ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Bachelor's Thesis Nr. 308b

Systems Group, Department of Computer Science, ETH Zurich

Formal semantics for Devicetrees

by

Sandro Rüegge

Supervised by

Lukas Humbel, Prof. Dr. Timothy Roscoe

March 2020 - September 2020

inf | Informatik
Computer Science

Abstract

The world of computers is becoming more complex and this trend does not stop at the hardware boundary. Formal models can help to reduce errors as well as development time. In a perfect future world, hardware manufacturers would provide such formal descriptions for their products. We are not quite there yet, hence by using existing machine readable, but informal descriptions we can save a lot of manual work and learn about the shortcomings and benefits of both, the informal description and our formalization.

To this end I analyze Devicetrees and Sockeye to find out what they provide us with and what some common subset is. Then I show that Devicetrees can be used to generate prototype Sockeye memory decoding networks doing a lot of usually manual work in an automated fashion. Finally I present a compiler that performs such a translation for some default interpretation of the Devicetree memory representation.

Acknowledgements

Ahead of all else I would like to thank my supervisors Lukas Humbel and Prof. Dr. Timothy Roscoe for their invaluable support and input during this thesis. Thanks also to David Cock for the feedback on my draft. Furthermore I would like to thank the ETH Systems Group, in particular the Sockeye team, for the regular meetings where they provided feedback and insights.

Contents

1 Introduction

We are experiencing a fast and ongoing trend towards more diverse hardware. Even platforms as small in size as mobile phones consist of a lot of heterogeneous, interconnected components. As an example the Devicetree for the colibri-eval board (more on these two later) has 498 nodes. The components can be of different architectures and use interconnects to perform interactions between them. This complexity increases the challenge of writing reliable and correct software to manage such systems by a multi-fold. The precise hardware behaviour generally has to be extracted from long hardware descriptions in plain English, a process that is both error prone and very time consuming.

Formal models are an effective tool to describe properties of such a systems in a concise and exact way. Leveraging their precision enables reasoning about system behaviour from SOCs to whole data centers. Such models don't trivially arise for existing hardware. On the contrary the task of writing a formal description for a piece of hardware still requires in-depth knowledge of the platform. As a consequence it is very time-consuming to run or test systems that leverage formal knowledge on more than a select few platforms. To facilitate prototype creation of formal descriptions on new platforms it would be of interest to reuse existing descriptions of hardware.

The focus in this work will be put on two specific hardware description frameworks. On one side we have Devicetrees (subsection 3.1), a highly capable but informal hierarchical data structure for passing hardware information to a piece of client software. They are in active use today on hundreds of different platforms and have proven themselves to be useful in particular in configuring Linux. However they were not designed with a formal background in mind. Sockeye (subsection 3.2), on the other hand, is a research project, with the main focus on the development of a model, rather than creation of descriptions for all available boards. In fact such definitions are only available for a handful of boards.

I analyze the capabilities of the two different projects for describing hardware features. We would like to know what Sockeye might be able to learn from Devicetrees and vice versa. Since Sockeye claims to be complete in some sense we wonder whether we can find systems that are expressible in Devicetree but not in Sockeye. Now assuming Sockeye was actually complete, can we express what Devicetrees are missing to become more complete? Based on this analysis I then assign a default formal semantic to a subset of Devicetrees. Default in the sense that it is a useful interpretation of the data but not guaranteed to be correct. in all cases.

Finally we would also like a tool to transport a common subset of information from Devicetrees to Sockeye. To this end I present a new compiler for generating a prototype Sockeye memory description from a valid Devicetree. Focus is put

on a formalization step where we assign a formal meaning to the contents of the Devicetree while still allowing developers to define an alternative interpretation if so desired. In the interest of access to the jungle of Devicetrees present already in the Linux and FreeBSD source trees, we make sure that we are able to parse all their `.dts` files. Furthermore we check whether the default semantics we choose for interpretation are applicable in a wide range of cases from the mentioned sources. Lastly we would also like to check at least for some example that the generated definitions can be used to query for knowledge presented by the original description.

To give readers an idea what awaits them, I give a short outline of the contents of this thesis. We start with an overview of the related work (section 2) that influenced this thesis to various degrees. It is then followed by an in-depth description of the two projects central to this thesis, namely Devicetrees and Sockeye, in section 3, the background section. Next section 4 analyzes the Devicetree semantics in more depth. In section 5 I present a compiler that parses Devicetree files, performs formalization of the contents and automatically generates a Sockeye memory decoding net.

2 Related Work

There is already a substantial body of work in the field of hardware descriptions and usage thereof. In this section I would like to summarize only a few of them that give some relevant context for this thesis.

Projects like the Barrelfish [13] operating system by the ETH Systems Group [12] already leverage formal knowledge about hardware. It is a multi kernel research operating system that can be used to explore platform management on heterogeneous hardware. Each CPU core runs a separate kernel on top of which the operating system runs distributed in single-core processes. A central component called the System Knowledge Base (SKB) collects information about the system. It is based on Eclipse/CLP as Prolog engine, a language for first order logic, which uses facts and rules to create a set of relations. Queries can be executed against its state to determine properties of the system like aliases of memory locations.

When looking closely at real hardware one has to come to the conclusion that a shared view of memory by a whole platform is an illusion. A solution to describing memory in its true complexity is provided by the work Achermann et al. [9] They propose address decoding networks (ADN) that allow different views of addressability in a single system to be described.

In his master thesis [11] Daniel Schwyn introduces the domain specific language Sockeye as a syntax for ADNs. He shows how it can be used in Barrelfish OS to reduce device-specific code both statically during compilation as well as dynamically on a running system. Page table configurations can be generated when compiling the operating system and generic device code can dynamically query the SKB for information about the hardware to correctly configure the platform during runtime. As Sockeye is the target language of this thesis it has its own space in 3.2.

Another project attempting description of SOCs (although in an informal way) are Devicetrees. They are widely used today already, in particular on ARM platforms. In subsection 3.1 I provide an in-depth description. There are some efforts to verify Devicetree code using static checkers [14] for property definitions in yaml format. Though they are mostly syntactical, they can still be very useful to catch human errors early on in the development process.

Sadik Arslan and Geylani Kardas recently presented a model-driven approach to Devicetree software development [10]. Thanks to visual representations, static code checking and more restrictions on the data representation than in the Devicetree specification, they are able to substantially support developers in the production of Devicetrees.

In contrast to the methods mentioned up to this point, there are also dynamic

approaches. They allow the operating system to discover and configure devices during runtime, possibly getting the required information directly from the hardware in question. ACPI [5] is a widely used standard that provides component information to the OS. It has its own machine language that needs to be executed by the OS using some interpreter. Though it is criticized for the complexity of its specification and the security implications of external code execution inside the kernel, its wide availability and reliability prove effective in real world use. The peripheral component interconnect (PCI) specification includes a PCI configuration space that contains some information about the devices present that the operating system can use to select a driver. Though for more complex configuration tasks the given information might be insufficient without broader knowledge about the specific hardware.

