waits for callbacks from the Sigfox Cloud and sends the data received from the Sigfox Cloud to the database.

- The database acts as storage for the data measured by the probes. This data is timestamped as well. When the database receives a request containing data from SigfoxDB API it inserts the data into the database under the specific probe. If this is the first time data is received from a specific probe, a new entry is made in the database. When the database receives a request from SensorThings API, requesting data from a specific probe, the database will send the requested data back to SensorThings API.
- SensorThings API waits for requests from the website requesting data from a specific probe. SensorThings API sends this request to the database and after getting a response from the database, SensorThings API will send back the requested data to the website. However, if the request to the database was unsuccessful, some default data (error data) will be sent to the website instead.
- The website allows the user to request and view data measured by the probes. The website will receive a request from a user for a specific probe and forward that request to SensorThings API. The website will then send back the data received from SensorThings API to the user.
- The user is an arbitrary user of the system that makes requests for data from an arbitrary probe through the website and awaits a response from the website, otherwise the user is idle.
- It is possible to scale the APIs, the database, and the website by having multiple instances of these running at once.

Regarding the timing aspects of the system, we have the following information:

- A probe will take measurements for 15 minutes and then send the measurements to the Sigfox Cloud. It takes at most 1 second for a probe to send the measurements.
- We do not have any performance-related data on the Sigfox Cloud.
- We performed stress tests on our APIs and website and experienced a worst-case scenario of a response time of 1 second. This means that the APIs and website will take at most 1 second to process a request.

3.1.1 What We Want To Verify

The purpose of modelling a system in UPPAAL and UPPAAL SMC is to be able to guarantee that the modelled system satisfies certain properties. These properties include safety, reachability, and liveness properties. Based on what we consider important about our P7 system's functionality, we have formulated the following properties that we want to verify:

1. Regarding probes:
   (a) Are the probes always able to send data to the database?
   (b) Is the data correctly inserted?
   (c) How much time does it take to insert the data in the worst case?
2. Regarding users:
   (a) Are users always able to read the data from the database?
   (b) Is the data read correctly?
   (c) How much time does it take to respond to a user's request in the worst case?
3. What influence does incorporation of stochastic behavior (e.g. package loss) have?

3.2 UPPAAL Model

The UPPAAL model of the system is constructed to allow us to assert whether our system fulfills the properties described in section 3.1.1. Therefore, the model has to support that a probe can insert data into a database and that a user can query the database for a given probe. In addition, the model should allow to assert properties about how much time certain processes need. In our model, we have decided that the unit of time is seconds. It takes 15 minutes for the `Probe` templates to make a measurement. The other time constraints in the model are overestimates based on the stress tests we performed on our P7 system. A transition will always take exactly the worst-case time. We do not consider time ranges, because it increases the state-space and we are only interested in verifying worst-case guarantees in UPPAAL. Contrarily, we utilize time ranges in the UPPAAL SMC model (cf. section 3.6). This is to approximate a realistic performance of the model. Additionally, UPPAAL SMC verifies queries based on statistical evidence from simulations. As such, the size of the state-space is less of an issue for UPPAAL SMC.

We have implemented value passing by utilizing global placeholder variables that are accessed during binary synchronization. Another way of implementing value passing is by utilizing the channels. An example of this can be seen in section 3.3.

3.2.1 Template Description

When we began to model our P7 system we first used the package diagram seen in Fig. 3.1.2. This diagram helped us decide upon which templates the model should contain. Our system is clearly split into certain components (such as APIs, database, and probes) which made it somewhat straightforward to choose an initial separation of templates for the model. Having an idea of what we wanted to verify, as stated in section 3.1.1, also meant that we could shape the model to fit this purpose. Because we have previous experience with the UPPAAL tool we had a good understanding of how to quickly test our initial model ideas. This section describes the individual templates of our model.

Because the Sigfox Cloud component (seen in Fig. 3.1.2) is a third-party part of the system that we do not control, we find it hard to reasonably make assumptions about constraints upon it. The Sigfox Cloud component could be modeled such that we could reason about what the effect could possibly be and what constraints need to be satisfied in order for it to not be a bottleneck. We do however assume that it is fully scalable, as it is third-party, which in turn makes it uninteresting for us to model. Furthermore, we do not believe it is important to model, because it only serves to relay messages between the probes and our API. Therefore, potential message loss or time constraints on the Sigfox Cloud could easily be implemented in the other components of our system. We have therefore chosen to create an abstraction of the subsystem, consisting of the Probe, Sigfox Cloud, and SigfoxDB API, by removing the intermediate Sigfox Cloud. As a result, there will be a synchronization from the Probe directly to the SigfoxDB API, where the time constraints from the Sigfox Cloud are added.

The Probe Template

The `Probe` template is one of the templates (along with the `User` template) that initiates the chain of communication between the templates in the model. The probe lies at the "bottom" end of the system (cf. Fig. 3.1.2). The template of the probe (cf. Fig. 3.2.1) has three locations: `Idle`, `Sending`, and `NotifyObserver`, with `Idle` being the initial location. From the `Idle` location the template can take a transition to the `Sending` location. From `Sending` the template goes to `NotifyObserver`, which is

a committed location, before going back to `Idle`. The template has a local clock (called `probeClock`) and a function called `measure()`. The function is meant to model the behavior of the probe measuring some property. However, in this model, the `measure()` function always returns 1 since we abstract away from the measuring aspect of the probe. We do this because we are mostly interested in verifying the communication between the templates. Therefore, the probe can only be either idle or sending a message. The reason for a separate state for sending is to be able to model that the probes take 1 second to send and that they only can do it after having measured. The third state `NotifyObserver` exists as we use observer templates to aid verification. The `probeClock` is used to model the different timing aspects of the probe. This is achieved by implementing guards and invariants over the `probeClock` in the template.

The `Probe` is in the `Idle` location for 900 time units before having to take the transition to the `Sending` location. This is to model our P7 system's time constraints where measurements are sent once around every 15 minutes (cf. section 3.1). In the `Sending` location the template has to delay 1 second before it takes the transition to `NotifyObserver`. This is to model that sending the measurement takes 1 second (as an upper bound). Furthermore for the template to take the transition from `Sending` it has to broadcast over the `probeToDBAPI` broadcast channel. Here we use a broadcast channel since the Sigfox protocol is based on radio broadcast technology, which does not guarantee that a message is received. When the transition from `Sending` to `NotifyObserver` becomes available and fires, we pass the probe's ID along with its measurement by assigning their values to the global variables `id1` and `mes`, respectively. Afterwards the template enters the `NotifyObserver` location. Since this location is committed, the model cannot delay and has to take the transition to `Idle` (unless another location in the current state is also committed, in which case it can choose between exiting either location). The location is committed to make sure that sending and notifying the observer is done as an atomic operation. When the transition to `Idle` fires, the template initiates synchronization on the `ObserveProbe` channel corresponding to its own `ProbeID`. This channel is a broadcast channel so that the probe template does not have to wait for the observer template to be ready for the synchronization. After this synchronization happens, the `Probe` template is in the `Idle` location again and the behavior loops.

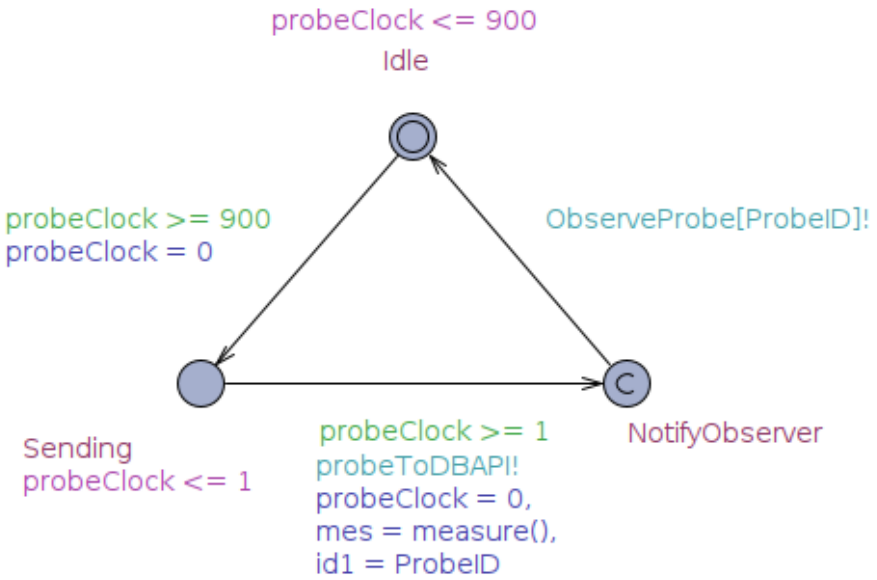


Figure 3.2.1: The `Probe` template.

The DBAPI Template

The `DBAPI` template can be seen in Fig. 3.2.2. The template consists of two locations, the initial location `Idle`, and a `Sending` location. Each location has a transition leading to itself. The purpose of these transitions is to allow the model to listen for broadcast synchronizations from probes at all times so that no probe message is lost due to the `DBAPI` template being unable to receive it (cf. section 3.1). We use a queue in the template to store probe messages. When the template is in the `Idle` location and has at least one probe message in its queue, it takes the transition to `Sending` due to the `CanSend` channel being urgent. The transition is created this way to force the model to move to the `Sending` location if there is something to send to the `DB` template so that it cannot delay in `Idle` forever. While in the `Sending` location the template can still collect probe messages, but after at most 1 second it has to take the transition back to `Idle`. During this transition, we dequeue the element at the front of the queue and send that to the database template.

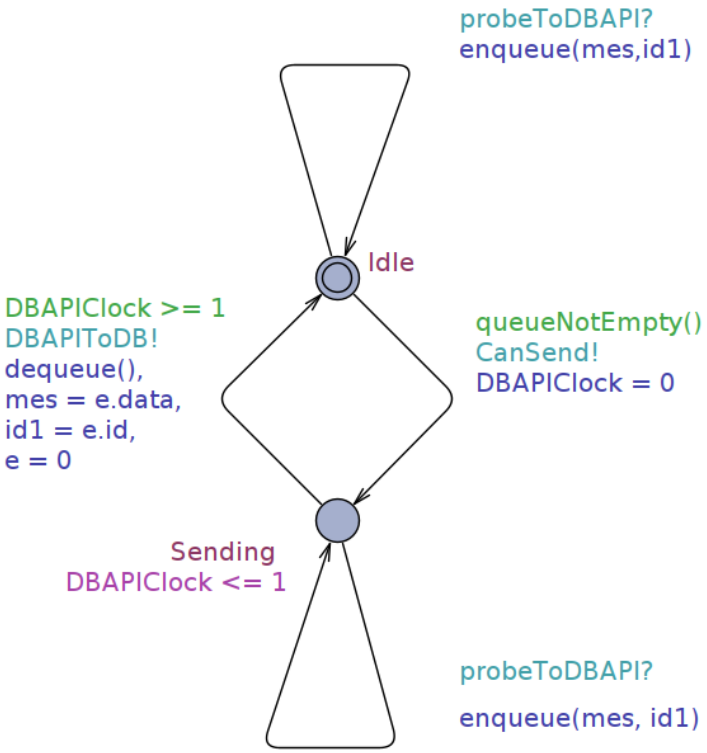


Figure 3.2.2: The `DBAPI` template.

The queue allows us to store a fixed number of messages from probes. This is necessary to avoid that a message is lost, due to the `DBAPI` being "busy" with handling another message, a scenario that is increasingly likely to occur as the number of probes is increased. The queue is implemented as a bounded array and two integer values, front and back. The size of the queue is set to the number of probes. The `DBAPI` can dequeue a message each second, and the probes send each 900 time units, therefore, the `DBAPI` can manage to dequeue 900 messages before the probes send another message. This means that as long as the number of probes does not exceed 900, the current queue size is sufficient.

The DB Template

The `DB` template can be in either of three locations as seen in Fig. 3.2.3. The locations being `Idle`, `SendingResponseToSensorThingsAPI`, and `NotifyObserver`. The `DB` template takes one of two transitions based on incoming synchronizations from other templates. These synchronizations are initiated by either the `SensorThingsAPI` or `DBAPI` templates. Incoming synchronizations from the `Sensor ThingsAPI` template allow the template to take the transition to the `SendingResponseToSensor ThingsAPI` location. This will save the ID of the `SensorthingsAPI` locally. The ID of the probe is used to read from the database in the function `readDB()`, and the `sensorthingsAPIID` is used to send the data back to the correct `SensorThingsAPI`. The action of sending the data back is represented by the transition from the `SendingResponseToSensorThingsAPI` location that initiates synchronization on the `DBToSensorThingsAPI` channel. The data being sent is stored in the global `mes` variable. The second transition from the `Idle` location represents data from a probe being inserted into the database. The location `NotifyObserver` is used for an observer template, that we use to verify properties of the model.

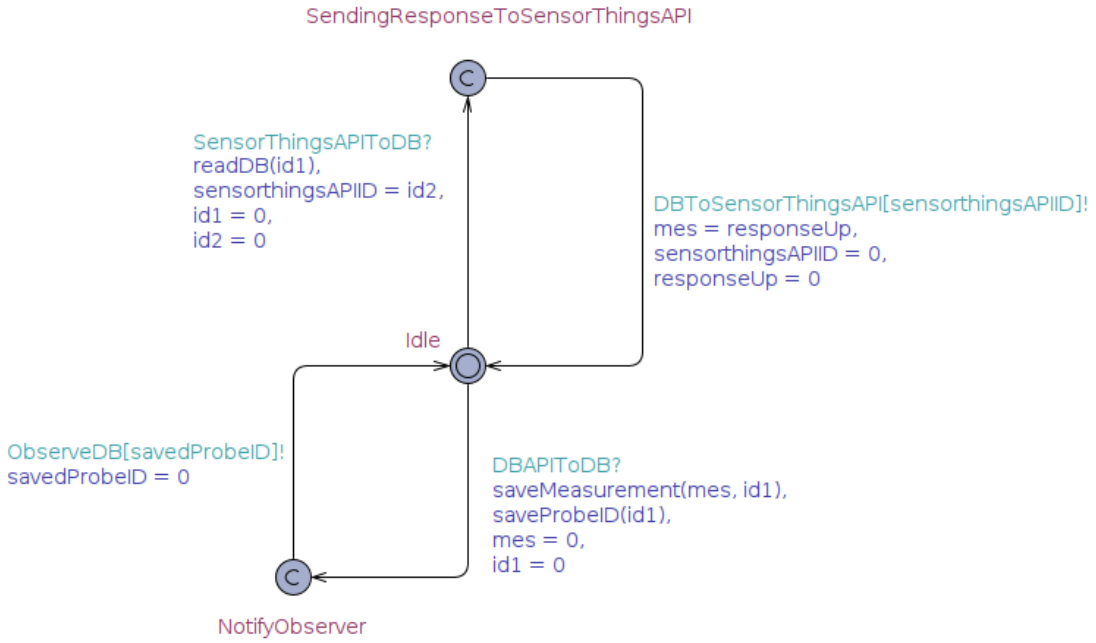


Figure 3.2.3: Template of the `DB`.

The `DB` template has no internal time constraints. However, it is bounded by the time constraints of the templates that it synchronizes with. Each of the APIs take 1 second to process a request. We can see this 1 second delay as both being for the APIs processing the request and for the database to read and write data.

The SensorThingsAPI Template

The `SensorThingsAPI` template is seen in Fig. 3.2.4 and consists of four locations, namely, `Idle`, `Sending`, `WaitingForResponse`, and `SendingResponse`. From the initial `Idle` location the template is ready to receive a synchronization from the `Website` template requesting data from a specific probe. The synchronization happens over the channel `WebsiteToSensorThingsAPI` and sends the template to the `Sending` location. When synchronizing, it reads the global variables and saves the ID of the

`Probe` and the ID of the `Website` that initiated the synchronization with the purpose of, respectively, sending the ID of the `Probe` to the `DB`, and knowing which `Website` to respond to. The transition to the `WaitingForResponse` location initiates synchronization with the `Database` template and passes on the ID of the current `SensorThingsAPI` template instance and the ID of the `Probe`, through the global variables. The invariant on the `Sending` location along with the guard on the transition is used to model that sending a request to the `DB` takes at most 1 second. The template then waits for a response from the `DB` template in terms of synchronizing over the `DBToSensorThingsAPI` channel. When the synchronization happens, it saves the data received from the `DB` template and goes to the `SendingResponse` location. After at most 1 second, the template initiates synchronization to the same `Website` template as it received a request from, passing the data from the `DB` template to it through the global variable `mes`, and the `SensorThingsAPI` template is now ready to receive another request.

Figure 3.2.4: Template of the `SensorThingsAPI`.

The Website Template

The `Website` template is used to "display" the data that a `User` requests. As such the template waits until some data is requested by a `User` and then it relays this request to the `SensorThingsAPI` template. Afterward, it waits for the response to the request. The template consists of four locations: `Idle`, `Sending`, `WaitingForResponse`, and `SendingResponse` with `Idle` being the initial location. Furthermore, the template has a clock and three functions for locally saving different values. These functions are `saveProbeID(id_Probe ProbeID)`, `saveSensorThingsAPIID(id_SensorThingsAPI sensorThingsAPIID)`, and `saveResponse(T_data response)`.

The template can be seen in Fig. 3.2.5. The template starts in the `Idle` location where it waits for the `User` template to initiate synchronization on the `UserToWebsite` channel. When this happens, the transition from `Idle` to `Sending` fires. During this transition the template saves the `probeID` for which the `User` requested data along with the `userID`. The `userID` is used to eventually give the response to the correct `User`. Furthermore, the clock and global variables `id1` and `id2` are set to zero. When the `Website` template enters the `Sending` location it must wait 1 second before synchronizing with the `SensorThingsAPI` template over the `WebsiteToSensorThingsAPI` channel. If the template can synchronize after at most 1 second then the transition to the `WaitingForResponse` location fires. During this transition the saved `probeID` and `websiteID` are passed to the global variables `id1` and `id2`, respectively. In the `WaitingForResponse` location the template waits until it can synchronize on the `SensorThingsToWebsite` channel. When this happens, the transition from `WaitingForResponse`

to `SendingResponse` fires. During this transition the template calls the `saveResponse()` function to save the response. Lastly, in the `SendingResponse` location the template has to wait for at most 1 second and then synchronize on the `WebsiteToUser` channel with the `User` who made the initial request to send the response to the `User`. This fires the transition back to `Idle`. During this transition the global variable `mes` is set to the saved response to allow the `User` to access it.

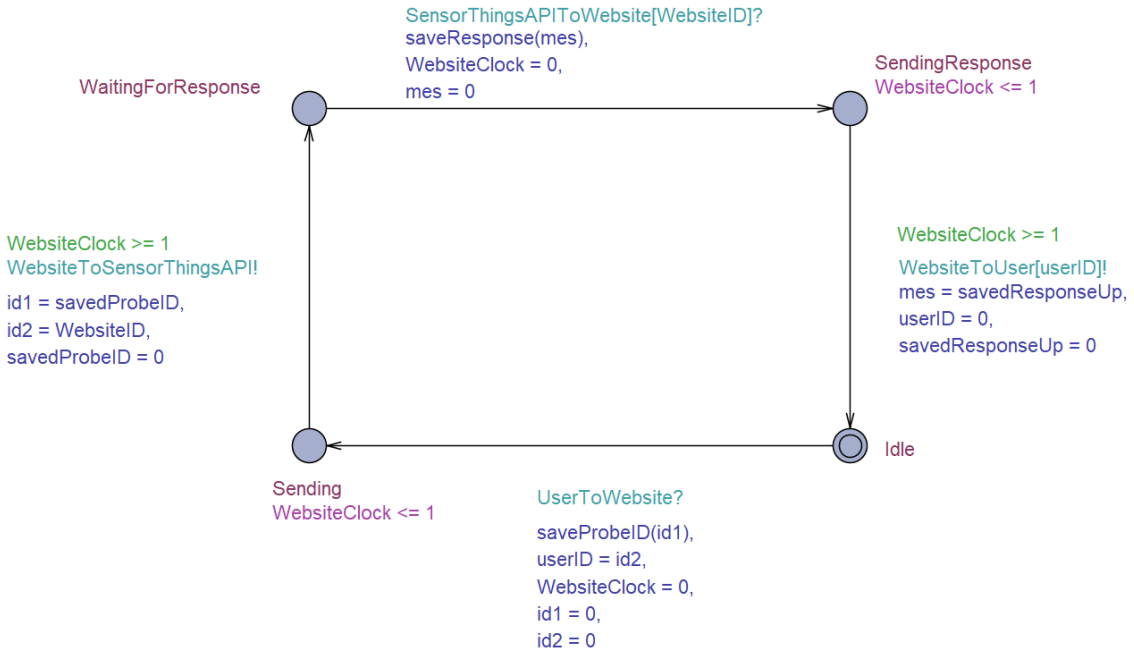


Figure 3.2.5: Template of the `Website`.

The User Template

The `User` template consists of four locations, namely, `Idle`, `ObservedRequest`, `WaitingFor Response`, and `ObservedResponse`, as seen in Fig. 3.2.6. The initial location is the `Idle` location. The `User` can initiate a synchronization to the `Website` through the `UserToWebsite` channel, but only when the guard `UserClock` $\geq$ `1` is satisfied. The ID of the probe that a `User` wants information about is stored in `id1` and the `User`'s own ID is stored in `id2`. Then follows a committed location, `ObservedRequest`, which is used only for an observer template. After initiating a synchronization for the observer, the `User` will wait for a response in `WaitingForResponse` until the `WebsiteToUser` synchronization is initiated. The location `ObservedResponse` is only used to verify properties via an observer template.

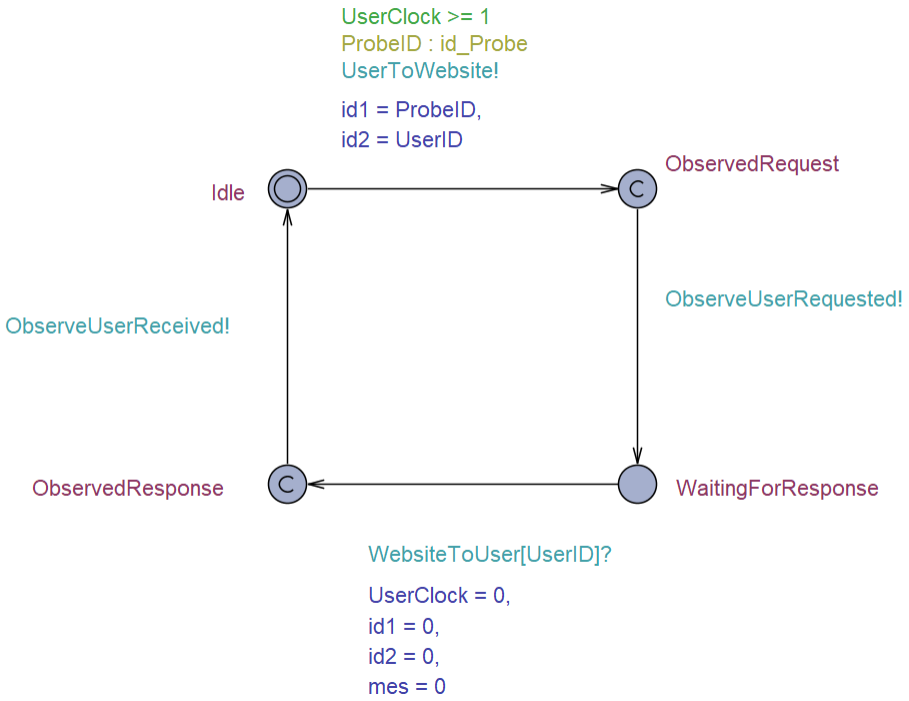


Figure 3.2.6: Template of a `User`

3.3 Value Passing by Channels

When we started modelling our P7 project we tried two different strategies of value passing. One of them is using global variables and the other is using arrays of channels. The model that uses global variables is described in section 3.2. This section will describe the `User` template of the model that uses value passing via channels. We will not be describing the model in its entirety as the structure of the model is almost identical to the one described in section 3.2.

The idea behind value passing via channels is to make (potentially multidimensional) arrays of channels so that one channel corresponds to one value that can be sent. Doing value passing this way drastically increases the number of channels that are needed in the model. This is especially true if the number of different values that can be sent between templates is high. However, this strategy eliminates the need for global variables altogether and, therefore, also the need for resetting these. Through comparison of the two models, we have found that there is very little difference in the time it takes to verify the queries, and presumably the size of the state-space.

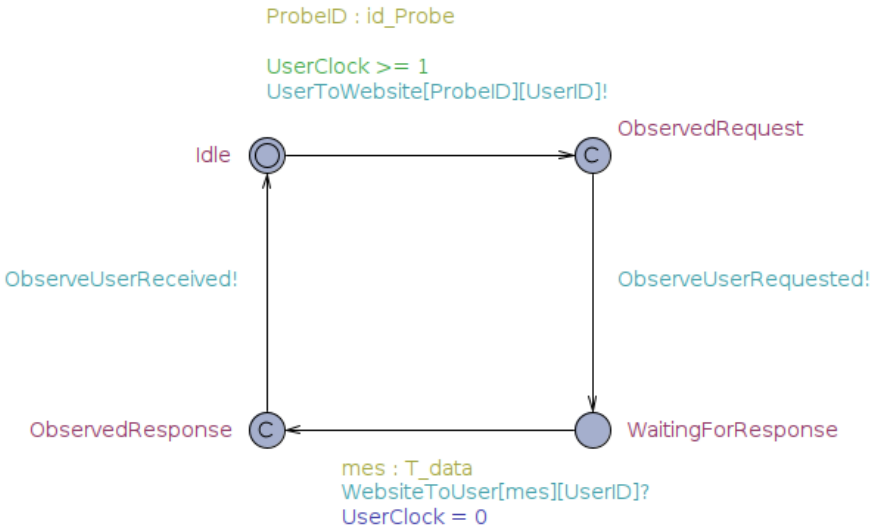


Figure 3.3.1: The `User` template using channels for value passing.

Figure 3.3.1 shows the `User` template. This template is almost identical to the one shown in Fig. 3.2.6. However, the channels are different since this template uses value passing via channels. The difference is that when a user makes a request in `Idle`, the template picks any `probeID`, and synchronizes over the `UserToWebsite` channel that corresponds exactly to the `probeID` and the user's own ID. In this way, the `Website` being synchronized with knows which `probeID` is being requested by the user, and which user to respond to, based on the channel used for the synchronization. When waiting for a response from the `Website`, the `User` listens on the channel `WebsiteToUser` that corresponds to its own `userID`. However, since `WebsiteToUser` is also used to transfer the data, it has to also listen to the channel that corresponds to a given `mes` value. Since the `User` does not know the data it is supposed to receive before it receives it, it can select any given response when waiting for synchronization. Therefore, it listens for synchronization on `WebsiteToUser` for its `userID` and for any `mes`. This is denoted by the select statement on the transition from `WaitingForResponse` to `Idle` that states that it can choose any `mes` given that it is part of the custom data type `T_data`.

