Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 180

Systems Group, Department of Computer Science, ETH Zurich

Hardware Configuration With Dynamically-Queried Formal Models

by

Daniel Schwyn

Supervised by

Reto Acherman
Dr. David Cock
Prof. Dr. Timothy Roscoe

April 2017 – October 2017

inf | Informatik
Computer Science

Abstract

Hardware is getting increasingly complex and heterogeneous. With different components having different views of the system, the traditional assumption of unique physical addresses has become an illusion. To adapt to such hardware, an operating system (OS) needs to understand the complex address translation chains and be able to handle the presence of multiple address spaces. This thesis takes a recently proposed model that formally captures these aspects and applies it to hardware configuration in the Barrelfish OS.

To this end, I present Sockeye, a domain specific language that uses the model to describe hardware. I then show, that code relying on knowledge about the address spaces in a system can be statically generated from these specifications. Furthermore, the model is successfully applied to device management, showing that it can also be used to configure hardware at runtime. The implementation presented here does not rely on any platform specific code and it reduced the amount of such code in Barrelfish's device manager by over 30%. Applying the model to further hardware configuration tasks is expected to have similar effects.

Acknowledgements

First of all, I want to thank my advisers Timothy Roscoe, David Cock and Reto Achermann for their guidance and support. Their feedback and critical questions were a valuable contribution to this thesis.

I'd also like to thank the rest of the Barrelfish team and other members of the Systems Group at ETH for the interesting discussions during meetings and coffee breaks.

Last but not least my thanks go to my family and friends for their ongoing support during my studies at ETH.

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Hardware is getting increasingly heterogeneous. Processors of different types and even architectures are present on the same chip alongside a wide variety of peripheral devices. Furthermore, devices and increasingly also cores can appear and disappear at any time e.g. for heat and power management purposes [11]. Operating systems (OSs) need to be able to run on and configure these systems ranging from small systems on a chip (SoCs) to server platforms with thousands of cores.

Hardcoding support for every platform into the system is not an option. Supporting new platforms would mean adapting the kernel and other OS parts, resulting in increasingly unmaintainable code. Furthermore, recompiling the OS for every existing hardware combination would be simply infeasible. Thus, the OS needs to be able to gather information about the platform it is running on and adapt to the hardware. There are several existing approaches to solve this problem, and I discuss the most important ones in more detail in Chapter 2.

However, diversity is only part of the challenge today's system software faces. Traditionally a computer system has been modelled as a processor with a memory management unit (MMU), which translates the requested *virtual* addresses to *physical* addresses. Physical addresses then correspond to device registers or locations in memory. This view is getting more and more unsuitable for writing correct systems code. Today's computer systems are becoming increasingly complex networks of CPUs, GPUs, accelerators, memory nodes, and peripheral devices. Address translation often consists of multiple stages and a globally shared view on the system has become simply an illusion: A "physical" address can denote different things depending on where it is

used.

To illustrate these difficulties we take a look at the Texas Instruments OMAP4460. It is representative of the SoCs found in a wide range of multimedia devices nowadays. It has been used in products like the Amazon Kindle Fire 7" and also in the PandaboardES, which is used as the test platform for this thesis. It features an ARMv7 Cortex-A9 multiprocessor with two cores, two Cortex-M3 cores and other processors like a GPU. Furthermore, there are DMA capable devices, which can also issue loads and stores to memory. Detailed documentation is publicly available in form of a Technical Reference Manual [22]. Address translation in this system is by no means as simple as the traditional model assumes (Figure 3.1 shows a simplifed addressing diagram). A good example for this are the two M3 cores. They share a first level (`L1`) MMU and thus always use the same virtual address space. After the first translation the "physical" addresses are either forwarded to local ROM or RAM or to another second level (`L2`) MMU that offers a window into the level 3 (`L3`) interconnect. The M3 local ROM and RAM are in turn accessible from the `L3` interconnect, meaning that different addresses issued by the `L1` MMU might address the same memory cell, some directly and some with a detour through the `L3` interconnect. The A9 cores come with their own MMUs and local ROM which can only be accessed from their local interconnect. The "physical"address spaces these MMUs translate to are therefore different from the one for the M3 cores' `L1` MMU. There are also several subsystems on the chip that add even more address spaces.

This sort of complex address translation chains and the fact, that even comparatively simple systems like the OMAP have several address spaces, make it hard to reason about software running on these platforms. Without an accurate model, it is hard to write correct code that interacts directly with hardware. In 2001 Chou et al. found that error rates in Linux drivers are three to seven times higher than in the rest of the kernel [5]. Swift et al. reported that in Windows XP, drivers accounted for 85% of all recent system failures in 2003 [21]. Since then the complexity of hardware has only increased. Better correctness guarantees are needed. However, without a model that is able to capture the complexity, synthesis or formal verification approaches are impossible.

Recently Achermann et al. presented a model that formalises address decoding [1]. It captures the aforementioned address translation chains and interactions between different address spaces. The authors also present a concrete syntax in which hardware can be described using the model and show that they can model real systems. For

the rest of this thesis I will refer to the model as address decoding nets (ADNs). It is discussed in more detail in Chapter 3.

Ideally such formal specifications would be supplied by the hardware manufacturer. Adding support for a new platform to an OS requires a big effort compared to the relatively short lifetime of today's systems. If these specifications could be used to configure hardware, this effort would be greatly reduced.

This thesis shows that the ADNs can be used for hardware configuration by applying them to automatic configuration in the Barrelfish OS. Barrelfish's multikernel architecture [3] pushes configuration tasks out of the kernel to a reasoning engine called the System Knowledge Base (SKB). The SKB allows one to describe problems declaratively, making it easier to reason about them. This makes Barrelfish a good fit to apply a formal model to a low-level task like hardware configuration.

I start with a review of current approaches to tackle hardware diversity in OSs and talk about how declarative techniques are already used in Barrelfish (Chaper 2). In Chapter 3, I discuss the important aspects of ADNs and Barrelfish's SKB for this thesis. To be able to describe hardware, I designed a domain specific language (DSL) based on the concrete syntax proposed for ADNs. Its syntax, language features and the implemented compiler are described in Chapter 4. For hardware configuration algorithms to be able to use knowledge about the system's address spaces, I designed an SKB representation for ADNs. The representation and how it can be queried is discussed in Chapter 5. To validate the approach I then implemented two configuration tasks in Barrelfish running on a PandaboardES. I first show that ADNs can be used for static code generation by constructing kernel page tables at compile time. For the second task I extended the device manager. I use the ADN representation in the SKB to discover devices and cores. Upon discovery of a device, an appropriate driver is started. To find the addresses of the device registers the driver needs access to, the device manager also queries the ADN. By outsourcing all address space knowledge to the formal model, the code dealing with ARMv7 device management becomes mostly platform independent. Having a platform independent and maybe, with some additional work, even an architecture independent device manager would be a huge win for maintainability. Providing a formal description of a platform would suffice for adding device management support. The implementation is described in Chapter 6. I finally conclude in Chapter 7 and give an overview of future work.

Chapter 2

Related Work

As the problem of adapting an OS to an increasingly diverse zoo of hardware platforms is not new, several standards for acquiring hardware information have emerged. Many systems support ACPI [12], a standard through which system firmware can provide OSs with configuration information. With PCI [13] available devices can be discovered and information about their functionality is provided. However, often more information than these standards provide is required to correctly configure a system. Furthermore, many SoCs do not implement these standards. ACPI has also been heavily criticised for the complexity of the standard [8]. Other concerns address the fact that configuration information is stored as bytecode, which needs to be executed inside the kernel. It has even been called a trojan horse [20] as this potentially allows the injection of malicious code into the OS kernel.

Another approach to decouple hardware information from the OS kernel are device trees [9]. A device tree is a hierarchical data structure to describe hardware topologies statically. They were originally used to pass hardware information from firmware to systems software, but support is now also built directly into software. Linux for example makes heavy use of them to decouple hardware information about platforms with no discovery mechanisms from the rest of the kernel. However, their tendency to rely on conventions rather than strict specifications make them error prone [17]. As device trees do not define clear semantics, it is hard to verify their correctness. Using them as a basis for formal reasoning about hardware properties or verifying systems software is downright impossible. Device trees also assume a globally shared view on the system. With different processor types, accelerators and DMA capable devices, even simple

systems nowadays have multiple address spaces, and different parts of the system can have different apertures into the address spaces of other components. With the OMAP we have already seen an example. The assumption of globally unique addresses has become too simplistic if not plain wrong. In Section 3.1, I show how ADNs tackle the problem.

Device trees are either supplied by the manufacturer or need to be deduced from hardware sheets. For some hardware they are hard to come by, for other they are incomplete. If documentation is available, extracting the right information from these sheets is tedious. Address space information is intertwined with programming models and lengthy description of which device register bits have what functionality. The document describing the OMAP4460 [22] is 5820 pages long! Although the documentation for the OMAP is one of the better examples, it still falls for the unique address space illusion when presenting some of the information. If formal specifications would be supplied by hardware manufacturers the labour needed to parse data sheets would be greatly reduced. In the course of this thesis I designed a DSL to describe hardware using ADNs.

Barrelfish tries to reduce the complexity of platform dependent mechanism code by pushing policy decisions to the SKB. The SKB allows to describe desired properties of allocation policies declaratively. In Chapter 3.2 it is discussed in more detail. Notable efforts to configure hardware with declarative algorithms include the work by Schüpbach et al. [18]. They applied high-level declarative language techniques to PCI programming. They were able to allocate resources more efficiently than current solutions used in Linux and written in C. Altough incuring a performance overhead, it was acceptable and compensated by the improved allocation policy and lower code complexity. A wide range of PCI devices also exhibits so called “quirks”, bugs that require special treatment when configuring the hardware. The linux kernel includes an extensive collection of workarounds for such quirks, all written in C and therefore hard to maintain. Schüpbach et al. found that by declaratively describing the desired policy, these quirks can be handled much cleaner. By adding address space knowledge to the SKB and using it to configure hardware, I build on their work.

Parallel to this thesis, Humbel et al. [14] worked on applying the formal model to declarative interrupt programming.

Chapter 3

Background

The first part of this chapter is an introduction to the formal model this thesis relies on. In the second part, Barrelfish's System Knowledge Base is discussed to provide the necessary background about the service and the constraint logic programming (CLP) system it uses.

3.1 Address Decoding Nets

This thesis relies on a formal model developed by Achermann et al. I refer tit as an ADNs. ADNs formally capture address decoding. In the following I give a description of how they model address spaces. As the goal of this thesis is to apply the model to hardware configuration, the description is given from a practical point of view. I refer to the original work for a more formal specification [1].

In Chapter 1, I already used the Texas Instruments OMAP4460 to illustrate the complexity of today's computer systems. I use it again here to illustrate how ADNs work. A heavily simplified addressing block diagram of the OMAP is depicted in Figure 3.1. We only show the interaction between the address spaces as seen by the Cortex-A9 cores and the Cortex-M3 cores. Also note, that the `L1` MMUs and the corresponding virtual address spaces are not shown in the diagram. The top level blocks (`Cortex-A9 1`, `Cortex-A9 2` and `Cortex-M3 MIF`) depict what would traditionally be described as the physical address spaces of the cores.