3 Background

To be able to understand the differences between Devicetrees and Sockeye we need a solid background. This chapter focuses fully on the two projects. Devicetrees in particular have some very interesting behaviours that influence later decisions and are thus explained in detail.

3.1 Devicetrees (DT)

A piece of software managing a platform needs to be able to acquire knowledge about the hardware that is present. Devicetrees were originally designed by Open Firmware [7] to solve the problem of firmware needing to tell client code about non discoverable devices on a platform. As such Devicetrees also provide a solution to the problem of passing the necessary information to operating systems when devices themselves are not able to provide it. The trees are usually passed to the kernel by a piece of firmware. A common way to achieve this is to load a binary representation of the Devicetree into memory and tell the kernel about it's location. U-Boot [15] is a prime example for a boot loader that handles Devicetrees.

As mentioned in the introduction we are interested in particular in the Devicetrees of the Linux [6] and FreeBSD[8] source trees and we will now have a quick look at the corresponding toolchains. Devicetree files in the Linux source can be found in architecture specific directories of the form `arch/<arch>/boot/dts`. The tool chain in the Linux source tree accepts Devicetree files, optionally with includes, and processes them first using the C pre processor. This allows including `.h` files and using macros for specifying values in the Devicetree. Although this is a very useful feature for maintainability by sharing constants between the Devicetree and C code, it is arguably also a shortcoming of DTs that this is necessary. Next it uses the Devicetree compiler [4] (DTC) to create a binary representation of the Devicetree, called a flat Devicetree, that is understood by the Linux kernel. During runtime, the tree is accessible in a pseudo file system at `/proc/device-tree/`. The tree can be dynamically edited at runtime; for example to reflect state changes. Some Devicetrees additionally rely on the firmware to overwrite certain properties at runtime with dynamical values. There are several examples of such behaviour in the Linux source tree, one of which can be found at `arch/arm/boot/dts/imx6q-h100.dts`. The toolchain for Devicetrees in the FreeBSD source tree is very similar.

To get started, I include a heavily cut down version of the Devicetree used with the Colibri iMX8X [2] on the Aster Carrier Board [1] in Listing 1. This is a board employed by the ETH Systems Group for research and the tree was extracted from a running Linux system on the board. It shows how CPUs and main

memory are usually represented. In the original Devicetree, there is a lot more content with device specific information, but in the end all trivial in structure, and I go into depth about the format in a dedicated section subsubsection 3.1.1. In brief, there are nodes and properties in a hierarchical structure. Every node can contain an unlimited amount of properties and child nodes.

```
/dts-v1/;

/ {
  model = "Toradex Colibri iMX8QXP/DX on Colibri Evaluation Board
      V3";
  #address-cells = <0x02>;
  #size-cells = <0x02>;

  cpus {
    #address-cells = <0x02>;
    #size-cells = <0x00>;

    cpu@0 {
      // identify the device as a physical cpu core
      device_type = "cpu";
      // value used by the OS to find a suitable driver
      compatible = "arm,cortex-a35";
      // CPU identifier
      reg = <0x00 0x00>;
    };

    cpu@1 {
      device_type = "cpu";
      compatible = "arm,cortex-a35";
      reg = <0x00 0x01>;
    };

  };

  memory@80000000 {
    // identify the device as a physical memory
    device_type = "memory";
    // the memory addresses start at 0x80000000
    // the memory size is 0x40000000 addresses
  reg = <0x00 0x80000000 0x00 0x40000000>;
  };
};
```

Listing 1: Simplified Colibri Devicetree

3.1.1 Format

The Devicetree Specification v0.3 [3] defines some loose syntax without giving concrete semantics. However the actual format parsed by the DTC is an extended version that I had to extract from the compiler parser code and from real world examples in the Linux source tree. For reasons detailed in subsection 5.1

I decided to focus on the syntax parsed by the DTC. As a consequence that's what this section describes.

For simplified platforms the Devicetree structure is comparable to how hardware is pieced together. But when applying this logic to more complex platforms the analogy quickly falls apart. Nonetheless let's imagine a simplified view of a standard consumer PC. One can imagine the motherboard as a whole being the root node. The CPUs placed onto it are represented as children of the root, as is the main memory (RAM). The PCI bridge is right on the motherboard as well and has some PCI devices or other bridges connected that are represented as its children. Listing 2 shows such a very simplified structure.

```
/dts-v1/;

// This example is for visualization purposes
// "Real" device trees need a lot more properties

// motherboard
/ {
    #address-cells = <0x2>;
    #size-cells = <0x2>;

    // collection for cores
    cpus {
        // single core
        cpu@0{
            reg = <0x0>;
        };
    };

    // RAM, around 8GiB or memory starting at 0x10000000
    memory@10000000 {
        reg = <0x0 0x10000000 0x2 0x00000000>;
    };

    // PCI Host bridge accessible at 0x0
    pci@0x0 {
        reg = <0x0 0x0 0x0 0x0fffffff>
    }
};
```

Listing 2: Simplified view of a PC

In Appendix A there is a short description of the format using EBNF. Here I continue with an informal description of what Devicetrees look like and some accompanying examples to get the relevant basis for later concepts in this thesis.

A source file contains a list of statements. Every statement or definition ends with a semicolon (';').The source file has to start with a Devicetree version statement `/dts-v1/`. This can be followed by some `/memreserve/` statements that have to appear before the first root tree in the source file. They define entries for the memory reservation table. Each tree definition is also a statement.

Devicetrees themselves are trees of named nodes with associated properties. In this hierarchical structure there is a single root node named '/'. Each node encloses its contents in curly braces and can optionally be assigned some unique labels to refer to it from other parts of the tree. The DTC allows multiple root trees in Devicetree files that will be read top to bottom and merged according to a process described in subsection 5.2. Furthermore so-called overwrites can be specified. They are in structure very similar to root tree definitions but they start with a reference to some node instead of a node name and will be merged with the referenced node on compilation. To facilitate code reuse additional Devicetree files can be included using the `/include/` statement with a quoted file name and are inserted as a whole at the location of the inclusion. The example in Listing 3 shows what that might look like. This allows a tree to be built sequentially where later definitions can add properties to already existing nodes.

```
// file: main.dts
/dts-v1/;
/include/ "extra.dts"

/ {
    label1: node2 {
        d = "not d in node 2";
    };
};

// reference by label
&label1 {
    new_property = "new prop";
};

// reference by path in the tree
&{/node1} {
    b = "not b in node 1";
};

// file: extra.dts
/dts-v1/;

/ {
    node1 {
        a = "a";
        b = "b";
    };
    node2 {
        c = "c";
        d = "d";
    };
};

// Resulting tree
/dts-v1/;

/ {
  node1 {
```

```
    a = "a";
    b = "not b in node 1";
  };

  label1: node2 {
    c = "c";
    d = "not d in node 2";
    new_property = "new prop";
  };
};
```