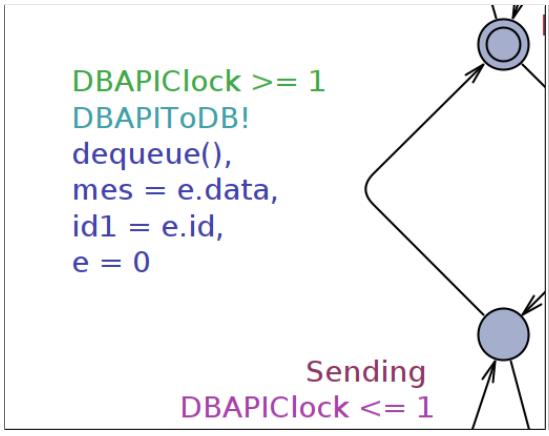


Figure 3.3.2: The `DBAPI` template using global variables for value passing.

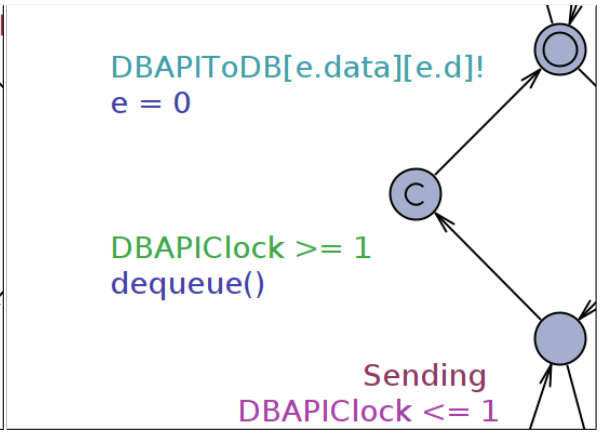


Figure 3.3.3: The `DBAPI` template using channels for value passing.

In UPPAAL the synchronization of a transition is done before the update. In case two templates synchronize with each other, the sender transition's update is executed first, and the receiving transition's update is executed afterwards. In the global variable version of the system (seen in Fig. 3.3.2), there is only one channel, and the global variables (`mes` and `id1`) are set by the sender, and afterwards, read, and reset by the receiver to decrease the size of the state-space. The `dequeue` function assigns the element being dequeued to the variable `e` . This trick cannot be used as easily in the local variables version (seen in Fig. 3.3.3). This is because in this version there is an array of channels, and the indices used to find the correct channel are local variables that need to be set before synchronization can happen. Since a transition's update always happens after its synchronization, we need two distinct transitions that will happen in sequence rather than one. We will also split the update such that the local variables are set during the first transition, and the synchronization happens in the second transition.

3.4 Verification and Results for UPPAAL

As mentioned in section 3.1.1, we modeled the system to be able to verify a list of properties about our system. For verifying the properties we utilize observer templates and UPPAAL queries. The syntax of UPPAAL queries uses a subset of CTL.

3.4.1 Observers

To help with the verification process we have created two observer templates. These are not part of the modeled system, but can be seen as "observing" the model. We use the observers to check timing aspects of the model, debugging, and to check the synchronization sequences in the model. The two observers are very similar in their structure but are used to observe two different parts of the model. They have four locations each: `Idle` , `Waiting` , `Success` , and `BAD` . The idea behind these observers is that they are idle until a specific synchronization happens in one of the templates in the actual model. The given observer resets their local clock and moves to the `Waiting` location. From this location, the observer waits until another specific synchronization in the model occurs. If the second synchronization occurs within the threshold, `BAD_TIME` , then the observer template moves to the `Success` location. However, if the time it takes for the second synchronization to occur is longer than the given threshold then the observer is forced into the `BAD` location. Using these observers it is easy to verify if the time to go from one point in the model to another takes too long by simply writing a reachability query that asks if there is any path that eventually leads to the `BAD` location.

The first observer, seen in Fig. 3.4.1, monitors the time it takes for the `Probe` to send data and until the data is in the database. The second observer, seen in Fig. 3.4.2, is made to monitor the time from a `User` sending a request until the same `User` gets a response. Thus, the two observers are used to verify the first and second property described in section 3.1.1, except aspect 1.b and 2.b.

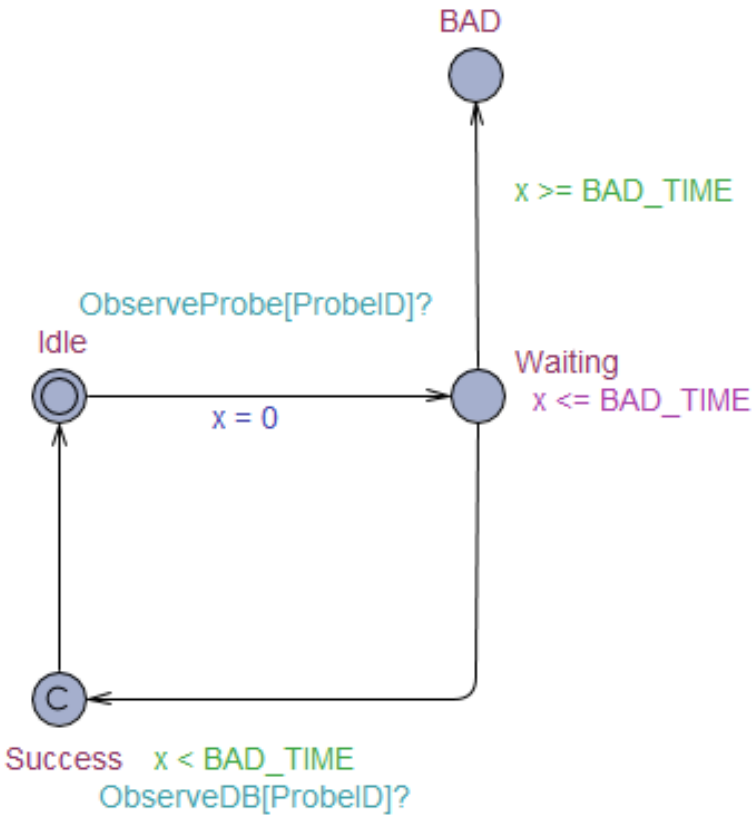


Figure 3.4.1: The `Probe` to `Database` observer.

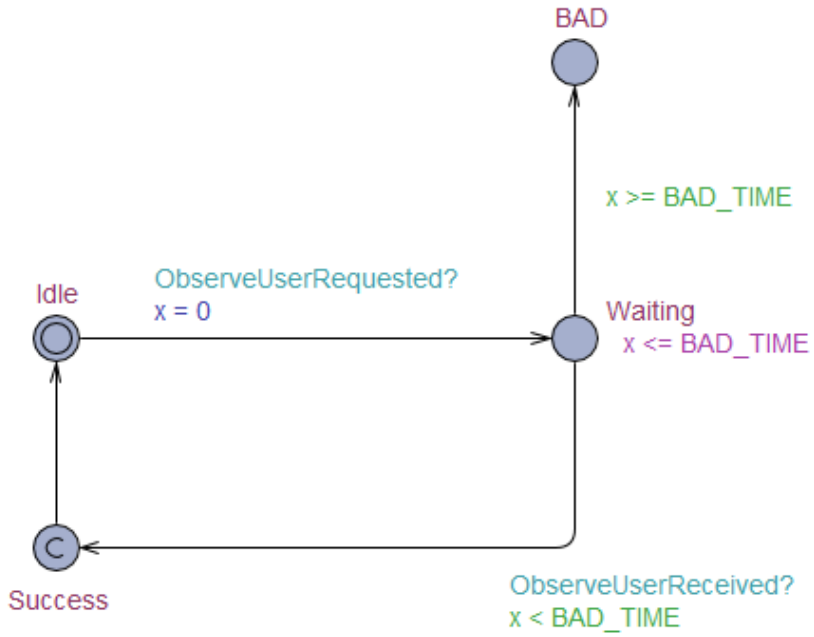


Figure 3.4.2: The `User` observer.

3.4.2 Queries

This section describes the queries we have made using the verification tool in UPPAAL. All of the queries were made with the model containing one instance of each type of template. The queries are used along with the observer templates to verify the properties mentioned in section 3.1.1.

`ObserverUser.Waiting --> ObserverUser.Success`
This query verifies that when the UserObserver goes to the `Waiting` location, then the `ObserverUser` will eventually enter the `Success` location. It describes a liveness property because it states that if there exists a path in which the first part of the state formula is satisfied, then the second part is eventually also satisfied. The property describes that when a user has requested data, the user will always receive a response within 5 time units(i.e. BAD_TIME). The `ObserverUser` will only enter the `Success` location if the templates (i.e. the `Website`, `SensorThingsAPI`, and `DB` template) can process the `User`'s request within `BAD_TIME`. Therefore, this asserts aspect 2c of the properties in section 3.1.1. We also tried to verify `A <> ObserverUser.Waiting`, but this formula is not satisfied because the user never has to make a request. However, if the user is forced to make a request at some point in time, the property is satisfied. This indicates that the user is always able to make a request but can decide not to in the current model. This asserts aspect 2a of the properties in section 3.1.1.

`A[] not ObserverUser.BAD`
The query verifies that the `ObserverUser` template can never reach the `BAD` location. It describes a safety property since it checks that all reachable states satisfy the state formula, i.e. "something bad will never happen". If the `ObserverUser` template would reach the `BAD` location, it would mean that the other templates do not always process the `User`'s request in less than `BAD_TIME` (i.e. 5 time units). This asserts aspect 2c in section 3.1.1, but also that the required set of synchronization from the `User` template to the `DB` template and back up to the `User` template is always possible to perform.

`Probe(0).Sending --> DB(0).data[0] == 1`
This query verifies that if the `Probe(0)` template instance ever comes to the `Sending` location, then the correct data will eventually be in the `Database` template, at the correct index. It describes a liveness property since it states that if the first state formula is satisfied, eventually, the second state formula is too. The query is specific to the arbitrarily chosen `Probe(0)`, but should also apply to all instances of `Probe`. This query assert aspect 1b of the properties in section 3.1.1.

`A[] not exists(id : id_Probe) ObserverProbeToDB(id).BAD`
This query verifies that none of the `ObserverProbeToDB` template instances can reach the `BAD` location in any trace. This is a safety property. It checks that passing data from the `Probe` template to the `Database` template cannot take more than `BAD_TIME`, i.e. 1000 time units in this case. This query asserts aspect 1c of the properties found in section 3.1.1.

`A <> ObserverProbeToDB(0).Success`
This query verifies that in all traces the `ObserverProbeToDB` can eventually reach the `Success` location. This means that passing the data from the `Probe` template to the `Database` template can always happen in less than 1000 time units. This query asserts aspect 1a of the properties found in section 3.1.1.

`A[] not deadlock`
This query verifies that on all paths at every step of the path, the model can always make a valid transition or delay into a valid state. This is also a safety property. The motivation behind this property is not to verify that the model cannot reach a deadlock state. Instead, we use it as a debug property to verify that e.g. a safety property does not succeed because of an inevitable deadlock. Note that the same deduction can not be done if the not deadlock property fails.

3.5 UPPAAL Reflections

In this section, we reflect on using UPPAAL. We will also give some tips on what to do and not do when using UPPAAL based on our personal experiences with the tool.

3.5.1 Pre-Modelling Phase

The first step before modeling should be to read the UPPAAL tutorial [29]. The tutorial describes among other things how to model atomicity, synchronization, and ways of making the state-space of the models smaller to mitigate the state-space explosion problem.

Furthermore, it is important to realize that one system can be modeled in many different ways, and these models have individual pros and cons. We found that it is difficult to make a model that is representative, easy, and quick to verify. For this reason, it makes sense to decide what needs to be verified about the system and let that determine how the system should be modeled. In our case, we found out that we wanted to verify the correctness and performance of the system. Therefore, it made sense for us to create a model that allows us to verify that the data could be sent from the probe to the user before a certain time constraint.

During verification, it is the model that is being verified and the verification is only useful as long as the modeling gap is sufficiently small. However, in our model, we initially modeled the probe as making 16 measurements during a 15-minute cycle much like the real system. We later found out that modeling the probe that way made the verification take much longer, as it significantly increased the number of states in the model. For this reason, we decided to instead have a single transition that serves as an abstraction of the measuring. Even though this increases the modeling gap, we did not see it as an issue because we are mostly interested in verifying the communication between the components.

3.5.2 Reducing State-Space

When performing model checking, one of the biggest issues that is ever-present is the state-space explosion problem [29, p. 31]. While modeling we quickly encountered this problem ourselves due to the complexity of our model. Initially, we tried to model the system as detailed as possible. This included data transfers between components such as all HTTP responses. In addition, our `Probe` template was first modeled with an incrementing integer that could count between 0 and 16. This, along with the fact that we never reset variables, lead to the state-space becoming too large for formal verification, very quickly. Verifying certain properties with the model having more than one probe instance was impossible on our hardware. Intuitively, this makes sense as having a variable per probe that increments between 0 and 16 leads to 17 different states in itself. Having two probes leads to 17^2 states, and so on. This is, of course, only looking at this one variable in one template of the model of the entire system. However, as much as this makes sense in hindsight, it was not intuitive for us when we began to model.

To combat the state-space explosion problem we encourage the number of variables in templates to be kept at a minimum. Furthermore, these variables should have a clearly defined range to limit their potential values and should be reset when not used. This is so that the model does not contain unnecessary states due to a variable value that does not influence the behavior in that state. Lastly, it is important to choose an appropriate level of abstraction for each template. This is so that not everything in the model is of the finest detail, but only if such detail provides information necessary for verifying the properties of interest.

3.5.3 Using Observers to Help Verification

When we first started to verify different properties about our models in UPPAAL we did so using the built-in verification tool. However, after some time we found that we had issues expressing some of the queries that we wanted to verify. To combat this we began to employ the use of observer templates. The observers that we use for our model are described in section 3.4.1. The observers can, for example, be used to formulate queries in the verification tool in UPPAAL or can contain failure states that represent that something happened in the model that should not occur (e.g. too much time passed). We found that using observers allowed us to verify properties that we had trouble expressing only using the subset of CTL that UPPAAL supports.

3.5.4 Debugging a UPPAAL Model

Debugging of a UPPAAL model refers to findings errors that are related to the implementation of the model, and not errors that are related to whether the system is modeled correctly.

Three ways we found useful for debugging UPPAAL models are:

- Using the simulator to see which transitions can be fired in the different states of the model.
- Analyzing the proof trace given by the verificator, proving that a certain property/query does not hold.
- Having properties/queries which work as unit tests. That is, not necessarily interesting properties, but properties that, if they do not hold, would indicate an error in the model's behavior.

3.6 UPPAAL SMC Model

In this section, we introduce the UPPAAL SMC model of our P7 system. The purpose of modelling the system in UPPAAL SMC is to be able to verify property 3 mentioned in section 3.1.1. That is, to model stochastic behavior and verify properties about a model that has such behavior. It makes more sense to verify property 3 in UPPAAL SMC compared to standard UPPAAL, as it allows for modelling stochastic behavior such as package loss.

UPPAAL SMC does not do an exhaustive exploration of the state-space of the model. Rather, it runs a certain number of simulations over the model when checking if certain properties hold. This also means that compared to standard UPPAAL verification, UPPAAL SMC does not give discrete answers. It instead gives estimates and probabilities based on the simulations over the model [30]. The UPPAAL SMC model is created using UPPAAL Stratego version 4.1.20-7. We have made the same estimates about time for the UPPAAL SMC model as for the UPPAAL model. Lastly, we have decided to only create a model using global variables to do value passing rather than also making a model using channel-based value passing as we did for UPPAAL.

3.6.1 Model Changes

This section describes the changes made to the UPPAAL model as part of converting it into a SMC model. The SMC model of the system is similar to the one described in section 3.2. The changes we have made are:

- Allow templates to take transitions within time ranges rather than forcing them to wait until the worst-case time.
- Modifications to the probe template so that we can model loss of messages.
- Converted channels from binary synchronization (binary communication) to broadcast channels. This is because binary synchronization is not possible in UPPAAL SMC.

The probe in UPPAAL SMC (seen in Fig. 3.6.1) is different from the probe in standard UPPAAL (seen in Fig. 3.2.1). We have chosen to remove all clock guards on transitions that initiate synchronization. This is to model that outgoing communication may range between 0 and the upper bound specified in the invariant. An example of this change can be seen in the transition from the `Sending` location to the `NotifyObserver` location which now can take from 0 to 1 time units. UPPAAL SMC will choose a random delay in that range during the simulations. This corresponds to how the system behaves in practice and makes the simulations and the model's performance more realistic.

We have chosen to give some transitions a probabilistic weight. This models the fact that, in our P7 system, the probe communicates with the Sigfox Cloud via a wireless connection, which has a risk of failing. This means, that packages can be lost in the communication from the `Probe` to the `DBAPI`. In our model, we assume that a package has a 99% chance of being delivered, and 1% chance of not being delivered. In both cases the `Probe` advances to the `HaveSent` location, from whence it must advance to the `Idle` location without delay, because the `HaveSent` location is an urgent location. Otherwise, the `Probe` template is the same as described in section 3.2.1.

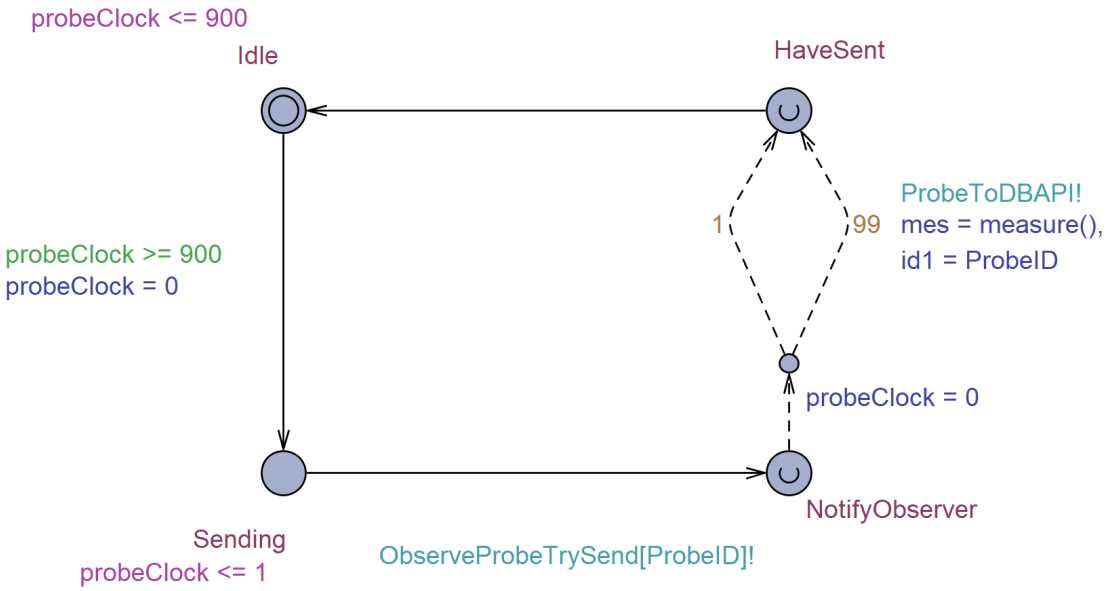


Figure 3.6.1: Model of the `Probe` in UPPAAL SMC.

As mentioned in [30, pp 2-3], all channels must be broadcast channels. This means that binary synchronization is not directly supported in UPPAAL SMC as it is in standard UPPAAL. A broadcast synchronization

happens between one sender and any number of receivers, which could be 0, whereas a binary synchronization happens between one sender and exactly one receiver. [30, section 7.1] describes how to implement binary synchronizations based on broadcast synchronization in UPPAAL SMC. For each template that is capable of receiving a broadcast synchronization, we use global variables to keep track of their current locations. Then, the transitions that initiate broadcast synchronization are guarded wrt. the location of the receiver. This asserts that the broadcast synchronization can only be initiated when the receiver is ready, thus effectively simulating binary synchronization. In case multiple templates are capable of receiving the same broadcast synchronization at the same time (this usually happens if there are multiple instances of the same template type), we will convert the global variable representing the template's location and the channel responsible for the communication, into arrays.

3.7 Verification and Results for UPPAAL SMC

This section will cover how we use observer templates and UPPAAL SMC queries to verify property 3 mentioned in section 3.1.1.

3.7.1 Observers

The purpose of the observers is the same as described in section 3.4.1, namely, to aid in verifying properties of the model. To do so we have made the `ProbeDBObserver` and the `UserObserver` templates (seen in Fig. 3.7.1 and Fig. 3.7.2, respectively). Their purpose is to help verify how many times a probe fails to send a message (package loss) and the response time for a user's request, respectively.

A `ProbeDBObserver` is instantiated for each `Probe`. It is designed to observe each time the `Probe` tries to send and each time the `DB` successfully stores a measurement from the specific `Probe`. An `ObserveProbeTrySend` transition must be followed by an `ObserveDB` transition. If n `ObserveProbeTrySend` transitions are made in sequence followed by one `ObserveDB` transition, then we know that the `Probe` failed $n-1$ times in trying to send. This is always true since the time it takes for passing and inserting a measurement into the `DB` is less than the time it takes to make a single measurement. `totalFails` accounts for the `Probe's` total number of failures. `fails` accounts for the number of fails in succession.

The `UserObserver` template is similar to the `UserObserver` template from the UPPAAL model (cf. section 3.4.1). It observes the `User` requesting data and awaits the observation of the user receiving the response. The clock `x` is reset upon request and it reflects the time it takes from request to response.

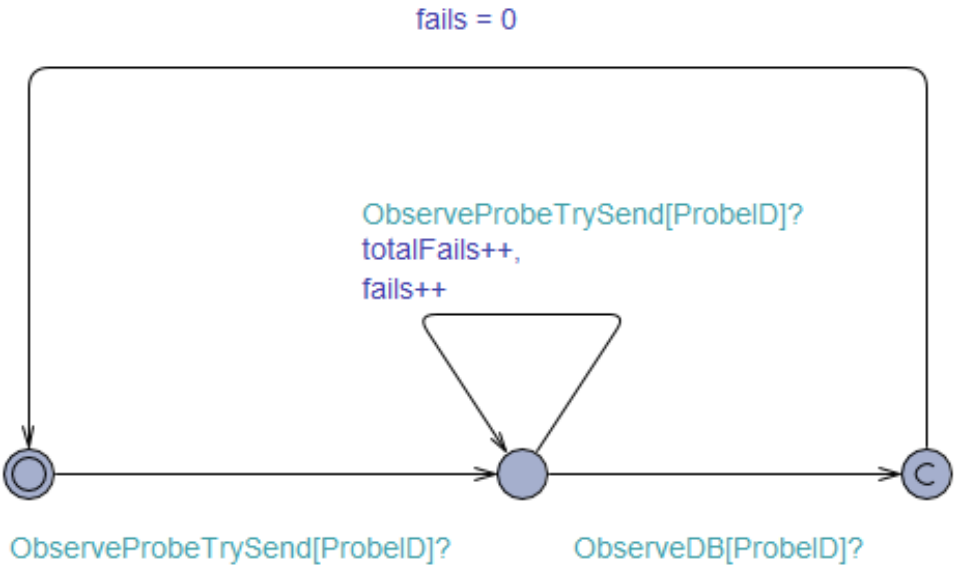


Figure 3.7.1: The `ProbeDBObserver` template.

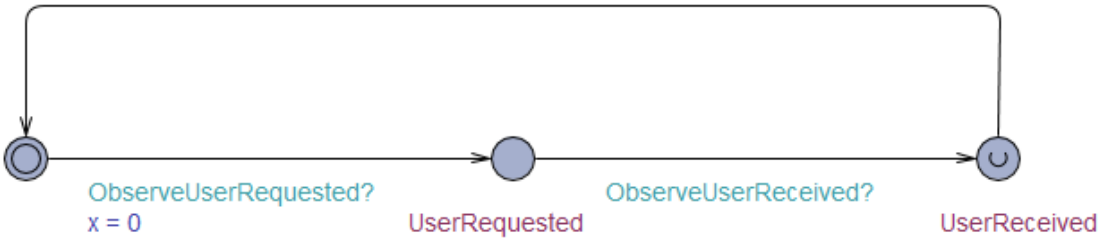


Figure 3.7.2: The `UserObserver` template.

3.7.2 Queries

This section covers the queries we made using the verification tool in UPPAAL SMC.

`Pr[<= 10](<> UserObserver.UserReceived && UserObserver.x <= 5)`
This query uses the `UserObserver` template to give an indication of the response time for users' requests for data in the database. The query estimates the probability of reaching a state, within 10 time units, where the `UserObserver` template is in location `UserReceived` and clock `x` is less than or equal to 5 time units. The `UserObserver` template initializes the clock `x` to zero when the user requests data from the database via the website. The `UserReceived` location indicates that the user has received a response containing the data. This is simulated an arbitrary number of times until it reaches a confidence of 0.95. The probability of this query being satisfied is in the range [0.901855, 1] with confidence 0.95. This means that UPPAAL SMC is 95% sure of that within 10 time units the probability of a user receiving a response for its request within 5 time units is between 90.1855% and 100%. UPPAAL SMC allows the modeller to increase the confidence (to e.g. 99%). This will increase the time for calculating the result of the query, but the result will be more precise (i.e. the probability range will shrink).

UPPAAL SMC also provides graphical representations of the results of the queries, such as frequency histograms, probability density distributions, and cumulative probability distributions. For example, based on the frequency histogram, seen in Fig. 3.7.3, it can be seen that the shortest response time was approximately 2.7 time units, which happened in 1 out of 29 simulations, at worse approximately 6.5 time units, and on average 4.445 ± 0.382 time units, with a confidence level of 95%. And the probabilities for the different outcomes can be seen in the probability density distribution of the query (seen in Fig. 3.7.4). This says that after e.g. 3.2 time units, there is a probability of 0.33 (bucket height) · 0.10044 (bucket width) ≈ 3.3% of a user having received a response. Lastly, the cumulative probability distribution (seen in Fig. 3.7.5) can also be used to visually represent the distribution of the results to e.g. conclude that after approximately 5.3 time units, the probability of the user having received a response is approximately 0.75.