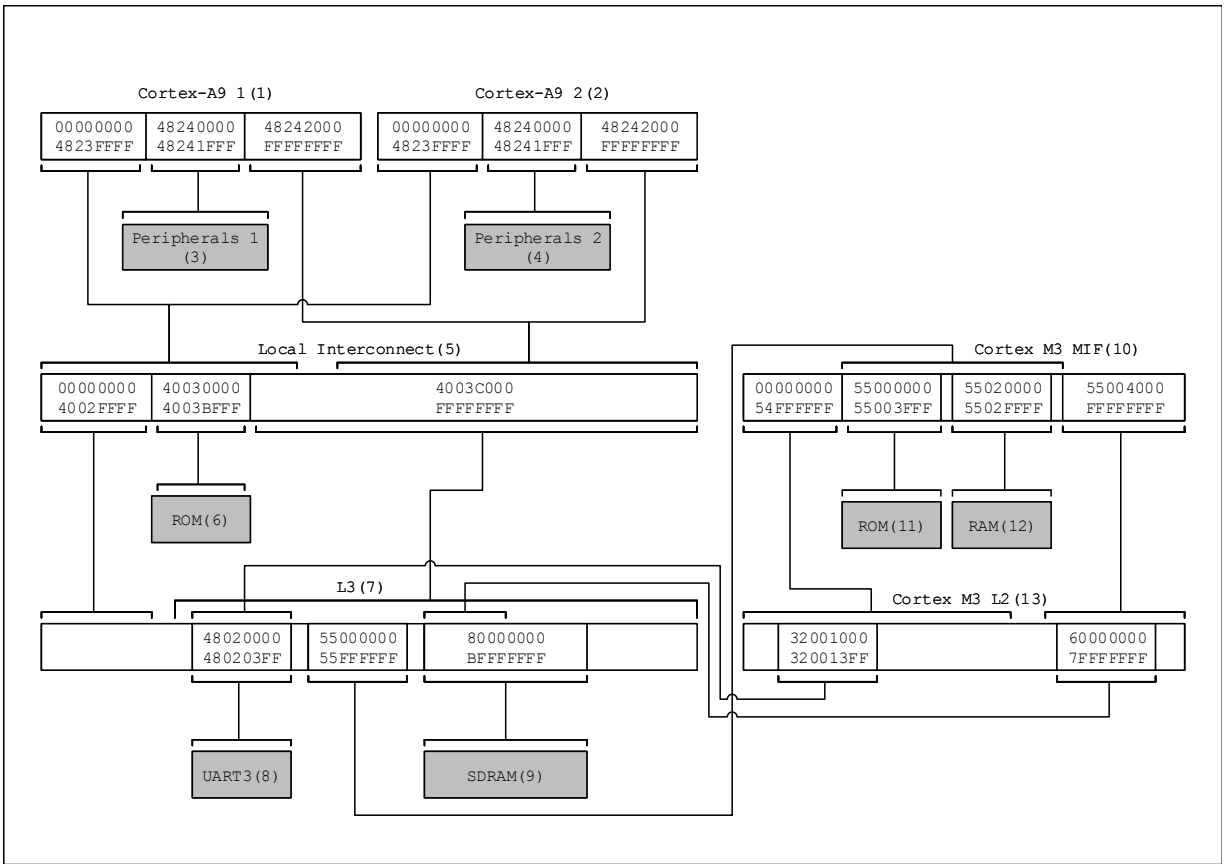

Figure 3.1: Simplified addressing block diagram of the Texas Instruments OMAP 4460 SoC (adapted from [1])

There are two different kinds of blocks in the diagram. The dark ones represent hardware components where address translation terminates. In our case, these are memory nodes or device registers that can be read from or written to. The other kind of nodes take incoming addresses and forward them to other nodes. This is e.g. the case for MMUs. In an ADN, hardware components are represented by *nodes*. In the first case a node is said to *accept* an address and in the second one it *translates* the address.

If we look at the two A9 cores, we can see, that they each have their own private peripheral region. If we decode address `0x48240000`, we end in different nodes depending on the core we start from. Therefore, an address is not enough to completly define a location in a system. It needs to be qualified by the identifier of the node it is used at. In an ADN addresses and node identifiers are natural numbers. In the diagram the node identifiers are written in parantheses behind the component names. The tuple of a node identifier and an andress qualified by it is called a *name*. Nodes therefore do not simply

translate addresses to other addresses but to names. A node is completely defined by its set of accepted addresses and the set of translated addresses, along with the names it translates them to.

We can now define the *accepted names set* and the *decode relation* of an ADN:

Accepted names set The set of all names where the name's address is accepted by the node corresponding to the name's node identifier.

Decode relation The set of all pairs of names where the first name's address is translated to the second name by the node identified by the first name.

From these two we can define the *resolution function*:

Resolution function The resolution function maps each name to a set of accepted names, where each pair of the mapped name and an element of the image set is in the reflexive transitive closure of the decode relation.

Assume that all dark nodes in the diagram have the same size as their direct predecessor and their accepted range starts at `0x0`. Examples of accepted names would then be `(9,0x0)` or `(11,3FFF)`. The following would be examples for pairs of names in the decode relation:

```
((7,0x55000000),(10,0x55000000))
((1,0x48240000),(3,0x0))
```

To illustrate how the resolution function works, let's look at the following name:

```
(10,0x32001000)
```

We could first calculate the reflexive transitive closure of the decode relation. However, the easier and more intuitive way is to interpret the tuples in the relation as edges in a directed graph with the names as nodes.

A particular name then resolves to all accepted names that can be reached by starting from that name and following the edges (including the name itself if is in the set of accepted names). `(10,0x32001000)` is not an accepted name so we follow the only outgoing edge to `(13,0x32001000)`, which is not an accepted name either. We then get to `(7,0x48020000)` and finally to `(8,0x0)`. The node identified by `8` accepts address `0x0`, meaning that `(10,0x32001000)` resolves to `(8,0x0)`. As there were no alternative edges to take, and none of the names along the way was an accepted name, this is in fact the only name to which it resolves.

By modelling systems as a directed graph, ADNs can capture even complicated address translation chains. In this thesis they will be used to model the OMAP4460 and provide the basis for correctly configuring hardware.

3.2 Barrelfish's System Knowledge Base

In Barrelfish there is a central system service called the SKB. Its purpose is to store gathered knowledge about the system and make it available to clients [19]. I extend this knowledge to address spaces using ADNs. The SKB enables the usage of high-level declarative language techniques. As already mentioned in Chapter 2, these techniques provide a more natural way to describe policies than C does. Barrelfish derives configuration parameters derived from these policies. The parameters are then used in fast, low-level C code to configure the system accordingly. This separation of policy and mechanism code helps to keep complexity out of low-level code and makes it easier to reason about complex policies. It also allows for complex algorithm's to be run off the system's fast path.

The SKB is implemented using an embedded version of ECLiPSe [7], a CLP system. The data in the knowledge base is represented in ECLiPSe's Prolog dialect. Algorithms and queries run against the SKB are also written in Prolog. Important aspects of Prolog and ECLiPSe for this thesis are discussed in the next section.

The SKB comes with a client library for C programs. For maximum expressiveness, the queries are written as C strings.

3.3 ECLiPSe

The language used to write ECLiPSe programs is a superset of Prolog. In the following I present the most important concepts for this thesis and refer to the ECLiPSe Tutorial Introduction [6] for a more complete description.

3.3.1 Basics

Prolog only has a single data type called a *term*. In ECLiPSe they are built from *numbers*, *strings*, *atoms*, *lists* and *structures* (also called *compound terms*). Numbers, strings and lists are self explanatory. Atoms are symbolic constants (similar to an enumeration type constant). They are either single quoted or start with a lower case letter. Structures are aggregated terms e.g.

```
author(george, orwell)
book('romeo and juliet', author(william, shakespeare), tragedy)
```

They have a name and a fixed number of arguments called their *aritiy*. The combination of a structure's name and aritiy is called its *functor*. It is often written as *name/arity* e.g. `author/2` or `book/3`. For improved readability, ECLiPSe allows to give names to the arguments. We first specify a template with

```
:- local struct( author(firstname, lastname) ).
```

We can now write `author/2` structures like this:

```
author{
    firstname:george,
    lastname:orwell
}
```

Named structures support inheritance. By sepcifying a template like

```
:- local struct( book(title, author:author, genre) ).
```

We can conveniently write `book/3` structures like this:

```
book{
    title:'the lord of the rings',
    firstname:'j.r.r.',
    lastname:'tolkien',
    genre:fantasy
}
```

Alternatively we can still specify the author with the `author` field and a nested `author/2` structure.

In Prolog two terms are equal if they can be pattern matched. An important concept in Prolog is *unification*. It is an extension of pattern matching, that allows the terms to contain variables. A variable in Prolog starts with an upper case letter or an underscore. Two terms can be unified if they are either equal or if the contained variables can be instantiated with values such that the terms become equal.

Prolog programs are based on predicate invocations. If we invoke a predicate, ECLiPSe will return its truth value, e.g.

```
:- member(2,[1,2]).
Yes
```

Such a predicate invocation is called a *goal*.

A predicate is defined by one or several *clauses*. A clause is either a *fact* or a *rule*. Facts are clauses without a body like `mouse(jerry).` They simply state that something is true. Rules are clauses that make the truth value dependent on other goals like `animal(X) :- mouse(X).` Goals can be combined with a comma (conjunction) or a semicolon (disjunction).

To see how ECLiPSe finds solutions for goals, we look at the goal `animal(jerry).` Assume the following definitions.

```
animal(X) :- cat(X).
animal(X) :- mouse(X).

cat(tom).
mouse(jerry).
```

ECLiPSe tries to unify the goal with each of the `animal/1` clauses in order. Both clauses can be unified with the goal by instantiating `X` with `jerry`. ECLiPSe chooses the first clause and remembers that there would be other choices. It creates a *choice-point*. Now it has to find a solution for `cat(jerry).` As there are no clauses of the `cat/1` predicate that unify with the goal, the process reaches a dead end. If there wouldn't be any previous choice-points, ECLiPSe would now report that there are no solutions. However, there is a choice-point to which it can *backtrack* and make a different choice. When choosing the second clause of `animal/1`, it can make the initial goal true: It again instantiates `X` with `jerry`. As `mouse(jerry)` is a fact, the goal is true.

If we put variables in a goal, ECLiPSe will report instantiations that make the goal true:

```
:- animal(X).
X = tom
Yes (solution 1, maybe more)
```

If we ask for more solutions, ECLiPSe will backtrack and enumerate all solutions. We can ask for all solutions at once by using the built-in predicate **`findall`**`/3`. If we are not interested in the values that make the goal true, we can do the following:

```
:- animal(_).
Yes
```

If we do not specify a value for a name in a structure, ECLiPSe assumes we do not care. The following two are equivalent:

```
author{
    lastname:rowling
}

author{
    firstname:_,
    lastname:rowling
}
```

3.3.2 Constraints

ECLiPSe includes an interval constraint solver library. It allows to constrain variables to ranges of integers or even real numbers. However, in this thesis we only deal with integer constraints. To constrain a variable to an interval we can write

```
:- X :: 1..3.
X = X{1..3}
Yes
```

As we can see, ECLiPSe does not enumerate possible values for constrained variables until we ask it to do so with `labeling/1`:

```
:- X :: 1..3, labeling([X]).
X = 1
Yes (solution 1, maybe more)
```

We can also use arithmetic constraints to relate variables:

```
:- X :: 1..3, Y #> X, Y #< 3.
X = 1
Y = 2
Yes
```

We can also get information about the domain of variables. To e.g get the bounds we can do:

```
:- X :: 1..3, get_bounds(X, Min, Max).
X = X{1..3}
Min = 1
Max = 3
Yes
```

However, this only gives lower and upper bounds, not information about non continuous domains:

```
:- X :: 1..3, X #\= 2, get_bounds(X, Min, Max).
X = X{[1,3]}
Min = 1
Max = 3
Yes
```

We will see in Section 5.2 that there are problems which are solved much more efficiently with constraints than with pure backtracking.

Chapter 4

Sockeye

On platforms that implement dynamic discovery mechanisms (cf. Chapter 2) the ADN describing the platform could probably (at least partially) be inferred from the discovered information. However, this is beyond the scope of this thesis. Also, various platforms do not implement such mechanisms e.g. the PandaboardES used for this thesis. Even on platforms that have such mechanisms, an OS often requires more information to correctly configure the hardware. To use ADNs for hardware configuration, we therefore need a way to produce specifications describing ADNs. Hardware specifications are typically written by hand. Thus, a language used for the purpose must be well readable and allow a programmer to work in an efficient way. A very important feature for efficiency is the possibility to reuse code. In the context of hardware specifications one wants to be able to reuse specifications written for components like a particular processor type.

The original ADN paper proposes a concrete syntax. To cater to the above requirements, I augmented this syntax. The resulting DSL is called *Sockeye*. Sockeye's syntax and features and their semantics are described in Section 4.1. I also built a compiler for Sockeye and its implementation is described in Section 4.4.

A more user oriented description can be found in the Barrelfish Technical Note on Sockeye [16].

4.1 Language Overview

Sockeye was designed as a DSL that is primarily written by hand as opposed to being generated from specifications in another format. The main design goals for Sockeye were therefore

- **Readability:** The produced code should be well readable such that specifications can be understood and maintained.
- **Efficiency:** The developer should be able to write as little code as possible.

To achieve the first goal, Sockeye uses similar lexical conventions as C does. C is still mostly the language of choice for systems programming. Hence, a lot of developers writing hardware specifications should, at least to some degree, be familiar with it. The exact conventions used are documented in Appendix A. To further improve readability I made some changes to the ADN syntax. The resulting basic syntax along with its semantics is described in Section 4.2.