Listing 3: Sequential Devicetree build

Devicetree properties have a name, some associated data and a set of unique labels. The data is a possibly empty list of the following three data types: string, byte array, cell array. The first two are self explanatory and the third is similar to the byte array but each element has default size 32bits. The size can be adapted to 8, 16, 32 or 64 bits with a prefix. The DT Specification v0.3 [3] also allows integers to be defined with arithmetic expressions in cell arrays. There are no surprises hidden in these expressions, they provide the usual C style options. Properties can reference a node in a cell array or as a string value by using their path in the Devicetree or a label assigned to the node. In the first case, the DTC turns it into a unique integer associated with the referenced node. This value is also guaranteed to be put in the `phandle` property (and sometimes the `linux,phandle` property for compatibility) of the referenced node. In the second case, the reference is transformed into a string that represents the path of the node in the tree where the nodes on the path are separated by a slash ('/'). Listing 4 shows what that would look like.

```
// dt_ref.dts
/dts-v1/;

/ {
    // references a node by label
    property1 = <&label1>;
    property2 = <&label2>;
    // references a node by path
    property3 = <&{/node1/node2}>;
    // references a node as a string
    property4 = &label2;

    label1: node1 {
        label2: node2 {};
    };
};


// result
/dts-v1/;

/ {
  property1 = <0x01>;
  property2 = <0x02>;
```

```
  property3 = <0x02>;
    property4 = "/node1/node2";

  label1: node1 {
    phandle = <0x01>;
    label2: node2 {
      phandle = <0x02>;
    };
  };
};
```

Listing 4: Node References

Nodes and properties can also be deleted with respective statements called `/delete-node/` and `/delete-property/`. These statements need to occur at the same position in the tree as the original element was declared. Listing 5 shows an example. Should a node only be removed if there is no reference to it in the tree, then it can be annotated with `/omit-if-no-ref/`.

```
// input tree 1
/dts-v1/;

/ {
    property1 = "a";
    property2 = "b";

    node1 {
    };

    node2 {
    };
};

// input tree 2
/dts-v1/;

/ {
    /delete-property/ property1;
    /delete-node/ node1;
};

// Merged Result
/dts-v1/;

/ {
    property2 = "b";

    node2 {
    };
};
```

Listing 5: Example for deletions

The DT specification [3] loosely defines some standard properties to allow for uniform expression of commonly-used hardware properties. The `compatible`

property, for example, is used by the operating system to select a suitable driver for some device. The others defined in the specification mainly describe memory and interrupts which we will look at in depth in the coming sections. These properties are only roughly specified and conformance is not enforced. Rather developers are encouraged to use these properties in a way most useful to them.

3.1.2 Memory/Addressability

There are four standard properties mainly used for memory style addressability. The meaning of these addresses can vary but they are generally used as a name for some object like a memory location or a CPU thread identifier.

- `#address-cells` defines the number of cells required to express an address, defaults to 2
- `#size-cells` describes the number of cells required to express the size of an address range, defaults to 1
- The `reg` property specifies address blocks belonging to a node. Its data is an alternating sequence of address and size specifiers. The number of cells required for each specifier is defined by the two aforementioned properties respectively.
- The domains of a node A and its parent can be joined by defining translations between the two domains utilizing the `ranges` property on node A. This translation can be the identity translation by defining an empty `ranges` property. In this way Devicetrees are able to specify multiple disjoint memory domains as well as different views of the same memory location. The property data is a sequence of tuples, each consisting of a child-address, parent-address and range-size specifier. The number of cells required to define the child-address and range-size are defined by the `#address-cells` and `#size-cells` properties respectively on the node on which the `range` property is located. The parent-address requires the number of cells specified by the parent of the aforementioned node.

Every node in the tree represents an isolated naming context which resolves only the addresses of its direct children and it is responsible for defining the `#address-cells` and `#size-cells` properties for its domain. To make the complicated reading of the `reg` and `ranges` properties more clear the Listing 6 contains some examples.

```
1 / {
2     #address-cells = <0x2>;
3     #size-cells = <0x1>;
4     node1 {
5
```

```
6           #address-cells = <0x3>;
7           #size-cells = <0x2>;
8
9           // cells values always taken from the direct parent
10          // address block at 0x1000       with size 0x2000
11          // address block at 0x1 00000000 with size 0x3000
12          reg = <0x0 0x1000 0x2000 0x1 0x0 0x3000>;
13
14          // 0x5000 many child addresses starting
15          // at 0x1 00000000 00000000 are available
16          // in the parent domain starting at 0x2 00000000
17          ranges = <0x1 00000000 00000000 0x2 00000000 0x5000>;
18
19          child1 {
20              // defines a single address block
21              // at 0x1 00000000 00000000 with size 0x2 00000000
22              reg = <0x1 0x0 0x0 0x2 0x0>;
23          };
24      };
25  };
```

Listing 6: reg and ranges examples

3.1.3 Interrupts

This section is kept fairly short because interrupts did not make it into a backend of the final compiler due to time constraints. Interrupts are defined outside of the tree structure. This is achieved by the means of references to the interrupt-parent(s) of a node. If an interrupt generating node does not specify a parent it defaults to the parent node in the tree structure. There is also a way to describe multi-child, multi-parent relationships contrary to memory addressing where this is not possible.

- The `#interrupt-cells` property defines the number of cells required to describe an interrupt specifier.
- `interrupts` is a cell array of of interrupt specifications each of which is `#interrupt-cells` cells in length.
- `interrupt-parent` contains a reference to the interrupt parent of a node which defaults to the parent in the tree.
- The Property `interrupt-map` defines a translation between interrupt specifiers similar to what `ranges` does for memory addresses.
- `interrupts-extended` can be used as an alternative to `interrups` to define multiple interrupt parents because it allows specifying a parent per interrupt specifier.
- `interrupt-controller` is an empty property marking a node as an interrupt controller.

3.2 Sockeye

Sockeye [11] is a domain specific language (DSL) to specify an address translation network (ADN). It can be used to describe hardware features like memory, interrupts or power management. Highly complex address resolution processes can be modelled thanks to its support for full directed graphs without static restrictions on cycles. There are two versions of Sockeye and in this work we focus on Sockeye v2.

The language is based on the concept of nodes. Every node has a set of accepting and a set of mapping addresses. If some address is in the accepting set the node is able to answer queries itself. An address in the mapping set defines a translation to another node. The accepting and mapping sets may overlap. A prime example for such an overlap are caches. They accept the address if it is a cache hit and map it to some underlying device like RAM otherwise.