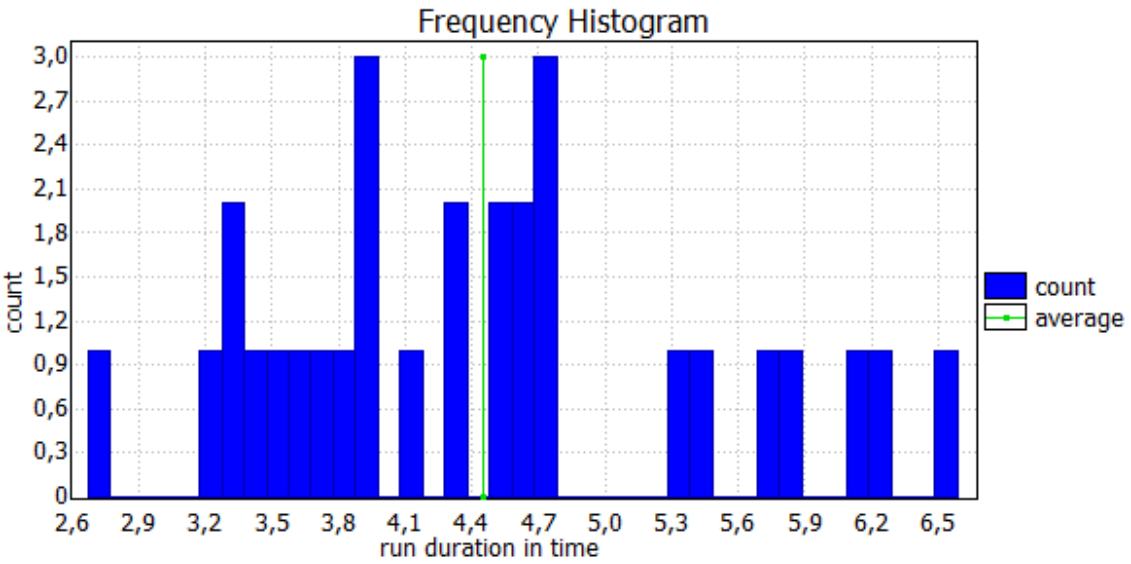
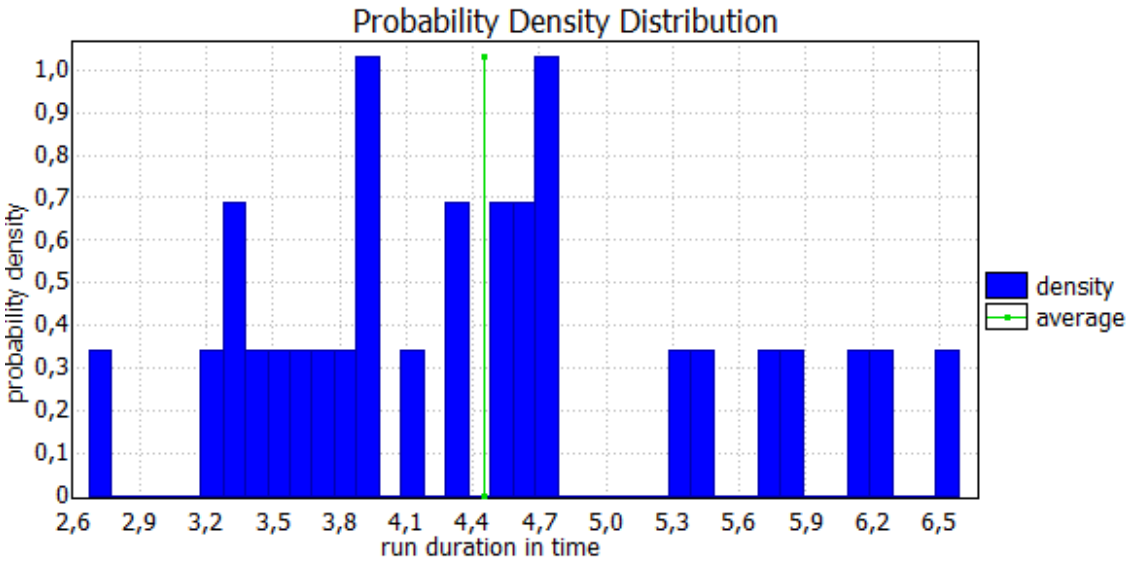
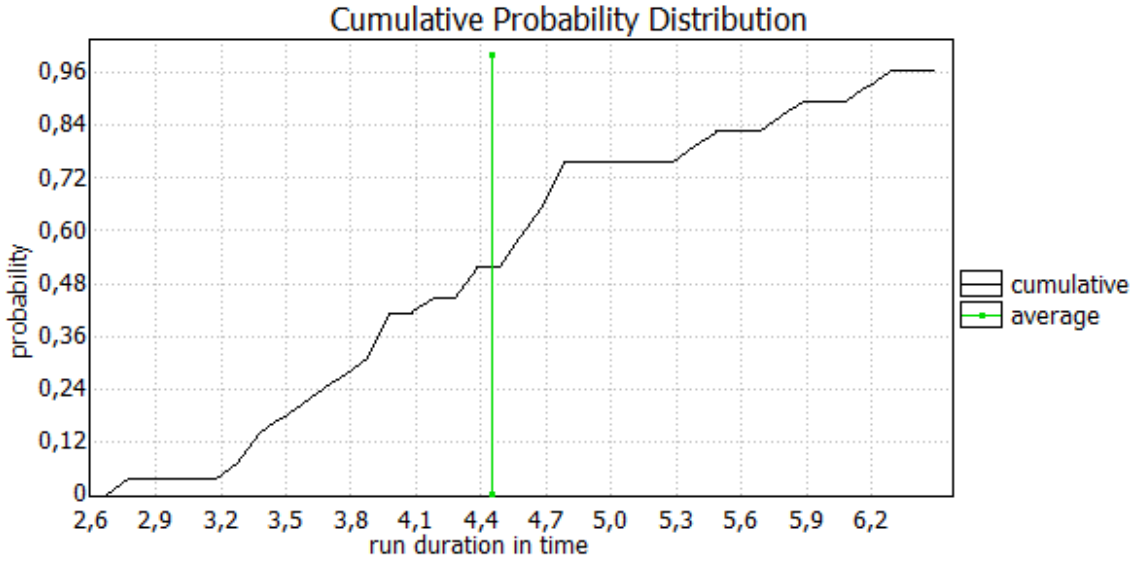


Figure 3.7.3: Frequency histogram for the above query.



Parameters: α=0.05, ε=0.05, bucket width=0.10044, bucket count=39
Runs: 29 in total, 29 (100%) displayed, 0 (0%) remaining
Span of displayed sample: [2.671, 6.588]
Mean from displayed sample: 4.445 ± 0.382 (95% CI)

Figure 3.7.4: Probability density distribution diagram for the above query.



Parameters: α=0.05, ε=0.05, bucket width=0.10044, bucket count=39
Runs: 29 in total, 29 (100%) displayed, 0 (0%) remaining
Span of displayed sample: [2.671, 6.588]
Mean from displayed sample: 4.445 ± 0.382 (95% CI)

Figure 3.7.5: Cumulative probability distribution diagram for the above query.

`Pr[<= 1500] (<> forall(id : id_Probe)DB(0).data[id] == 1)`

This query is related to the performance of the message chain from the probes to the database, as it indicates the amount of time before all probes have successfully managed to get a measurement inserted into the database. This is expressed by estimating the probability of reaching a state, within 1,500 time

units, where the database is populated. A probe can only measure the number 1, so the `DB` template has received data from each probe when all entries in the array `data` is 1. The probability of this query being satisfied is also in the range [0.901855, 1] with confidence 0.95. This asserts property 3 of the properties in section 3.1.1.

`Pr[<= 10000] ([] ProbeDBObserver(0).totalFails == 0)`
This query utilizes the `ProbeDBObserver` template to give an indication of how often a probe fails to send a message. The query estimates the probability of always being in a state, within 10,000 time units, satisfying that `totalFails` is equal to 0. Simply put, the state property fails if a probe measurement does not reach the database. `totalFails` is incremented in the `ProbeDBObserver(0)` each time it observes `Probe(0)` make a measurement that does not lead to a corresponding update in the database. The probability of this query being satisfied is in the range [0.855061,0.954924] with confidence 0.95. This asserts property 3 of the properties in section 3.1.1.

`E[<= 100000; 100] (max : ProbeDBObserver(0).totalFails)`
This query utilizes the `ProbeDBObserver` to estimate the maximum value of `totalFails` within 100,000 time units based on 100 simulations. The result is estimated to be 1.03 ± 0.181, which means that in the 100 simulation, each being 100.000 time units, `totalFails` was on average 1.03 ± 0.181. This is the expected behavior as a probe can manage to send 100 times in 100.000 time units, and is modelled to fail in 1 out of 100 times. This asserts property 3 of the properties in section 3.1.1.

3.8 UPPAAL SMC Reflections

This section covers the reflections we have made during use of UPPAAL SMC to model our P7 system.

3.8.1 The Power of SMC

UPPAAL SMC can express properties that standard UPPAAL cannot [30, p. 1] as it e.g. allows to model the stochastic behavior of systems, such as package loss, which decreases the modelling gap. However, UPPAAL SMC does not provide a qualitative answer to whether or not a system satisfies a given property or not, but instead gives a probability with a given confidence, based on a set of simulations over the system. This gives the tool a better performance when evaluating queries which makes it able to handle larger models than standard UPPAAL.

3.8.2 Timelock Confusion

While working with the UPPAAL SMC model we have faced more timelocks than when working with the standard UPPAAL model. A timelock describes the case where the only action a template can perform is to delay, but delaying would make the state invalid. Compared to deadlocks, we have found that timelocks are more difficult to debug. This means that we generally have used more time debugging the UPPAAL SMC model compared to the standard UPPAAL model. This is despite the fact that we, at the time of creating the UPPAAL SMC model, had more experience with UPPAAL and the system we wanted to model, compared to when we made the initial standard UPPAAL model. The increased debugging difficulty might be due to the fact the UPAAL SMC can not give a counter-example trace.

3.8.3 Binary Synchronization based on Broadcast Synchronization

As mentioned in section 3.6.1, we had to implement binary synchronization using broadcast synchronization. The way binary synchronization is implemented based on broadcast synchronization is shown in Fig. 3.8.1, which is a figure taken from [30]. In standard UPPAAL we can use one channel for multiple receivers because only two templates can be engaged in synchronization at once (binary synchronization). However, since UPPAAL SMC only implements broadcast synchronization we have to have one channel for each receiving template (as seen in Fig. 3.8.1 with the channels a1 and a2). If we do not have one channel for each receiver, then all the receivers can potentially synchronize with a single broadcast message. The guards for these transitions would be the conjunction of the guard from the original sending transition, the guard from the corresponding receiving transition, and a guard that ensures that the receiving componenet is in the correct location by using some global variable. This makes it cumbersome to change the number of receiving templates, and makes it cumbersome to have a high number of receiving templates.

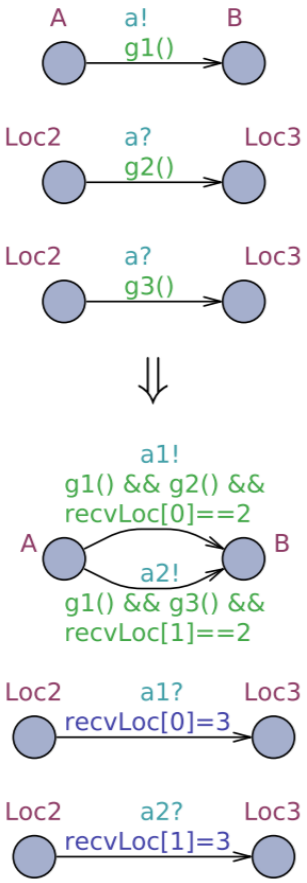


Figure 3.8.1: A figure from [30], about how to implement binary synchronization using broadcast synchronization.

Automating the Translation Process

Automating the process of implementing binary synchronization using broadcast synchronization would ease the workflow when working with both UPPAAL and UPPAAL SMC. The process of implementing binary synchronization using broadcast synchronization is a matter of a set of user input to the UPPAAL GUI, such as manipulating transitions and variables. All user input to the UPPAAL GUI, related to templates,

scripts, and formulae, are stored as plain text files i.e. .xml files. Therefore, the process of implementing binary synchronization using broadcast synchronization can be automated in terms of reading and manipulating those files, however, the visual aspect of the resulting templates may suffer when loaded into the UPPAAL GUI.

It should be noted that there is a case where it is impossible to perform this translation. It is the case where the receiving component of the binary synchronization is in a location with an invariant. This is because, in this situation, the receiver can force the transmitter to synchronize. If the transmitter does not synchronize, then the system can end in a deadlock. In this situation, the synchronization cannot be modelled in UPPAAL SMC [30].

Chapter 4

Frama-C

In this chapter, we look at the tool Frama-C [2]. Frama-C is an open-source platform that combines several analysis tools to analyze programs written in the C programming language [31]. The design of the C programming language is not heavily focused on safety. It is instead the programmers' responsibility to make the source code safe (e.g. memory safe). Furthermore, the C programming language is a very popular programming language [32]. This motivates the use of tools, such as Frama-C, to help detect errors in C programs. Frama-C can verify C code through the use of supported plugins. Multiple plugins that handle different types of verification exist, but for the purposes of this project, we will only use Eva and WP. The purpose of Eva is to ensure that the code is free of RTEs, whereas the purpose of WP is to verify the behavior of the code in relation to written annotations, using deductive verification.

Eva can show the values of variables at the end of functions. This makes it easy to find the exact range of possible return values from a function, for example. The tool can also be used to turn one program into smaller, simpler programs, and it enables the user to more easily track the values of variables between declaration and use. Furthermore, if an Eva analysis does not raise any alarms on a piece of code, then that piece of code will be free of RTEs [33].

To properly use the WP plugin in Frama-C, the user must annotate their code with the ANSI / ISO C Specification Language (ACSL). These annotations are used to express assumptions, requirements, and guarantees about functions, loops, etc. that should be respected when they are encountered. The WP plugin will, given a correctly and sufficiently annotated program, be able to prove whether or not the behavior of the program is in accordance with the written annotations.

4.1 ACSL

ACSL is used to express several kinds of annotations. In this section, we will focus on function contract annotations, assertion annotations, and loop annotations, because they cover the most common types of annotations. More kinds of annotations are described in [34]. Annotations are written directly into the C code by making a comment and adding an "@" as the first character in the comment.

When writing a function contract, different clauses are used to specify the function's behavior:

- *Requires clauses* are pre-conditions that the caller of the function must fulfill when calling the function.
- *Ensures clauses* are post-conditions that the function guarantees are true when the function terminates correctly.
- *Assigns clauses* specify a list of global variables and/or formal parameters that the function may assign during its execution.

- *Allocates clauses* specify a list of global variables and/or formal parameters that the function may dynamically allocate on the heap during its execution.
- *Frees clauses* specify a list of dynamically allocated global variables and/or formal parameters that the function may free during its execution.
- *Behavior clauses* specify a possible behavior of the function. Behavior clauses contain assumption clauses and ensures clauses.
- *Assumption clauses*, specify when a behavior clause must hold.

Annotating functions in a program is often not enough. If a function contains a loop it is also necessary to specify a loop invariant. If the loop invariant is not specified the code cannot be analyzed [35].

- A *loop invariant* specifies a boolean expression that is true at the beginning of the first iteration, and after each successful iteration, which is an iteration that does not stop through a return or break statement.
- A *loop variant* is, contrary to the loop invariant, optional and is a numerical value. The variant must satisfy that each iteration of a loop decreases the value of the variant. The initial value of the variant must be positive, but this is not guaranteed for the loop's last iteration.
- A *loop assigns* specifies what variables are assigned in the loop. The loop assign should not contain variables that only exist in the scope of the loop[1].

Annotations written in ACSL rely on other constructs than those already mentioned, some of which are described here:

- `\valid` is an ACSL function that takes a pointer (that is not a void pointer) and returns a boolean value. It returns true iff the given pointer is valid, i.e. it is safe to deference the pointer and write to it, at the time of calling `\valid`, and returns false otherwise.
- `\valid_read` is also an ACSL function that takes a non-void pointer and returns a boolean value. It returns a boolean value indicating whether the content of the pointer can be safely read, but does not say anything about writing to it. `\valid(p)` implies `\valid_read(p)` for all non-void pointers p.
- `valid_string` a function that takes a pointer and returns a boolean indicating if the pointer is pointing to a valid string.
- `valid_read_string` is a function that takes a pointer and returns a boolean indicating whether it points to a readable valid string. Note that `valid_read_string` does not say anything about writing validity.
- `\result` refers to the returned value of the function. `\result` is not allowed in function contracts for void functions, loop annotations, or requires clauses.
- `\old` is a function that takes a global variable or formal parameter and returns the value of the variable at the time of calling the function. `\old` can only be used in ensures, assigns, allocates, and frees clauses.
- `\nothing` is an expression used in assigns, frees, allocates, and loop assigns clauses to denote that nothing is assigned to, freed, or allocated.

[1]There is an exception for `for` loops specifically. The loop variable(s) of the for loop should also be mentioned in the loop assigns clause.

- `\at` is used to denote at what point in the program a variable should be evaluated. The `at` function takes two arguments, a variable and a label. The label can either be manually defined (as labels in C) or a built-in ACSL label. The built-in labels are explained in table 4.1.1.

Label	Permitted Locations	Meaning
Here	Assertions Loop annotations Function contracts	State where the annotation appears Pre-state of function
Old	Function contracts	Pre-state of function
Pre	Loop invariant	Pre-state of loop
Post	Assigns and ensures clauses	Post-state of loop or function
LoopEntry	Loop annotations	State prior to first loop entry
LoopCurrent	Loop annoations	State at beginning of current loop iteration
Init	All annotations	State before call to main

Table 4.1.1: Explanation of the labels built into ACSL [36].

ACSL contains a multitude of other constructs to annotate the code. For a deeper understanding of the specification language, we refer to the ACSL implementation manual [34].

4.2 The Eva Plugin

Eva is an abbreviation of Evolved Value Analysis and is one of the plugins in the Frama-C platform. The Eva plugin automatically computes sets of all the possible values a variable can have in a program. Eva uses proof obligations to reason about possible RTEs in a program. A proof obligation is a type of warning that suggests making ACSL assertion annotations in the code to highlight potential RTEs. Eva supports automatic code annotation as suggested by proof obligations.

This paragraph is based on [33]. Eva allows the programmer to specify annotations that can detect RTEs in the code. The types of bugs that can be detected by Eva include invalid memory access, invalid pointer arithmetic, invalid function pointer access, etc. Furthermore, if Eva does not raise any alarms during the analysis of a piece of code, then the code is formally verified to not have any RTEs. However, Eva sometimes gives false warnings and alarms about potential bugs in the code due to over-approximation. To mitigate false alarms and warnings, Eva offers different degrees of precision. A default run with the Eva plugin is set to a precision between 0 and 1 out of 11, this is done to ensure fast analysis. However, the depth/precision of the analysis can be increased, causing the analysis to take more time, but be less over-approximated which potentially avoids some false alarms and warnings. The `-eva-precision` option is a shortcut to adjusting multiple aspects of the analysis. The aspects include minimum loop unrolling, auto loop unrolling, slevel, and more. The slevel can be thought of as a limit to how many states in the analysis should be stored separately. The slevel reached by the analysis increases with loops and branches. Once the slevel is at the specified limit, the analysis will merge the stored states to avoid further increase analysis time. Because of this merge, the results will be less accurate, but it will still be a correct over-approximation.

To demonstrate the Eva plugin, we have written a small program in C with errors in it. The C program can be seen in listing 4.1. In line 10, the `i`'th element in `a` is accessed. Given the values of `b` and `c`, `i` can be as high as 100, meaning that line 10 accesses a value outside the array.

```
1  #include <stdio.h>
2
3  int a[100];
4  int b = 3;
5  int c = 97;
6
7  int main()
8  {
9      for (int i = 0; i <= c+b; i++)
10         a[i] += 10;
11
12     return 0;
13 }
```

Listing 4.1: Example C code that invokes a proof obligation.

A C compiler will compile and run the program without complaining, but, as seen in listing 4.2, Eva claims that the program is non terminating. This indicates that Eva interprets access out of bounds errors as fatal. In addition, Eva gives two alarms: "Warning: accessing out of bounds index. assert i < 100" (cf. line 6), and "Warning: signed overflow. assert a[i] + 10 <= 2147483647" (cf. line 8). The former alarm is a reminder to ensure that the variable `i` is not incremented to be larger than the size of the `a` array. The latter says that when incrementing the value stored in `a[i]` by 10, there might be an integer overflow. Furthermore, Eva gives a summary about how much of the code was analyzed, both in terms of statements and functions (cf. lines 15-16).

```
1  [eva:initial-state] Values of globals at initialization
2    a[0..99] in {0}
3    b in {3}
4    c in {97}
5  [eva] main.c:9: starting to merge loop iterations
6  [eva:alarm] main.c:10: Warning: accessing out of bounds index. assert i < 100;
7  [eva:alarm] main.c:10: Warning:
8    signed overflow. assert a[i] + 10 <= 2147483647;
9  [eva] done for function main
10 [eva] ====== VALUES COMPUTED ======
11 [eva:final-states] Values at end of function main:
12   NON TERMINATING FUNCTION
13 [eva:summary] ====== ANALYSIS SUMMARY ======
14   ------------------------------------------------------------------------------
15   1 function analyzed (out of 1): 100% coverage.
16   In this function, 6 statements reached (out of 9): 66% coverage.
17   ------------------------------------------------------------------------------
18   No errors or warnings raised during the analysis.
19   ------------------------------------------------------------------------------
20   2 alarms generated by the analysis:
21        1 access out of bounds index
22        1 integer overflow
23   ------------------------------------------------------------------------------
24   No logical properties have been reached by the analysis.
25   ------------------------------------------------------------------------------
```

Listing 4.2: Eva output from example C code

Eva is only able to show the values of variables at the end of functions. For functions that never terminate or functions that have at least one non-terminating path, Eva is not able to say anything about the variables. Furthermore, in case Eva encounters a non-terminating path, the analysis is stopped, so other functions that are called afterward are not analyzed. That is why the output from listing 4.2 shows no values for variables

at the end of main.

Listing 4.3 is the output given by Eva if we change the program such that $b+c < 100$ (i.e. no out of bounds error), and when precision is set to 5 out of 11. When the precision is set to 5, we avoid the alarm about signed overflow, due to the tool performing loop-unrolling, and thus analyzing each iteration of the loop, ensuring that the value being calculated is not larger than 2,147,483,647. Lastly, it can also be seen that Eva outputs the values of global variables at the start and end of the program as well the values of variables at the beginning and end of functions as well as the return values of functions (cf. listing 4.3, lines 2-4 and 10-11).

```
1  [eva:initial-state] Values of globals at initialization
2    a[0..99] in {0}
3    b in {2}
4    c in {97}
5  [eva:loop-unroll] main.c:8: Automatic loop unrolling.
6  [eva] main.c:8: Trace partitioning superposing up to 100 states
7  [eva] done for function main
8  [eva] ====== VALUES COMPUTED ======
9  [eva:final-states] Values at end of function main:
10  a[0..99] in {10}
11   __retres in {0}
12 [eva:summary] ====== ANALYSIS SUMMARY ======
13   ------------------------------------------------------------------------------
14   1 function analyzed (out of 1): 100% coverage.
15   In this function, 9 statements reached (out of 9): 100% coverage.
16   ------------------------------------------------------------------------------
17   No errors or warnings raised during the analysis.
18   ------------------------------------------------------------------------------
19   0 alarms generated by the analysis.
20   ------------------------------------------------------------------------------
21   No logical properties have been reached by the analysis.
22   ------------------------------------------------------------------------------
```

Listing 4.3: Eva output from example C code

4.3 The WP Plugin

This section is based on [36]. Contrary to the Eva plugin the WP plugin is not primarily used to ensure that there are no RTEs in the code. The WP plugin is used to generate proofs based on ACSL annotations. The proofs generated by the WP plugin are dependent on the fact that no RTEs occur in the program. Thus for the proofs generated by WP to be sound, the WP plugin should be used with the `-wp-rte` option or in combination with Eva.

The WP plugin is built on the weakest pre-condition calculus. Running the WP plugin on an annotated program gives a mathematical proof about the ACSL annotations. The weakest pre-condition calculus can be explained with Hoare triples. A Hoare triple is on the form

$$\{P\}\ stmt\ \{Q\} \tag{4.1}$$

where P and Q are predicates and are respectively known as the pre-condition and post-condition. In this context P and Q are ACSL requires clauses and ensures clauses respectively. For a given predicate S, $\{S\}$ is the set of program states that satisfy S. *stmt* is a list of program statements. In this context, *stmt* could be the body of a function. The Hoare triple $\{P\}$ *stmt* $\{Q\}$ means that any program state that satisfies P can execute *stmt*, and will then be guaranteed to satisfy Q.

WP is able to calculate the weakest pre-condition, given the program statements (*stmt*) and the post-condition (Q). In this context "weakest pre-condition" is the pre-condition that includes all program states that can execute *stmt* and then satisfy Q. This can be done with the function $wp(stmt, Q)$. The function returns a copy of Q where the variables used in *stmt* have been updated. For example, if *stmt* is `{x = x + 1}`, and Q is `x < 10`, then $wp(stmt, Q)$ will be `x + 1 < 10`. When WP has calculated the weakest pre-condition, it will compare it to the given ACSL pre-condition and provide an error message if the given ACSL pre-condition is weaker than the calculated weakest precondition. This is because, if the given ACSL pre-condition is weaker, then the post-condition cannot be ensured.

The swap function described in 4.4 is a simple example of the usage of ACSL annotations to specify the behavior of some code.

```
1  const char* c = "";
2
3  /*@ requires \valid(a) && \valid(b);
4      assigns *a, *b ;
5      ensures \old(*a) == *b ;
6      ensures \old(*b) == *a ; */
7  void swap(int* a, int* b) {
8      int temp = *a;
9      *a = *b;
10     *b = temp;
11 }
```

Listing 4.4: Example of an annotated Swap function. Code taken from [36].

Using the WP tool on the example in listing 4.4 with the following command:

```
frama-c -wp -wp-model "Typed+var+int+float" -wp-timeout 20 swap.c
```

gives the output seen in listing 4.5. This output states that all the goals set by the clauses have been proven. This means that the function adheres to its function contract.

```
1 [kernel] Parsing swap.c (with preprocessing)
2 [wp] Warning: Missing RTE guards
3 [wp] 4 goals scheduled
4 [wp] [Cache] found:1
5 [wp] Proved goals:    4 / 4
6   Qed:              3  (0.69ms-1ms-3ms)
7   Alt-Ergo 2.2.0:   1  (12ms) (32) (cached: 1)
```

Listing 4.5: Output from example code.