The second goal is mainly achieved by enabling developers to reuse code. Even within the same platform, there is potential for reusing code. This can already be seen in the heavily simplified version of the OMAP4460 depicted in Figure 3.1. Figure 4.1 shows a slightly modified version of the diagram, splitting it into reusable modules.

Sockeye allows one to package reusable parts into modules. These modules can then be instantiated multiple times within the same platform specification. If placed in a separate file, code can even be reused in other specifications. The language features that make this possible are described in Section 4.3.

4.2 Basic Syntax

In the following the basic syntax for Sockeye is described. The differences to the orignal ADN syntax are stated in Section 4.2.5.

I use EBNF to describe the syntax. Terminals are **bold** while non-terminals are *italic*. The non-terminals *letter*, *decimal* and *hexadecimal* are self explanatory. The non-terminal *identifier* represents alphanumerical strings starting with a letter. Additionally, dashes and underscores are allowed. The exact definitions for these non-terminals can be found in Appendix A.

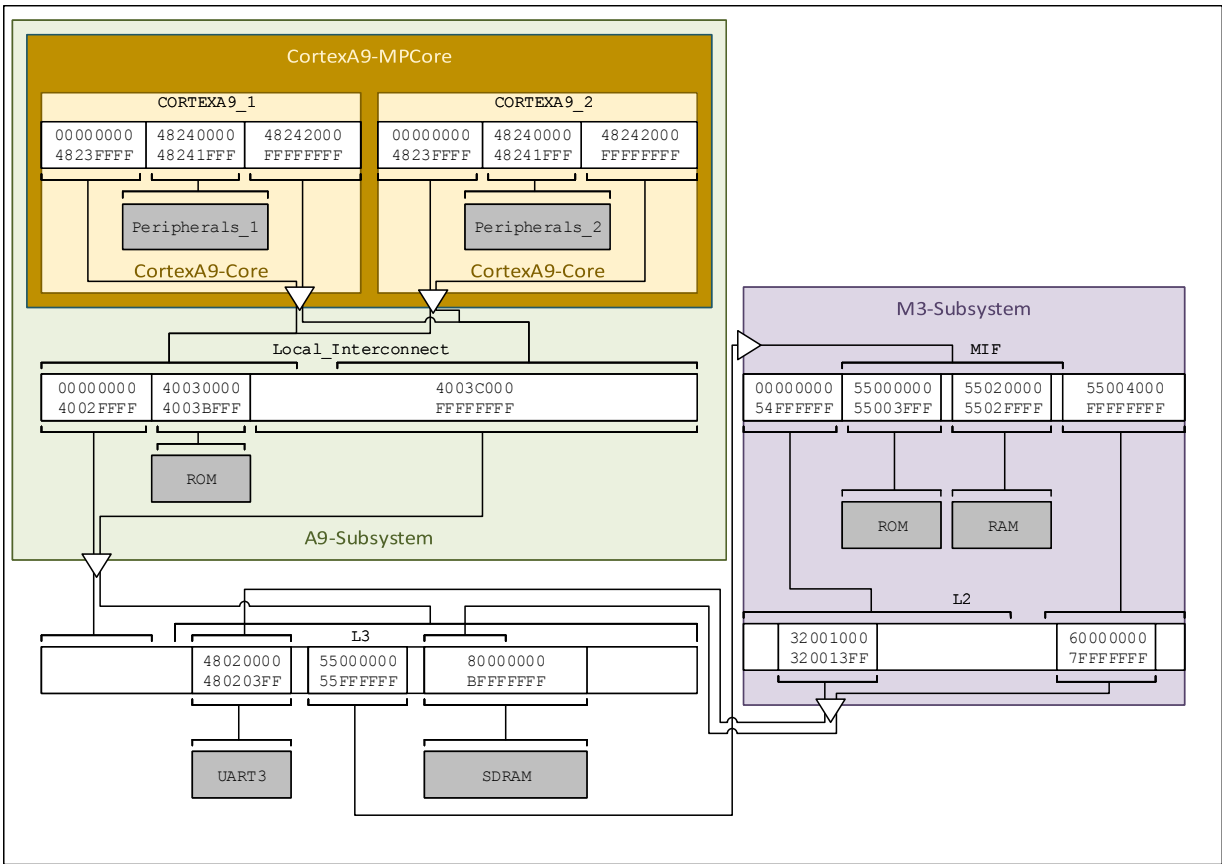

Figure 4.1: Simplified modular view of the Texas Instruments OMAP 4460 SoC

Additionally, there are examples for every syntactic construct. The examples are all taken from the simpliefied version of the OMAP4460 depicted in Figure 4.1.

4.2.1 Node Declarations

A node declaration contains one or more identifiers and the node specification. If several identifiers are given, an equivalent node is declared for each of them. The order in which the nodes are declared does not matter.

Syntax

$$node\_decl = \left\{ identifier \textbf{ is } node\_spec \;\middle|\; identifier \{ \textbf{,} \; identifier \} \textbf{ are } node\_spec \right\}$$

Example

```
SDRAM is ...

Peripherals_1,
Peripherals_2 are ...
```

4.2.2 Node Specifications

A node specification consists of a type, a set of accepted address blocks, a set of address mappings to other nodes, a set of reserved address blocks and an overlay. All of these are optional. Not specifying a set is equivalent to specifying an empty set. The type does not have formal semantics but is used to attach meta-data to nodes. Currently there are three types: `core`, `device` and `memory`. A fourth internal type `other` is given to nodes for which no type is specified. The `core` type designates the node as a processor core. The `device` type specifies that the accepted addresses are device registers while the `memory` type is for memory nodes like RAM or ROM. If an overlay node is given, all addresses not captured by the accept and translate sets will be translated to the same addresses in the overlay node. Overlays are specified as a node identifier and a number of address bits. An overlay will span addresses from `0x0` to $\texttt{0x2}^{\texttt{bits}} - \texttt{1}$. The reserved address blocks are only relevant in conjunction with overlays and are used to exclude specific addresses from the overlay.

Syntax

$$
\begin{aligned}
\textit{node\_spec} &= \left[\ \textit{type}\ \right]\ \left[\ \textit{accept}\ \right]\ \left[\ \textit{map}\ \right]\ \left[\ \textit{reserved}\ \right]\ \left[\ \textit{overlay}\ \right] \\
\textit{type} &= \textbf{core} \mid \textbf{device} \mid \textbf{memory} \\
\textit{accept} &= \textbf{accept [}\ \{\ \textit{block\_spec}\ \}\ \textbf{]} \\
\textit{map} &= \textbf{map [}\ \{\ \textit{map\_spec}\ \}\ \textbf{]} \\
\textit{reserved} &= \textbf{reserved [}\ \{\ \textit{block\_spec}\ \}\ \textbf{]} \\
\textit{overlay} &= \textbf{over}\ \textit{identifier/decimal}
\end{aligned}
$$

Example

```
SDRAM is memory accept [...]
L3 is map [...]
Local_Interconnect is reserved [...] over L3/32
```

4.2.3 Block Specifications

To talk about a continuous range of addresses at once instead of having to specify each address separately, Sockeye offers address blocks. A block is specified by its start and end address. If the start and end address are the same, the end address can be omitted. Sockeye also supports specifying a block as its base address and the number of address bits the block spans: A block from `0x0` to `0xFFF` with a size of 4kB can be specified as `0x0/12`. Addresses are specified as hexadecimal literals.

Syntax

$$block\_spec = hexadecimal\left[-\ hexadecimal \;\middle|\; Idecimal\right]$$

Example

```
UART3 is accept [0x0-0xFFF]
UART3 is accept [0x0/12]    // same as 0x0-0xFFF
ROM is accept [0x44]        // same as 0x44-0x44
```

4.2.4 Map Specifications

A map specification is a source address block, a target node identifier and optionally a target base address to which the source block is translated within the target node. If no target base address is given, the block is translated to the target node starting at `0x0`. Multiple translation targets can be specified by giving a comma-separated list of targets.

Syntax

$$map\_spec = block\_spec\ \textbf{to}\ identifier\ \left[\textbf{at}\ hexadecimal\right]\ \left\{\textbf{,}\ identifier\ \left[\textbf{at}\ hexadecimal\right]\right\}$$

Example

```
/*
 * Translate 0x60000000-0x7FFFFFFF
 * to L3 at 0x80000000-0x9FFFFFFF:
 */
L3 is map [0x60000000/29 to L3 at 0x80000000]

/*
 * This is the same as 0x80000000/30 to SDRAM at 0x0:
 */
L3 is map [0x80000000/30 to SDRAM]
```

4.2.5 Differences to the Original Syntax

The following syntax was originally proposed for ADNs:

$$\begin{aligned}
net_s &= \left\{\mathbb{N}\ \textbf{is}\ node_s \;\middle|\; \mathbb{N}\textbf{..}\mathbb{N}\ \textbf{are}\ node_s\right\} \\
node_s &= \left[\textbf{accept [}\ \left\{block_s\right\}\ \textbf{]}\right]\ \left[\textbf{map [}\ \left\{map_s\right\}\ \textbf{]}\right]\ \left[\textbf{over}\ \mathbb{N}\right] \\
map_s &= block_s\ \textbf{to}\ \mathbb{N}\ \left[\textbf{at}\ \mathbb{N}\right]\ \left\{\textbf{,}\ \mathbb{N}\ \left[\textbf{at}\ \mathbb{N}\right]\right\} \\
block_s &= \mathbb{N}\textbf{–}\mathbb{N}
\end{aligned}$$

To improve readability and address some practical issues, the basic syntax for Sockeye slightly differs from this. The following changes were made:

Node identifiers: Sockeye uses strings to identify nodes while the original syntax uses natural numbers. By using strings, nodes can be given more meaningful names,

which greatly improves maintainability. However, this is equivalent as a one-to-one mapping from strings to natural numbers can easily be found (e.g. by enumerating the strings in their alphanumerical order).

Node types: Sockeye allows to attach meta-data to nodes that is not captured by the formal model. Currently this is restricted to node types. The type is used to designate a node as a processor core, a memory node or a collection of device registers. Types are purely meta-data, they do not have any formal semantics.

Overlays: The original syntax also allows to specify an overlay for a node. An overlay is a default translation target: All addresses that are neither accepted nor explicitly translated are implicitly translated to the same address in the overlay target node. However, in practice a hardware component only uses a certain amount of bits for addresses. In Sockeye an overlay is restricted to a specified amount of address bits. Additionally, to be able to exclude specific address regions from an overlay, Sockeye introduces a set of reserved addresses. Neither of these changes affect the expressiveness: By explicitly adding addresses affected by the overlay to the translate set of a node, the node can be transformed into an equivalent node without overlays. Conversely a finite overlay can always be expressed with explicit translation entries. Infinite overlays do not occur with real hardware.

Default translation target address: In the original syntax, if no target base address is given for a map specification, the address block is translated to the same block in the target node. In Sockeye they are translated to the block of the same size starting at `0x0`. This was changed due to the mapping to `0x0` being used more often in practice. Omitting the target base address is only syntactic sugar. Changing the convention has no impact on the expressiveness of the language as the original behaviour can still be achieved by stating the base address explicitly.

4.3 Advanced Features

The basic syntax allows to fully describe an ADN. However, apart from specifying multiple equal nodes at once (see Section 4.2.1), it does not offer any form of code reuse. As Figure 4.1 shows, a hardware component might appear multiple times within a platform (like the Cortex-A9 core). Components like the Cortex-A9 MPCore are also used across different platforms. Instead of having to respecify the same thing over

and over again, a developer should be able to write such specifications once, and reuse them. With its module and import system, Sockeye enables code reuse for exactly these use cases.

The features enabling code reuse along with their syntax are described in the following sections. As in Section 4.2 I use EBNF, **bold** terminals and *italic* non-terminals. An example specification describing the simplified OMAP4460 in Figure 4.1 can be found in Section 4.3.4.

It is important to note that all the features described in this section do not add to the expressiveness of Sockeye. All specifications using these features can be transformed to equivalent ones using only the basic syntax from Section 4.2. In fact, we will see in Section 4.4 that this is exactly what the Sockeye compiler does. However, they allow one to describe hardware with much less redundancy in the code.