Nodes must be combined into modules for reusability. Inputs and outputs of a module can be defined to specify what can be accessed from other modules by instantiating it. Modules describe specific viewpoints of the hardware. A graph created in this language is called a decoding network and it specifies the resolution of a name (here address) in a given context. Listing 7 shows a standard example from the Sockeye compiler source code that illustrates what a simple model could look like.

```
1  /*
2   * Example Module for illustration purposes
3   * Created by Lukas Humbel
4   */
5
6  module DRAM {
7      input memory (0 bits 40) GDDR0
8      GDDR0 accepts [(0x000000000 to 0x0fedfffff)]
9  }
10
11 module SOCKET {
12     instance RAM of DRAM
13     RAM instantiates DRAM
14
15     memory (0 bits 40) SOCKET
16     SOCKET maps [
17         (0x000000000 to 0x0fedfffff) to RAM.GDDR0 at (0x000000000
               to 0x0fedfffff)
18     ]
19 }
```

Listing 7: Simple Sockeye example

4 Semantics for Devicetrees

We are interested in semantics for Devicetrees. But as this chapter will show, it is not trivial to know how to interpret the information contained in a tree. On the contrary it is hardly possible to define a single semantic applicable for each and every tree. In the interest of achieving usefulness rather than perfection we analyze Devicetrees as they are found in the Linux and FreeBSD source trees. Then we develop semantics for only a small subset of the properties, which will be used in section 5 to generate a Sockeye memory address decoding net.

4.1 Devicetree Analysis

Devicetree's biggest strength is flexibility. They provide developers with a direct way to talk to their drivers, enabling them to pass any data desired through unrestricted additional properties. Thanks to so very few restrictions on property definitions they can be adapted to a lot of different use cases. It would have arguably been harder to get such wide spread usage with a more restricted model.

This freedom comes at a price though. The loss of generally applicable semantics for the contents of the DT. To illustrate this, let's have a look at the representation of memory structures. Without even considering complex configurations that break the historical concept of memory as a single set of unique physical addresses with a shared view by the whole platform, already several issues arise when trying to understand the provided values.

For a `reg` field it's not possible to know for sure what the integer values presented are supposed to be because the endianness is not specified. Furthermore we don't even know if the values, if interpreted correctly, represent some kind of memory address or anything different like a CPU thread ID or a chip select on a bus. As a consequence, it is very hard to verify that an interpretation of the value would be correct and dangerous to make non-trivial assumptions.

Furthermore, if we were aware that something represents memory, Devicetrees do not specify who is an actor performing operations on addresses and who isn't. For example CPU cores are usually actors on the memory domain of the root node but their nodes are not even contained in that domain. Finally there are memory configurations that cannot be expressed at all using the standard properties. A single node cannot be present in two partially or fully disjoint parent domains at the same time. Although one could imagine workarounds with out-of-tree references like with interrupts these would be nonstandard solutions.

The Linux documentation does contain so-called bindings where some additional information on the usage of standard or custom properties is available. If they

are written in a specific format, they can even be used with the Devicetree schema project [14] to check the source code for certain types of mistakes. But to be completely sure about correct interpretation of values one should have a look at the corresponding driver sources to see how they are used.

4.2 Useful Semantics

In the introduction to this section I already hinted at creating useful semantics rather than ones that are applicable to all Devicetrees. As we will see in subsection 5.3 and section 6 all considered DTs can be compiled without errors this way. However this does not imply correctness of the results. Nonetheless we show correctness for some example.

The Devicetree Specification does contain some descriptions for the memory properties. But since they are not formal I had a look at several Devicetree source files, most notably the one from the colibri-eval board, to create the semantics described in the next few paragraphs.

This work defines semantics for memory properties exclusively. There is no way one could possibly define semantics for all the properties that are in use, in particular not in a useful time frame, because most manufacturers can and do define their own for special purposes. The properties from subsubsection 3.1.2 are refined to provide the necessary clarity to be able to generate Sockeye memory descriptions.

We describe the semantics in terms of Sockeye syntax. This allows a concise description that has a well defined meaning without going overboard with explanations. It is assumed that values are given in big endian byte order.

Each node of a Devicetree maps to exactly one Sockeye module. This allows us to preserve all the viewpoints of different devices in the tree. Per Devicetree Specification [3] every node defines its own separate memory domain that contains the address blocks of its direct children. In other words the address blocks in a `reg` field of a node are always names defined in the naming context of its direct parent. We achieve this by creating a Sockeye node called `LOCAL_DOMAIN` in each module, representing the corresponding node's naming context. Furthermore, a Sockeye node, called `LOCAL`, is added to every module. It accepts the address blocks defined by the `reg` field of the corresponding Devicetree node. To allow the respective parents to resolve these addresses they instantiate the modules corresponding to their children and create identity maps to all address blocks accepted or mapped (no maps there yet, they are added in the next step) to the children's `LOCAL` nodes. This way we are able to represent the separate naming contexts and allow a parent to resolve addresses of its direct children. Listing 8 shows an example Devicetree and the resulting Sockeye code created with the semantics defined up to here. Each module has a comment to show with

which Devicetree node it is associated and empty Sockeye nodes were removed for brevity.

```
// Devicetree
/dts-v1/;
/ {
    #address-cells = <0x1>;
    #size-cells = <0x1>;

    bus@0 {
        #address-cells = <0x1>;
        #size-cells = <0x1>;
        reg = <0x0 0x1000>;

        child@0 {
            reg = <0x0 0x1000>;
        };
    };
};

// Sockeye
// DT Node name: / path: /
module ROOT {
  // memory domain of the corresponding Devicetree node
  memory (0x0 bits 32) LOCAL_domain

  instance ROOT_BUS_0 of ROOT_BUS_0
  ROOT_BUS_0 instantiates ROOT_BUS_0
  LOCAL_domain maps [(0x0 to 0xfff) to ROOT_BUS_0.LOCAL at (0x0 to
      0xfff)]
}

// DT Node name: bus@0 path: /bus@0
module ROOT_BUS_0 {
  input memory (0x0 bits 32) LOCAL
  LOCAL accepts [(0x0 to 0xfff)]

  // memory domain of the corresponding Devicetree node
  memory (0x0 bits 32) LOCAL_domain

  instance ROOT_BUS_0_CHILD_0 of ROOT_BUS_0_CHILD_0
  ROOT_BUS_0_CHILD_0 instantiates ROOT_BUS_0_CHILD_0
  LOCAL_domain maps [(0x0 to 0xfff) to ROOT_BUS_0_CHILD_0.LOCAL at
      (0x0 to 0xfff)]
}

// DT Node name: child@0 path: /bus@0/child@0
module ROOT_BUS_0_CHILD_0 {
  input memory (0x0 bits 32) LOCAL
  LOCAL accepts [(0x0 to 0xfff)]
}
```