4.4 Frama-C Program Examples

To give an example of the use of the Eva and WP plugin for Frama-C we have annotated a small C-program that we have written during a previous semester. This program is a simple terminal program that allows doing certain queries about football matches in a tournament that are recorded in an accompanying .txt file. Since we use this program only to give an example of how to do an Eva analysis, and how to annotate a program for a WP analysis, we have cut away all but one query. This query, along with basic functionality in the program such as the menu, is annotated with special ACSL comments as described in section 4.3. All functions have been annotated with function contracts and all loops are annotated with loop annotations. Furthermore, some code has been re-written so that it is easier to annotate while maintaining the same functionality. Lastly, some functionality has been simplified or "assumed as being correct" while

annotating. Functionality that has been simplified includes user input via `scanf()` in the form of menu choice (this has been assigned a fixed value) and `fscanf()` for reading a football match from the input file. Functionality that has been "assumed as being correct" includes the C function `printf()`. It is difficult to handle such functions with our level of experience with Frama-C. The following sections show some of our results from the Eva and WP analysis of the C-program. The fully annotated program can be found in appendix B.

It should be noted that the program does not need to be annotated to do an Eva analysis. The annotations are created for running the WP analysis.

4.4.1 Eva Example

For WP to be able to guarantee the correctness of the program we need to ensure the absence of RTEs. This is done with the EVA plugin. We use the following command:

```
frama-c -eva -eva-precision 11 non-annotated-exam.c
```

We specify that we want to use precision level 11, which is the highest level of precision. The function contracts and loop annotations written for WP are temporarily cut from the program, to avoid Eva generating alarms related to the annotations. The output given by Eva is seen in listing 4.6. According to Eva, the code is free of RTEs and with only a single warning, which is that a floating point value is not represented exactly. In this case, the warning can be safely ignored.

```
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------------
  5 functions analyzed (out of 5): 100% coverage.
  In these functions, 111 statements reached (out of 131): 84% coverage.
  ------------------------------------------------------------------------------
  Some errors and warnings have been raised during the analysis:
    by the Eva analyzer:      0 errors    0 warnings
    by the Frama-C kernel:    0 errors    1 warning
  ------------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------------
  No logical properties have been reached by the analysis.
  ------------------------------------------------------------------------------
```

Listing 4.6: EVA analysis summary from example program

The analysis covers all the functions, in total 84% of the statements are covered. The reason for it not being 100% is that some values have been hardcoded to simplify the annotation process. The hard coded values cause some branches of the program to never be chosen.

4.4.2 WP Examples

To do a WP analysis of the program we run Frama-C with the following command:

```
frama-c -wp -wp-model "Typed+var+int+real" -wp-timeout 20 -wp-cache none file.c
```

We specify which models to use according to [36]. *Typed* is a memory model used by WP. *var* allows a combination of memory models to be used depending on the verification. *int* means that we assume that machine integers are used, which means that we assume that overflows can happen. *real* means that we are using mathematical reals rather than floating point operations. We had to use *real* to avoid an error with conversion between types. We set the max time allowed to spend verifying a property to 20 seconds rather than the default 10 seconds.

The following two sections describe two of the functions in the C program and detail how we have annotated them. The annotations shown in these sections show minimal annotations that pass a WP analysis. However, we found that these annotations cannot be used to verify more complex properties about the functions. We have chosen to include them in this section to show how easy it can be to write "correct", but uninteresting annotations. This also documents our progress as this was our initial approach to writing annotations for WP. section 4.4.3 shows in-depth annotations that can be used to infer more interesting properties of the annotated functions.

mainLoop

The `mainLoop` function (seen in listing 4.7) is responsible for prompting the user for a query number and printing the answer to the query based on the input. The original purpose of the program was to have an infinite loop with an exit clause based on user input. For the purpose of verifying with Frama-C, we have changed the code such that the loop can only be iterated three times. We have done this to ease the loop unrolling for Frama-C. We have removed all calls to `printf()` and `scanf()`. We have assumed that the behavior of the function does not change with this restriction.

Function Contract

We have made a function contract that describes requirements and guarantees of the function, and loop annotations that describe invariants and guarantees of the loop. As seen in line 2 the function requires that the formal parameter `matchArray` can be read and that the formal parameter `arrayLength` is at least 0. This means that if the function is called with an invalid `matchArray` or a negative `arrayLength`, Frama-C would raise a warning. As seen in line 3, the function assigns nothing, which means that calling the function will not change any of the parameters or any global variables. Lastly, as seen in line 4 the function ensures that all its formal parameters are the same as when the function was called. This is essentially the same as `assigns \nothing`. In non-void functions, the `ensures` clause typically says something about the returned value of the function.

Loop annotations

For the while loop in line 11, we have made loop annotations. This contract consists of a loop invariant annotation and a loop assigns annotation. The loop invariant is very similar to the loop condition in line 11. The reason that they are not the same is because the loop invariant has to be satisfied after the last iteration, and the while condition does not. In line 10 the loop assigns annotation specifies the variables that are assigned inside the loop.

```
1  /* The main loop of the program that does different things depending on user input
       ↪ if isRunPrint is not enabled */
2  /*@ requires \valid_read(matchArray) && arrayLength >= 0 ;
3      assigns \nothing ;
4      ensures \old(isRunPrint) == isRunPrint && \old(*matchArray) == *matchArray && \
       ↪ old(arrayLength) == arrayLength ;*/
5  void mainLoop(match *matchArray, int arrayLength, int isRunPrint){
6      int menuChoice = 1, wrongChoice = 0, run = 1;
7      int iter = 3;
8
9      /*@ loop invariant run && iter >= 0 ;
10         loop assigns menuChoice, iter ;*/
11     while(run && iter > 0){
12         menuChoice = 1;
13
14         if(isRunPrint){
```

```
15              sevenGoalMatches(matchArray, arrayLength);
16          }
17
18          else if(menuChoice == 1){
19              sevenGoalMatches(matchArray, arrayLength);
20          }
21          iter--;
22      }
23  }
```

Listing 4.7: The annotated mainLoop function.

stringCompare

The `stringCompare` function (seen in listing 4.8) is a replacement for `strcmp` in `string.h`. We have made this function because `strcmp` has requirements that are unknown to us. This is a problem since it meant that WP could not properly analyze code that utilize `strcmp`. `stringCompare` takes 4 parameters; `s1`, `s2`, `n1`, and `n2`. `s1` and `s2` are strings, and `n1` and `n2` are the lengths of `s1` and `s2`, respectively. Like `strcmp`, this function returns 0 if the contents of `s1` and `s2` is the same, and it returns a positive number if `s1` is higher than `s2` lexicographically and a negative number otherwise.

Function Contract

The function annotations in lines 1-6 require that the lengths of the strings are at least 0. The function ensures that if it is passed the same string (i.e. `s1` = `s2`) then the function will return 0. It also ensures that if the function returns 0, then the content of `s1` and `s2` must be the same in all their entries and that the lengths of `s1` and `s2` are the same.

Loop Annotations

We have made loop annotations for the loop in line 13. The loop invariant in line 11 says that the variable `i` is always at least 0. This is ensured by the fact that it is initialized to 0, and is incremented in the loop. The loop assigns annotation in line 12 specifies that the loop alters the values of `i`, `s1`, and `s2` in the loop and that it is not allowed to write to other variables.

```
1  /*@  requires 0 <= n1 ;
2       requires 0 <= n2 ;
3       requires n1 == strlen(s1) && n2 == strlen(s2) ;
4       ensures (s1 == s2 ==> \result == 0) ;
5       ensures \result == 0 ==> (\forall integer k ; 0 <= k < n1 ==> s1[k] == s2[k] );
6       ensures \result == 0 ==> n1 == n2 ;*/
7  int stringCompare(const char* s1, const char* s2, int n1, int n2) {
8      if (s1 == s2)
9          return (0);
10     int i = 0;
11     /*@ loop invariant 0 <= i;
12         loop assigns i , s1, s2; */
13     while (*s1 == *s2++)
14     {
15         ++i;
16         if (*s1++ == '\0')
17             return (0);
18     }
19     return (*(unsigned char*)s1 - *(unsigned char*)--s2);
20 }
```

Listing 4.8: The annotated stringCompare function.

While `mainLoop` and `stringCompare` pass all goals set by WP with the annotations shown, the annotations themselves are not that interesting. They are very superficial and do not verify interesting properties of the two functions. Even though this is not an error, it means that the power of Frama-C is wasted. Because of this, we are not satisfied with the annotations. We will therefore try to create more interesting annotations on smaller pieces of code.

4.4.3 Examples of More Rigorous Annotations

The following subsections describe examples of functions that have been more thoroughly annotated than the ones shown in section 4.4.2. Note that the `arrayMax` and `stringCopy` functions described in section 4.4.3 and section 4.4.3, respectively, are not used in the program described in section 4.4.

arrayMax

In this section we give an example of an `arrayMax` function (seen in listing 4.9). This function takes as parameters an array of integers and the size of the array. It then returns the largest element in that array. To verify the function we use the command:

```
frama-c -wp -wp-rte exam.c
```

The reasons for using this command as opposed to the one shown in section 4.4.2 can be seen in section 4.4.3.

Function Contract
The function contract of `arrayMax` can be seen in line 1. This function contract is made up of five clauses: two requires clauses, one assigns clause, and two ensures clauses. The first requires clause, labeled as `validSize`, states that the given size of the array must be between 0 and `SIZE_MAX`, which denotes the maximum size of `size_t`. The second requires clause, labeled as `validArr`, states that the given array must consist entirely of valid readable pointers. This is done through the built-in predicate called `\valid_read` which is used on all indices of the array. The third clause, the assigns clause, simply states that the function does not assign anything other than local variables (given by the `\nothing` construct). The fourth clause, labeled as `Largest`, states that for all elements in the given array, the result of the function is either equal to, or larger than those elements. This ensures that the value returned by the function is larger than or equal to each individual element in the array. The fifth and final clause, labeled `existsInArray`, states that the result of the function must exist in the given array.

Together these clauses say that assuming the array is a valid array of integers, and the size is valid, the returned value will be the largest integer in the array. They also state that the function does not write to any non-local memory-locations. However, since the function contains a loop we also have to annotate this for WP to be able to verify the function contract.

Loop Annotations
The beginning of the loop annotations can be seen in line 11. The loop annotations consist of loop invariants, a loop assigns, and a loop variant. The first invariant, labeled as `largest`, states the same as the ensures labeled `largest`. That is, that the result is larger than or equal to every element in the array. However, this invariant only holds up to the current iteration of the loop and not the complete size of the array. The second invariant, labeled `pos`, ensures that the iteration counter `i` is between 0 and `size` (including these bounds). The third and last invariant, labeled `existsInArr`, states the same as the ensures clause of the same name. However, as with `largest`, it only does so up to the current iteration of

the loop. For the loop assigns we can see that the loop assigns both the local variable `i` and `result`. Lastly, we have included a loop variant in the form of `size - i`. This loop variant enables WP to reason about the termination of the loop. It should be noted that the loop variant is not required by WP.

With all of these annotations in place, WP can correctly verify that the function adheres to its function contract. As opposed to the function contracts described in section 4.4.2 and section 4.4.2 these annotations actually tell us relevant and interesting things about the `arrayMax` function. Most notably the two ensures clauses of the function contract verify that the function returns the largest element of a given array of integers.

```
1  /*@ requires validSize: SIZE_MAX > size > 0 ;
2      requires validArr: \valid_read(arr + (0 .. size - 1)) ;
3      assigns \nothing ;
4      ensures largest: \forall integer i; 0 <= i < size ==> \result >= arr[i] ;
5      ensures existsInArr: \exists integer i; 0 <= i <= size && arr[i] == \result ;
6      */
7  int arrayMax(int* arr, size_t size) {
8      int result = arr[0];
9      size_t i = 0;
10
11     /*@  loop invariant largest: \forall integer k; 0 <= k < i ==> result >= arr[k];
12          loop invariant pos: 0 <= i <= size;
13          loop invariant existsInArr: \exists integer k; 0 <= k <= i && arr[k] ==
       ↪ result ;
14          loop assigns i, result;
15          loop variant size - i;
16          */
17     while (i < size) {
18         if (arr[i] > result)
19             result = arr[i];
20
21         i++;
22     }
23
24     return result;
25 }
```

Listing 4.9: The annotated `arrayMax` function.

stringCopy

To give an example of a function that takes strings as parameters as well as one that writes to a string, we have annotated a `stringCopy` function. The function works like `strcpy` from `string.h`. The function takes two `char*` s as input. One of these is the source and the other is the destination. The function and its annotations can be seen in listing 4.10.

Function Contract

The function contract of the `stringCopy` function can be seen in line 5 of listing 4.10. This function contract consists of nine clauses: six requires clauses, one assigns clause, and two ensures clauses. The first requires clause, labeled as `validDest`, states that the `dest` parameter must be a `valid_string`. The second requires clause, labeled as `validReadSrc`, stated that the `src` parameter must be a `valid_read_string`. `valid_string` and `valid_read_string` are explained in section 4.1. The

third requires clause, labeled as `destLargest`, states that the length of the `dest` string must be larger than, or equal to, the length of the `src` string. The fourth requires clause, labeled as `noDestOverflow`, states that the string length of `dest` must be strictly less than `SIZE_MAX`. `SIZE_MAX` translates to the maximum value of `size_t`. The fifth requires clause, labeled as `separatedStrings`, states that the entirety of both the `src` string and the `dest` string have to be separated in memory. This means that no parts of the two strings can be in the same memory. The sixth requires clause, labeled as `nullCharEnd`, states that after the last character of the `src` string there must be the null character to correctly end the string. The assigns clause states that the function assigns every part of the `dest` string up to the length of the `src` string. The first ensures clause, labeled as `copied`, states that after the function returns then `src` and `dest` should be identical strings. The second ensures clause, labeled as `stillValidDest`, states that after the function returns then after the last character of `dest` there will be a null character indicating that `dest` is still a valid string.

Together all of these clauses state that given two valid strings (one of which can be written to) that have no overlap in memory, the function will correctly copy the content of the `src` string into the `dest` string. However, as with the `arrayMax` function, the `stringCopy` function contains a loop that must be annotated for WP to correctly analyze the function.

Loop Annotations

The loop annotations can be seen in line 25. These are made up of four loop invariants and a loop assigns clause. The first loop invariant, labeled as `validRange`, states that `i` will always be between 0 and `destStrlen`, and that `srcStrlen` is less than or equal to `destStrlen`. The second invariant, labeled as `intermediateCopied`, states that `dest[k]` and `src[k]` will be the same character for each k between 0 and `i` (not including `i`). In other words, this invariant states that after X iterations of the loop, X characters from `src` have been copied into `dest`. The third invariant, labeled as `srcPos`, states that the original `src` pointer plus the current iteration count `i` equals the `srccopy` pointer. The fourth invariant, labeled as `destPos`, states the same as the third invariant, but for `dest` and `destcopy` rather than `src` and `srccopy`, respectively. Lastly, the loop assigns clause states that the loop assigns `i`, the `srccopy` and `destcopy` pointers, and the content of `dest` up to the length of `src` - 1.

These annotations together pass a WP analysis and therefore ensures that the function behaves correctly. It should be noted that it was difficult to get the annotations to work on this version of the function that uses pointer arithmetics. To get the WP analysis to pass we had to introduce the `srccopy` and `destcopy` pointers so that we could avoid incrementing the base pointers of `src` and `dest`. It would be easier to write the function so that we access the strings as arrays and then annotate this. Nevertheless, we feel that this example is more interesting as it shows how to do annotations with pointer arithmetics.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdint.h>
4
5  /*@ requires validDest: valid_string(dest) ;
6      requires validReadSrc: valid_read_string(src);
7      requires destLargest: strlen(dest) >= strlen(src);
8      requires noDestOverflow: SIZE_MAX > strlen(dest);
9      requires separatedStrings: \separated(src + (0 .. strlen(src)), dest + (0 ..
       ↪ strlen(dest))) ;
10     requires nullCharEnd: src[strlen(src)] == '\0';
11
12     assigns *(dest + (0 .. strlen(src)));
```

```
13
14      ensures copied: \forall integer k; 0 <= k <= strlen(src) ==> dest[k] == src[k];
15      ensures stillValidDest: *(dest + strlen(src)) == '\0';
16      */
17  void stringCopy(char* dest, const char* src) {
18      size_t i = 0;
19      size_t srcStrlen = strlen(src);
20      size_t destStrlen = strlen(dest);
21
22      char* destcopy = &(dest[0]);
23      char* srccopy = &(src[0]);
24
25      /*@ loop invariant validRange: 0 <= i <= srcStrlen <= destStrlen ;
26          loop invariant intermediateCopied: \forall integer k; 0 <= k < i ==> dest[k]
        ↪   == src[k];
27          loop invariant srcPos: src + i == srccopy ;
28          loop invariant destPos: dest + i == destcopy ;
29
30          loop assigns i, srccopy, destcopy, *(dest + (0 .. srcStrlen - 1));
31
32      */
33      while (i < srcStrlen) {
34         *destcopy = *srccopy;
35          i = i + 1;
36          srccopy = srccopy + 1;
37          destcopy = destcopy + 1;
38      }
39      *destcopy = *srccopy;
40  }
```

Listing 4.10: The annotated `stringCopy` function.

Alternative stringCompare Annotations

The code seen in listing 4.11 shows a more thoroughly annotated `stringCompare` function. These annotations are made by a StackOverflow user called “Virgile” [37]. While we did try to do the annotations ourselves, we felt that we did not have the knowledge required to do so sufficiently. We hit a lot of roadblocks that could not be overcome by using the available Frama-C WP or ACSL material found in [36] and [34], respectively. We, therefore, went to StackOverflow with our code along with our issues and got an answer in the form of some general helpful comments and the code seen in 4.11. This answer gave us some valuable insights that we feel are understated in the official documentation. Firstly, the importance of solid loop invariants is somewhat understated from what we could gather ourselves. The response we got from StackOverflow showed us just how precise one must be when designing these for one’s annotations. Secondly the use of `valid_string` when using strings in the annotated program to require valid strings is something we could not find in the manuals (we used the more generic `valid` which gave us some issues). Furthermore we learned that WP cannot handle “strings” in C that are initialised as `const char* hello = "hello";`, instead a “string” must be explicitly initialised as char arrays `const char hello[] = { 'h', 'e', 'l', 'l', 'o', '\0'};`. The insights gathered from the StackOverflow response was one of the things that allowed us to eventually annotate functions such as `arrayMax` which is shown in listing 4.9.

```
/*@
    requires validPointers: valid_read_string(s1) && valid_read_string(s2);
    requires validLengthS1: SIZE_MAX >= strlen(s1) >= 0;
    requires validLengthS2: SIZE_MAX >= strlen(s2) >= 0;
    assigns \nothing ;
    allocates \nothing ;
    frees \nothing ;
    behavior allEqual:
        assumes \forall integer k; 0 <= k <= strlen(s1) ==> s1[k] == s2[k];
        ensures \result == 0;
    behavior SomeDifferent:
        assumes \exists integer k; 0 <= k <= strlen(s1) && s1[k] != s2[k];
        ensures \result != 0;

    disjoint behaviors;
    complete behaviors;
    */
int stringCompare(const char* s1, const char* s2) {
    if (s1 == s2)
        return 0;
    size_t i = 0;

    /*@ assert \valid_read(s1) ; */
    /*@ assert \valid_read(s2) ; */
    /*@ loop invariant index: 0 <= i <= strlen(\at(s1,Pre));
        loop invariant index_1: 0 <= i <= strlen(\at(s2,Pre));
        loop invariant s1_pos: s1 == \at(s1,Pre)+i;
        loop invariant s2_pos: s2 == \at(s2,Pre)+i;
        loop invariant equal: \forall integer j; 0 <= j < i ==> \at(s1,Pre)[j] == \
    ↪ at(s2,Pre)[j];
        loop invariant not_eos: \forall integer j; 0 <= j < i ==> \at(s1,Pre)[j] !=
    ↪ '\0';
        loop assigns i , s1, s2; */
    while (*s1 == *s2)
    {
        ++i;
        if (*s1 == '\0')
            return 0;
        ++s1;
        ++s2;
    }

    return *s1 - *s2;
}
```

Listing 4.11: Virgile's `stringCompare` annotations.

Virgile also expressed that the command (cf. section 4.4.2) we used was wrong. There is no reason to change the memory model to solve a verification problem. However, he suggests adding `-wp-rte` to also catch any RTEs that might arise. This means that we should use the command:

```
frama-c -wp -wp-rte file.c
```

4.5 Frama-C Reflections

In this section we document our reflections on our use of Frama-C along with general reflections on the tool itself.

4.5.1 Installation

Installing the Frama-C tool on Linux is quite easy. However, we experienced issues trying to get it to run correctly on our Windows machines. We found a guide for doing so and by following it we got parts of the tool to work. To get Frama-C working on a Windows machine we had to set up a Windows Subsystem for Linux (WSL), which was straightforward due to a large amount of documentation. As of yet, we have not gotten the Frama-C GUI working on Windows. It is unclear to us whether this is even possible at this point.

4.5.2 Eva

In general, Eva is much easier to use than WP. Eva does not require that one annotates a program before analyzing it and the error messages that it gives are much easier to address. It also seems that running an Eva analysis before running a WP analysis has some benefits. We feel that running the analyses in this order and rectifying the errors that Eva gave us, helped a few of the issues we saw in our WP analysis. Furthermore, it seems that Eva can take ACSL annotations into account when doing an analysis. We do not, as of yet, know to what extent this is done but we have used this feature to create, change, and remove some annotations made for WP based on the result of Eva.

Output

When generating output Eva will, besides errors and warnings, emit alarms if any potential RTEs are present. These alarms are in some cases false alarms and can be removed by increasing the precision of the analysis. However, it took some time to understand how alarms were to be understood compared to errors and warnings. This was especially confusing because some alarms are given as `[eva:alarm] strcpy.c:29: Warning: loop invariant got status unknown`. Here the alarm is related to a loop invariant annotation written for WP. This is not a warning, despite the message indicating so. Also since it is not an alarm that can cause an RTE, Eva does not include it in the analysis summary. This means that we end up with an output indicating that there are alarms, and a summary stating that there are not any alarms.

Overestimation

The Eva plugin overestimates the potential of RTEs, meaning that it in some cases will warn about a potential RTE that can never occur. This has been a source of confusion in some scenarios. However, the overestimation is also what makes an Eva analysis sound and therefore an error-free Eva analysis is a valid claim to the fact that a program does not have any RTEs.

4.5.3 WP

This section goes in-depth with our reflections on the WP plugin of Frama-C.

Challenges with strlen

There is a mismatch between the syntax used in the ACSL implementation manual and the actual correct syntax. The manual introduces a function `\strlen()`. However this is not a valid function to use for annotations, to use the `\strlen()` function the "\" needs to be omitted thus only writing `strlen()`. This is confusing because we initially thought that the `\strlen()` function was just not implemented.

It is Difficult to Annotate Existing Programs

When annotating the program, we found out that it would have been a lot easier to annotate the program as it was being written, rather than writing the annotations afterward. This is because we found that we had to make significant changes to the code to prove the annotations with Frama-C. We think that annotating the code while writing it generates better and more robust code.

Sparse Number of Examples

The Frama-C manual for WP does provide some examples of how to annotate code. However, the manual has been the only source of examples of annotation. Ideally, if more examples would have been available, this would have eased the learning process. In addition to the lack of examples, the number of forums writing about Frama-C and how to use it is also small. In general, a larger user-base would ease the learning curve.

Difficult to Take Input from Files

We found that reading from a file is very difficult to annotate. It would be beneficial to have a way of telling Frama-C / WP to ignore certain parts of the program or certain functions and assume that they function correctly. In the end, we had to abstract those functions and parts of the program away and lose some details about how the behavior of the program changes because of the file's contents. Skipping may be a viable option to ignore function calls that are assumed to behave correctly. It is possible to write wrapper functions for those library functions and skip those, so in effect, it is possible to skip library functions.

Complexity of WP

Throughout our experience with Frama-C, it has become clear that the tool is difficult to pick up and use for new users. Even with our knowledge of verification, we have had a lot of issues with using the WP plugin to do deductive verification. Even after reading the entire manual both for ACSL and WP, we have found annotating a simple program to be a daunting task. To add to this, it has been difficult for us to find external sources that can help with the problems we encounter. More than once we have encountered issues that we could not solve on our own (including what we could find in manuals and on other sites). It is hard to imagine Frama-C seeing widespread use in the industry unless serious measures are taken to improve the documentation and other sources of help.

Loop Invariants

When using the WP plugin to do deductive verification on code that has loops, the creation of loop invariants is of utmost importance. For these invariants it must hold that:

1. The invariant must hold before the first iteration of the loop.
2. If the invariant held before an iteration of the loop, it must also hold after that iteration[2].

These rules are fairly easy to understand and creating loop invariants might seem easy. However, while some of the WP documentation we have found does say to create loop invariants (such as [35]), it understates just how thorough these invariants have to be. Our group has not had much experience with creating loop invariants before. Therefore the first loop invariants we created were trivial. This meant that even though they worked, they could not be used to prove any interesting properties of the loops or the functions that contained these loops. It was not until we saw the annotations documented in section 4.4.3 that we understood how thorough our loop invariants had to be.

It would have been nice if the documentation itself put more emphasis on loop invariants since they are so important when using WP.

Function Complexity

Looking at the function described in section 4.4.3 it is worth noting how much work has gone into annotating a very simple function. The annotations themselves make up more lines than the actual function. This speaks to the complexity of annotating even simple functions so that they can be analyzed by WP. We have also seen that it can be somewhat easy to fall into the trap of writing annotations that can be verified but say nothing interesting. Especially as the complexity of the annotated functions increases, it seems easier to write generic function contracts instead of dealing with the larger task of writing useful annotations. We have also encountered a lot of problems that were not easily solvable while annotating functions even if those functions were not of high complexity. It seems that it requires a very solid understanding of ACSL as well as the WP plugin to use the tool to a satisfying degree.

Accessing Contents of Char Arrays

When writing the code while annotating it, it can be a good idea to consider different solutions for the same problem. While learning to use the WP plugin we worked on some examples with arrays of characters. To iterate through an array of characters two obvious solutions exist: array indexing and pointer arithmetic. We found that incrementing the value of the base pointer to iterate through the array was much more difficult to express in ACSL than indexing.

Dealing with Errors

When annotating a function with ACSL and running WP on it, the errors that WP gives might be due to other things than one would think. We experienced multiple times that WP said that some of our ensures clauses could not be proven. While it is easy to think that such an error is due to an ensures clause being wrong, we found that it was often something else that was wrong. Most of the time the ensures errors were actually due to faulty or lacking loop invariants or requires clauses. We found that trying to improve these clauses (loop invariants and requires) before trying to redo an ensures clause often led to the best results. Therefore we recommend using this approach if the user is somewhat sure of their ensures clauses.

Furthermore, the error messages given by the WP plugin can be difficult to understand. The error messages are not very informative and are often expressed in a way similar to the following example: “function_assigns_part3: Timeout”. Note that this example is an error concerning an assigns clause. In general,

[2]This rule does not apply if the loop does not terminate normally e.g. through a `break` or `return` statement.

this is somewhat understandable when the error messages only concern the functionality that we have written. However, when utilizing standard library functions such as `strcmp()`, WP will associate some of the error messages with the standard library functions and the error messages generated for those are not always intuitive. E.g. for a function printBlankLine that uses `printf()` we received the error message seen in listing 4.12.