4.3.1 Modules

The main constructs to enable code reuse in Sockeye are modules. A module encapsulates an ADN. By instantiating the module, the contained ADN can be integrated into a larger ADN. However, just fusing the encapsulated ADN with the surrounding one would lead to problems: If a module is instantiated twice or even if two modules declare nodes with the same identifier, name clashes would occur. One way to resolve them, would be to enforce globally unique names. However, this would defeat the purpose of a module: A developer would than have to care about the exact contents of the module, instead of just being able to use it. This is why, a module instantiation always creates a new namespace nested inside the namespace it is instantiated in. To further guarantee encapsulation, modules have a clearly defined interface in the form of *ports*. Namespace boundaries can not be crossed by simply referencing nodes inside another namespace. This is enforced by the syntax, as specifying a namespace when referencing a node is not possible (see Sections 4.2.2 and 4.2.4). Ports allow to cross these boundaries. The direction in which the namespace boundary can be crossed depends on the port type:

Input ports allow to cross a namespace boundary into a subspace. To put it differently, an input port allows a node outside a module to forward addresses to a node inside the module.

Output ports do the opposite: They allow a node inside the module to forward addresses to a node outside the module.

A port therefore has an *ingoing end* (on the side *from* which the crossing happens) and an *outgoing end* (on the side *to* which the crossing happens). In Figure 4.1 the ports are marked as arrows pointing from their ingoing to their outgoing end. Arrows that point into a module are input ports and the ones that point out of modules are output ports.

When a module is instantiated, a list of port mappings can be specified. A port mapping creates a proxy node on the ingoing end of a port. Nodes on the ingoing side of the port can reference the node on the other end via this proxy node. Like overlays, ports also have a width, that specifies the size of the address range forwarded by the proxy node.

In Figure 4.1 we identified the Cortex-A9 MPCore as a potentially reusable module. Its specifications [2] state, that it can come with one to four cores and that the address of the private peripheral region is implementation dependent. To be able to reuse modules even across such differences, they are parametrisable. A module with parameters is called a *module template*: The topology of the ADN it encapsulates depends on the values of the parameters the template is instantiated with. Module parameters are typed and there are two types of parameters: *address parameters* and *natural parameters*. Address parameters allow to parametrize addresses in node specifications (like the base address of the private peripheral region of a Cortex-A9 core). Natural parameters are used to parametrise the number of certain nodes in the encapsulated ADN. The construct that makes this possible is described in the next section. Parameters can also be passed on as arguments to module template instantiations in the module body.

A module declaration starts with the keyword `module` and a unique module name. An optional list of typed parameters can be specified, making the module a template. The module body is enclosed in curly braces and starts with the port definitions. The rest of the body is node declarations and module instantiations. If the module has address parameters the names of the parameters can be used wherever in the body an address is expected. Doing arithmetic on parameters is currently not supported.

Module instantiations start with the module name and in the case of a module template with the list of arguments. After that, the identifier of the namespace in which the module should be instantiated has to be given, followed by an optional list of port mappings.

Syntax

$$
\begin{aligned}
\textit{mod\_decl} &= \textbf{module}\ \textit{identifier}\,[\textit{param\_list}]\ \textbf{\{}\ \{\textit{input\_port} \mid \textit{output\_port}\}\ \textit{body}\ \textbf{\}}\\
\textit{param\_list} &= \textbf{(}\,[\ \textit{parameter}\{\textbf{,}\ \textit{parameter}\}\ ]\,\textbf{)}\\
\textit{parameter} &= \textit{param\_type}\ \textit{identifier}\\
\textit{param\_list} &= \textbf{(}\,[\ \textit{parameter}\{\textbf{,}\ \textit{parameter}\}\ ]\,\textbf{)}\\
\textit{param\_type} &= \textbf{addr} \mid \textbf{nat}\\
\textit{input\_port} &= \textbf{input}\ \textit{identifier/decimal}\{\textbf{,}\ \textit{identifier/decimal}\}\\
\textit{output\_port} &= \textbf{output}\ \textit{identifier/decimal}\{\textbf{,}\ \textit{identifier/decimal}\}\\
\textit{body} &= \{\ \textit{node\_decl} \mid \textit{mod\_inst}\ \}
\end{aligned}
$$

$$
\begin{aligned}
\textit{mod\_inst} &= \textit{identifier}\,[\textit{arg\_list}]\ \textbf{as}\ \textit{identifier}\ [\ \textbf{with}\ \{\textit{input\_map} \mid \textit{output\_map}\}\ ]\\
\textit{argument} &= \textit{decimal} \mid \textit{hexadecimal} \mid \textit{identifier}\\
\textit{arg\_list} &= \textbf{(}\,[\ \textit{argument}\{\textbf{,}\ \textit{argument}\}\ ]\,\textbf{)}\\
\textit{input\_map} &= \textit{identifier}\ \textbf{>}\ \textit{identifier}\\
\textit{output\_map} &= \textit{identifier}\ \textbf{<}\ \textit{identifier}
\end{aligned}
$$

Example

```
/*
 * Module for a Cortex-A9 core
 * The address of the private peripheral region is parametrised
 */
module CortexA9-Core(addr periphbase) {
    input CPU/32
    output Interconnect/32

    Peripherals is device accept [0x0/13]

    CPU is map [
            periphbase/13 to Peripherals
        ]
        over Interconnect/32
}

/*
 * Instantiate the module and map the ports
 */
CortexA9-Core(periphbase) as Core with
        CORTEXA9_1 > CPU
        Local_Interconnect < Interconnect
```

4.3.2 Templated Identifiers

In the case of the `CortexA9-MPCore` module in Figure 4.1, the number of cores can vary. We would therefore give the `CortexA9-MPCore` module a parameter of type `nat` that controls the number of cores. Now we need to be able to say that we want to instantiate the `CortexA9-Core` module as many times as there should be cores. Templated identifiers allow us to do this.

There are two forms of templated identifiers: *interval templates* and *simple templates*. A templated identifier is called an interval template if it contains one or several intervals e.g.

```
CORTEXA9_{[1..2]}
```

It represents the set of identifiers, that is obtained when replacing the intervals with all possible combinations of their members. In the above example these are

```
CORTEXA9_1
CORTEXA9_2
```

In a system that has two Cortex-A9 MPCore modules with two cores each, we might want to number them with two indices, one for the MPCore module and one for the core:

```
CORTEXA9_{[1..2]}_{[1..2]}
```

This interval template would represent the following four identifiers:

```
CORTEXA9_1_1
CORTEXA9_1_2
CORTEXA9_2_1
CORTEXA9_2_2
```

Interval templates can be used as identifiers in node and port declarations, and as namespace identifiers in module instantiations. In each of these cases it would be equivalent to repeat the syntactic construct for all of the identifiers represented by the interval template. Optionally the intervals can be assigned to index variables that can then be used in the corresponding syntactic construct. Writing

```
CORTEXA9_{c in [1..2]} are map [0x48240000 to Peripherals_{c}]
```

is equivalent to writing

```
CORTEXA9_1 is map [0x48240000 to Peripherals_1]
CORTEXA9_2 is map [0x48240000 to Peripherals_2]
```

An identifier template that just uses an already defined index variable but does not itself contain new intervals is called a simple template. Simple templates can be used in any place a node identifier is expected. This includes the places where interval templates can be used, identifiers of translation destination nodes, and overlays. Identifier templates can not be used in module parameter or index variable names.

While this already allows one to save a few lines of code, there is a more powerful use case: When an interval template is used inside a module, the natural parameters of the module can be used as interval limits. To come back to the use case from the beginning

of this section, assume we have the following module declaration for a simple version of a single Cortex-A9 core:

```
module CortexA9-Core {
    input CPU/32

    CPU is core map [ ... ]
}
```

We can now declare a module for the Cortex-A9 MPCore with a variable number of cores:

```
module CortexA9-MPCore(nat cores) {
    CortexA9-Core as Core_{c in [1..cores]} with
        CORTEXA9_{c} > CPU
}
```

The number of times the `CortexA9-Core` module is instantiated depends on the parameter `cores`. By giving the index variable a name, we can make the names of the proxy nodes for the processors depend on the module instance: The processor from the first `CortexA9-Core` module is mapped to `CORTEXA9_1`, the one from the second one to `CORTEXA9_2` and so on. This allows to parametrise a module over the number of times certain nodes or other modules are contained in it.

Syntax

$$\begin{aligned}
\mathit{interval_templ} &= \mathit{identifier}\{\ \left[\mathit{var}\ \mathbf{in}\ \right]\ \mathit{interval}\}\left[\mathit{interval_templ} \mid \mathit{simple_templ} \mid \mathit{identifier}\ \right] \\
\mathit{simple_templ} &= \mathit{identifier}\{\mathit{var}\}\left[\mathit{simple_templ} \mid \mathit{identifier}\ \right] \\
\mathit{var} &= \mathit{identifier} \\
\mathit{limit} &= \mathit{decimal} \mid \mathit{identifier} \\
\mathit{interval} &= [\mathit{limit}..\mathit{limit}]
\end{aligned}$$

4.3.3 Imports

With modules and templated identifiers partial specifications are reusable inside the same file and therefore the same specificiation they were defined in. Imports allow to put modules into separate files, such that code can also be reused across different specifications in a library-like fashion. Furthermore, they can help improve readability by giving the ability to split large specifications across several files. Imports have to be specified at the very top of a Sockeye file. An import will cause all modules from the specified file to be loaded. Anything declared outside a module will be ignored.

Syntax

$$import = \textbf{import}\ \{\ letter \mid \textbf{/}\ \}$$

4.3.4 Sockeye Files

Finally, a complete Sockeye file consists of imports, module declarations and the specification body (node declarations and module instantiations).

Syntax

$$sockeye = \{\ import\ \}\ \{\ mod\_decl\ \}\ \{\ node\_decl \mid mod\_inst\ \}$$

Listing 4.1 shows a Sockeye specification that describes the simplified OMAP4460 depicted in Figure 4.1.

```
module CortexA9-Core(addr periphbase) {
    input CPU/32
    output Interconnect/32

    Peripherals is device accept [0x0/13]

    CPU is map [
            periphbase/13 to Peripherals
        ]
        over Interconnect/32
```

```
}

module CortexA9-MPCore(nat cores, addr periphbase) {
    output Interconnect/32

    CortexA9-Core(periphbase) as Core_{c in [1..cores]} with
        CORTEXA9_{c} > CPU
        Interconnect < Interconnect
}

module A9-Subsystem {
    output Interconnect/32

    ROM is memory accept [0x0-0xBFFF]

    Local_Interconnect is map [
            0x40030000 to ROM
        ]
        over Interconnect/32

    CortexA9-MPCore(2, 0x48240000) as MPU with
        Local_Interconnect < Interconnect
}

module M3-Subsystem {
    input MIF/32
    output L3/32

    ROM is memory accept [0x0/14]
    RAM is memory accept [0x0/16]

    MIF is map [

        ]
        over L2

    L2 is map [
```

```
            0x32001000/10 to L3 at 0x48020000
            0x60000000/29 to L3 at 0x80000000
        ]
}

UART3 is device accept [0x0/10]
SDRAM is memory accept [0x0/30]

L3 is map [
    0x48020000/10 to UART3
    0x55000000/24 to M3_MIF at 0x55000000
    0x80000000/30 to SDRAM
]

A9-Subsystem as A9_SS with
    L3 < Interconnect

M3-Subsystem as M3_SS with
    M3_MIF > MIF
    L3 < L3
```

Listing 4.1: Sockeye specification for a simplified OMAP4460

Appendix B shows the EBNF specification for the whole syntax in one listing.

4.4 Compiler

In the previous sections I described the Sockeye hardware description language. To be able to use the Sockeye specifications for hardware configuration, I also implemented a compiler. The current implementation only features a backend for generating ADN representations for use in Barrelfish's SKB. However, the design makes it easy to add additional backends. This allows to extend the compiler to e.g. generate code that can be fed into an automated theorem prover. One could then prove certain properties about hardware as proposed in the original work about ADNs. In this section I describe the transformations and checks the compiler performs before running the code generator. The SKB representation for ADNs is discussed in detail in Chapter 5.

4.4.1 Implementation

The compiler first parses the specification and resolves the imports. It then makes sure that the described ADN is correct. Namely it checks two properties:

- Uniqueness of node identifiers
- Referential integrity

These properties are easy to check for specifications only using basic syntax. However, when advanced features are used, this is difficult: With templated identifiers aliasing can occur and the presence of ports complicates checking that a referenced node actually exists. This is why the compiler transforms the specifications and translates the advanced syntactical constructs into equivalent basic constructs. The transformation from Sockeye to "Basic Sockeye" is done in several separate steps:

Parsing: Checks syntax and resolves imports.