Listing 8: Semantics Example 1

Devicetrees allow parent and child domains to be joined using the `ranges` property. This is represented by maps between the `LOCAL` node and the `LOCAL_DOMAIN`

node of a module. Note that earlier we said maps defined on the `LOCAL_DOMAIN` nodes are for accepting and mapping address blocks of the children. This allows address translation to continue along a chain of nodes, giving a parent node access to its grandchildren through the `LOCAL` node of its own children. If the `ranges` property exists and is not empty then the translations can be directly used as a Sockeye map. If it exists but is empty then it is defined as an identity map from the `LOCAL` node to every address block in the `LOCAL_DOMAIN` node. Listing 9 is very similar to Listing 8 with the addition of a ranges property on the bus. The additional output generated is marked with comments.

```
// Devicetree
/dts-v1/;
/ {
    #address-cells = <0x1>;
    #size-cells = <0x1>;

    bus@0 {
        #address-cells = <0x1>;
        #size-cells = <0x1>;
        reg = <0x0 0x1000>;
        ranges = <0x0 0x1000 0x1000>;

        child@0 {
            reg = <0x0 0x1000>;
        };
    };
};

// Sockeye
// DT Node name: / path: /
module ROOT {
  // memory domain of the corresponding Devicetree node
  memory (0x0 bits 32) LOCAL_domain

  instance ROOT_BUS_0 of ROOT_BUS_0
  ROOT_BUS_0 instantiates ROOT_BUS_0
  LOCAL_domain maps [(0x0 to 0xfff) to ROOT_BUS_0.LOCAL at (0x0 to
      0xfff)]
  // ADDITIONAL OUTPUT:
  LOCAL_domain maps [(0x1000 to 0x1fff) to ROOT_BUS_0.LOCAL at
      (0x1000 to 0x1fff)]
}

// DT Node name: bus@0 path: /bus@0
module ROOT_BUS_0 {
  input memory (0x0 bits 32) LOCAL
  LOCAL accepts [(0x0 to 0xfff)]
  // ADDITIONAL OUTPUT:
  LOCAL maps [(0x1000 to 0x1fff) to LOCAL_domain at (0x0 to 0xfff)]

  // memory domain of the corresponding Devicetree node
  memory (0x0 bits 32) LOCAL_domain

  instance ROOT_BUS_0_CHILD_0 of ROOT_BUS_0_CHILD_0
  ROOT_BUS_0_CHILD_0 instantiates ROOT_BUS_0_CHILD_0
  LOCAL_domain maps [(0x0 to 0xfff) to ROOT_BUS_0_CHILD_0.LOCAL at
```

```
      (0x0 to 0xfff)]
}

// DT Node name: child@0 path: /bus@0/child@0
module ROOT_BUS_0_CHILD_0 {
  input memory (0x0 bits 32) LOCAL
  LOCAL accepts [(0x0 to 0xfff)]
}
```

Listing 9: Semantics Example 2

The direct mapping between Devicetree nodes and Sockeye modules could allow us to later combine memory, interrupt and power decoding networks. By joining the different types of networks into single modules per Devicetree node we can preserve knowledge of which viewpoints of the system belong to the same device. This separation by devices is in line with how developers need to think about hardware when working with Devicetrees.

The attentive reader might have noticed that memreserves are not handled in the semantics defined. This is because I was not able to identify a reliable, OS independent interpretation and consequently no formalization using Sockeye.

4.3 Devicetrees vs Sockeye

As a formal DSL with a well defined syntax and semantic, Sockeye has the obvious advantage that interpretation of a description file is well-defined. However these restrictions limit its expressivity. For example, it does not allow passing of free format information to a driver and there is no inherent way of specifying a relation from nodes to devices or drivers other than their node and module name. This is in accordance with Sockeye's design to solve addressability modelling and not driver information passing.

As a consequence I decided not to propose extending Sockeye for transportation of free form data. Instead I would suggest creating separate Prolog facts about a device that represent the additional information present in a Devicetree. Using matching by Sockeye node name, one could find the path in the tree and with that query additional information about a device. This would allow free form properties to be passed on.

Using the semantics from the previous subsection we know that a Devicetree memory description can always be represented in Sockeye. This is due to the very similar but more restricted abstract structure. As in Sockeye, the DT memory representation allows nodes with a set of addresses and a set of address translations between these nodes. The graph implied by a Devicetree has the additional restriction that it has to be a tree. As a consequence, not all Sockeye decoding nets can be translated into a Devicetree.

If one was interested in translating a Sockeye decoding net to a Devicetree, one would have to define a different representation for memory where address translations can not only happen in a parent and child relationship but instead between arbitrary nodes. This would remove the tree restriction from the memory graph. The gain in expressivity would be paid for in complexity of the representation.

5 Compiler

An important part of this thesis was the creation of a compiler that takes a Devicetree as an input and creates a corresponding Sockeye description as output. This section provides an in-depth look at the inner workings and the design decisions taken during the creation. Because of the issues explained in subsection 4.1 it is not possible to create a reliable memory model without additional assumptions. Though an output could certainly be used for some testing on new platforms where unsupervised correctness is not important and errors can be fixed when needed.

In a first attempt I investigated creating a new backend for the existing DTC. This would have saved the work of analyzing the precise DTC behaviour, implementing a new parser and would have ensured compatible behaviour in current usage scenarios of Devicetrees. There were two main issues with this approach. Firstly, the parser performs some evaluations and transformations on the Devicetree. Secondly, it is harder to reason about correct behaviour of C code than in a more formal language setting. Ideally the architecture would allow us to exactly say what kind of transformations we assume. To be able to control the input fully and to pave the way for more thorough analysis of this compiler in future work, I decided to go with the alternative option presented in the thesis proposal: Writing a new compiler in Haskell. It is well suited for verification tasks and it allowed me to make a clean separation between 'dumb' parsing, building an internal representation, formalizing the contents of the tree and generating code.

There are four main stages to the compiler:

- In the first stage, the parsing stage, we parse the file without interpreting or modifying the actual contents. The goal is to get as much information from the tree into an internal parser representation without losing anything.
- Secondly, we build the internal representation. This requires merging the multiple input trees. Interpretation is still kept to a minimum but some contents are moved from outside of the tree structure into properties. Arithmetic expressions in cell arrays are also evaluated to reduce the complexity of the AST. These are optimizations to make the formalization process later on more concise.
- The third stage performs formalization of the tree. These are from an abstract point of view just semantical interpretations of the Devicetree contents. They take the information present in the tree and create compiler internal properties that have a well-defined meaning. Default interpretations for memory and interrupt properties are provided as a starting point.

- Finally there is the backend stage where the formal compiler properties are used to generate code. Currently a single backend is implemented for Sockeye. These outputs should not be assumed to be correct without manual verification as without device specific formalization implementations they only represent default interpretations of the data given in the Devicetree and have no guaranteed validity.