```
[wp] [Alt-Ergo 2.2.0] Goal typed\_printBlankLine\_call\_printf\_va\_1\_requires : Timeout
↪ (10s)
```

Listing 4.12: WP's error message when calling `printf()`.

When trying to annotate the program it is often challenging to understand why a certain annotation does not hold. This is in part because the error messages do not give a lot of information. Some information that would be very nice to have would be a counter-example as to why a certain ensures clause, loop invariant, etc. does not hold. We encountered a great deal of frustration while trying to express certain functionality, and not being able to understand why the property does not hold. Being provided with counter-examples could have drastically eased the process.

Chapter 5

OpenTitan Security Analysis

OpenTitan is a new (unfinished) collaborative open-source secure chip design project that acts as a Root of Trust (RoT). [38] defines RoT as "A system element that provides services, including verification of system, software & data integrity and confidentiality, and data (software and information) integrity attestation between other trusted devices in a system or network." The project is inspired by Google's custom made RoT chip Titan, which Google uses to verify that data-centers boot from a safe state with verified code [39]. OpenTitan is being built by a partnership consisting of Google, lowRISC, and several other companies [40]. According to [41], OpenTitan is motivated by tech giants' and governments' rising need for safety against hostile nation states that try to infiltrate software and hardware manufacturing supply chains to carry out long-term espionage.

The OpenTitan chip can be seen as a microcontroller with its own boot sequence, kernel, and application layer. The chip can be implemented into a system as a standalone security module or as an integrated part of the CPU. We will investigate the security of the OpenTitan chip where it is implemented to ensure secure boot as an integrated part of the host's CPU. The host may be a smartphone device, a server in a data center, etc.

OpenTitan acts as a RoT that is inherently trusted to do safety critical operations such as key management and boot code verification for the host during secure boot. The software and hardware of the OpenTitan chip effectively act as a single point of failure for the security of the entire host. The most fundamental part of the chip is its own secure boot. As the entire OpenTitan stack, and thus the host is reliant on the security of the OpenTitan boot. For this reason, we want to analyze the security of the initial OpenTitan boot process. The security analysis will result in security goals about code validation (hashing and signing), key infrastructure, the privilege hierarchy, etc. These goals and their respective security mechanisms will reveal critical parts of the boot software of which we will, in future work, verify the security using the formal verification techniques such as those explored in previous chapters.

5.1 Cryptography

The purpose of this section is to introduce cryptography. The cryptographic concepts are necessary to define before analyzing OpenTitan as OpenTitan is, simply put, a microcontroller that performs cryptographic operations.

Encryption

Encryption is the process of encoding a message, using a key and an encryption algorithm, so it can only be read by entities that have a corresponding key [42]. Decryption is the reverse process, i.e. transforming the encrypted back to the original message, using a key and a decryption algorithm.

Asymmetric Encryption

Asymmetric key pairs are defined as a public key and a private key. When asymmetric encryption is used to ensure that the receiver (owner of the keys) is the only one who can read an encrypted message the encryption key is denoted as the public key and the decryption key as the private key [43]. This is not the case for the asymmetric encryption in OpenTitan that we discuss in this report. Instead, asymmetric encryption is used to ensure that only the sender (owner of the keys) is able to encrypt a message that the receiver can then decrypt and verify against the original message. As such, we define the private key as being the encryption key and the public key as being the decryption key. The public key is mathematically related to the private key. Asymmetric algorithms are designed so that it **should be** infeasible to compute the private key from the public key. It is, however, considered common practice to handle the public key securely, e.g. by restricting access to it.

An asymmetric encryption system defines two functions: *Encrypt* and *Decrypt* [44]. *Encrypt* takes the message and private key as input and outputs the encrypted message (some bit string value). *Decrypt* takes the encrypted message and public key as input and outputs the original message.

The benefit of asymmetric encryption is that the private key never needs to be shared. This makes the key transmission less at risk. The disadvantage is that the mathematical complexity of asymmetric encryption algorithms makes them less efficient. The keys are big and are computationally heavy to generate compared to their symmetric counterpart. In practice, asymmetric keys can only encrypt short messages since the key size would otherwise have to be unfeasibly big [45].

Symmetric Encryption

A symmetric key can be considered both the “private key” and “public key”. Symmetric encryption requires only one key, i.e. the key works for both encryption and decryption [46].

Symmetric encryption algorithms are commonly much simpler and more efficient than asymmetric encryption algorithms. As a result, symmetric encryption can feasibly be used for the encryption of larger messages [45].

Signature

A signature, also known as a digital signature, can be considered proof of owning a specific secret key. Asymmetric encryption and signing are similar but distinct. A signing algorithm commonly uses asymmetric keys to implement authentication. The sender uses the private key, which is not shared, to write a signature to a message. The signature is bundled with the original message during transmission to the receiver, i.e. the message is not a secret. The receiver uses the public key to verify the integrity of the message and the authenticity of the sender [47].

A signature algorithm defines two functions: *Sign* and *Verify* [44]. *Sign* takes the message and private key as input and outputs a signature (some bit string value in a given space). *Verify* takes the message, signature, and public key as input and outputs *true* or *false* (indicating whether the signature matches the public key and message).

A typical signing algorithm first hashes the message, then pads the hash, and then finally signs the padded hash [47].

Hybrid Encryption

Hybrid encryption implements both asymmetric encryption and symmetric encryption [45]. The idea is to have “safe” key transmission and benefit from the efficiency of symmetric encryption.

Assume that the receiver has received the sender’s public key. First, a symmetric session key is generated. The message is encrypted by a symmetric encryption algorithm taking the session key as input. The session key is encrypted by an asymmetric encryption algorithm using the sender’s private key. The encrypted message and the encrypted session key are then sent to the receiver.

The receiver first decrypts the session key using the sender’s public key. Next, the receiver decrypts the message using the session key.

HMAC

HMAC, also known as hash-based message authentication code, is a form of cryptographic code that is computed from a message and a private key as input. HMAC computation is similar to signature creation. It provides both message authentication and data integrity. HMAC is, simply put, an intricate procedure for hashing (typically SHA-2 or SHA-3), padding, and appending the key and message. It is designed to make it difficult to retrieve the key from the HMAC [48].

5.2 OpenTitan Description

This section will present a description of the OpenTitan project [40]. Note that the OpenTitan documentation does not clearly state whether the notion of Silicon Owner/Creator public and private key pairs are equivalent to the Owner Identity and Creator Identity, see appendix C for examples. We assume for the rest of this chapter that the notions are equivalent. Thus, the Creator Identity and Owner Identity are asymmetric keys used to, among others, sign and verify the signature of `mask_ROM` / `ROM_EXT` and `BL0` / `Kernel`, respectively.

5.2.1 Logical Security Model

This section is written based on [49]. The logical security model of OpenTitan consists of 4 entities: Silicon Creator, Silicon Owner, Application Provider, and End User. The first three entities are responsible for creating an OpenTitan compliant chip, and the End User is not documented. Note that ROM, as depicted on Fig. 5.2.1, is referred to as `mask_ROM`.

The Silicon Creator manufactures, packages, provisions (supplies the software for `mask_ROM` and `ROM_EXT` stages), and tests the chip. As proof that this has happened, the Silicon Creator assigns the chip with an asymmetric cryptographic key, known as a "Creator Identity". The Silicon Owner is responsible for performing Ownership Assignment and functional provisioning. Ownership Assignment is the process of assigning a Silicon Owner’s "Owner Identity" to the chip. The Silicon Owner of the chip can change through a specific process called "Ownership Transfer" (cf. section 5.2.2). Functional provisioning is concerned with the Silicon Owner deploying a software stack to the chip, such as a `Kernel` and a boot loader (boot loader stages). Lastly, the Application Provider provides applications that utilize the software stack provided by the Silicon Owner.

The software stack on the chip consists of stages and the different entities (i.e. Silicon Creator, Silicon Owner, and Application Provider) own different stages. The notion of ownership of a stage is defined as having the private key used for signing a stage (e.g. Silicon Creator signs `mask_ROM` and `ROM_EXT` with the private key of the Creator Identity). The different stages are: `App{N}` (N is the number of applications), `Kernel`, `BL{N}` (N is the number of boot loader stages), `ROM_EXT`, and `mask_ROM`. The applications (`App{N}`) run in the least privileged state of the RISC-V processor, namely User Mode. The other stages run in Machine Mode, which is the highest privileged state. The applications are owned by the Application Provider. `Kernel` and `BL{N}` are owned by the Silicon Owner, and the `ROM_EXT` and `mask_ROM` are owned by the Silicon Creator. This is summarized in Fig. 5.2.1.

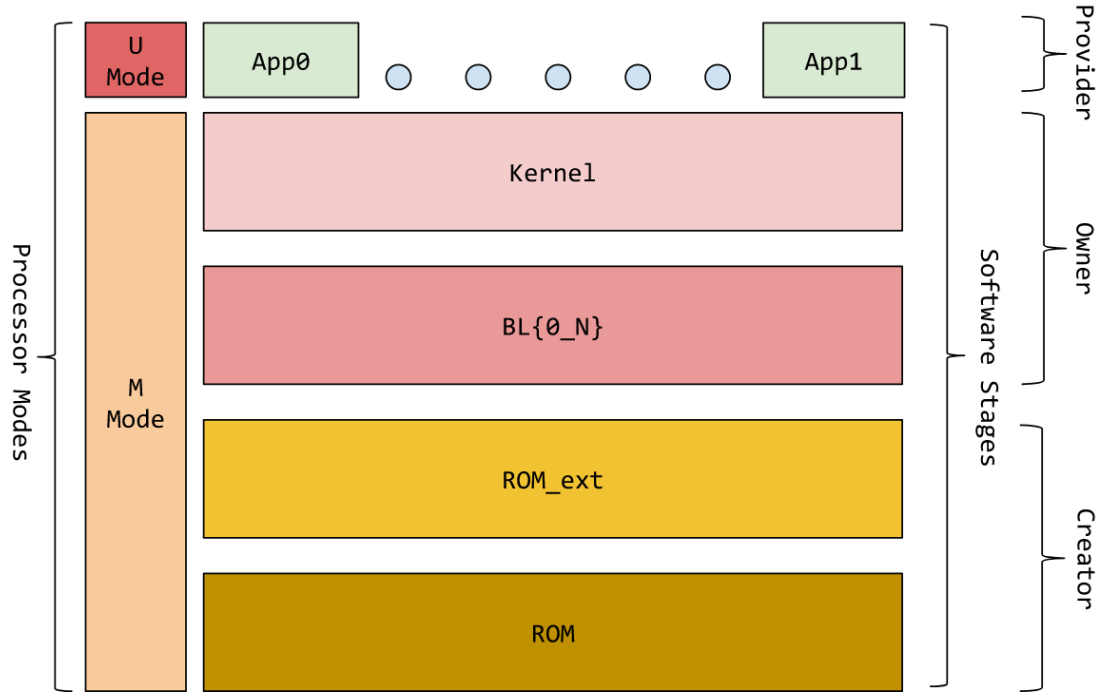


Figure 5.2.1: The software stack of the OpenTitan chip when the entities are distinct. Showing the stages and their processor modes (privilege state) and the (distinct) entities owning them (Provider, Owner, and Creator) [49].

`mask_ROM` is responsible for validating and transferring execution to `ROM_EXT`. `ROM_EXT` is responsible for validating and transferring execution to the initial boot loader (`BL0`). An additional role of `mask_ROM` and `ROM_EXT` is to lock out specific parts of the chip that are not needed later, i.e. make them immutable, unreadable, or both. Both `mask_ROM` and `ROM_EXT` should only execute once per system restart. OpenTitan refers to all the boot loader stages `BL0..N` as `BL0`. We will use the same terminology for the rest of this chapter.

The `BL0` is responsible for validating the `Kernel` and transferring execution to it. The `BL0` does also perform lock outs. Functionality such as key management and general chip control is provided by the `Kernel`. The `Kernel` is also responsible for validating, executing, and scheduling the Application Provider's applications. Lastly, the `Kernel` is also responsible for providing isolation between the Application Provider's applications.

Regarding the execution of the software, there are two assumptions. The first is that from the execution of `mask_ROM` until the `Kernel` is executed, the execution is one way. This means that once a stage is done executing and is left, it is not possible to go back to and re-execute the stage. For example, it is not possible to re-execute the software in `mask_ROM` once the execution has been transferred to `ROM_EXT`. The other assumption is, that when reaching `Kernel` the execution can become dynamic instead of one way. This means that it is possible to switch between executing in the `Kernel` and executing the Application Provider's applications. Which allows for switching between User Mode and Machine Mode (i.e. switch between different privilege levels).

5.2.2 Ownership Transfer

This section is based on [50]. The motivation for ownership transfer is to install a new Silicon Owner and all its implementation specifics (boot loader, kernel, Owner Identity, etc.). There are two types of ownership transfer, a transfer from a Silicon Creator to a Silicon Owner, and from a Silicon Owner to another Silicon Owner. Ownership transfer can also be disabled which means that once an initial owner is assigned, that owner cannot change.

With regards to ownership transfer, the device can be in two different states, namely unowned and owned. When in the unowned state the device is ready to be assigned a new owner, and when in the owned state the device has an owner but it can change through the ownership transfer process. The overall steps in ownership transfer can be seen in Fig. 5.2.2. Note that if the device is unowned, then the ownership transfer process starts in the *Unowned State*, and otherwise it starts in the *Owned State*.

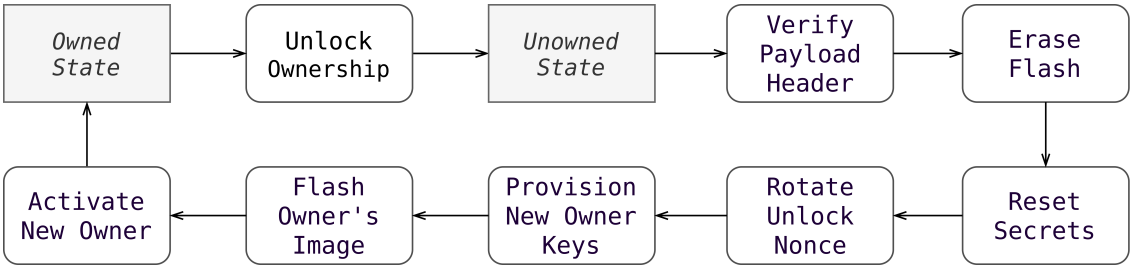


Figure 5.2.2: Ownership transfer stages of OpenTitan.

Unlock Ownership

Unlock ownership is the process of changing the device state from owned to unowned. This is done by sending a signed unlock command to the `ROM_EXT` from the `Kernel` /app layer.

The host receives, upon query, the ownership transfer nonce and `device_id` from the chip `Kernel` /app layer. The host gets the unlock command signed, by sending a command signature request to the Device Registry [1] The host then forwards the signed unlock command to the `Kernel` /app layer on the chip, which may validate the command. The `Kernel` /app layer further forwards the unlock command to the `ROM_EXT`. The `ROM_EXT` verifies the command and triggers the unlock operation on the next boot. The unlock result is propagated back to the Device Registry when the unlock is complete.

[1] Device Registry: An external (cloud) service used for vouching for e.g. unlock commands [50].

Verify Payload Header

The OpenTitan chip receives an ownership transfer payload (e.g. containing new Silicon Owner software) from the host. The payload header includes the new Silicon Owner's key endorsement manifest, which is signed. The `ROM_EXT` verifies the manifest by verifying the signature to either the Silicon Creator Identity or Silicon Owner Identity. Essentially either the Silicon Creator or the current Silicon Owner must vouch for the new Silicon Owner.

Erase Flash

All the previous Silicon Owner code and data are cleared from the flash.

Reset Secrets

The `OwnerRootSecret` [2] is changed to a new random bit string by deterministic random bit generation (DRBG). DRBG is supported by the Cryptographically Secure Random Number Generator (CSRNG) hardware [52].

Rotate Unlock Nonce

The DBRG supplies a random 64-bit value to be used as an unlock nonce. The purpose of the unlock nonce is to protect the unlock command protocol against replay attacks [53] and thus make it possible to try again (upon failure).

Provision New Owner Keys

This step consists of the previous Silicon Owner approving the new Owner Identity public key. This step guarantees that the new Silicon Owner is a trusted entity. The new Silicon Owner is then able and allowed to load its software onto the chip.

Flash Owner's Image

The new Silicon Owner's software is written into the flash.

Activate New Owner

When the new Silicon Owner's software is booted for the first time by the `ROM_EXT`, the software sends an activate owner command to the `ROM_EXT`. The `ROM_EXT` then sets the new Silicon Owner as the Silicon Owner of the chip and concludes the ownership transfer process.

5.2.3 Key Manager and Key Derivation

This section is based on [51] and [54], and covers the essential parts related to the key manager and the derivation of keys in OpenTitan. The key manager does not store keys but implements a one-way function called `KM_DERIVE`, that derives a key or a seed for generating identities, based on the input given to it. The input required for deriving the specific keys and seeds is described in OpenTitan's key derivation scheme (seen in [54]). If the boot code is initiated with altered ROM code or hardware configuration the derived keys will not be correct. This means that the boot will fail since the boot stages are not able to validate the signature of their succeeding boot stage.

[2] OwnerRootSecret: A random value with fitting entropy provisioned at ownership transfer that is used for key derivation [51].

The key manager can be in six different states:

- Reset (entered immediately after reboot)
- Initialized
- `CreatorRootKey`
- `OwnerIntermediateKey`
- `OwnerRootKey`
- Disabled (entered after successful boot)

The rest of this section will only consider the Initialized, `CreatorRootKey`, `OwnerIntermediateKey`, and `OwnerRootKey` states. They (except the Initialized state) will be denoted as intermediate states. The intermediate states are represented by the derivation of the root key with the same name. In each intermediate state, it is possible to derive specific keys and seeds from the associated root key (and other values).

Advancing to an intermediate state is seen as a one-way checkpoint, meaning that once a key representing an intermediate state has been derived and the key manager advances its state, it cannot change back to previous states and derive the keys and seeds possible at those intermediate states.

The transitions between the intermediate states of the key manager, starting from the Initialized state, are the following:

- Initialized (`mask_ROM` stage)

  In the Initialized state, the `CreatorRootKey` is derived from the `RootKey` and other environment dependent values (life cycle state, hash of ROM, HardwareRevisionSecret, etc.). This results in advancing to the `CreatorRootKey` intermediate state.

- `CreatorRootKey` (`ROM_EXT` stage)

  In the `CreatorRootKey` intermediate state the `CreatorIdentitySeed` is derived from the `CreatorRootKey`. The `CreatorIdentity` is then derived from the `CreatorIdentitySeed`. In addition, the `OwnerIntermediateKey` is derived from the `CreatorRootKey`. This derivation advances the key manager's state to the `OwnerIntermediateKey` intermediate state.

- `OwnerIntermediateKey` (`BL0` stage)

  In the `OwnerIntermediateKey` intermediate state the `OwnerIdentitySeed` is derived from the `OwnerIntermediateKey`. The `OwnerIdentity` is then derived from the `OwnerIdentitySeed`. In addition, the `OwnerRootKey` is derived from the `OwnerIntermediateKey`. This derivation advances the key manager's state to the `OwnerRootKey` intermediate state.

- `OwnerRootKey` (`Kernel` stage)

  In the `OwnerRootKey` intermediate state the `VersionedKey`[3] is derived. The key manager can advance from the `OwnerRootKey` intermediate state to the Disabled state (this is non-revocable and requires a total reboot to transition to the Reset state).

[3] VersionedKey: No documentation about the usage of this key.

5.3 Security Analysis

Many generic definitions of security exist, however, security related to this project will be defined as "The ability of the OpenTitan chip to safely perform the initial boot process or terminate in the presence of knowledgeable and resourceful malicious entities (such as nation states) with physical access to the device and with interest in compromising the boot process of the chip and thereby the safety of the host."

The purpose of the security analysis of the OpenTitan initial boot process is to deduce what security measures OpenTitan has to have in place for the initial boot to be able to satisfy its goal of booting correctly in the presence of an adversary. The security analysis (based on information found in [55]) defines the system and the context wherein it operates. It then goes on to describe a set of security policies, goals, and mechanisms that we have defined. Security policies are abstract rules/policies that OpenTitan has to meet. Based on them, less abstract and verifiable security goals are deduced. The goals are fulfilled by security mechanisms that describe what technologies, methods, or alike are implemented to satisfy the goals.

5.3.1 System Definition

We define the system as the software and hardware parts of the OpenTitan project that constitute the initial boot process (i.e. until `BL0` is executed), consisting of the following hardware:

- CPU (Ibex RISC-V Core + Ibex RISC-V Core Wrapper)
- ROM
- SRAM
- Flash

and software:

- mask_rom_start.S
- mask_rom.c
- rom_ext_start.S
- rom_ext.c
- rom_ext_manifest.S
- rom_ext_manifest_parser.c

The focus will mostly be on the software aspect of the initial boot process, but if it is not enough to make guarantees about the safety of the initial boot process, the hardware will also be considered. In addition, the `ROM_EXT` is involved in the process of Ownership Transfer, but that aspect of the `ROM_EXT` 's functionality will not be considered.

Memory Layout

The memory consists of three different types: ROM, Flash info, and Flash [56]. A simplified memory layout can be seen in Fig. 5.3.1. Note that the memory sizes are figurative. The `creator certificate` , `owner certificate` , `BL0` , and `Kernel` are not in scope of the system (i.e. the initial boot process). An owner slot is, simplified, a data structure that stores and has stored the id and other information about the current owner and previous owners of the chip, respectively [50].

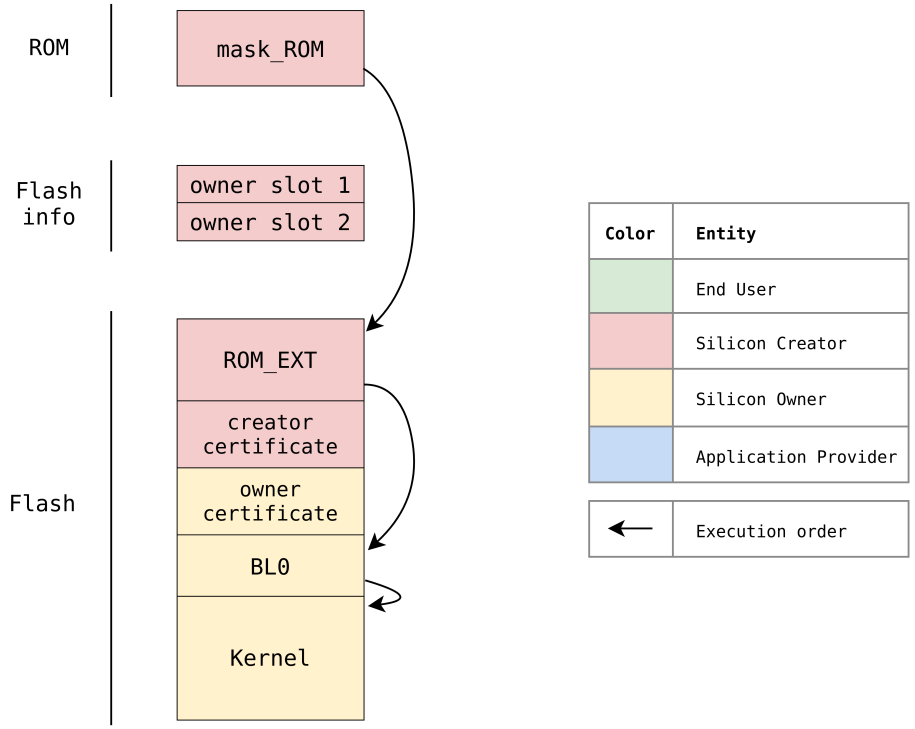


Figure 5.3.1: Memory layout of the OpenTitan boot stage code.

Flow of Execution

Our interpretation of the flow of execution is that the first software that is executed is `mask_rom_start.S` which initiates the execution of `mask_rom.c`. `mask_rom.c` starts execution of `rom_ext_start.S` which then executes `rom_ext.c`. The files regarding the ROM Extension manifest (i.e. `rom_ext_manifest.S` and `rom_ext_manifest_parser.c`) are used to facilitate the retrieval of the content of the manifest.

A visual representation of the boot stages and flow of execution can be seen in Fig. 5.3.2. Based on our system definition, we will only consider the two stages: `ROM` stage and `ROM_EXT` stage. The dotted line from `ROM_EXT` to `Kernel` indicates that it is possible to boot without a `BL0`. This claim is based on the information in [56] and [50] which, respectively, indicates that `ROM_EXT` can load either a `BL0` or a `Kernel` image, and that a `BL0` can be omitted if there is fixed flash size allocation.

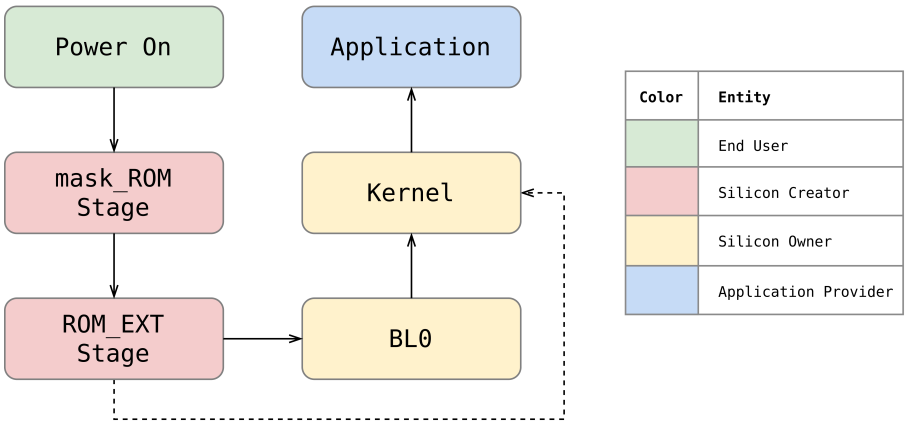


Figure 5.3.2: Boot stages of OpenTitan.

mask_ROM Stage

The OpenTitan documentation states that "the ROM is the metal mask ROM, sometimes known as Boot ROM" [56]. This is inconsistent with the GitHub repository which denotes two separate C files; `boot_rom.c` and `mask_rom.c`, indicating that these are two separate stages [57]. The `boot ROM` README states that the `boot ROM` code always is the first piece of code to be run [58]. To solve this inconsistency we contacted the developers of OpenTitan [59]. They replied that the `boot_rom.c` is temporary and will not be included in the final OpenTitan chip. The `mask_rom.c` file is not finalized and still subject to much change. They stated that the boot process will, eventually, follow the stages described in [56] (which is also subject to change). As such, this section is based on the "secure boot" documentation of OpenTitan seen in [56].

The `mask_ROM` is initiated by `Power On` and its main purpose is to validate and transfer execution to the active `ROM_EXT` slot. The initial PMP [4] region covers the entire flash region and is configured by some reset logic to be read-only and locked. The PMP region is respected by machine mode and can only be unlocked by a total system reset. It is currently undecided whether or not the initial PMP region is set by hardware or software (presumably the `mask_ROM` code) [61]. Then, all of the Static Random Access Memory (SRAM), except for retention SRAM, is cleared.

First, `mask_ROM` must determine which `ROM_EXT` slot to boot [61]. The reference to the prioritized `ROM_EXT` slot is loaded from the flash Boot Info[5]. Note that if the prioritized `ROM_EXT` slot fails to be validated or fails to boot then the `mask_ROM` can try to boot one of the other `ROM_EXT` slots [61]. For the selected `ROM_EXT` slot it is determined if the `ROM_EXT` is empty. This empty check is done by testing the presence of some magic value in the `ROM_EXT` header. If this magic value is missing the boot failure logic is executed. The `mask_ROM` verifies the signature of the `ROM_EXT` (RSA 3k signature stored in the `ROM_EXT manifest` [6] [62]) using the Creator Identity public key. This is to validate that the `ROM_EXT` image has been created by the Silicon Creator for this chip. If validation fails the boot failure logic is executed.

[4]Physical Memory Protection (PMP): This feature is supported by the processor and allows for control of physical memory [60].
[5]Flash Boot Info: Have not found any documentation of this.
[6]ROM_EXT manifest: a data structure which stores e.g. the ROM_EXT image and the signature of the hash of the image.

Then, mask_ROM performs system state measurements (no documentation of system state measurements) and derive the CreatorRootKey from the key manager [63]. The key manager is then locked down preventing further root key requests [61]. The CreatorRootKey is passed on to the ROM_EXT later during execution transfer [61]. The ROM_EXT manifest specifies peripheral lockdown info [62]. The mask_ROM locks down peripherals as requested in the ROM_EXT manifest [61]. A new PMP region is created covering the ROM_EXT image restricting the address space access to read, execute, and to otherwise be locked.

Lastly, the mask_ROM is now ready to transfer execution to ROM_EXT and does so by executing a jump to the entry point specified in the ROM_EXT manifest . If the ROM_EXT fails to boot, the execution will return to the mask_ROM and it will handle the failure by performing the boot failure policy described in the ownership blob (no documentation of ownership blob).

ROM_EXT Stage

The execution of ROM_EXT is initiated from mask_ROM . The first action performed in the ROM_EXT stage is to check if a boot service request (i.e. UPDATE_FIRMWARE , REFRESH_ATTESTATION , TRANSFER_OWNERSHIP , or UNLOCK_OWNERSHIP) has been made. If a request has been made, the boot service is performed. The second step is to determine which Silicon Owner image (i.e. BL0 / Kernel image) to execute. This is done by reading the Boot Info page [7]. A hash of the image is computed and compared to the hash given in the Silicon Owner Manifest (no documentation of the Silicon Owner Manifest). The hash in the manifest is verified by verifying its signature using the Owner Identity public key. If one of these two checks fails, the boot process is aborted. If the checks succeeded, a structure called a Boot Information Structure is placed at the beginning of SRAM. It contains information about the boot process e.g. which ROM_EXT slot was used (i.e. slot a or slot b). Lastly, a PMP region is created which covers the executable region of the Silicon Owner software. The region is restricted to read and execute and is locked. The execution then jumps to the entry point of the Silicon Owner software.

The Input to the System

- The first boot loader (BL0).
- Physical input that may cause bit flip in memory. E.g. electrical current or laser pulse attack described in [64].

The Output of the System

- Execution of the first boot loader (BL0).

5.3.2 The Context of the System

The context of the system defines the environment the system is deployed in and the ways it can affect the system both directly, i.e. via the system's input, and indirectly. The context of the system is relevant to consider wrt. security, as the system, is affected by it. The context of the system is limited to the components that mask_ROM and ROM_EXT use during boot. The following paragraphs cover these components:

[7] Boot Info page: Data structure containing Owner Identity, Boot Policy (a data structure that contains information about the steps of the boot process [56]), etc.

Flash Controller: Contains functionality to e.g. read, program, and execute flash. It also facilitates the host to read the flash, but not to program nor execute [65]. According to [66], it is e.g. used by `mask_ROM` to read the Boot Policy, Boot Info page, and Boot Reason.

Life Cycle Controller: The life cycle controller's main purpose is to handle the initiation of life cycle transitions. A life cycle transition is a transition from e.g. the `UNOWNED` state to the `OWNED` state [67]. There are multiple different life cycle states which may differ in access rights and privileges (e.g. restricting key manager and flash controller) [68].

Key Manager: Generates (derives) keys and provides access to them from software [51]. It is e.g. used for deriving the Creator Identity public key when validating the `ROM_EXT` , during the `mask_ROM` stage [56].

Key derivation scheme: The scheme/strategy used for derivation of some keys used throughout OpenTitan, e.g. Creator Identity and Owner Identity. The scheme used in OpenTitan utilizes a symmetric key manager. Key derivation is performed during boot to derive CreatorRootKey, which is necessary for later derivations of keys that are to be used for authentication [54].

CSRNG: The cryptographically secure random number generator (CSRNG) [69] module can perform both deterministic random number generation (DRNG) and true random number generation (TRNG). This functionality is only possible if the CSRNG is used together with a secure entropy source. The CSRNG module can be used when implementing the key manager's key derivation function [54].

Entropy Source: The entropy source module uses a physical true random number generator (PTRNG) to generate random values. These random values are used as non-deterministic seeds for the CSRNG [70]. Good entropy (randomness) is important for the security of cryptographic systems as patterns in "randomness" can be exploited [71].

AES: The advanced encryption standard (AES) module is a cryptographic accelerator, i.e. a module that is designed to be efficient for symmetric encryption and decryption [72]. The AES documentation does not specify what relies upon it, but it can be used by the key manager [51] and the CSRNG [69].

HMAC (SHA256): The HMAC module is designed to compute the hash (HMAC code) of a message and secret key (see the definition section 5.1). It is used during the `mask_ROM` stage to calculate the hash of the `ROM_EXT` image using, as a minimum, the HMAC-SHA256 hash algorithm and a 256-bit private key [66] [73].

OTBN: The OpenTitan Big Number Accelerator (OTBN) is a processor that performs asymmetric cryptographic operations, that is, it executes asymmetric cryptographic code [74]. It is used during the `mask_ROM` stage to verify the signature of the `ROM_EXT` image using the RSA-3072 algorithm, as a minimum, and a public key [66] [73].

PMP: Physical Memory Protection (PMP) is a hardware unit that is used to apply restrictions regarding write, read, and execute, to memory regions. PMP is not specifically documented but is referenced throughout the OpenTitan documentation [66] [75]. PMP restrictions may apply to both flash and ROM memory.

SRAM Controller: The SRAM controller is responsible for reporting SRAM data integrity failures and performing SRAM memory scrambling[8] [77].

Alert Handler: The alert handler is a module that waits for and collects alerts (interrupts that indicate security threats) from all the other components of the chip [78]. The alerts are converted to interrupts that the processor can handle. If the processor cannot be trusted, the alert handler can itself act on the security threats (i.e. the alerts).