Type checking: Ensures type safety of the module parameters.

Module template instantiation: Turns module templates into fully defined modules by resolving parameter values. This then allows to instantiate identifier templates and check that all identifiers are unique.

Module instantiation: Instantiates modules and checks that all referenced nodes exist.

Note that the abstract syntax produced from this transformation differs from the basic syntax in one point: Node identifiers are not simply strings, but a name qualified by a list of namespaces (both represented by strings). This is necessary to ensure the uniqueness of identifiers in the presence of module instantiations. A string-only representation could have been achieved by e.g. concatenating all the strings with a separator in between. However, the chosen solution is more practical as one does not need to parse the string to separate out the parts.

In the following, the most important transformations and checks of each step are detailed. For a comprehensive list of checks performed by the compiler, see Appendix C.

Parsing As in every compiler, the parser checks that the syntax is correct and creates an abstract syntax tree (AST). It then recursively resolves the imports and parses the imported files. The resulting set of ASTs is merged into a single one.

Type Checking Before any transformations can be done, the AST needs to be type checked. The type checker first makes sure that all module names are unique. This is especially important as the definitions can come from several files, that are not necessarily all written by the same developer. It then goes on to check that all referenced parameters and index variables are actually defined and all parameters are used in accordance with their type. It also checks that all instantiated modules are defined and that they are instantiated with the correct number of arguments.

Module Template Instantiation In the final AST, node identifiers are represented by a name and a list of namespaces. Additionally all module instantiations create a new namespace. This means that we only need to check uniqueness of identifiers within the same module. However, as module parameters can be used in identifier templates, this check is still hard to perform. Depending on the values of the parameters, aliasing can occur: The templates `CPU_{[1..c]}` and `CPU_{[4..d]}` for example might represent overlapping sets of identifiers. After the parameter values are resolved, uniqueness is much easier to check.

In the module instantiation stage, module templates are transformed to fully defined modules as follows: For each combination of argument values with which a module template is instantiated, the compiler creates a new module. The body of the module is the same as in the module template, but all occurrences of the template parameters are replaced with their values. This transformation then allows to instantiate the identifier templates in the module body. This allows for easy checking whether all identifiers are unique.

Module Instantiation After checking that all node identifiers are indeed unique, the compiler still needs to ensure that all referenced nodes are actually declared. For nodes that are explicitly declared, this can easily be verified. However, with modules, the referenced nodes could also be proxy nodes, implicitly declared by port mappings. To be able to also check the integrity of such references, the compiler instantiates all modules. It qualifies the node names with the nested namespace they are in and creates explicit node declarations for the proxy nodes. With all nodes now explicitly declared, it is easy to verify referential integrity.

4.5 Evaluation

The original work on ADNs already demonstrates that real systems can be modelled with the syntax proposed there. Sockeye's syntax has the same expressiveness. However, it adds features to reuse code and describe systems more efficiently. To evaluate Sockeye, we first look at the code complexity of Sockeye specifications. We then compare Sockeye to device trees, the format for hardware specifications used in Linux.

To implement the hardware configuration tasks (Chapter 6), I needed specifications for the OMAP4460. I wrote those using Sockeye. The specification for the OMAP contains under 1000 lines of code[1]. Note that the focus for the specifications I wrote, was on the Cortex-A9 subsystem and the devices relevant for the use cases in Chapter 6. Therefore, these specifications do not yet cover the complete system. For the same reason, only the Cortex-A9 MPU was described as a reusable module. The module is about 40 lines of code, which can be shared with other specifications containing the same mulitprocessor. One could do the same for other parts of the system (e.g. the Cortex-M3 subsystem), to increase reuseability. The module for a single A9 core is about 20 lines of code. On the OMAP there are only two of these cores. The reduction in code from reusing this module in the OMAP specifications instead of describing each core separately, is therefore minor. However, other SoCs feature considerably more cores. E.g. the Cavium ThunderX, an ARMv8 based server SoC, can have up to 96 cores in a two socket configuration [4]. With module parameters and templated identifiers, the amount of code two write is the same, no matter whether a system has 1 or 1000 cores.

A device tree for the OMAP [10] has over 5000 lines of code[2]. Though the Sockeye specifications are incomplete, it is unlikely that adding the few missing parts would blow up the size by a factor of 5. However, the main advantage of Sockeye over device trees is the way hardware is modelled. As mentioned in Chapter 2, device trees assume physical addresses to be globally unique, making it hard to model contemporary hardware with them. Sockeye on the other hand, models hardware using ADNs, which capture the complex interactions between different address spaces. Additionally ADNs have precise semantics, while device trees are just a file format. Sockeye specifications, unlike device trees, can therefore also be used for formal reasoning about hardware.

[1] The lines of code were counted by looking at the total number of lines in the files. These numbers therefore include blank lines and comments.

[2] For the numbers to be comparable, the lines were counted the same way as above.

4.6 Summary

Sockeye is a DSL to describe hardware, based on the syntax proposed for ADNs. To make the language practical, I altered this syntax slightly and added features that allow the reuse of code. The expressiveness of the resulting concrete syntax for Sockeye is equivalent to the one from the ADN paper. To use Sockeye specifications for this thesis, I also implemented a compiler. It ensures the correctness of the described ADN by checking uniqueness of node identifiers and referential integrity. By using ADNs as the hardware model, Sockeye specifications can capture the complexity of contemporary hardware. The precise semantics of ADNs allow for formal reasoning about hardware based on the specifications. Both of these features make Sockeye a more powerful tool to describe hardware than other specification formats like device trees.

Chapter 5

SKB Schema

ADNs carry low-level knowledge about the system's address spaces (see Section 3.1). In Barrelfish such knowledge is stored in the SKB where it can be accessed with high-level, declarative algorithms (cf. Section 3.2). To extend the SKB's knowledge to address spaces, I designed a Prolog representation for ADNs. In database terms, such a representation is also called a *schema*. In this chapter I discuss the design of the SKB schema for ADNs and how it can be queried.

5.1 Representation

The design of the SKB schema was guided by two things:

Queries: The main operation on ADNs is address resolution. The schema needs to support resolution queries that are efficient enough to not have a huge impact on system performance.

Code Generation: Generating the representation from the Sockeye compiler's abstract syntax (see Section 4.4) should be as straight forward as possible.

I therefore started out with an exact representation of the abstract syntax for Sockeye. Using strings as node identifiers, qualified by the list of namespaces the node is in, also proved useful for querying ADNs: Querying a node using the same meaningful name as in the specifications, results in much more readable queries than using natural numbers. To further improve query readability, the Prolog representation uses named fields for the structures modelling the ADN (see Section 3.3). An additional structure

was added for names. While not directly part of the representation, they are a central concept in ADNs. They are needed to express resolution queries: The resolution function is defined in terms of the accepted names set and the decode relation. The decode relation is in turn also defined on names. The `name/3` structure therefore is used as the input and output type for queries. The template specifications for the named structures are shown in Listing 5.1.

```
:- local struct(node(
    id:node_id,
    spec:node_spec
)).

:- local struct(node_id(
    name,
    namespace
)).

:- local struct(node_spec(
    type,
    accept,
    translate
)).
```

```
:- local struct(map(
    src_block,
    dest_node,
    dest_base
)).

:- local struct(block(
    base,
    limit
)).

:- local struct(name(
    node_id:node_id,
    address
)).
```

Listing 5.1: Prolog representation for ADNs

The represented ADN is defined by the `node/2` predicate. Recall from Section 3.3 that predicates are defined by clauses being either facts or rules. In our case we assert `node/2` facts to state that all these nodes exist and make up the represented ADN.

Note the absence of the `overlay` and `reserved` properties in `node_spec/3`. An overlay is a default translation target and the reserved set specifies addresses explicitly not covered by the overlay. An equivalent node without an overlay or reserved addresses can be obtained as follows: Find all addresses covered by the overlay that are neither accepted, reserved or explicitly translated. Then, for each of them, add an entry to the translate set that translates the address to the same address in the overlay node. The Prolog backend for the Sockeye compiler implements this transformation on address block level. Having to consider only the translate sets of the nodes, simplifies the im-

plementation of the decode relation.

In the next section I discuss the implementation of resolution queries on top of this represenation.

5.2 Queries

To implement queries to the SKB representation of an ADN we need Prolog implementations of the definitions in Section 3.1:

- Accepted names set
- Decode relation
- Resolution function

Prolog programs are based on predicates. To represent a set we can implement a predicate that is true for all elements of the set and false for everything else. Similarly we can represent binary relations with predicates of arity two that are true when the pair of arguments is in the relation. I implemented a predicate `accepted_name/1` that represents the set of accepted names and a predicate `decode_step/2` that represents the decode relation. The predicate `decodes_to/2` then represents the reflexive transitive closure of the decode relation by checking the arguments for equality and calling `decode_step/2` recursively. The `resolve/2` predicate is just the conjunction of `decodes_to/2` and `accepted_name/1` invoked on the second argument.

To illustrate how these predicates can be used, we again look at a simplified version of the OMAP4460. The ADN for it is shown in Figure 5.1. For simplicity we ignore namespaces in this example.

To test whether a name is accepted we pose the query

```
:- accepted_name(
    name{
        name:'CORTEXA9_1',
        address:0x2A
    }
).
No
```

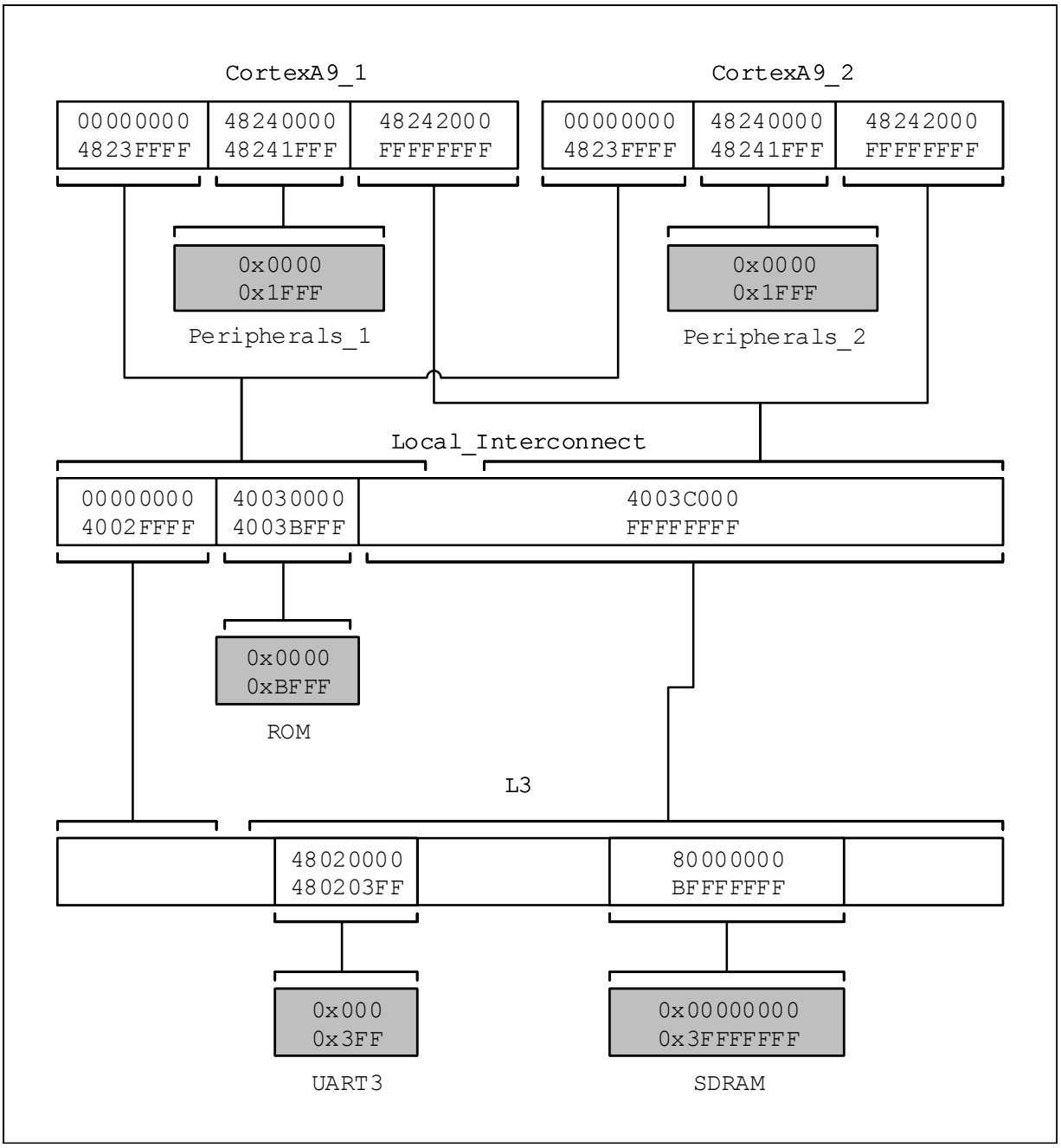

Figure 5.1: Simplified ADN of the Texas Instruments OMAP 4460 SoC

To test whether a name resolves to a name in a specific node we can use the following query:

```
:- resolve(
    name{
        name:'CORTEXA9_1',
        address:0x80000000
    },
    name{
        name:'SDRAM'
    }
).
Yes
```

However, if we add variables to our queries, ECLiPSe will try to find instantiations that make the given goal true. We might want to use the following query to find an address in node `'CORTEXA9_1'` that resolves to some address in node `'UART3'`:

```
resolve(
    name{
        name:'CORTEXA9_1',
        address:A
    },
    name{
        name:'UART3'
    }
).
```

With a plain Prolog implementation, ECLiPSe would enumerate all possible solutions for `A` by backtracking. However, using backtracking on addresses in the 32bit address space of the Cortex-A9 cores creates 2^{32} choice-points. This brute-force approach is highly inefficient and with larger address spaces it gets even worse.