In principle this just automates parts of the process of manually assigning a specific semantic to Devicetree properties and allows code generation from such an interpretation specification in the form of formalization code in Haskell.

5.1 Parser

The Devicetree specification does not put very strict requirements on the format of a Devicetree file. As the explicit goal of the thesis was to be able to parse all Devicetree files in the Linux and FreeBSD source trees, I decided to design the parser to allow for a super set of the syntax parsed by the DTC.

In contrast to the behaviour of the DTC parser, I perform neither interpretations, evaluations nor transformations. These are tasks for the build and formalization stage. Everything is transformed into an internal representation called ParserAST as-is. The compiler runs under the assumption that the Devicetrees it receives are valid and that they do not contain any references to nonexistent nodes. In that sense it is a 'dump' parser. This allows important decisions to be performed in a dedicated compiler stage.

Another difference worth mentioning is the handling of include statements. To perform includes, the DTC, when happening upon a valid include statement, will pause parsing of current file and parse the included file first before resuming. Includes are allowed to recurse but not cycle. This behaviour can be emulated using the c pre processor (CPP). That is useful because a lot of files in the Linux source tree already use CPP features and it is unnecessary to re-implement the include feature as valid includes in the DTC are almost equivalent to just pasting the contents of the included file at the include statement location, which is what the CPP is good at. The main difference is that the DTC is very picky about the location of Devicetree version and memreserve statements inside the file. The parser from this work can deal with these occurrences between root and reference trees without any issues.

5.2 Build AST

The abstract syntax tree used in the compiler was supposed to be simple and easy to work with. Unfortunately, this collides with the intention of providing

the formalization stage with all of the available data. I did not want to restrict the power of the interpretations under any circumstance which is why I decided to keep all the tedious labelling of properties and even single values in the AST. This is a change from the first attempt at the internal AST where I removed some of most likely obsolete information from the tree. Listing 10 shows what the AST of a very simple Devicetree could look like. As you might notice, there are a lot of places where a label can appear in a Devicetree. The object `Tree.Node` is defined by Haskell `Data.Tree` and is used for representing the actual tree structure. `Node`, on the other hand, is the actual Devicetree node. Each such node has a list of properties that in return contain a list of data sections that can be of the different types mentioned in the Devicetree format section (3.1.1).

```
Tree.Node {
  rootLabel = Labeled {
    element = Node {
      nodeName = "/",
        nodeProperties = [
          Labeled {
            element = Property {
              propertyName = "prop1",
              propertyData = [
                Labeled {
                  element = String "Hello",
                  labels = []
                }
              ],
              postDataLabels = []
            },
            labels = []
          }]},
    labels = []
  },
  subForest = []
}
```

Listing 10: Example AST

To make the process of formalization as intuitive as possible, I kept the structure of the AST very close to the one of the Devicetree itself. It is still a tree of named nodes with associated properties. Everything that is outside of the tree structure is represented with properties defined by the compiler. What makes the structure rather verbose are all the possibilities to label parts of the Devicetree structure. The whole AST definition in Haskell code is included in Appendix B.

To build the AST we need to perform three tasks:

- merge root and overwrite trees in order
- remove nodes that are marked with omit-if-no-ref from the tree if they are

not referenced

- represent information provided in unconventional form in the generic structure
- evaluate expressions

The process of merging was modeled to behave like the DTC. The process starts with an empty tree and iteratively merges the trees from the source top down, one by one, into the result tree. Merging two nodes causes new properties to be created and existing ones with the same name to be overwritten. Missing child nodes are added and existing child nodes with the same name are merged recursively with the same process. The set of labels on properties or nodes is united upon merging. Node deletions and property deletions remove the specified elements from the tree and clear their label set. Override nodes in the input are handled almost the same. Only instead of starting the process at the root of the current result tree, it starts at the referenced node.

In the DTC the "omission if there is no reference" feature has interesting behaviours. Creating a node with the `/omit-if-no-ref/` annotation, then deleting it and finally creating a new node at the same path causes the new node to be annotated with `/omit-if-no-ref/` as well. If a marked node A references another marked node B then only the first node in the chain, here A, is removed. This is presumably because technically the node B was referenced at some point. For compatibility reasons my compiler replicates these behaviours as shown in Listing 11.

```
/dts-v1/;

/ {
  /omit-if-no-ref/
  node1 {
  };

  /omit-if-no-ref/
  node2 {
    p = <&label3>;
  };

  /omit-if-no-ref/
  label3: node3 {
  };
};

/ {
  /delete-node/ node1;
};

/ {
  node1 {
  };
};
```

```
26
27 // resulting tree
28 /dts-v1/;
29
30 / {
31   label3: node3 {
32   };
33 };
```

Listing 11: Example for omission behaviour

There are statements outside of the tree definitions that define reserved memory regions. They are currently ignored in the formalization and code generation, but for completeness they are included in the AST nonetheless. To avoid creating special cases in the structure, they are turned into a property in the root node named `__meta__memreserve`.

The DTC handles references occuring in cell arrays by assigning each referenced node a unique 32-bit integer and placing it in the `phandle` property of the referenced node. Then it replaces all references in cell arrays with the associated `phandle` value. References that are not inside a cell array are turned into a string that represents the node's path in the tree with slashes between the node names on the path. Because it is not possible to differentiate between integers or strings that are references and ones that are not, I decided to keep references as a separate type in the AST. `phandle` values are generated anyway to allow a formalization that requires them to have access.

Expressions in cell-arrays are evaluated using unlimited integer arithmetic from Haskell. The effort of making sure that the precise semantics of the evaluations are equivalent to the DTC evaluations was not deemed worth the gained precision in the given context.

5.3 Formalization

Devicetree properties do not have formal semantics for interpreting their values. DT developers can choose their own for their devices. To allow code generation from such a basis we need a special step, the formalization step, where interpretation decisions are made and can be influenced depending on the tree in question.

Backends as described in subsection 5.4 define a unique property name prefix under which the well defined properties for that backend reside. They will later be used for actual code generation. When implementing a formalization, it is possible to use all the information provided by the original Devicetree to produce the formal backend properties.

A formalization in the most basic sense is just a function that maps an AST to a new AST. In practice, we use this to add properties with a specific prefix such that the meaning of the new properties has a well defined semantic and they can be differentiated from the original properties. This allows users who use non-standard properties or properties in a non standard way to convert them into a form that has a known meaning. It is recommended to use the `compatible` property for matching to specific devices that need non-default semantics.