SBHS: The OpenTitan Secure Boot Hardware Support is not documented at this point in time, but is referenced to as a mechanism used to disallow transfer of execution to `ROM_EXT` from `ROM` without first validating the `ROM_EXT` [56].

5.3.3 Security Policies

Security Policies are abstract rules that the system should satisfy to be secure. They are not concrete enough to be verified and are therefore broken further down into verifiable security goals. The security policies below are not exhaustive for the entire OpenTitan project but limited to those relevant for the system defined in section 5.3.1.

- P1: It should only be possible to execute code that has been validated (authenticity/integrity).
- P2: All boot stages must only succeed in validating the succeeding boot stage if the environment that the boot was initiated from is safe.
- P3: Cryptography keys and other secrets must not be leaked.
- P4: There is a privilege hierarchy that is respected (i.e. access rights: read, write, and execute).

5.3.4 Security Goals

Security goals are verifiable statements about the system, that we want to verify in future work. They are broken down into security mechanisms which together ensure the fulfillment of the security goals. We only consider the goals which are related to our definition of the system, which means that there is potentially other than the mentioned goals that are used to satisfy the given policy.

P1: It should only be possible to execute code that has been validated (authenticity/integrity)

- G1: The hash of the `ROM_EXT` image and the signature of the hash must be validated by `mask_ROM` before it is executed to ensure authenticity and integrity of the image.
- G2: The hash of the `BL0` image and the signature of the hash must be validated by `ROM_EXT` before it is executed to ensure authenticity and integrity of the image.
- G3: The hash of the `ROM_EXT` image must be signed by the Silicon Creator. The hash and signature must not be changed.
- G4: The hash of the `BL0` image must be signed by the Silicon Owner. The hash and signature must only be changed as a result of `BL0` image overwrite during ownership transfer.

[8] Memory scrambling: A security mechanism turning main memory code into scrambled code to prevent analysis of and attacks on remanent data [76].

P2: All boot stages must only succeed in validating the succeeding boot stage if the environment that the boot was initiated from is safe.

- G5: `mask_ROM` must validate `ROM_EXT` using a key that is dependent on the environment. If the environment is not as expected the validation must not succeed.
- G6: `ROM_EXT` must validate `BL0` using a key that is dependent on the environment. If the environment is not as expected the validation must not succeed.

P3: Cryptography keys and other secrets must not be leaked.

- G7: There must be a dedicated hardware component to manage all cryptography keys.
- G8: Only intended receivers are able to receive a correct key.
- G9: The location and value of secret information in memory should not be persistent after its respective stage(s) has terminated.

P4: There is a privilege hierarchy that is respected (i.e. access rights: read, write, and execute)

- G10: Only software with write access to some memory section may modify it.
- G11: Only software with read access to some memory section may read it.
- G12: Only software with execute access to some memory section may execute it.
- G13: Previously executed boot stage(s) may not be reexecuted by downstream software (i.e. execution is one way).
- G14: Previously executed boot stage code may not be modified by downstream software.

5.3.5 Security Mechanisms

Security mechanisms are implementation specific choices that help to fulfill security goals. In this case, it is what the OpenTitan developers have implemented to fulfill the security goals we have defined for the OpenTitan project.

We have decided to not consider all the security mechanisms to satisfy all the security goals. The choice of which goals to consider depended both on the ease of fulfilling them, but also on their relevance to the system we are considering (i.e. are the goals related to the `ROM` and `ROM_EXT` stages). This means that the goals which are not highly relevant but we evaluate to be easy to fulfill are still considered. It also means that highly relevant goals that we evaluate to be very difficult to fulfill are not considered, but left for future work.

G1: The hash of the `ROM_EXT` image and the signature of the hash must be validated by `mask_ROM` before it is executed to ensure authenticity and integrity of the image.

- HMAC: Computes the HMAC hash of the `ROM_EXT` image using the HMAC-SHA256 hash algorithm and a 256-bit private key [73].
- OTBN: Verifies the signature of the hash of the `ROM_EXT` image using the RSA-3072 algorithm [73] and the Creator Identity public key.
- Creator Identity public key: Used to verify the signature of the hash of the `ROM_EXT` image.

- OpenTitan Secure Boot HW Support: It enforces that the `ROM_EXT` must be validated before leaving the `mask_ROM` stage.

G2: The hash of the `BL0` image and the signature of the hash must be validated by `ROM_EXT` before it is executed to ensure authenticity and integrity of the image.

- HMAC: Computes the hash of the `BL0` image using the HMAC-SHA256 hash algorithm and a 256-bit private key [73].
- OTBN: Verifies the signature of the hash of the `BL0` image using the RSA-3072 algorithm and the Owner Identity public key [73].
- Owner Identity public key: Used to verify the signature of the hash of the `BL0` image.

G3: The hash of the `ROM_EXT` image must be signed by the Silicon Creator. The hash and signature must not be changed.

- Creator Identity private key: Silicon Creator use the RSA-3072 algorithm and their private key to sign the `ROM_EXT` image hash, which is computed using the HMAC-SHA256 algorithm.
- PMP: `mask_ROM` creates a PMP region that disallows writes to the entire `ROM_EXT` image including the hash and signature [56].

G4: The hash of the `BL0` image must be signed by the Silicon Owner. The hash and signature must only be changed as a result of `BL0` image overwrite during ownership transfer.

- Owner Identity private key: Silicon Owner uses the RSA-3072 algorithm and the private key of the Owner Identity to sign the `BL0` image hash, which is computed using the HMAC-SHA256 algorithm.
- PMP: `ROM_EXT` creates a PMP region that disallows writes to all of the Silicon Owner code, thus including, the `BL0` image, hash, and signature.

G5: `mask_ROM` must validate `ROM_EXT` using a key that is dependent on the environment. If the environment is not as expected the validation must not succeed.

- Key Derivation Scheme: `mask_ROM` verifies the `ROM_EXT` using the Creator Identity public key. The Creator Identity is derived from the `CreatorRootKey` which is derived from the underlying `RootKey`, `HardwareRevisionSecret`, life cycle state, debug mode state, etc. This means, that the verification of `ROM_EXT` only succeeds if the environment is safe.
- Key Manager: The Key Manager is a dedicated hardware component that derives the `CreatorRootKey` and `CreatorIdentity`.
- OTBN: Verifies the signature of the `ROM_EXT` hash using the RSA-3072 algorithm and the Creator Identity public key [73].

G6: `ROM_EXT` must validate `BL0` using a key that is dependent on the environment. If the environment is not as expected the validation must not succeed.

- Key Derivation Scheme: `ROM_EXT` verifies the `BL0` using the Owner Identity public key. The Owner Identity is derived from the `OwnerIntermediateKey` which is derived from the underlying `CreatorRootKey` and `OwnerRootSecret`, etc. [54]. This means, that the verification of `BL0` only succeeds if the environment is safe.

- Key Manager: The Key Manager is a dedicated hardware component that derives the `OwnerIntermediateKey`, `OwnerIdentity`, and `CreatorRootKey`.
- OTBN: Verifies the signature of the `BL0` hash using the RSA-3072 algorithm and the Owner Identity public key [73].

G7: There must be a dedicated hardware component to manage all cryptography keys.

- Key Manager: The key manager is a dedicated hardware component that derives identities and keys. The key manager does not directly store keys nor receive requests for reading identities and keys. Instead, it receives requests for deriving keys and identities.

  The key manager hides several seed values (e.g. RootKey) from software through hardware design. The key manager is disabled during the kernel execution and is first enabled again during reboot.

G8: Only intended receivers are able to receive a correct key.

- Key Derivation Scheme: The key derivation scheme of OpenTitan is designed such that it should only be possible for intended entities to supply the correct input to the key derivation function.

G9: The location and value of secret information in memory should not be persistent after its respective stage(s) has terminated.

- SRAM Controller: Scrambling of SRAM data and addresses is always enabled. The SRAM controller is supplied with a fresh scrambling key by the software after a system reset [77].
- Clearing of software registers and clearing and/or scrambling of flash to e.g. erase the private key of the Creator Identity after the `ROM_EXT` stage [54].

G10, G11, G12: Only software with write/read/execute access to some memory section may modify/read/execute it.

- PMP: To create a PMP region that restricts access to and modification of memory sections.

5.3.6 Threat Model

The system is defined as the software and hardware parts of the OpenTitan project that constitute the initial boot process until `BL0` is executed (as mentioned in section 5.3.1). Therefore, we will only consider the threats that are related to the initial boot process.

While the focus of this report is on the verification of software, we will still consider safety threats involving hardware/physical attacks. The reason being that (some) physical attacks on hardware may be simulated and verified in software with Frama-C as presented in [64]. Additionally, the hardware could be modeled in UPPAAL with inspiration drawn from METAMOC [79].

Before defining the threat model for the system we have some assumptions.

- The host and chip boots concurrently. This means that neither the host nor the chip can be accessed via the Internet during boot.
- Physical attacks can be either passive or active. Passive physical attacks consist of monitoring e.g. the electrical power outputs of the hardware (power-monitoring attack) or the electromagnetic radiation from the hardware (electromagnetic attack) [80]. Active physical attacks can be done by e.g.

shooting laser pulses or electrical current onto memory that then can modify assembly code and values [64]. Additionally, we consider the following as active physical attacks: reading directly from memory such as flash, deploying new malicious software on the hardware, and replacing hardware modules with malicious hardware modules.

- Memory safety bugs such as buffer overflow and null pointer dereference pose a security threat. The OpenTitan software is mainly written in C and C++ where the user is responsible for memory management and is thus potentially memory unsafe. We will consider memory safety bugs as they, even though these bugs may be unlikely in a safety critical system like OpenTitan, may occur and be exploited by attackers.
- The Silicon Creator, Silicon Owner, and Application Provider are all trustworthy. This assumption is made as otherwise the possible attacks to consider are too comprehensive.
- The attackers (malicious entities) are knowledgeable and resourceful (such as nation states) with an interest in compromising the safety of the Root of Trust which the host relies on. The attacker has physical access to the OpenTitan chip at some point in time. Hobbyists and less professional attackers are not considered as we evaluate that attacks from them are less likely considering the complexity and comprehensiveness of the system's security.

We use the STRIDE model [55] for the threat model analysis. The STRIDE model is used to consider the different aspects a system could be compromised. These are Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege.

Spoofing: Spoofing attacks are attacks where a malicious entity acts as another trusted entity. An example of this could be that a malicious entity could connect their own HMAC module to the OpenTitan chip that would spoof on HMAC requests during boot. The malicious HMAC could then send false hashes that could allow tampered code (`mask_ROM` , `ROM_EXT` , and `BL0`) to pass the verification check and be executed. If the security of the host depends on the OpenTitan chip's Root of Trust, the malicious entity could then compromise the security of the host.

The host sends a payload with the new `BL0` and `Kernel` during ownership transfer. This payload is signed with the new Owner Identity private key. A man-in-the-middle attack may intercept the payload from the Silicon Owner to the host or from the host to the chip. The attacker may then try a Known-plaintext attack which is an attack where access to both the message and the encryption of the message can be exploited to deduce secret information such as private keys. The attacker may then deduce the new Owner Identity private key based on the payload message (that is written in plaintext) and the signature, potentially leading to an elevation of privilege that could compromise the boot code.

Tampering: A malicious entity could be able to tamper with the code and/or data on an OpenTitan chip, e.g. via a physical attack. A tampering attack could be able to change variables' values which could cause software errors. It could also be possible to flip the access bits of PMP regions. The consequence of the latter could be that regions of memory get read, write, and execution rights, and if a malicious entity can move the program counter to that region, it could get its own code executed, which could compromise the host. For a tampering attack to succeed, the attacker has to modify the hash and signature to comply with the modified image, for integrity and authenticity verification to succeed (which is enforced by SBHS). This is highly unlikely to succeed. To circumvent the hashing, the attacker could tamper with the code/data after the hashing has been executed.

A memory safety bug such as a buffer overflow makes it possible for the attacker to change data or machine code [81]. This could lead to an information disclosure attack (described below), and elevation of privilege. An elevation of privilege attack could occur by the malicious entity overwriting functionality used e.g. for validating signatures.

Repudiation: A repudiation attack can occur when a malicious actor can make untrackable (unlogged) actions/changes. All code in OpenTitan requires a signature, which must be tied to a valid entity, in order to be executed. Therefore, such attacks are unlikely and we will not investigate repudiation attacks any further.

Information disclosure: An information disclosure attack is an attack where malicious entities acquire disclosed information that can be used for their benefits. This type of attack is possible if data is not correctly protected and thus accessible by malicious entities, e.g. a memory safety bug causing a faulty PMP region, faulty hiding of data from software, or faulty erasing data (data remanence [80]). The malicious entity could then e.g. read, write, and/or execute a restricted piece of memory. The consequence of this could be that the malicious entity acquires information that could elevate its privilege or allow it to execute its own code on the OpenTitan chip. This would compromise the host.

A memory safety bug such as a buffer overflow could allow a malicious entity to get its own code executed on the OpenTitan chip by e.g. overriding return addresses or function pointers. The malicious code could then output variables' values, providing the malicious entity with information about the internal state of the chip, which are by standard not readable. In addition, the malicious entity could also have its own code call functions on the OpenTitan chip and time the function calls (timing attack [80]), which could provide information about the length of cryptography keys and e.g. used hash functions.

The documentation states that "The `CreatorIdentitySeed` and the private portion of the Creator Identity shall be cleared before the ROM Extension hands over execution to the Silicon Owner first boot stage" [54]. If this clearing does not occur during `ROM_EXT` due to e.g. a memory safety bug a malicious entity could get access to the `CreatorIdentitySeed` and private portion of the Creator Identity. This could lead to the entity being able to elevate its privilege to that of the Silicon Creator.

The documentation also states that "Hardware secrets stored in OTP and flash shall be scrambled to increase the difficulty of physical attacks" [54] (OTP is defined as one time programmable memory) and thereby mitigate information disclosure. A memory safety bug during scrambling makes these secrets vulnerable for reading by a malicious entity with physical access to the flash and OTP.

Denial of service (DOS): A DOS attack is an attack that makes functionality unavailable rather than destroying or changing it. As mentioned in the first assumption above, the host nor the chip have an internet connection and are thus not susceptible to common internet-based DOS attacks during boot. A DOS attack could be achieved by a physical attack. We do however not consider this a threat as the attacker instead could choose to destroy the OpenTitan chip (or parts of it) which would have a greater consequence.

Elevation of privilege: An elevation of privilege attack can be described as a malicious entity that acquires access to otherwise restricted actions. This can be achieved through e.g. leaked secrets (passwords, keys, etc.) or software errors in the authentication process. An information disclosure attack can for example lead to access of secret keys, such as the private key of the Creator Identity or Owner Identity. The Creator

Identity and Owner Identity private keys can be used to sign new malicious boot code (`mask_ROM` and `ROM_EXT`) and boot loader (`BL0`) stages. This would make it possible to execute the new malicious boot code and thereby compromise the host.

Chapter 6

Discussion

In this chapter, we describe our overall reflections on the project. This is done as the last evaluation of the main points of our work throughout the semester.

6.1 Frama-C Versus UPPAAL

This section discusses the differences between Frama-C and UPPAAL in terms of their problem domains as well as a general comparison of the two tools, based on our experiences with them.

6.1.1 Application Domains

UPPAAL can be used to formally verify the properties of a design. Verifying a concrete implementation, while possible, is not ideal. This is because a UPPAAL model will be separate from the actual piece of code that it is modeling. E.g. it can verify properties of a communication protocol, but modeling the code (and hardware) would probably be an unfeasible task and likely suffer from state-space explosion. Frama-C on the other hand can be used to formally verify properties of C source code. This means that Frama-C does not suffer from the modeling gap in the same way that UPPAAL does.

One thing that Frama-C and UPPAAL have in common is that they are both formal methods. For UPPAAL this means that if the formulae in UPPAAL are correct and satisfied, the model always behaves as expressed by the formulae. However, the model may still be an incorrect model of the system being modeled. For Frama-C with the WP plugin, it means that if the annotations are correct and satisfied, then the code behaves as expressed by the annotations. Regarding the Eva plugin, if a piece of code does generate any alarms, then no RTEs can happen when executing it. However, Eva will also sometimes give false alarms, some of which are disregarded when increasing Eva's precision, as mentioned in section 4.2.

Verifiable Properties

UPPAAL can formally verify safety, liveness, and reachability properties of a model of a system (i.e. a network of timed automata). UPPAAL calculates a formal proof for the satisfiability of a property as being either fulfilled or unfulfilled. If UPPAAL query options like over- or under-approximation are used, then these same guarantees may not apply.

Queries in UPPAAL SMC are used to evaluate bounded properties. UPPAAL SMC can estimate the probability that a certain logical property is satisfied, estimate the value of model variables, and run simulations on the model variables. The output of these queries are not limited to the discrete value but may also include confidence interval, probability distribution, frequency histogram, etc.

Frama-C can verify the absence of RTEs through the Eva and WP plugins. WP can formally verify pre- and post-conditions of functions as well as verifying the value of variables, the validity of pointers, and

properties of loops, such as loop invariants. Because Eva creates an over-approximation, it is guaranteed to catch all RTEs, but there may be some additional false positives as well. Eva can also evaluate the values of variables at the end of every function covered by the analysis.

There are obvious parallels between UPPAAL and UPPAAL SMC in their approach to representing the model and expressing properties. Similarly, Eva and WP, with the help of the RTE plugin, can both analyze the code directly and can both identify RTEs. UPPAAL and WP share some similarities in the properties that they can verify (excluding RTEs). We think that most (if not all) WP annotations can be verified in UPPAAL. As an example, WP pre- and post-conditions for a function could be represented as two queries in UPPAAL for a CFG-like model. Firstly, query `A<> FunctionCall.pre` specifies that all traces leads to a location `pre` where the precondition is satisfied. Secondly, query `FunctionCall.pre -> FunctionCall.post` specifies that location `pre` will always lead to location `post` eventually, where the postcondition is satisfied. UPPAAL would, however, probably suffer from seemingly endless termination time for the queries if the model included everything written in the code. WP has the advantage of being able to work directly on the code. Eva, WP, UPPAAL, and UPPAAL SMC are all different in their approach to verification and provide distinct guarantees and results. None of the tools can realistically provide the same property verification guarantees on their own as a combination of them can provide.

6.1.2 Learning Curve

We found it more intuitive to express properties in UPPAAL, and develop a model than using Frama-C. However, we do also have prior experiences with UPPAAL, thus, this opinion might be biased. UPPAAL also provides a GUI for the development of models which eases the process of model development for novices. Frama-C does also have a GUI, but it does not ease the process of writing annotation as much as UPPAAL's GUI eases the process of creating models. In addition, we could not get Frama-C's GUI to work on Windows (cf. section 4.5). Lastly, UPPAAL does also provide counterexamples when properties are not satisfied, which significantly eases the process of debugging. With regards to UPPAAL SMC, we have similar opinions, however, UPPAAL SMC had a bit steeper learning curve than UPPAAL, due to the theory behind UPPAAL SMC.

As expressed in section 4.5 we experienced a *very* steep learning curve with Frama-C. We evaluate the reason for this to be that Frama-C requires very strict annotations to function correctly, and they are difficult to express. In addition, the error messages of Frama-C do not always give a good idea of what the actual issue is when annotations are not satisfied. This along with the fact that Frama-C does not provide counterexamples increases the difficulty of debugging.

6.1.3 Scalability

We encountered scalability issues with UPPAAL but managed some of them by transferring to UPPAAL SMC. However, UPPAAL SMC does not provide formal proofs of properties, as it is based on simulations. In addition, the transfer to UPPAAL SMC also involved pre-processing of the UPPAAL model as described in section 3.8. However, as also described in section 3.8 the pre-processing can be automated for some scenarios. For comparison, we did not encounter any scalability issues with Frama-C. However, we did not manage to execute it on a large code base (the largest code base we ran Frama-C on was approximately 310 lines, cf. appendix B) due to the difficulty and complexity of annotating programs, as expressed in section 4.5.

6.1.4 Maintenance

Both Frama-C and UPPAAL require maintenance. Frama-C depends on annotations and UPPAAL on a model and related properties, which both have to be maintained along with the development of the system being analyzed. One could argue that the design of a system changes less often than the implementation of a system. Thus, the UPPAAL model might require maintenance less often than Frama-C. However, the maintenance might be more extensive than the one required for Frama-C.

6.2 Continued Use of UPPAAL

UPPAAL seems to be a good candidate to model and verify some parts of the design of OpenTitan. Compared to Frama-C, UPPAAL is much easier for us to use as we have more formal experience with it. However, because UPPAAL does not work by verifying the actual code of a system but rather works on a model of the system we believe that it cannot be the only tool that we use. To verify the security and correctness of some of the boot code for OpenTitan we believe that we need to supplement UPPAAL with some tool that works closely with the code. In this report, we have explored Frama-C which could be a potential candidate for this. The relative ease of use that UPPAAL provides is a big factor in why we believe that we should continue to use it going forward. Furthermore, UPPAAL is a powerful tool (cf. chapter 3) that would allow us to model parts of both the software and hardware of OpenTitan.

6.3 Continued Use of Frama-C

Having used Frama-C for the better part of two months at this point we have some reflections on its usefulness towards verifying parts of OpenTitan. In this section, we focus only on the Eva and WP plugins as these are the only ones we have actually used. Frama-C includes 28 other plugins that have different functionalities. However, we did not have time to explore all of them. Therefore, we decided to look at Eva and WP as these are some of the more common plugins that we can also see potential uses for. It is clear that the tool has potential and could be valuable for verifying parts of OpenTitan. However, the initial learning curve of WP is *very* steep. We have spent a lot of time figuring out WP and getting smaller examples to work. The available documentation from Frama-C's own website does not seem geared towards teaching new users how to use the tool. We have not felt that the documentation was adequate in a lot of cases and have had to resort to whatever else we could find. While we have no in-depth training within the field of static analysis, we have experience with formal methods in general. Even with this knowledge, we have still felt stuck at times. This does not give us the best of hope for the usefulness of the tool from a practical standpoint. If we feel that the tool is hard to use, then we doubt that the industry will invest the time it takes to learn it. Therefore it might not get adopted by the industry. While this seems pertinent to take into account, it is not the only factor to consider. As stated earlier, we recognize that the tool has strong potential for verifying C code. WP has constructs that make it suited to handle memory safety, which is relevant to us (cf. section 5.3.6). Because of this, we feel that the tool is still worth exploring going forward. If we can get to a point where we can narrow down the need-to-know for the tool to a subset that is sufficient for verifying boot code, then our future work might aid as documentation for the parts of the original documentation that we find insufficient.

6.4 Working with OpenTitan

As mentioned in the introduction to chapter 5, OpenTitan is a new and unfinished open-source project, which helps to solve an interesting and relevant problem. This makes it interesting to work with and facilitates easy access to insights into the project both regarding access to the documentation but also regarding communicating with the developers. The downside of it being a new and unfinished project is that it also introduces some problems. One problem is that the documentation is not complete and some parts contain conflicting information, which makes it difficult for newcomers to gain an insight into the project but also difficult to gain a correct insight.

We also personally experienced the documentation to change during the project, which could have the consequence that the understanding we had might be wrong. Knowing that the documentation is incomplete and is somewhat often changed also makes it difficult to evaluate whether the information is correct or is simply waited to be updated. We experienced a fairly steep learning curve due to the points mentioned above, but also that we personally do not have much experience with software and hardware operating at the level that OpenTitan does.

A benefit of OpenTitan being open-source is that it makes it possible to modify and run the actual system in a simulator since we have access to the source code. We did not do this during the project, but we know it is possible. Doing it could be useful for understanding how the different source files are used or for testing certain hypotheses e.g. about the flow of execution.

6.5 Vision of the Project

During the start of the semester, our project was throttled by a lack of vision. This is mainly because the platform that we were supposed to work with was not decided upon at the beginning of the project. This meant that we did not have a clear path forward and therefore some of our work felt out of place once we settled on a vision halfway through the project. As an example, the UPPAAL models shown in chapter 3 are of our P7 system because we found no obvious candidates for a system to model. We just knew that it would be beneficial to MAXSYT for us to reflect on creating models and verifying properties in UPPAAL. Had we had a clear vision for the project from the start we could potentially have modeled a system that would have been more appropriate. We still feel that the examples used both for UPPAAL and Frama-C are useful as they do show the tools on a more general level. However, if we had known from the start of the project that we were going to work on verifying the OpenTitan platform (which we only settled on in the last half of the project period) we could potentially have had enough time to verify examples more closely related to OpenTitan.

6.6 Difficulty of Verifying Boot Code

As seen in chapter 3 and chapter 4 we did not document the use of UPPAAL and Frama-C on boot code, but on less complex code to demonstrate and evaluate the tools. We did however try to verify pieces of the source code of rBoot [82] (an open-source boot loader) early in the project period. The verification was not very successful due to the complexity of the boot code and our inexperience with Frama-C. As such we chose to learn and reflect upon the tool on less complex C code. This section will discuss potential difficulties with verification of boot code in general and more specifically the OpenTitan boot code.

In chapter 4 we mentioned the problem with analyzing code with Frama-C, that utilize library defined functions such as `strcmp`. This could also be a problem when verifying boot code as boot code commonly depends on custom library functions. Another difficulty with verifying boot code is that it interacts closely with the hardware. This means that it will utilize external functions that are implemented in hardware. E.g. that a given function is programmed at a specific address in the ROM on the hardware platform. We experienced this early in the project period when we tried to verify pieces rBoot's source code with Frama-C. Listing 6.1 shows an example of this with an externally defined function `ets_memcpy` that is hard-coded at address `0x400018b4`. Analyzing the behavior of such a function proved to be very difficult due to the lack of implementation details.