Section 3.3 discusses how ECLiPSe allows to constrain variables to intervals and relate variables with arithmetic constraints. We can use this functionality for a more efficient implementation: Each entry in the translate set of a node maps a block of addresses to a block in another node. Instead of creating choice-points for all these addresses, we can constrain our variable to be in the source block. We then need an arithmetic constraint that correctly relates the source address to the target address. With this approach we only create a choice-point for each entry in the translate set. In the degenerate case where all the entries translate exactly one address, this is equivalent to a brute-force

search. However, in real hardware blocks almost never contain just one address. Often they are even the size of several kilobytes and larger.

When using constraints, ECLiPSe finds the domain for `A`. With the built-in predicate `labeling/1` we could now enumerate the addresses. Alternatively, by using e.g. `get_bounds/3` we can get the base and limit of the address range at which `UART3` is seen at the first Cortex-A9 core.

The SKB module implemented for this thesis includes a second version of the `resolve/2` predicate, which allows to write queries about address ranges. Instead of taking `name/3` structures as arguments, it operates on `region/4` structures. It is a wrapper around the basic `resolve/2` predicate and takes care of the conversion between names and regions. The template for the `region/4` structure is shown in Listing 5.2.

```
:- local struct(region(
    node_id:node_id,
    base,
    size
)).
```

Listing 5.2: Prolog representation for ADN regions

If we are interested in the region `UART3` is seen at from `CORTEXA9_1` rather than single addresses, we can rewrite the previous query as follows:

```
:- resolve(
    region{
        name:'CORTEXA9_1',
        base:B,
        size:S,
    },
    region{
        name:'UART3'
    }
).
B = 0x48020000
S = 0x3FF
Yes
```

We can now use this information to e.g. map the device into the virtual address space of a driver.

5.3 Evaluation

The Prolog representation and predicates for ADNs presented in this chapter are intended to be used in SKB queries. These queries should be easy to write and well readable to maximise maintainability of the code. Furthermore they need to allow expressive enough queries to be helpful in hardware configuration scenarios. In the following I give two examples of such queries to illustrate that this is possible.

At the end of the last section, we have already seen how the address region a node is seen at by another node can be found. This is a very common use case for hardware configuration. By strategically placing variables in the a `resolve/2` invocation one can also write more complex queries. To e.g. generate a memory map as seen from a specific core, the following goal can be used in a **`findall`**`/3` invocation:

```
resolve(
    region{
        name:'CORTEXA9_1',
        base:B,
        size:S,
    },
    region{
        name:DestName
    }
).
```

The following query would find a shared memory frame of at least 4KB accessible from both Cortex-A9 cores:

```
SharedRegion = region{
    name: 'SDRAM',
    size: Size
},
Size #=> 4096,
resolve(
    region{
        name:'CORTEXA9_1',
        base:A9Base
    },
    SharedRegion
),
resolve(
    region{
        name:'CORTEXM3_MIF',
        base:M3Base
    },
    SharedRegion
).
```

As we will see in Chapter 6, these two queries already cover more use cases than needed for this thesis. However, even more complex queries could be constructed with the above building blocks.

Evaluating performance in terms of execution time does not make sense outside the context of a concrete use case: First of all, the time a query takes to execute depends on the complexity of the query and the ADN it is executed on. Furthermore, acceptable execution times vary from use case to use case and also depend on how often a query is. I therefore do not provide measurements here, but instead evaluate the performance in the context of the use cases in Chapter 6.

5.4 Summary

For Barrelfish to be able to access the address space knowledge provided by ADNs, they need to be represented in the SKB. To facilitate code generation from Sockeye, I kept this representation as close as possible to the abstract syntax of the Sockeye com-

piler. On top of this representation I then implemented Prolog predicates to support resolution queries. By using constraints on address variables, the amount of backtracking required to execute resolution queries is reduced. All queries needed for this thesis can be constructed using resolution queries.

Chapter 6

Hardware Configuration in Barrelfish

For an OS to correctly configure the hardware it is running on, it first of all needs information about the hardware. There are several standards for an OS to retrieve such information at runtime (e.g. ACPI or PCI). However, SoCs tend not to implement these standards. Instead they rely on static specifications of their available devices and address space layout. The goal of this thesis is to show that ADNs can serve as such a specification and be used to configure hardware in an OS.

I first make the case that ADNs can be used to generate build time artefacts by statically generating kernel page tables. I then demonstrate, that ADNs can be used to configure hardware at runtime. I extended Barrelfish's device manager to discover available devices and cores from an ADN. Furthermore, I use them to retrieve the necessary address space information to start drivers. This allows Barrelfish to manage devices nearly platform independent.

As a test platform I used the PandaboardES. It features a Texas Instruments OMAP4460 SoC. Barrelfish currently has support for the two ARMv7 Cortex-A9 cores.

6.1 Preliminaries

This section is a brief overview of some concepts and system services relevant to hardware configuration in Barrelfish. I first give a quick overview over Barrelfish's architecture called the multikernel and how resources are managed. For more information about Barrelfish's architecture I refer to the original work about the multikernel by Bau-

mann et al. [3]. I then describe the coordination and synchronisation service that is used in device management. Finally I give an introduction to Barrelfish's device manager.

6.1.1 CPU Drivers

Barrelfish implements the multikernel architecture. This means, that each processor core runs a separate kernel. These kernels are referred to as *CPU drivers*. OSs with a monolithic architecture (e.g. Linux) incorporate most of their functionality into the kernel. CPU drivers in Barrelfish behave more like device drivers: They only abstract the necessary details specific to a core type to export an architecture independent interface to the system. This interface only includes privileged operations like page table manipulations or interrupt handling. The CPU driver also enforces resource access restrictions via capabilities. All other functionality is pushed to user space services.

6.1.2 Capabilities

Barrelfish uses capabilities to manage resources. A capability can be seen as a key that grants access to a certain resource. Capabilities in Barrelfish are typed according to the type of resource they grant permissions to. The most important ones for this thesis are device frame capabilities, which grant access to memory mapped device registers. When a driver wants to access a memory mapped device, it needs to map the registers into its virtual address space. To do so it needs to hand the CPU driver a corresponding capability when requesting the mapping. On ARMv7, drivers get the necessary capabilities from the device manager.

6.1.3 Octopus

I already introduced the SKB in Section 3.2 as Barrelfish's central system service that allows it to run high-level reasoning algorithms. On top of it Octopus is implemented. It provides synchronisation and coordination facilities across the system. The core of Octopus is a key-value storage engine. Clients can add and delete data or search for specific records. Octopus also features a publish-subscribe mechanism: Callbacks for changes to records that match a supplied template can be registered. Octopus will then

notify the subscribers whenever one of these records is added or deleted by calling the supplied function.

There is a client library for Octopus written in C. However its functionality can also be accessed from Prolog code executed in the SKB.

Zellweger et al. describe Octopus in more detail [23].

6.1.4 Kaluga

Device management in Barrelfish is event based: The device manager, Kaluga, uses the publish-subscribe mechanism offered by Octopus to get notified when devices and cores are discovered or removed. On x86 based platforms ACPI and PCI drivers add the Octopus records that are then consumed by Kaluga.

The PCI driver discovers devices connected over the PCI bus. It adds records for the devices it finds to Octopus. Information about available PCI device drivers is stored in the SKB. When a device is discovered, Kaluga queries the driver database to find an appropriate driver and starts it. The PCI driver then supplies the device driver with the required device frame capabilities. Being responsible for the PCI address space layout, it has the required knowledge to do so.

Records about available cores are added by the ACPI driver. When a new core is discovered, Kaluga starts a special driver called a *coreboot driver*. This driver in turn queries the SKB for the correct CPU driver binary. After that, it sets up the necessary data structures (e.g. kernel page tables). It then loads the CPU driver binary and starts the core.

The PandaboardES has no dynamic discovery mechanisms. For cores, Kaluga mimics the dynamic discovery behaviour by adding the necessary Octopus records itself. The information about available cores comes from static facts in the SKB. I changed this to use information from the ADN. For devices the current solution is more crude: The device drivers Kaluga should start, and the resources they need, are hardcoded in platform dependent device management code. I show that ADNs can be used to reduce the amount of this code towards unification of device management across platforms and architectures.

More information about Kaluga can be found in the Barrelfish technical note about device drivers [15].

6.2 Generating Kernel Page Tables

Barrelfish first starts on a single core called the *bootstrap core*. The CPU driver for this core is started by the bootloader. It then bootstraps the rest of the system. On ARMv7 all cores initially run with their MMUs disabled. Before a CPU driver can start on a core, its MMU needs to be configured with appropriate page tables. The CPU driver needs access to main memory and certain devices in the system e.g. interrupt controllers and system timers. They need to be mapped into the kernel's virtual address space.

For the bootstrap core, the setup of these page tables is currently hardcoded in C. To correctly set up the page tables, this code needs knowledge about the system's address spaces: It needs the physical addresses (as seen by the core) of main memory and the necessary devices. These addresses can vary from platform to platform.

For this thesis I removed the platform dependent C code that sets up the kernel page tables at runtime. Instead, the data structures are statically generated. The hardcoded knowledge about address spaces is replaced by queries to the ADN for the platform.

6.2.1 Implementation

Chapter 5 discusses the representation of ADNs in the SKB and how they can be queried. By including ECLiPSe in the toolchain for building Barrelfish, we can basically run a minimal version of the SKB at compile time. This allows for the SKB implementation of ADNs to also be used for static code generation. The "offline SKB" includes the ADN representation for the OMAP generated from Sockeye. Additionally, the driver database for the OMAP is loaded. This database contains meta-information about the CPU driver for the A9 cores. I augmented this description with the ADN identifiers of the memory nodes and devices the kernel needs. The kernel page table generator then queries the ADN to find the physical addresses of main memory and the devices as seen by the bootstrap core.

Also in Prolog, I implemented a generator for the C data structures that represent the page tables. From the information extracted from the ADN it generates a C file that contains these data structures.

To initialise cores on ARMv7, Barrelfish runs a small binary prior to the CPU driver. By linking the generated C file to this binary, it can use the data structures to configure the

MMU.

6.3 Device Management

In the previous section I demonstrated, that ADNs can be used to statically generate platform dependent code. In this section I show that they can be applied to hardware configuration at runtime. I do so by using them for device management.

As explained in Section 6.1.4, the drivers to start on the PandaboardES are currently hardcoded in Kaluga. The same is true for the physical addresses of the device registers they need access to. To work towards unification of device management, I took the process of how PCI devices are managed by Kaluga as a reference. This led to the following requirements for the implementation:

- Device discovery should be event based. Discovered devices should be published as Octopus records, which are then consumed by Kaluga.
- Driver information should be stored in the SKB. When a device is discovered, Kaluga should query the driver database to find an appropriate driver.
- Addresses of device registers should not be hardcoded. They should be derived from knowledge about the platform.