Because it is a lot of effort to implement the formalization process for all devices in a tree and because the semantics established in section 4 work in a lot of cases, I have implemented them as default formalization rules for memory. They are applied last in the formalization setp and only to nodes that are not marked as already memory formalized. This allows formalization writers to dynamically decide which nodes they want to handle. Listing 12 shows a simple example formalization that adds a property named `example` with string data `Hello World` to the root of the AST.

```
1 exampleFormalization :: Formalization
2 exampleFormalization = do
3   return $
4     rootNodeApply
5       ( setProperty "example" [AST.noLabels (AST.String "Hello
            World")]
6       )
7       tree
```

Listing 12: Example formalization

5.4 Code Generation

Code Generation requires a solid data basis to be able to work reliably. The previous compiler steps try to provide this as best as they can in a "best-effort" process. Backends like the one provided for Sockeye memory networks define properties, whose names are prefixed by a unique string per backend, with formal semantics. Based on these, they will generate their output.

There are five general properties already defined by the base compiler:

- `__meta__name` is a string that is the name of the node without any changes applied
- `__meta__unique_name` a string that is unique for this node, starts with a capital letter and consists only of capital letters A to Z, digits 0 to 9 and underscore ('_'). This should allow it to be (part of) a valid identifier in most languages. It can be used for example in Sockeye output to make sure that a node has a unique name. The calculation can be expressed in Haskell as:

```
nameFromPath :: AST.NodePath -> String
nameFromPath ["/"] = "ROOT"
nameFromPath path =
  let repl c = if isLetter c || isDigit c then c else '_'
   in map (toUpper . repl) ("ROOT/" ++ intercalate "/" (tail
       path))
```

Listing 13: unique name calculation

- `__meta__path` contains the path of this node in the tree. It is provided for convenience and to make sure it is the same wherever the path is used. It is calculated as `"/" ++ intercalate "/" path` where `path` is a Haskell list of strings, the node names on the path in hierarchical order.
- `__meta__memreserve` represents the memreserve statements that are found in Devicetrees outside of the normal tree structure.
- `__meta__omit-if-no-ref` is an empty property and is present on all nodes that were annotated as such.

The rest of this section revolves around the Sockeye memory decoding network backend. So, as to avoid reinventing the wheel, I use the same information structure as defined in the Devicetree specification [11]. The prefix used for this backend is `__sock_v2_mem__`.

- `__sock_v2_mem__address-bits` is a cell array with a single integer, the number of bits required at most to represent the addresses in the addressing domain defined by the given node.
- `__sock_v2_mem__reg` is a list of cell arrays with two numbers each. The first number in each array specifies the starting address of an address block and the second number defines the size of the block. This is possible since the Cells are represented with infinite integer variables in Haskell.
- `__sock_v2_mem__ranges` is also a list of cell arrays but with three numbers each. Per array first is a child base address, the second the parent base address and the third is the size of the translated range.

Based on these three properties, here the names without prefix are used, we are able to build a memory tree and generate the desired code. The process described here is performed bottom up in the AST. Every Devicetree node gets assigned a module in the output. If it turns out to be an empty module then it is not printed to reduce unnecessary output. If the given Devicetree node has children with `reg` fields then it will generate a Sockeye node called `LOCAL_domain` that represents its memory domain. It then instantiates all the Sockeye modules of the child nodes and maps all the addresses that are available

in the child's `LOCAL` node onto the local domain. The `LOCAL` node is then defined to be an input to the module and it contains all the address blocks that the Devicetree node defines itself in the `reg` field. Furthermore the local node gets map definitions to the local domain that represent the translations to the parent domain defined by the `ranges` property.

6 Evaluation

6.1 Parsing

The thesis proposal explicitly requires the resulting compiler to be able to parse all Devicetree files in the Linux and FreeBSD source trees. To verify compliance with the requirement, I have created a testing script called `auto-test.sh` that handles the bulk of the work for compiling all Devicetrees in a directory. It replaces DTC style includes with CPP includes and then runs the CPP on all `.dts` files in all the target directories. For each tested directory it prints the number of successful ('`+`'), failed due to the CPP ('`?`'), failed due to my compiler ('`-`') and total compilations. On the last line the total over all directories is printed. This allows generating a good overview of the performance of the compiler on the Devicetrees in the Linux and FreeBSD source trees.

6.1.1 Linux

Listing 14 shows the commands used for evaluation on the Linux source tree [6] A clean version of the master branch (commit: 5925fa68fe82) was used.

```
# Add the arm device tree directory to the includes.
# It is required by a lot trees and not present otherwise.
# executed from the root directory of the Linux source tree
ln -s ../arch/arm/boot/dts include/arm

# run the auto-test command with the include
# executed from the tests directory of my compiler source
# <lrd> is the Linux source tree root directory
./auto-test.sh --target CP \
  --include <lrd>/include
  <lrd>
```

Listing 14: Linux Evaluation

There are no CPP errors or Compiler errors reported for any of the Devicetrees found.

6.1.2 FreeBSD

Very similar to the Linux tests, the Listing 15 shows the commands used in the FreeBSD [8] test. Again I used a clean version of the master branch (commit: 7fbac817ea4).

```
# Add the arm dts directory to the includes.
# It is required by a lot trees and not present otherwise.
```

```
# executed from the root directory of the FreeBSD source tree
ln -s ../arm sys/gnu/dts/include/arm

# run the auto-test command with the include
# executed from the tests directory of my compiler source
# <fbrd> is the FreeBSD source tree root directory
./auto-test.sh --target CP\
  --include <fbrd>/include \
  --include <fbrd>/sys/gnu/dts/include/ \
  --include <fbrd>/sys/dts/mips \
  --include <fbrd>/sys/dts/arm \
  <fbrd>
```

Listing 15: FreeBSD Evaluation

Several errors were recorded. Although one is shown in the compiler results all of them have their source in the CPP. The compiler error is caused by several C macros that should be evaluated by the CPP but are not. I was not able to find the corresponding macro definitions anywhere in this version of the FreeBSD source tree. All the CPP errors are caused by a missing Devicetree file that should be included, called `skeleton.dtsi`. These are minor errors and do not appear to show missing parsing capabilities of the presented compiler.

6.2 Code Generation

Thorough evaluation of the quality and correctness of the generated code is very difficult due to the under-specification of Devicetrees discussed earlier in this thesis, as well as there not being a better way to define correctness than over the behaviour of the systems using the trees. To get a bit of an impression on how we perform nonetheless we present an example usage and show that the decoding net can be used to retrieve correct values.