```
1 //From rboot-private.h
2 extern void ets_memcpy(void*, const void*, uint32_t);
3
4 //From eagle.rom.addr.v6.ld
5 PROVIDE ( ets_memcpy = 0x400018b4 );
```

Listing 6.1: An rBoot extern function defined in hardware [82].

Additionally, we found in the security analysis of OpenTitan that the security goals rely on security mechanisms implemented as hardware modules.

The close interaction with hardware means that the functionality of boot code relies on the hardware it interacts with and the layout of memory. As an example, we experienced both in rBoot and OpenTitan, explicit conversions of hard-coded memory addresses to function pointers related to transferring execution. An example from OpenTitan can be seen in listing 6.2, where `rom_ext_get_entry` returns a memory address, which is then converted to a function pointer and called.

```
1 (...)
2
3 typedef void(boot_fn)(void);
4
5 void mask_rom_boot(void) {
6     (...)
7
8     boot_fn *rom_ext_entry = (boot_fn *)rom_ext_get_entry(rom_ext);
9
10    // Jump to ROM_EXT entry point.
11    rom_ext_entry();
12
13    (...)
14 }
```

Listing 6.2: Example of execution transfer from `mask_ROM` to `ROM_EXT` in OpenTitan [83].

We believe this dependency on hardware and memory layout to cause challenges compared to analyzing software that accesses memory through an abstraction (e.g. the abstraction provided by modern operating systems). The reason is that it is necessary to provide a model of the hardware and memory to fully model and analyze the boot code to e.g. determine if it is memory safe. This issue is also addressed in [84]. The authors describe how they implemented automatic generation of a memory model (i.e. the layout of memory) in CBMC, based on linker scripts, to solve issues related to statically analyzing boot code. Furthermore, as described in chapter 3, we experienced problems with state-space explosion and adding hardware specific details may exacerbate the issue.

C code accesses memory through pointers and is commonly used in boot code. As mentioned in section 4.4, pointers made verification with Frama-C difficult. This difficulty is also documented in [64], which men-

tions that Frama-C has trouble verifying code containing pointers because static analyzers have difficulties with gaining enough information about pointers and arrays to perform a proper analysis.

Lastly, as illustrated in chapter 4, we found it difficult to write meaningful annotations that cover all the interesting aspects of a program's behavior, even for non-complex programs. Based on our experience, boot code is more complex than the code we have analyzed in this report. Therefore, we evaluate the task of verifying boot code as a generally challenging task.

Chapter 7

Conclusions

In this chapter, we present the conclusions for the project and give a problem statement for future work. The conclusions are made with the initial problem statement in mind. The initial problem statement is:

How can code be formally verified, and what needs to be formally verified about the security of OpenTitan?

In terms of verification, we have looked at the tools UPPAAL, UPPAAL SMC, and Frama-C. We think that the UPPAAL tool in itself offers a lot of good options to verify the design of OpenTitan. The tool is in our experience the easiest tool to use out of those investigated in this report. This is in part due to the provided counterexamples which are useful during debugging. UPPAAL allows for verification of reachability, liveness, and safety properties which are useful properties to verify for the OpenTitan security goals (cf. section 5.3.4). If UPPAAL confirms the satisfiability of a property, it is guaranteed by a formal proof that the model satisfies the property. However, if the state-space is over- or under-approximated, UPPAAL cannot always give this guarantee. The challenges in UPPAAL verification are the inevitable modeling gap and the state-space explosion problem. The model should be as close to the implemented system as possible to make sure that the queries are usable. However, if the model is too detailed, then the size of the state-space will increase drastically, meaning that the queries will take a seemingly endless time to evaluate. We learned that some model abstraction is needed for non-trivial systems to mitigate the state-space explosion problem and that the abstraction should be dictated by the properties that need to be verified. The state-space reduction can be improved by careful introduction of variables and rigorous variable reset. Additionally, we found that the use of observers is advantageous to use when expressing queries.

UPPAAL SMC's stochastic approach to model checking is different from the approach of standard UPPAAL. The query answers are calculated from simulations of the model, which allows for considerable verification time and memory improvements compared to UPPAAL. This means that it could be possible to model closer to the implementation as the state-space explosion is less of an issue. UPPAAL SMC gives probability distributions, estimations, and simulations of logical properties being satisfied rather than formal proofs. Stochastic behavior in the OpenTitan boot code is not as prevalent as in distributed systems and communication protocols. It could however be meaningful to model the probability of hardware failures as hardware attacks showed to be a considerable part of the threat model (cf. section 5.3.6). Additionally, the performance benefits of UPPAAL SMC would allow for modeling OpenTitan closer to its actual implementation. Debugging is, as a consequence of the statistical calculation approach, more difficult as UPPAAL SMC does not provide counterexamples. Our approach of first developing a UPPAAL model utilizing the provided counterexamples, and then expanding the model with stochastic behavior can be used to circumvent this somewhat. This is possible because translating from UPPAAL to UPPAAL SMC allows for large reuse of the model. During translation, we found that simulating binary synchronization with broadcast synchronization was a straightforward although manual process.

Frama-C can be used to analyze code directly. As a result, the modeling gap is small. The Eva plugin can be used to perform a value analysis and discover whether there are any RTEs in the program. Eva is

a sound analyzer meaning that if Eva does not raise any alarms, then there can be no RTEs. Eva analyzes an over-approximation of the program states, this means that false positives can occur. The WP plugin is used to verify if annotations denoting program behavior are satisfied. The WP plugin is much harder to use than the Eva plugin mostly because it requires the user to write annotations about program behavior written in ACSL. The annotations needed for the WP plugin to work are often more difficult to write than the functions they annotate. The documentation is in some areas lacking and there is limited documentation from the user base to supplement it. As an example, we found from experience that loop invariants must be thorough to verify meaningful properties, but this is not emphasized in the official documentation. WP error messages are not always sufficiently descriptive and may require a deeper understanding of how WP works to deduce what actually causes the error. Additionally, WP does not provide counterexamples when an annotation is not satisfied. This adds to the development and debugging difficulty. Verifying the absence of RTEs with Eva should be done before WP verification. This is because an RTE may invalidate the results verified by WP. Through the use of Frama-C, we have learned that the tool is powerful and we believe that it can be used to verify relevant properties, such as memory safety, for boot code found in OpenTitan.

To analyze the security of the OpenTitan boot code we performed a security analysis (cf. chapter 5). During the security analysis, we analyzed how the OpenTitan chip works in detail. The security analysis resulted in a series of verifiable security goals about boot code validation, execution transfer, concealing of secrets, and the privilege hierarchy. These security goals and their underlying security mechanisms may be targeted by an attacker. The threat model investigates how the security, and thus the security goals, may be compromised by various software and hardware attacks. This will serve as the foundation for future work where we will prove by formal verification of the boot code if some selected security goals are satisfied and of low risk of being compromised.

We have found several challenges to verification of boot code and specifically OpenTitan boot code that must be taken into account in future work. Boot code is often reliant on external functions that are hard-coded into hardware. Pointers are common in boot code and make Frama-C verification more difficult. The OpenTitan security mechanisms are very low level and large parts of these are implemented through hardware. The OpenTitan code base is not developed with Frama-C verification in mind. This will likely mean that verification will require rewrites of some of the code. The OpenTitan project is open-source and under development. This means that we can expect frequent updates to the code (and design) which would require maintenance and iterations of the models and annotations for verification to be useful.

This results in our problem statement, which is:

How can the OpenTitan security goals, defined in section 5.3.4, be verified by UPPAAL, Frama-C, or other verification tools?

Chapter 8

Future Work

The work with the project has led to an idea of how the correctness and security of a system can be verified with the help of different tools such as UPPAAL and Frama-C. The tools have different approaches to verification, UPPAAL verifies a model by doing model checking as described in section 2.1. Whereas WP and Eva (the Frama-C plugins) use the code directly to do static analysis as described in section 2.2. WP also has to rely on annotations in the code, which Eva does not. We believe it might be possible to combine the tools, to help verify the hardware as well as the software of OpenTitan. In future work, we want to apply some of the knowledge we have acquired about formal verification tools to verify some of the security goals for OpenTitan mentioned in section 5.3.4.

8.1 UPPAAL for Modeling Hardware

We could possibly apply a combination of Frama-C and UPPAAL to verify the OpenTitan boot code. We have already mentioned the possibility of modeling the hardware in UPPAAL in section 5.3.6. In order to model the OpenTitan hardware, inspiration could be drawn from METAMOC, a modular method for doing worst-case execution time analysis [79]. The method models programs, main memory, and pipeline in UPPAAL, and with given cache specifications can estimate worst-case execution times. It may be possible with a similar approach to model the hardware of the OpenTitan project, and then use Frama-C to do verification of the software.

8.2 UPPAAL Model from Frama-C CFG

The METAMOC approach will, given an annotated program, generate a control flow graph based on it, and use the control flow graph as a foundation for building a UPPAAL model of the program. We believe something similar may be possible with the help of Frama-C.

The Frama-C Eva plugin computes a control flow graph as an intermediary during analysis [85]. According to the Frama-C Eva help command, it should be possible to output the CFG in dot format into an output file using the command `-eva-traces-dot`. This output is not UPPAAL readable. UPPAAL expects an XML file with information about edges, locations, etc. For the CFG to be readable for UPPAAL it will require some work into creating a converter that translates the dot formatted output.

UPPAAL could be used to verify e.g. the security goal "The hash of the `ROM_EXT` image and the signature of the hash must be validated by `mask_ROM` before it is executed to ensure authenticity and integrity of the image." as a safety property. One of the challenges in doing this would be that the generated model is too large. Then it will be unfeasible to verify useful properties in UPPAAL, as the verification will be limited by the state-space explosion problem.

The model could also be used in UPPAAL SMC which, due to its statistical approach, can handle queries on larger models. It could be useful to evaluate the safety against physical attacks (cf. section 5.3.6) which may involve some probability of happening. Here it could be necessary to add hardware components to the model.

If this approach is feasible it would allow for automatic model generation of software in UPPAAL. Whether or not the generated model is useful in practice is for future work to consider.

8.3 WP Replacement

We are concerned that it might be too extensive a task to use Frama-C's WP plugin to perform verification of OpenTitan. If we feel that WP is the wrong tool for the task it could also be a possibility to look into a tool such as CBMC as mentioned in section 2.2.1. We believe that the steep learning curve we have experienced with Frama-C and especially WP as described in section 6.3 could be a considerable hindrance for future work. It would also be a possibility to only utilize Eva from the Frama-C platform and combine that with CBMC and/or UPPAAL.

Bibliography

[1] MAXSYT. *MAXSYT Security Consultancy in Embedded Product Solutions*. URL: https://novi.dk/dk/maxsyt (visited on 12/21/2020).

[2] Loïc Correnson et al. *Frama-C Homepage*. URL: http://frama-c.com/ (visited on 10/27/2020).

[3] UPPAAL. *UPPAAL Homepage*. URL: http://www.uppaal.org/ (visited on 11/10/2020).

[4] *BIOS*. URL: https://en.wikipedia.org/wiki/BIOS (visited on 10/27/2020).

[5] W. Ecker, W. Müller, and R. Dömer. *Hardware-dependent Software Principles and Practice*. eng. Dordrecht.

[6] *Real Mode OS Warning*. URL: https://wiki.osdev.org/Real_Mode_OS_Warning (visited on 10/27/2020).

[7] V. Zimmer, M. Rothman, and S. Marisetty. *Beyond BIOS : Developing with the Unified Extensible Firmware Interface, Third Edition*. eng. Boston.

[8] Rebecca Shapiro. *TYPES FOR THE CHAIN OF TRUST: NO (LOADER) WRITE LEFT BEHIND*. URL: https://www.cs.dartmouth.edu/~trdata/reports/TR2018-863.pdf (visited on 10/28/2020).

[9] Microsoft. *Measured Boot*. URL: https://docs.microsoft.com/en-us/windows/win32/w8cookbook/measured-boot (visited on 10/30/2020).

[10] V. Zimmer, M. Rothman, and S. Marisetty. *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*. Intel Press, 2010. ISBN: 9781934053294. URL: https://books.google.dk/books?id=t%5C_iwuAAACAAJ.

[11] Microsoft. *Secure boot*. URL: https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-secure-boot (visited on 10/27/2020).

[12] Edmund M. Clarke et al., eds. *Handbook of Model Checking*. Springer, 2018. ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8. URL: https://doi.org/10.1007/978-3-319-10575-8.

[13] TAPAAL. *TAPAAL Homepage*. URL: https://www.tapaal.net/ (visited on 11/10/2020).

[14] PRISM. *PRISM Homepage*. URL: http://www.prismmodelchecker.org/ (visited on 11/10/2020).

[15] Luca Aceto et al. *Reactive Systems: Modelling, Specification and Verification*. English. Cambridge University Press, 2007. ISBN: 9780521875462.

[16] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "Stochastic Model Checking". In: *Formal Methods for Performance Evaluation, 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures*. Ed. by Marco Bernardo and Jane Hillston. Vol. 4486. Lecture Notes in Computer Science. Springer, 2007, pp. 220–270. DOI: 10.1007/978-3-540-72522-0_6. URL: https://doi.org/10.1007/978-3-540-72522-0%5C_6.

[17] Anders Møller and Michael I. Schwartzbach. *Static Program Analysis*. Department of Computer Science, Aarhus University, http://cs.au.dk/~amoeller/spa/. Oct. 2018.

[18] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176. ISBN: 3-540-21299-X.

[19] Daniel Kroening. *CBMC homepage*. URL: http://www.cprover.org/cbmc/ (visited on 12/20/2020).

[20] Wikipedia. *Mathematical proof*. URL: https://en.wikipedia.org/wiki/Mathematical_proof (visited on 11/06/2020).

[21] Fenner Tanswell. "A Problem with the Dependence of Informal Proofs on Formal Proofs†". In: *Philosophia Mathematica* 23.3 (Mar. 2015), pp. 295–310. ISSN: 0031-8019. DOI: 10.1093/philmat/nkv008. eprint: https://academic.oup.com/philmat/article-pdf/23/3/295/4254304/nkv008.pdf. URL: https://doi.org/10.1093/philmat/nkv008.

[22] John Harrison. *Theorem Proving for Verification*. University Lecture. 2008. URL: https://www.cl.cam.ac.uk/~jrh13/slides/cav-09jul08/slides.pdf (visited on 01/06/2021).

[23] Martin Ouimet and K. Lundqvist. "Formal Software Verification: Model Checking and Theorem Proving". In: 2007.

[24] Coq. *Coq Homepage*. URL: https://coq.inria.fr/ (visited on 11/10/2020).

[25] Wikipedia. *What is Coq?* URL: https://coq.inria.fr/about-coq (visited on 11/06/2020).

[26] Bjarke Hilmer Møller et al. *Gathering Data on Water Streams near Agriculture*. URL: https://projekter.aau.dk/projekter/da/studentthesis/gathering-data-on-water-streams-near-agriculture(3b9ac056-6a39-48c1-b37d-8df6132eb415).html (visited on 12/18/2020).

[27] Sigfox. *Sigfox home page*. URL: https://www.sigfox.com/en (visited on 12/30/2020).

[28] inc MongoDB. *Mongodb home page*. URL: https://www.mongodb.com/2 (visited on 12/30/2020).

[29] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on Uppaal*. URL: http://people.cs.aau.dk/~adavid/publications/21-tutorial.pdf (visited on 09/21/2020).

[30] Alexandre David et al. *Uppaal SMC Tutorial*. URL: http://people.cs.aau.dk/~kgl/SSFT2015/SMC%5C%20tutorial.pdf (visited on 10/05/2020).

[31] Loïc Correnson et al. *Frama-C User Manual: Release 21.1*. URL: http://frama-c.com/download/user-manual-21.1-Scandium.pdf (visited on 10/27/2020).

[32] TIOBE. *TIOBE Index for October 2020*. URL: https://www.tiobe.com/tiobe-index// (visited on 10/27/2020).

[33] David Bühler et al. *Eva – The Evolved Value Analysis plug-in*. URL: http://frama-c.com/download/eva-manual-21.1-Scandium.pdf (visited on 11/03/2020).

[34] Patrick Baudin et al. *ACSL: ANSI/ISO C Specification Language ACSL Version 1.16*. URL: https://www.frama-c.com/download/frama-c-acsl-implementation.pdf (visited on 11/24/2020).

[35] A. Blanchard, N. Kosmatov, and F. Loulergue. "A Lesson on Verification of IoT Software with Frama-C". In: *2018 International Conference on High Performance Computing Simulation (HPCS)*. 2018, pp. 21–30. DOI: 10.1109/HPCS.2018.00018.

[36] Patrick Baudin et al. *Frama-C / WP*. URL: http://frama-c.com/download/wp-manual-21.1-Scandium.pdf (visited on 11/03/2020).

[37] Virgile. *User Virgile*. URL: https://stackoverflow.com/users/1633665/virgile (visited on 12/11/2020).

[38] Stephen M. Papa William D. Casper. *Root of Trust*. URL: https://link.springer.com/referenceworkentry/10.1007/978-1-4419-5906-5_789 (visited on 11/06/2020).

[39] VentureBeat. *Google announces OpenTitan, an open source silicon root of trust project Homepage*. URL: https://venturebeat.com/2019/11/05/google-announces-opentitan-an-open-source-silicon-root-of-trust-project/#:~:text=The%20project%20is%20managed%20by,transparent%2C%20trustworthy%2C%20and%20secure. (visited on 12/14/2020).

[40] Google. *OpenTitan*. URL: https://opentitan.org/ (visited on 11/03/2020).

[41] TechCrunch. *Google launches OpenTitan, an open-source secure chip design project*. URL: https://techcrunch.com/2019/11/05/google-opentitan-secure-chip/ (visited on 12/14/2020).

[42] Wikipedia. *Encryption*. URL: https://en.wikipedia.org/wiki/Encryption (visited on 12/20/2020).

[43] Wikipedia. *Public-key cryptography*. URL: https://en.wikipedia.org/wiki/Public-key_cryptography (visited on 12/20/2020).

[44] Thomas Pornin. *Security StackExchange: Trying to understand RSA and it's terminology?* URL: https://security.stackexchange.com/questions/68822/trying-to-understand-rsa-and-its-terminology (visited on 12/20/2020).

[45] Thomas Pornin. *Security StackExchange: SHA, RSA and the relation between them*. URL: https://security.stackexchange.com/questions/9260/sha-rsa-and-the-relation-between-them#answer-9265 (visited on 12/20/2020).

[46] Wikipedia. *Symmetric-key algorithm*. URL: https://en.wikipedia.org/wiki/Symmetric-key_algorithm (visited on 12/20/2020).

[47] Wikipedia. *Digital signature*. URL: https://en.wikipedia.org/wiki/Digital_signature (visited on 12/09/2020).

[48] Wikipedia. *HMAC*. URL: https://en.wikipedia.org/wiki/HMAC (visited on 12/09/2020).

[49] Google. *OpenTitaLogical Security Model*. URL: https://docs.opentitan.org/doc/security/logical_security_model/ (visited on 11/06/2020).

[50] OpenTitan. *Ownership Transfer*. URL: https://docs.opentitan.org/doc/security/specs/ownership_transfer (visited on 12/04/2020).

[51] OpenTitan. *OpenTitan Key Manager HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/keymgr/doc/index.html (visited on 12/14/2020).

[52] OpenTitan. *CSRNG HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/csrng/doc/ (visited on 12/04/2020).

[53] Wikipedia. *Replay Attacls*. URL: https://en.wikipedia.org/wiki/Replay_attack (visited on 12/04/2020).

[54] OpenTitan. *Identities and Root Keys*. URL: https://docs.opentitan.org/doc/security/specs/identities_and_root_keys/ (visited on 12/14/2020).

[55] Danny Bøgsted Poulsen and René Rydhof Hansen. *Del 1 Sikkerhed? - SikSoft 01*. Oct. 23, 2020.

[56] OpenTitan. *OpenTitan Secure Boot*. URL: https://docs.opentitan.org/doc/security/specs/secure_boot/ (visited on 11/11/2020).

[57] OpenTitan. *OpenTitan device source code*. URL: https://github.com/lowRISC/opentitan/tree/master/sw/device (visited on 11/24/2020).

[58] OpenTitan. *Boot ROM: README.md*. URL: https://github.com/lowRISC/opentitan/blob/master/sw/device/boot_rom/README.md (visited on 11/24/2020).

[59] OpenTitan. *OpenTitan Issue about Documentation*. URL: https://github.com/lowRISC/opentitan/issues/4315#event-4071904280 (visited on 12/04/2020).

[60] Dayeol Lee and David Kohlbrenner. *RISC-V Background*. URL: http://docs.keystone-enclave.org/en/latest/Getting-Started/How-Keystone-Works/RISC-V-Background.html (visited on 11/06/2020).

[61] OpenTitan. *Pseudo-code for Mask ROM Secure Boot Process*. URL: https://docs.opentitan.org/sw/device/mask_rom/boot/ (visited on 11/24/2020).

[62] OpenTitan. *ROM_EXT Manifest Format*. URL: https://docs.opentitan.org/sw/device/rom_exts/manifest/ (visited on 11/24/2020).

[63] OpenTitan. *Identities and Root Keys*. URL: https://docs.opentitan.org/doc/security/specs/identities_and_root_keys/ (visited on 11/24/2020).

[64] P. Berthomé et al. "Attack Model for Verification of Interval Security Properties for Smart Card C Codes". In: *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. PLAS '10. Toronto, Canada: Association for Computing Machinery, 2010. ISBN: 9781605588278. DOI: 10.1145/1814217.1814219. URL: https://doi.org/10.1145/1814217.1814219.

[65] OpenTitan. *Flash Controller HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/flash_ctrl/doc/index.html (visited on 12/14/2020).

[66] OpenTitan. *Secure Boot Process*. URL: https://github.com/lowRISC/opentitan/blob/master/sw/device/mask_rom/docs/index.md (visited on 12/14/2020).

[67] OpenTitan. *Life Cycle Controller Technical Specification*. URL: https://docs.opentitan.org/hw/ip/lc_ctrl/doc/ (visited on 12/14/2020).

[68] OpenTitan. *Device Life Cycle*. URL: https://docs.opentitan.org/doc/security/specs/device_life_cycle/ (visited on 12/14/2020).

[69] OpenTitan. *CSRNG HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/csrng/doc/ (visited on 12/14/2020).

[70] OpenTitan. *ENTROPY_SRC HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/entropy_src/doc/ (visited on 12/14/2020).

[71] Wikipedia. *Entropy (computing)*. URL: https://en.wikipedia.org/wiki/Entropy_(computing) (visited on 12/14/2020).

[72] OpenTitan. *AES HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/aes/doc/ (visited on 12/14/2020).

[73] Google. *OpenTitan Use Cases*. URL: https://docs.opentitan.org/doc/security/use_cases/ (visited on 11/06/2020).

[74] OpenTitan. *Big Number Accelerator*. URL: https://docs.opentitan.org/hw/ip/otbn/doc/ (visited on 12/14/2020).

[75] OpenTitan. *OpenTitan Security Model Specification*. URL: https://docs.opentitan.org/doc/security/specs/ (visited on 12/14/2020).

[76] Wikipedia. *Memory controller*. URL: https://en.wikipedia.org/wiki/Memory_controller#SCRAMBLING (visited on 12/14/2020).

[77] OpenTitan. *SRAM Controller Technical Specification*. URL: https://docs.opentitan.org/hw/ip/sram_ctrl/doc/ (visited on 12/14/2020).

[78] OpenTitan. *Alert Handler Technical Specification*. URL: https://docs.opentitan.org/hw/ip/alert_handler/doc/ (visited on 12/14/2020).

[79] Andreas Engelbredt Dalsgaard et al. "METAMOC; Modular Execution Time Analysis using Model Checking". English. In: *Proceedings of the 10th International Workshop on Worst-Case Execution-Time Analysis (WCET2010)*. 2010.

[80] Wikipedia. *Side-channel attack*. URL: https://en.wikipedia.org/wiki/Side-channel_attack (visited on 12/19/2020).

[81] Wikipedia. *Buffer overflow*. URL: https://en.wikipedia.org/wiki/Buffer_overflow (visited on 01/05/2021).

[82] Richard Burton. *rBoot Github*. URL: https://github.com/raburton/rboot (visited on 01/08/2021).

[83] OpenTitan. *mask_rom.c*. URL: https://github.com/lowRISC/opentitan/blob/master/sw/device/mask_rom/mask_rom.c (visited on 01/08/2021).

[84] Cook B. et al. *Model Checking Boot Code from AWS Data Centers. In: Chockler H., Weissenbacher G. (eds) Computer Aided Verification. CAV 2018. Lecture Notes in Computer Science, vol 10982. Springer, Cham.* July 18, 2018. URL: https://doi.org/10.1007/978-3-319-96142-2_28.

[85] Virgile Prevosto. *EXTRACTING INFORMATION FROM FRAMA-C PROGRAMMATICALLY*. URL: https://frama-c.com/2012/09/04/Extracting-information-from-Frama-C-programmatically.html (visited on 12/21/2020).

[86] StackOverflow. *What sort of things are UEFI "applications" actually used for?* URL: https://stackoverflow.com/questions/26825927/what-sort-of-things-are-uefi-applications-actually-used-for (visited on 12/30/2020).

[87] Wikipedia. *Unified Extensible Firmware Interface*. URL: https://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface (visited on 12/30/2020).

[88] UEFI Forum inc. *Unified Extensible Firmware Interface (UEFI) Specification*. URL: https://uefi.org/sites/default/files/resources/UEFI_Spec_2_8_final.pdf (visited on 10/28/2020).

[89] EDK II. *3.6 Protocols and handles*. URL: https://edk2-docs.gitbook.io/edk-ii-uefi-driver-writer-s-guide/3_foundation/36_protocols_and_handles (visited on 12/30/2020).

[90] Michael Kinney et al. *3.4 Handle database*. URL: https://edk2-docs.gitbook.io/edk-ii-uefi-driver-writer-s-guide/3_foundation/34_handle_database (visited on 01/07/2021).

[91] Google. *Device Attestation*. URL: https://docs.opentitan.org/doc/security/specs/attestation/ (visited on 11/06/2020).

[92] Wikipedia. *Message authentication code*. URL: https://en.wikipedia.org/wiki/Message_authentication_code (visited on 12/14/2020).

[93] tutorialpoint. *Message Authentication*. URL: https://www.tutorialspoint.com/cryptography/message_authentication.htm (visited on 12/14/2020).

Appendix A

UEFI Concepts

A.1 UEFI System Table

Each UEFI application (e.g. OS boot loader, shell, Linux kernel, and firmware updater) and UEFI driver (e.g. file system drivers) will contain a pointer to the system table [86, 87]. The system table stores a UEFI boot services table, UEFI runtime services table, and protocol services. These services can be thought of as functions that the UEFI firmware offers to the applications and drivers. A driver or application can therefore through the pointer to the system table access a system's configuration information [7, ch. 2].

The boot services table exposes functions that UEFI applications need to use to e.g. allocate memory. The boot services are available until an UEFI application calls `ExitBootServices()` to exit the boot services. The call to `ExitBootServices()` will happen when the UEFI application is ready to control the rest of the system's boot [88, ch. 8].

The runtime services provided through the UEFI runtime services table are available before and after the call to `ExitBootServices()`. The runtime services table exposes functions such as `GetTime()`, `SetTime()`, and `ResetSystem()` [88, ch. 2].

The protocol services provided by the system table are used to provide an interface to the components of the computer system such as network and disk [7, ch. 2].

A.2 Protocols

A protocol is essentially a data structure that contains functionality used to communicate between modules (e.g. drivers and applications) [89]. UEFI protocols are made up of two components: a set of function pointers and specification defined data structures or APIs. A protocol must have a globally unique identifier (GUID) which is used to identify and locate the protocol in the Handle Database. It is essential that each GUID is unique. If multiple protocols have the same GUID, the system does not know which one to use in a certain scenario. This will no doubt lead to crashes of the system. Protocols will often also include a protocol interface structure. A protocol interface structure consists of data structures and/or procedures. Protocols can be used by any UEFI code during system boot [7, ch. 2].

New protocols can be created over time to extend UEFI systems. There are also many protocols that are not defined in the UEFI specification. Protocols are also what allows the UEFI firmware specification to adapt over time [7, ch. 2].

A.3 Handle Database

The Handle Database consists of handles and protocols. Protocols may contain services, data fields, both, or none of them. Handles are groupings of protocols. When UEFI initialization begins the system firmware, along with drivers and applications, create handles and attach protocols to them. The Handle Database can then be globally accessed by any UEFI image [7, ch. 2]. A figure of the handle database can be seen in Fig. A.3.1

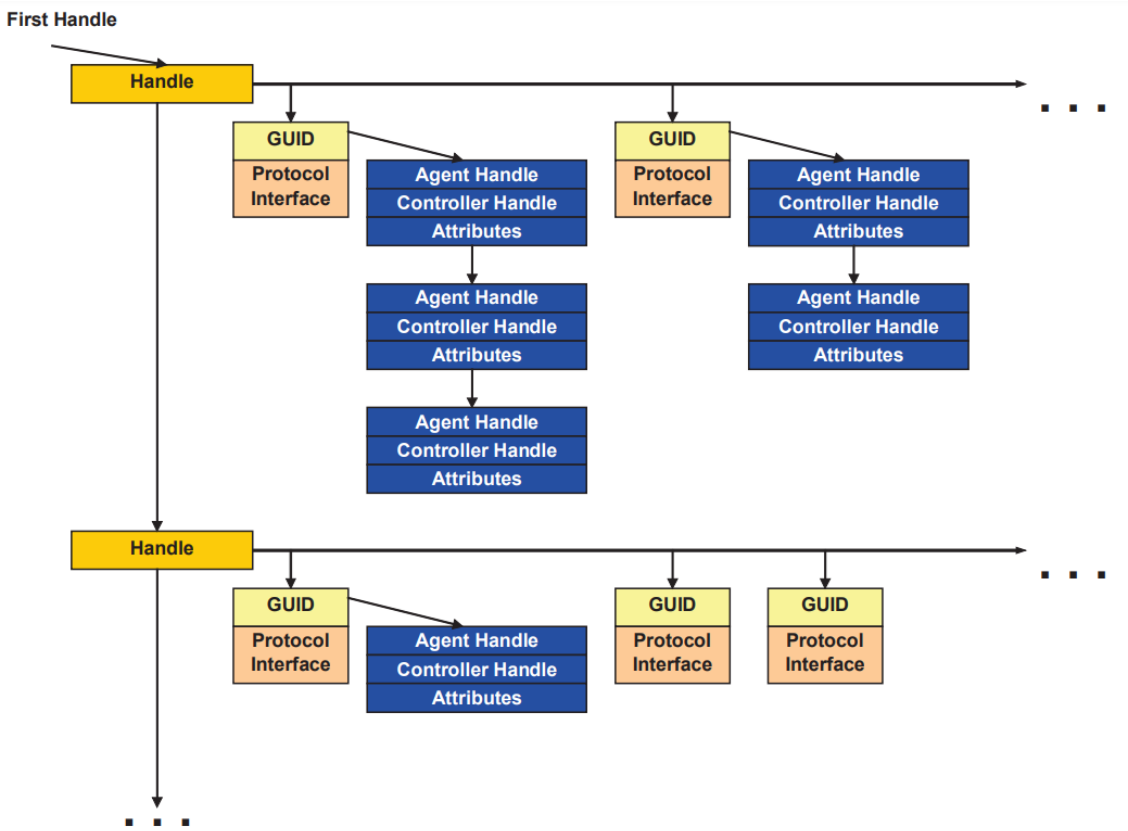


Figure A.3.1: A figure that shows the layout of the Handle Database [7, ch. 2].

Each handle has a number that provides a key to a database entry. A handle also has a type that is dependent on the types of protocols (such as drivers and applications, devices, etc.) that are attached to it [7, ch. 2]. The different types of handles and their relation to each other can be seen in Fig. A.3.2.

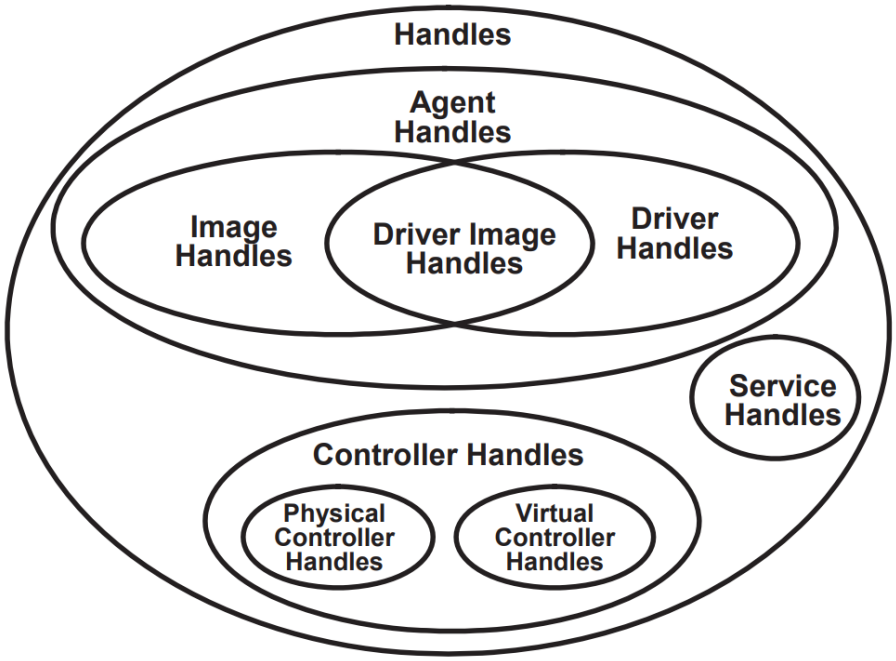


Figure A.3.2: The different types of handles in UEFI and their relation to each other [7, ch. 2].

Below is a list of definitions for some of the handles seen in Fig. A.3.2. We have chosen to include the following handles as they are shown in Fig. A.3.1. The definitions are taken from [90].

- An image handle is for an UEFI driver image that is loaded into memory.
- A driver handle supports all UEFI protocols.
- A driver image handle is a combination of an image handle and a driver handle.
- An agent handle is a general term encompassing the three aforementioned handles.
- A controller handle is a handle for a boot device or console present on the current platform.

Appendix B

Annotated Example Program

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define NUM_OF_TEAMS 12
#define NUM_GAMES_ROUND 6
#define POINTS_FOR_WINNING 3
#define POINTS_FOR_DRAW 1


typedef struct{
    char day[4];
    char date[6];
    char time[6];
    char homeTeam[4];
    char visTeam[4];
    int homeGoal;
    int visGoal;
    double viewers;
} match;

typedef struct{
    char team[4];
    int points;
    int gamesPlayed;
    int wins;
    int losses;
    int draws;
    int goalDiff;
    int goals;
} result;

match* readFileToArray(int *arrayLengthP);
int countLines(FILE *matchFile);
match createMatch();
void printMatch(match aMatch);
void printMatchHeader();
void runPrint(match *matchArray, int arrayLength, int isRunPrint);
void printMenu(int wrongChoice);
void mainLoop(match *matchArray, int arrayLength, int isRunPrint);
void printBlankLine();
void sevenGoalMatches(match *matchArray, int arrayLength);
void mostGoalRound(match *matchArray, int arrayLength);
void mostVisTeamWins(match *matchArray, int arrayLength);
int visGreaterHome(match *matchArray, int arrayLength, char *currentTeam);
void fewestViewers(match *matchArray, int arrayLength);
double countViewers(match *matchArray, int arrayLength, char *currentTeam);
void dateTimeMatches(match *matchArray, int arrayLength, int isRunPrint);
```

```
49 int elementCompareInt(const void* e1, const void* e2);
50 void createRankings(match *matchArray, int arrayLength);
51 void emptyResultArray(result *resultArray);
52 void printResult(result aResult);
53 void fillResultArray(match *matchArray, int arrayLength, result *resultArray);
54 void createResult(match *matchArray, int arrayLength, result *resultArray, int
       ↪ entryResultArray, char *currentTeam);
55 void printResultHeader();
56 int elementCompareResult(const void* e1, const void* e2);
57 int stringLength(const char* str);
58
59 /*@  requires 0 <= n1 ;
60      requires 0 <= n2 ;
61      ensures (s1 == s2 ==> \result == 0) ;*/
62 int stringCompare(const char* s1, const char* s2, int n1, int n2) {
63     if (s1 == s2)
64         return (0);
65     int i = 0;
66     /*@ loop invariant 0 <= i;
67         loop assigns i , s1, s2;
68         */
69     while (*s1 == *s2++)
70     {
71         ++i;
72         if (*s1++ == '\0')
73             return (0);
74     }
75     return (*(unsigned char*)s1 - *(unsigned char*)--s2);
76 }
77 /*@ assigns \nothing ;
78      ensures \result >= 0 ;*/
79 int stringLength(const char* str) {
80     if (str[0] == '\0')
81         return 0;
82     else return 1 + stringLength(str + 1);
83 }
84
85 /*@ requires n >= 0 ;
86     requires \valid_read (src);
87     assigns dest[0..n] ;
88     ensures \forall integer k; 0 <= k <= n ==> dest[k] == src[k];
89  */
90 char* stringCopy(char* dest, const char* src, int n) {
91     char* os1 = dest;
92
93     /*@ loop assigns n, dest[0..n], *dest ;
94         loop invariant \true ;
95         loop variant n;*/
96     while (*dest++ = *src++)
97         --n;
98     return (os1);
99 }
100
101
102
103 /*@  allocates \nothing ;*/
104 int main(int argc, char* argv[]) {
105     match* matchArrayP;
106     match matchArray[13];
```

```
107     int arrayLength, isRunPrint = 0;
108     int* arrayLengthP;
109     char* print;
110     int n = stringLength("--print");
111     stringCopy(print, "--print", n);
112     arrayLengthP = &arrayLength;
113 
114     /*@ assert \valid(arrayLengthP) ;*/
115     /* Passes temporary array to main array */
116 
117     int i;
118     match tempArray[12];
119     *arrayLengthP = 12;
120     arrayLength = *arrayLengthP;
121 
122     /*@ loop invariant 0 <= i <= arrayLength;
123         loop assigns i, tempArray[0 .. arrayLength] ;*/
124     for (i = 0; i < arrayLength; i++) {
125         match newMatch;
126         //strcpy(newMatch.day, "Son");
127         newMatch.day[0] = 'S';
128         newMatch.day[1] = 'o';
129         newMatch.day[2] = 'n';
130         newMatch.day[3] = '\0';
131         //strcpy(newMatch.date, "21/07");
132         newMatch.date[0] = '2';
133         newMatch.date[1] = '1';
134         newMatch.date[2] = '/';
135         newMatch.date[3] = '0';
136         newMatch.date[4] = '7';
137         newMatch.date[5] = '\0';
138         //strcpy(newMatch.time, "19.00");
139         newMatch.time[0] = '1';
140         newMatch.time[1] = '9';
141         newMatch.time[2] = '.';
142         newMatch.time[3] = '0';
143         newMatch.time[4] = '0';
144         newMatch.time[5] = '\0';
145         //strcpy(newMatch.homeTeam, "AAB");
146         newMatch.homeTeam[0] = 'A';
147         newMatch.homeTeam[1] = 'A';
148         newMatch.homeTeam[2] = 'B';
149         newMatch.homeTeam[3] = '\0';
150         //strcpy(newMatch.visTeam, "FCK");
151         newMatch.visTeam[0] = 'F';
152         newMatch.visTeam[1] = 'C';
153         newMatch.visTeam[2] = 'K';
154         newMatch.visTeam[3] = '\0';
155         newMatch.homeGoal = 2;
156         newMatch.visGoal = 1;
157         newMatch.viewers = 7.062;
158         tempArray[i] = newMatch;
159     }
160 
161     matchArrayP = tempArray;
162     for (int i = 0; i < arrayLength; i++)
163     {
164         matchArray[i] = matchArrayP[i];
165     }
```

```
166     int n1 = stringLength(argv[1]);
167     int n2 = stringLength(print);
168     if (0 <= n1 && 0 <= n2)
169     {
170         if (argc > 1 && !(stringCompare(argv[1], print,n1,n2)) && arrayLength >= 0)
        ↪ {
171             isRunPrint = 1;
172             runPrint(matchArray, arrayLength, isRunPrint);
173         }
174         else if (matchArray != NULL && arrayLength >= 0) {
175             mainLoop(matchArray, arrayLength, isRunPrint);
176         }
177     }
178 
179     return 0;
180 }
181 
182 /* Reads the file and puts its contents in a temporary array */
183 /*@ requires \valid(arrayLengthP) ;
184     assigns *arrayLengthP;*/
185 match* readFileToArray(int *arrayLengthP){
186     int i;
187     match tempArray[12];
188     *arrayLengthP = 12;
189     int arrayLength = *arrayLengthP;
190 
191     /*@ loop invariant 0 <= i <= arrayLength;
192         loop assigns i, tempArray[0 .. arrayLength] ;*/
193     for(i = 0; i < arrayLength; i++){
194         match newMatch;
195         //strcpy(newMatch.day, "Son");
196         newMatch.day[0] = 'S';
197         newMatch.day[1] = 'o';
198         newMatch.day[2] = 'n';
199         newMatch.day[3] = '\0';
200         //strcpy(newMatch.date, "21/07");
201         newMatch.date[0] = '2';
202         newMatch.date[1] = '1';
203         newMatch.date[2] = '/';
204         newMatch.date[3] = '0';
205         newMatch.date[4] = '7';
206         newMatch.date[5] = '\0';
207         //strcpy(newMatch.time, "19.00");
208         newMatch.time[0] = '1';
209         newMatch.time[1] = '9';
210         newMatch.time[2] = '.';
211         newMatch.time[3] = '0';
212         newMatch.time[4] = '0';
213         newMatch.time[5] = '\0';
214         //strcpy(newMatch.homeTeam, "AAB");
215         newMatch.homeTeam[0] = 'A';
216         newMatch.homeTeam[1] = 'A';
217         newMatch.homeTeam[2] = 'B';
218         newMatch.homeTeam[3] = '\0';
219         //strcpy(newMatch.visTeam, "FCK");
220         newMatch.visTeam[0] = 'F';
221         newMatch.visTeam[1] = 'C';
222         newMatch.visTeam[2] = 'K';
223         newMatch.visTeam[3] = '\0';
```

```
224         newMatch.homeGoal = 2;
225         newMatch.visGoal = 1;
226         newMatch.viewers = 7.062;
227         tempArray[i] = newMatch;
228     }
229 
230     return tempArray;
231 
232 }
233 
234 /*@ requires \valid(matchFile) ;
235     assigns *matchFile ;
236     ensures \result > 0 ;*/
237 /* Counts the number of non-empty lines in the open file */
238 int countLines(FILE *matchFile){
239     int numOfLines = 1;
240     char currentLine, preLine = '\0';
241 
242     if (matchFile != NULL) {
243         int loopIter = (currentLine = fgetc(matchFile)) != EOF;
244         if (loopIter)
245             /*@ loop invariant numOfLines > 0 ;
246                 loop assigns *matchFile, loopIter, preLine, numOfLines, currentLine
    ↪ ;*/
247             while (loopIter) {
248                 /* If there are no 2 empty lines together, adds 1 to numOfLines */
249                 if (currentLine == '\n' && preLine != '\n') {
250                     numOfLines++;
251                     /*@ assert numOfLines <= 2147483647; */
252                 }
253                 preLine = currentLine;
254                 if (matchFile != NULL)
255                     loopIter = (currentLine = fgetc(matchFile)) != EOF;
256                 else
257                     break;
258             }
259     }
260 
261     return numOfLines;
262 }
263 
264 
265 
266 /* If isRunPrint is enabled this function will be called */
267 /*@ requires \valid_read(matchArray) && arrayLength >= 0; */
268 void runPrint(match *matchArray, int arrayLength, int isRunPrint){
269         sevenGoalMatches(matchArray, arrayLength);
270 
271 }
272 /* The main loop of the program that does different things depending on user input
    ↪ if isRunPrint is not enabled */
273 /*@ requires \valid_read(matchArray) && arrayLength >= 0 ;
274     assigns \nothing ;
275     ensures \old(isRunPrint) == isRunPrint && \old(*matchArray) == *matchArray && \
    ↪ old(arrayLength) == arrayLength ;*/
276 void mainLoop(match *matchArray, int arrayLength, int isRunPrint){
277     int menuChoice = 1, wrongChoice = 0, run = 1;
278     int iter = 3;
279 
```

```
280     /*@ loop invariant run && iter >= 0 ;
281         loop assigns menuChoice, iter ;*/
282     while(run && iter > 0){
283         menuChoice = 1;
284 
285         if(isRunPrint){
286             sevenGoalMatches(matchArray, arrayLength);
287         }
288 
289         else if(menuChoice == 1){
290             sevenGoalMatches(matchArray, arrayLength);
291         }
292         iter--;
293     }
294 }
295 
296 /* Prints every match in which 7 or more goals have been scored */
297 /*@ requires \valid_read(matchArray) ;
298     assigns \nothing ;
299     ensures \old(arrayLength) == arrayLength ;*/
300 void sevenGoalMatches(match *matchArray, int arrayLength){
301     int i = 0;
302 
303     /*@ loop invariant 0 <= i ;
304         loop assigns i ;*/
305     for(i = 0; i < arrayLength; i++){
306         if((matchArray[i].homeGoal + matchArray[i].visGoal) >= 7){
307         }
308     }
309 }
```