The implementation presented here uses knowledge from ADNs to meet all of these requirements.

6.3.1 Implementation

To meet the first requirement, I implemented two predicates in the SKB that add Octopus records based on the information in the ADN. The first one adds device discovery records for all nodes with type `device`. The second one does the same for cores. Kaluga invokes these predicates in its initialisation phase.

For the second requirement, Kaluga needs to be able to find suitable device drivers with the information contained in the Octopus record. PCI uses vendor and device IDs to identify devices. The records added by the PCI driver include these IDs. The PCI device driver descriptions in the SKB include a list of supported vendor and device ID combinations. A suitable driver is found by matching the IDs in the record with the

entries in this list. For this process to work with devices discovered in the ADN, we need to be able to identify devices. A natural choice for doing so, is the identifier of the node that represents the device in the ADN. The record added by the device discovery predicate includes this identifier. To find the right driver, the identifier is then matched with the supported identifiers in the ADN device driver descriptions.

The last requirement matches the core use case for ADNs: Getting information about address spaces. As demonstrated in Chapter 5 we can find the physical addresses of the required device registers with resolution queries. For PCI the capabilities are managed by the PCI driver. As we have no such root driver on platforms without PCI, this task is simply handled by Kaluga itself.

Device drivers are started directly by Kaluga. CPU drivers require an intermediate step: the coreboot driver. When a core discovery record is added to Octopus, Kaluga starts the coreboot driver for the type of core that was discovered. This was already implemented and I did not change this process. The coreboot driver first finds the correct CPU driver in the driver database. Among other tasks it then sets up the page tables that will be used by the new kernel. Up until now, it just reused the ones from the bootstrap kernel. This works for the OMAP as both A9 cores have a very similar view on the system. However, on more heterogeneous systems this would fail.

As mentioned in Section 6.2, the record for the CPU driver used on the PandaboardES was augmented: Information about the CPU drivers page table requirements were added. I extended the coreboot driver to use this information to set up the page tables. The implementation reuses the queries used for static generation. Instead of being passed to the code generator, the retrieved information is used by mechanism code in the coreboot driver to populate the page tables.

6.4 Evaluation

In this section I evaluate the presented implementation by first looking at query execution times. As the focus in this thesis was not on performance, it is not surprising that the measured execution times were acceptable but query performance needs to be improved. Rather, the big achievement of this thesis is the step towards unification of device management across different platforms. ADNs allow to push address space knowledge implicitly contained in mechanism code to hardware specifications. This is

Query	Time [ms]
Find device address (SCU)	8
Find device address (UART3)	302
Kernel page table	1307
Discover devices	3
Discover cores	3158

Table 6.1: ADN query execution times

visible in the code complexity of C an CLP code in the parts of the system affected by the implementation.

6.4.1 Performance

For the use of ADNs for hardware configuration to be practical, the queries need to execute fast enough to not have a large impact on system performance. However, they are never executed on the fast path. Therefore, the tolerance is higher. I measured the execution times of the queries used in the implementations presented in this chapter. These were

- Find the address of a device
- Find all addresses needed for a kernel page table
- Discover all devices
- Discover all cores

The measured times are found in Table 6.1.

At a first look, the first two rows in the table are surprising: The execution time seems to depend on the device we want to find the address of. To understand this, we need to look at how the query traverses the ADN to find the address. Recall from Section 5.2, that while constraints are used for addresses, translation set entries are traversed using backtracking. Backtracking results in a depth-first traversal order. The SCU (snoop control unit) is in the private peripheral region of the Cortex-A9 cores at `0x48240000`. The UART3 (one of the serial ports on the OMAP) is seen at `0x48020000`. The current implementation of the Sockeye compiler does not necessarily sort the translation set entries by their source addresses. In our case, the entry leading to the SCU comes

before the one for the UART3. With a depth-first traversal order, this means, that, to find the UART3, a much larger portion of the ADN has to be traversed, than is the case for the SCU. However, the query to find device addresses is only run once for each device Kaluga starts a driver for. Currently, these are less than 10 devices on the PandaboardES and we can tolerate the query taking longer for some of them. Additionally, the same query is used repeatedly to get the information necessary to construct kernel page tables. On the PandaboardES this query runs only once when starting the second A9 core and the approximately 1.3s in execution time is not an issue. Even less so, when generating the page tables for the bootstrap core at compile time. The problem would be a lot worse, if backtracking was also used for addresses. However, to use ADNs more broadly, query performance needs to be improved. A possible optimisation would be, to store a flattened version of the ADN as described in the original work: Flattening a node means contracting intermediate decoding steps between the node and reachable accepting nodes. This reduces the depth of the graph that has to be traversed drastically (i.e. to one level) and therefore would be faster. A flattened representation of a node can be obtained with the query shown in Listing 6.1 and Prolog's built-in predicate **`findall`**`/3`.

```
resolve(
    region{
        name:'CORTEXA9_1',
        base:SrcBase,
        size:SrcSize,
    },
    region{
        nodeId:DestId,
        base:DestBase
    }
).
```

Listing 6.1: ADN query to obtain a flattened version of a node

The flattened representation could be produced at compile time. Alternatively, by using memoisation, the flattened representation could be produced by the first query needing it and then get cached. The second approach would allow the use of the optimisation also in a more dynamic setting where the ADN could change at runtime.

An even higher discrepancy between execution times can be observed for the discov-

ery queries. Recall from Chapter 4 that the Sockeye specifications for the OMAP use a module for the A9 cores. The nodes representing the two cores originate from the same node declaration, just from different module instantiations. This means that they are in different namespaces but have the same name. In Section 4.3.1 I described how input port mappings create a proxy node outside the namespace they map into. These proxy nodes are used to attach a more meaningful name in the context of the OMAP to the cores. The reason why this leads to an expensive core discovery, has to do with how meta-data can be attached to nodes in Sockeye. Currently attaching meta-data to nodes is only possible when declaring nodes. The `core` node type used for discovery is therefore attached to the node declared inside the module. For clarity we want to use the proxy name of the core in the driver database. This means that we need to find the proxy node that maps to the declared one. This again involves expensive depth-first traversal of the ADN. Meta-data should be usable to convey platform specific information for certain nodes to the OS. It is therefore not always useful to attach it to reusable modules. In the future, it should be possible to attach meta-data to any node. As all devices in the current OMAP specification are declared in the top level namespace, discovering them is possible without any address resolution. This makes the query much faster.

6.4.2 Code Complexity

The goal of using ADNs for hardware configuration was to separate hardware knowledge from low level mechanism code. This has the effect of reducing code complexity especially of platform dependent code. To quantify this effect, I counted the lines of code in the system parts I changed[1]. The lines of code were counted for the implementation in the current version of Barrelfish and the one for this thesis. The lines in mechanism code (written in C) were counted separately from the lines in CLP code. The numbers are shown in Table 6.2.

With only 238 lines of code the ADN query implementation is of very low complexity. Furthermore, it is shared across all system components that use ADNs. With the ability to query the SKB at compile time, this code can even be shared with the build system.

The complexity of the mechanism code in the binary that initialises the cores before the CPU driver starts is reduced by about 13.5%. The cost of having to add 205 lines of CLP

[1]The lines of code were counted using 'SLOCCount' by David A. Wheeler.

	C LOC		CLP LOC	
Component	BF	T	BF	T
ADN (Shared)	–	–	–	238
Core init	657	568	–	205
Kaluga	1968	1830	–	86
ARMv7 specific	402	275	–	86
Coreboot	2476	2769	–	–

Table 6.2: Lines of code (LOC) comparison between Barrelfish (BF) and the implementation for this thesis (T)

code is outweighed by the benefit of keeping hardware knowledge out of the C code. Other parts of address space layout dependent code could also be generated. By doing so also in the CPU driver itself, adding support for new platforms would become much easier and cheaper.

In Kaluga the benefits of using ADNs were the most evident: While only 86 lines of CLP code were necessary for the records in the driver database, the platform specific code for ARMv7 systems was reduced by over 30%. The rest of this code either deals with other platforms than the PandaboardES or more complex Pandaboard drivers. Implementing ADN support for these drivers is conceptually not hard but would have gone beyond the scope of this thesis. By extending ADN support to these drivers and the other ARMv7 platforms supported by Barrelfish, the ARMv7 specific code could be removed entirely. Barrelfish would then have a mostly platform independent device manager!

The only part of the system where mechanism code complexity went up, is the coreboot driver. However, this is due to the fact that it did not support generating new kernel page tables when starting cores. Starting a secondary core with a different view on the system was therefore not possible. With the added ADN support, this should now work (at least if the core is of the same type as the bootstrap core). However, due to the lack of such a core with Barrelfish support on the PandaboardES this could not be tested.

6.5 Summary

I have implemented two hardware configuration tasks in Barrelfish using ADNs. One of them was generating page tables both statically and at runtime. Statically construction kernel page tables for the bootstrap core reduced the mechanism code complexity of the core initialisation binary by about 13.5%. Generating more address space knowledge dependent code would reduce this complexity further, also in the CPU driver itself. The second task was device management: Discovering devices and cores, starting drivers for them and providing the drivers with the required resources. By adding ADN support to Barrelfish's device manager, the platform dependent code for ARMv7 based systems could be reduced by over 30%. There is even the potential to remove it entirely. While query performance is acceptable for the use cases in this thesis, it needs to be improved. However, the provided query implementation is not optimised for performance and there are ways to improve it.

Chapter 7

Conclusion & Future Work

The goal of this thesis was to demonstrate that ADNs can be used to configure hardware. I did so by applying them to hardware configuration tasks in Barrelfish both statically and at runtime.

In Chapter 4 I presented the design of Sockeye, a DSL that can be used to describe hardware using ADNs. It extends the syntax originally proposed for ADNs with code reusability features to improve the efficiency when writing hardware specifications. Sockeye was subsequently used to describe the Texas Instruments OMAP4460 SoC used in the PandaboardES. The Pandaboard is the platform the hardware configuration tasks were implemented on.

Barrelfish features a system service called the SKB that allows it to run declarative algorithms to make policy decisions. In Chapter 5 I described how I extended the SKB with address space knowledge in the form of ADNs. I designed an SKB schema representing ADNs and implemented Prolog predicates to query them. The implemented predicates allow to write expressive queries in a readable manner. To obtain acceptable performance for this thesis, constraint solving techniques are used to resolve addresses.

Finally, in Chapter 6 I use the ADN knowledge in the SKB to implement two hardware configuration tasks. The first one is generating kernel page tables both statically and at runtime. By running a minimal version of the SKB at build time, the CLP code to query ADNs can be shared between the two implementations. Constructing the kernel page tables for the bootstrap core statically reduces the code complexity for the binary initialising a core before the CPU driver starts. This shows, that generating code that depends on address space knowledge from ADNs is a viable approach to improve

maintainability.

The second hardware configuration task is device management. Using ADNs, the device manager can discover devices and cores and allocate the required resources to drivers. For the drivers that are started using the ADN no platform dependent code is required. This shows that the approach has potential to unify device management in Barrelfish across platforms and even architectures.

The work presented in this thesis demonstrates the feasability and merits of using ADNs to configure hardware. However, there are more potential use cases for ADNs in an OS. In the following I present a few ideas for future work.

7.1 Improvements

To enable further use cases for ADNs in Barrelfish, some improvements on the work presented are necessary. Wider application of ADNs will increase the number of queries executed. Hence, slow queries will have a bigger impact on system performance. As already mentioned in Section 6.4, the performance could be improved by executing them on a flattened representation of the ADN. Some use cases will also be likely to need more meta-data attached to nodes. Adding a feature to Sockeye that would allow to attach arbitrary meta-data as key-value pairs to nodes would be useful. Meta-data should also be attachable to arbitrary nodes, not just to explicitly declared ones (cf. Section 4.2). This would allow to augment nodes declared in library modules with platform specific information. Another useful feature would be support for arithemic on module parameters.

7.2 Hardware Verification Using Sockeye

As mentioned in Section 4.4, the Sockeye compiler is designed such that adding additional backends is easily possible. A backend that produces code for an automatic theorem prover would allow to use Sockeye specifications to formally prove properties of the described hardware. One such property is for example well-formedness: An ADN is said to be well formed, if the decode relation is acyclic. In well-formed ADNs address resolution is guaranteed to terminate.