We use the presented compiler to generated the Sockeye memory decoding network of the colibri-eval board Devicetree. With the usual tool chain from the Sockeye compiler project, we compile this to the Prolog language. Thanks to the test framework already in place there, we are able to query the generated network for addresses. The Prolog extract used for this is shown in Listing 16. In this case we wanted to know the base address of the USDHC as viewed by the root node of the system which corresponds to real CPU addresses in the world of Devicetrees. The address we get is `0x5b010000` which is what we would expect from looking at the corresponding Devicetree.

```
run_test :-
    printf("Testing colibri.pl\n",[]),
    init,
    state_empty(S),
    add_ROOT(S,[],_),
    OutR = region(["LOCAL", "ROOT_USDHC_5B010000"], _, _),
```

```
    accept(OutR),
    InR = region(["LOCAL_domain"], [block(InB,InLimit)], _),
    decodes_rev(OutR, _, InR),
    printf("%p --> %p\n", [InR, OutR]),
    printf("USDHC Base Address: 0x%x\n", [InB]).
```

Listing 16: Code generation test code

7 Conclusions

There is no doubt that hardware is getting harder. Furthermore, it is of no surprise that there is a need for correct OS code to handle said hardware. Abstractions and formal specifications of hardware allow us to generalize platform management in a less error-prone and more verifiable way. In this work I showed that it is possible to define some semantics for Devicetrees and use these existing hardware descriptions to generate prototype platform specifications in Sockeye. By performing manual validation and implementing specific formalization tasks to assign more precise semantics to Devicetree contents, these prototypes can be turned into actual formal specs.

Still a lot of manual work is required to verify and correct or at least test the output. Furthermore, it is in no way a complete replacement for Sockeye specifications manually-written from hardware specifications because Devicetrees do not contain a complete view of the platform but just what is necessary for an operating system and device drivers to handle a platform.

8 Future Work

8.1 Sockeye Interrupt Backend

Originally it was planned for the thesis to also include a backend that produces an interrupt specification for the platform. The time did not suffice and so it was not part of the final work. In future work it could be of interest to also provide such a backend for a more complete Sockeye model of the platform. Interrupts are more difficult because their structure allows for parent references. The implied directed graph has to be created before interrupt specifier properties can be interpreted because it is dependent on ancestor information. Only then can the translations be propagated through the graph.

8.2 Language for Formalization

Currently the formalization process requires writing tedious code. Future work could try to move formalization away from writing Haskell code and into a more practical setting. A language could be defined that matches to the compatible field of Devicetree nodes and describes the meaning of their properties. Although it is questionable of how much general use such a language would be as it would probably require special cases for every domain it describes - like memory, interrupt or power.

8.3 Use Yaml bindings Linux source

The information provided in the Linux Devicetree bindings, which are like a loose documentation for device specific properties, could be used to assist in the development of formalization by providing an outline of properties that are available and what structures they might or should follow.

8.4 Super DT

By introducing more restrictions on the format of Devicetree properties and their semantics, a "Super Devicetree" could be created. It could still be used like the current Devicetrees because it is a subset of them but it could also reliably be converted to Sockeye specifications. This would grant the benefits of both the existing infrastructure and knowledge about Devicetrees but also the formal semantics of Sockeye and its uses in scientific operating systems.

References

[1] *Aster Carrier Board*. URL: https://developer.toradex.com/products/aster-carrier-board. (last access: 09.09.2020).

[2] *Colibri iMX8X*. URL: https://developer.toradex.com/products/colibri-som-family/modules/colibri-imx8x. (last access: 09.09.2020).

[3] "Devicetree Specification". Version Release v0.3. In: (2020). URL: https://www.devicetree.org/.

[4] *Devie Tree Compiler*. URL: https://git.kernel.org/pub/scm/utils/dtc/dtc.git/. ((last access: 22.09.2020).

[5] Unified Extensible Firmware Interface (UEFI) Forum. "Advanced Configuration and Power Interface (ACPI) Specification". Version Version 6.3. In: (2019). URL: https://uefi.org/sites/default/files/resources/ACPI_6_3_final_Jan30.pdf.

[6] The Linux Foundation. *Linux kernel source tree*. URL: https://github.com/torvalds/linux. (last access: 18.09.2020, commit: 925fa68fe82).

[7] Grant Likely. *Linux and the Device Tree*. URL: https://www.kernel.org/doc/html/latest/devicetree/usage-model.html. (last access: 06.09.2020).

[8] The FreeBSD Project. *FreeBSD*. URL: https://github.com/freebsd/freebsd. (last access: 18.09.2020, commit: 7fbac817ea4).

[9] Lukas Humbel et. al. Reto Achermann. "Formalizing Memory Accesses and Interrupts". In: (2017).

[10] Geylani Kardas Sadik Arslan. "DSML4DT:A domain-specific modeling language for device tree software". In: (2019).

[11] Daniel Schwyn. "Hardware Configuration With Dynamically-Queried Formal Models". In: (2017).

[12] *Systems Group*. URL: https://systems.ethz.ch/. (last access: 06.09.2020).

[13] *The Barrelfish OS*. URL: http://www.barrelfish.org/. (last access: 06.09.2020).

[14] *Tooling for devicetree validation*. URL: https://github.com/devicetree-org/dt-schema. (last access: 06.09.2020, commit: c38438e).

[15] *U-Boot*. URL: https://www.denx.de/wiki/U-Boot. (last access: 06.09.2020).

A Devicetree EBNF

Whitespaces, trivial definitions and includes were left out to make the description more readable.

```
device_tree_file ::= '/dts-v1/' ';' memreserve* root_node (
    root_node | ref_node )*

memreserve ::= '/memreserve/' ( hex | int ) ( hex | int ) ';'

root_node ::= '/' node_definition

ref_node ::= label* noderef node_definition

node_definition ::= '{' property* child_node* '}' ';'

child_node ::= label* '/omit-if-no-ref/'* ( '/delete-node/'
    node_name ';' | node_name node_definition )

noderef ::= '&' label | '&{' nodepath '}'

nodepath ::= '/' [ node_name ('/' node_name)* ['/'] ]

property ::= label* ( [ '/delete-property/' ] property_name |
    property_name '=' property_data ) ';'

property_data ::= data_section ( ',' data_section )*

data_section ::= string | noderef | byte_array | cell_array

byte_array ::= '[' ( digit digit )* ']'

cell_arary ::= [ /bits/ ( 8 | 16 | 32 | 64 ) ] '<' cell* '>'

cell ::= noderef | (expression) | int | hex | char

hex ::= '0x' hex_digit+

label ::= string ':'

expression ::= an integer expression
```

Listing 17: Devicetree EBNF

B Internal AST

```
type DeviceTree = Tree.Tree LNode

type Label = String
data Labeled a = Labeled
  { element :: a,
    labels :: [Label]
  }

type LNode = Labeled Node
data Node = Node
  { nodeName :: String,
    nodeProperties :: [LProperty]
  }

type LProperty = Labeled Property
data Property = Property
  { propertyName :: String,
    propertyData :: PropertyData,
    postDataLabels :: [Label]
  }

type PropertyData = [Labeled DataSection]
type LDataSection = Labeled DataSection
data DataSection
  = CellArray CellSize [Labeled Cell]
  | ByteArray [Labeled Word8]
  | String String
  | DataSectionRef Reference

data CellSize = B8 | B16 | B32 | B64 | BInternal

data Cell = Int Integer | CellRef Reference

data Reference
  = LabelReference Label
  | PathReference NodePath

type NodePath = [String]
```

Listing 18: Compiler AST