Listing B.1: Example of annotated superliga program.

Appendix C

OpenTitan Terminology Confusion

The OpenTitan documentation does not clearly state whether the notion of Silicon Owner/Creator public and private key pairs are equivalent to the Owner Identity and Creator Identity.

The documentation on the secure boot [56] and `mask_ROM` boot process overview [66] only mention that the Owner and Creator public keys are used to verify the signature of `ROM_EXT`, `BL0`, and `Kernel`. The documentation of the key manager [51], device attestation [91], and identities and root keys [54] only mention identities. We have tried to deduce the answer by looking at the use of the different notions. [54] mentions that the identities are asymmetric keys. [91] mentions that the identities are computed as a message-authentication-code (MAC), which by definition is symmetric [92, 93].

[54] mentions that the Creator identity "is used to attest to the authenticity of the physical device and the ROM and ROM_EXT configuration" which could imply verification of the signature.

[54] mentions that the private portion of the Creator identity and Owner identity should be deleted. This indicates that the notions are not equivalent since a derivation of the private keys (used to sign the boot stages) during every boot should pose a security threat.

[50] states that public keys are provisioned during ownership transfer. Among these are the `CODE_SIGN` key used for verifying the Owner's `BL0` ([56] mentions that the Owner public key is used). There is no mentioning of Owner keys (other than they are endorsed) or Owner Identities. Note that the `CODE_SIGN` may be a temporary key but this is not clearly stated.

The documentation of the logical security model [49] mentions the creation of identities and that the different entities own different stages if they have the key used to sign them. They do not mention any specifics about such a key. They mention that during ownership assignment (i.e. ownership transfer) a set of public keys are loaded onto the device to verify the owner's initial boot stage. However, they do not provide any specifics about the public keys, only that they are plural. It still does not clarify what keys are used to sign the different boot stages.