7.3 Static Code Generation

For this thesis I generated kernel page tables with the use of ADNs. However, there is a lot more code in Barrelfish's CPU driver that uses address space knowledge. One such example is a function that translates physical memory addresses to kernel virtual addresses. Another example are device maps that list the physical addresses of hardware registers. Both of them are currently hardcoded in Barrelfish and the correct version has to be selected during compilation depending on the platform and core the kernel is compiled for. Generating such code would reduce the effort needed to support further platforms.

7.4 Unifying Hardware Knowledge

The address space knowledge in Barrelfish's SKB comes from various sources. Some of it is static, some of it is discovered using mechanisms like ACPI and PCI. The representation of the knowledge differs from source to source. A common representation in the form of an ADN would enable further unification of configuration tasks across platforms and architectures.

One approach would be to do so via Prolog inference rules: The ADN representation could be inferred from the facts provided by the various sources. How exactly these inference rules would look and how the dynamic appearance and disappearance of devices and especially cores would be handled is an open question. One might try to infer individual nodes. However, the discovery of a e.g. a co-processor does not just add some nodes to the ADN, but might introduce a set of new address spaces and views on the system.

A more involved approach would be to lift Sockeye modules from a purely syntactical construct to a dynamic concept: To e.g. add a core to the system, the corresponding module template would be instantiated dynamically. The template arguments and port mappings would have to be inferred from the discovered information. However, the current implementation of the compiler cannot check the correctness of module templates (cf. Section 4.4). Only when the template is instantiated with concrete parameter values the uniqueness check for node identifiers is performed. Modular checking for Sockeye could for example be attempted with static code analysis, that e.g. checks for

potential aliasing between identifier templates. Alternatively the checks could be deferred to runtime, with the disadvantage that bugs in module specifications could not be detected at compile time. Solving these problems could potentially bring hardware configuration in Barrelfish to a new level of platform independence.

Appendix A

Lexical Conventions in Sockeye

The Sockeye parser follows similar conventions as used in C. The following conventions are used:

Encoding: The file should be encoded using plain text.

Whitespace: As in C, Sockeye considers sequences of space, newline, tab, and carriage return characters to be whitespace. Whitespace is generally not significant.

Comments: Sockeye supports C-style comments. Single line comments start with `//` and continue until the end of the line. Multiline comments are enclosed between `/*` and `*/`; anything in between is ignored and treated as white space. Nested comments are not supported.

Identifiers: Valid Sockeye identifiers are sequences of numbers (0-9), letters (a-z, A-Z), the underscore character "_" and the dash character "–". They must start with a letter.

$$identifier \rightarrow letter(letter \mid digit \mid \_ \mid -)^*$$
$$letter \rightarrow (\mathsf{A} \ldots \mathsf{Z} \mid \mathsf{a} \ldots \mathsf{z})$$
$$digit \rightarrow (\mathsf{0} \ldots \mathsf{9})$$

Case sensitivity: Sockeye is case sensitive, hence identifiers `UART3` and `uart3` are not the same.

Integer Literals: A Sockeye integer literal is a sequence of digits, optionally preceded by a radix specifier. As in C, decimal (base 10) literals have no specifier and hexadecimal literals start with `0x`.

$$decimal \rightarrow (\mathsf{0} \ldots \mathsf{9})^1$$
$$hexadecimal \rightarrow (\mathsf{0x})(\mathsf{0} \ldots \mathsf{9} \mid \mathsf{A} \ldots \mathsf{F} \mid \mathsf{a} \ldots \mathsf{f})^1$$

Reserved words: The following are reserved words in Sockeye:

```
accept, are, as, at, import, in, input, is, map,
module, output, over, reserved, to, with
```

Appendix B

Sockeye Syntax

The following listing shows the complete syntax for Sockeye in EBNF. Terminals are **bold** while non-terminals are *italic*. The non-terminals *identifier*, *letter*, *decimal* and *hexadecimal* correspoind to the ones defined in Appendix A.

$$sockeye = \{\ import\ \}\ \{\ mod\_decl\ \}\ \{\ node\_decl \mid mod\_inst\ \}$$

$$import = \textbf{import}\ \{\ letter \mid \textbf{/}\ \}$$

$$\begin{aligned}
mod_decl &= \textbf{module}\ identifier\,[param_list]\ \textbf{\{}\ \{input_port \mid output_port\}\ body\ \textbf{\}}\\
param_list &= \textbf{(}[\ parameter\{\textbf{,}\ parameter\}\ ]\textbf{)}\\
parameter &= param_type\ identifier\\
param_type &= \textbf{addr} \mid \textbf{nat}\\
input_port &= \textbf{input}\ identifier\textbf{/}decimal\{\textbf{,}\ identifier\textbf{/}decimal\}\\
output_port &= \textbf{output}\ identifier\textbf{/}decimal\{\textbf{,}\ identifier\textbf{/}decimal\}\\
body &= \{\ node_decl \mid mod_inst\ \}
\end{aligned}$$

$$\begin{aligned}
mod_inst &= identifier\,[arg_list]\ \textbf{as}\ identifier\ [\ \textbf{with}\ \{input_map \mid output_map\}\ ]\\
argument &= decimal \mid hexadecimal \mid identifier\\
arg_list &= \textbf{(}[\ argument\{\textbf{,}\ argument\}\ ]\textbf{)}
\end{aligned}$$

$$input\_map = identifier \ \mathbf{>} \ identifier$$
$$output\_map = identifier \ \mathbf{<} \ identifier$$

$$interval\_templ = identifier\mathbf{\{} \left[var \ \mathbf{in} \right] \ interval\mathbf{\}} \left[interval\_templ \mid simple\_templ \mid identifier \right]$$
$$simple\_templ = identifier\mathbf{\{}var\mathbf{\}} \left[simple\_templ \mid identifier \right]$$
$$var = identifier$$
$$limit = decimal \mid identifier$$
$$interval = \mathbf{[}limit\mathbf{..}limit\mathbf{]}$$

$$node\_decl = \left\{ identifier \ \mathbf{is} \ node\_spec \ \middle| \ identifier \{ \mathbf{,} \ identifier \} \ \mathbf{are} \ node\_spec \right\}$$

$$node\_spec = \left[\ type \ \right] \ \left[\ accept \ \right] \ \left[\ map \ \right] \ \left[\ reserved \ \right] \ \left[\ overlay \ \right]$$
$$type = \mathbf{core} \mid \mathbf{device} \mid \mathbf{memory}$$
$$accept = \mathbf{accept \ [} \ \{ \ block\_spec \ \} \ \mathbf{]}$$
$$map = \mathbf{map \ [} \ \{ \ map\_spec \ \} \ \mathbf{]}$$
$$reserved = \mathbf{reserved \ [} \ \{ \ block\_spec \ \} \ \mathbf{]}$$
$$overlay = \mathbf{over} \ identifier/decimal$$

$$block\_spec = hexadecimal \left[\mathbf{-} \ hexadecimal \ \middle| \ \mathbf{/}decimal \right]$$

$$map\_spec = block\_spec \ \mathbf{to} \ identifier \ \left[\mathbf{at} \ hexadecimal \right] \ \left\{ \mathbf{,} \ identifier \ \left[\mathbf{at} \ hexadecimal \right] \right\}$$

Appendix C

Sockeye Checks

The following is a list of all checks the Sockeye compiler performs, grouped by the transformation stages they are performed in.

C.1 Type Checks

Duplicate Modules This check makes sure that all module names in any of the imported files are unique.

Duplicate Parameters This check makes sure that no module has two parameters with the same name.

Duplicate Index Variables This check makes sure that no two index variables in the same scope have the same name.

Undefined Modules This check makes sure that all modules being instantiated actually exist.

Undefined Parameters This check makes sure that all referenced parameters are in scope.

Undefined Index Variables This check makes sure that all index variables referenced in templated identifiers are in scope.

Parameter Type Mismatch This check makes sure that parameters are used in a type safe way.

Argument Count Mismatch This check makes sure that module instantiations give the correct number of arguments to the module template being instantiated.

Argument Type Mismatch This check makes sure that the arguments passed to module templates have the correct type.

C.2 Checks during Module Template Instantiation

Module Instantiation Loops This check makes sure that there are no loops in module instantiations which would result in an infinite nesting of decoding subnets.

Duplicate Namespaces This check makes sure that all module instantiations in a module have a unique namespace.

Duplicate Identifiers This check makes sure that all node identifiers are unique. This includes output ports, declared nodes and nodes mapped to input ports of instantiated modules.

Duplicate Ports This check makes sure, that there are no duplicate input or output ports. Note that declaring an output port with the same identifier as an input port is allowed and results in all address resolutions going through the input port being passed through the module to the output port.

Duplicate Port Mapping This check makes sure that no port is mapped twice in the same module instantiation.

C.3 Checks during Module Instantiation

Mapping to Undefined Port This check makes sure that there are no port mappings to ports not declared by the instantiated module.

References to Undefined Nodes This check makes sure that all nodes referenced in translation sets, overlays and port mappings exist. It also checks that every input port has a corresponding node declaration.

Bibliography

[1] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. Formalizing memory accesses and interrupts. In *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems*, MARS 2017, pages 66–116. Open Publishing Association, 2017.

[2] ARM. *Cortex-A9 MPCore Technical Reference Manual*, Revision r4p1 edition, 2012. `http://infocenter.arm.com/help/topic/com.arm.doc.100511_0401_10_en/arm_cortexa9_trm_100511_0401_10_en.pdf`.

[3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44. ACM, 2009.

[4] Cavium. ThunderX CRB 2S Product Brief. Online, 2014. `http://www.cavium.com/pdfFiles/ThunderX_CRB_2S_Rev1.pdf?x=2`.

[5] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.

[6] Cisco. *ECL^{i}PS^e - A Tutorial Introduction*, February 2017. `http://eclipseclp.org/doc/tutorial.pdf`.

[7] Cisco. *ECL^{i}PS^e User Manual*, Release 6.1 edition, February 2017. `http://eclipseclp.org/doc/userman.pdf`.

[8] Jonathan Corbet. Kernel Development. Online, July 2001. Accessed 2017-10-06. `https://lwn.net/2001/0704/kernel.php3`.

[9] devicetree.org. *Devicetree Specification*, Release 0.1 edition, May 2016. https://www.devicetree.org/downloads/devicetree-specification-v0.1-20160524.pdf.

[10] dragongold. Online, 2015. https://gist.github.com/dragondgold/1aaabf93279006b703f3.

[11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376. ACM, 2011.

[12] Unified Extensible Firmware Interface Forum. *Advanced Configuration and Power Interface Specification*, Version 6.2 edition, May 2017. http://www.uefi.org/sites/default/files/resources/ACPI_6_2.pdf.

[13] PCI Special Interest Group. *PCI Local Bus Specification*, Revision 3.0 edition, February 2004.

[14] Lukas Humbel, Reto Achermann, David Cock, and Timothy Roscoe. Towards correct-by-construction interrupt routing on real hardware. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS '17. ACM, 2017.

[15] Barrelfish Project. Device drivers in Barrelfish. Barrelfish Technical Note 019, Systems Group, ETH Zurich, December 2013.

[16] Barrelfish Project. Sockeye in Barrelfish. Barrelfish Technical Note 025, Systems Group, ETH Zurich, August 2017.

[17] Mark Rutland. Device Tree - The Disaster So Far. Online, 2013. ECL Europe http://elinux.org/images/8/8e/Rutland-presentation_3.pdf.

[18] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. A declarative language approach to device configuration. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 119–132. ACM, 2011.

[19] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the barrelfish manycore operating system. In *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[20] Mark Shuttleworth. ACPI, firmware and your security. Online, March 2014. Accessed 2017-10-06. `http://www.markshuttleworth.com/archives/1332`.

[21] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 207–222, New York, NY, USA, 2003. ACM.

[22] Texas Instruments. *OMAP44xx Multimedia Device Technical Reference Manual*, Version AB edition, April 2014. `www.ti.com/lit/ug/swpu235ab/swpu235ab.pdf`.

[23] Gerd Zellweger, Adrian Schüpbach, and Timothy Roscoe. Unifying synchronization and events in a multicore os. In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 16:1–16:6. ACM, 2012.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Hardware Configuration With Dynamically-Queried Formal Models

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Schwyn	Daniel

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 18.10.2017	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.