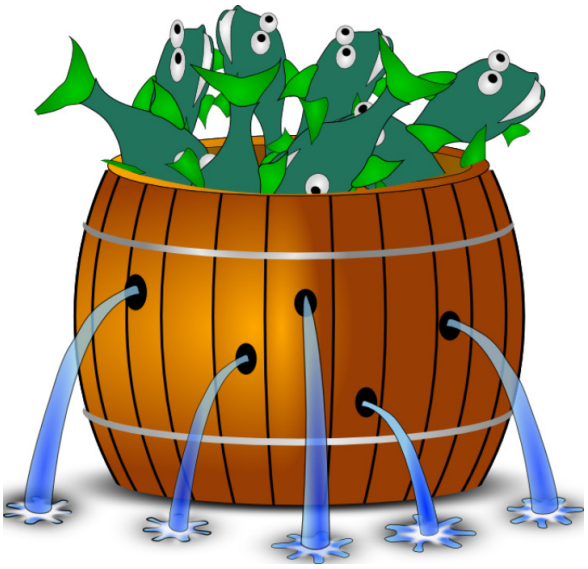
Barrelfish Project
ETH Zurich



Sockeye in Barrelfish

Barrelfish Technical Note 025

Barrelfish project

03.08.2017

Systems Group
Department of Computer Science
ETH Zurich
CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland
`http://www.barrelfish.org/`

Revision History

Revision	Date	Author(s)	Description
0.1	15.06.2017	DS	Initial Version
0.2	03.08.2017	DS	Describe Modularity Features

Contents

Chapter 1

Introduction and Usage

Sockeye[1] is a domain specific language to describe SoCs (Systems on a Chip).

Achermann et al. [1] propose a formal model to describe address spaces and interrupt routes in a system as a directed graph. They call such a graph a "decoding net". Each node in the graph can accept a set of addresses and translate another (not necessarily disjunct) set of addresses (when describing interrupt routes they accept or translate interrupt vectors). Starting at a specific node, addresses can be resolved by following the appropriate edges in the decoding net. When a node translates an address, resolution is continued at the referenced node. When a node accepts an address, resolution terminates

Achermann et al. [1] also propose a concrete syntax for specifying decoding nets. Sockeye is an implementation of the proposed language but adds some features to address issues encountered in practice.

The Sockeye compiler is written in Haskell using the Parsec parsing library. It generates Prolog files from the Sockeye files. These Prolog files contain facts that represent a decoding net (see Chapter 5). The Prolog files can then be loaded into Barrelfish's System Knowledgebase (SKB).

In the future the compiler should also be able to generate Isabelle code from Sockeye specifications to be able to verify hardware designs.

The source code for Sockeye can be found in `SOURCE/tools/sockeye`.

[1] Sockeye salmon (Oncorhynchus nerka), also called red salmon, kokanee salmon, or blueback salmon, is an anadromous species of salmon found in the Northern Pacific Ocean and rivers discharging into it. This species is a Pacific salmon that is primarily red in hue during spawning. They can grow up to 84 cm in length and weigh 2.3 to 7 kg. Source: Wikipedia

1.1 Command Line Options

```
$ sockeye [options] file
```

The available options are:

-P Generate a Prolog file that can be loaded into the SKB (default).

-i Add a directory to the search path where Sockeye looks for imports.

-o `filename` The path to the output file (required)

-d `filename` The path to the dependency output file (optional)

-h show usage information

The backend (capital letter options) specified last takes precedence. At the moment there is only a backend for generating Prolog for use in Barrelfish's SKB.

Multiple directories can be added by giving the `-i` options multiple times. Sockeye will first look for files in the current directory and then check the given directories in the order they were given.

When invoked with the `-d` option, the compiler will generate a dependency file for GNU make to be able to track changes in imported files.

The Sockeye file to compile is given via the `file` parameter.

Chapter 2

Lexical Conventions

The Sockeye parser follows a similar convention as opted by modern day programming languages like C and Java. Hence, Sockeye uses a Java-style-like parser based on the Haskell Parsec Library. The following conventions are used:

Encoding The file should be encoded using plain text.

Whitespace: As in C and Java, 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.

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

Case sensitivity: Sockeye is case sensitive hence identifiers `node1` and `Node2` 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`.

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

Reserved words: The following are reserved words in Sockeye:

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

Chapter 3

Syntax

In this chapter we define the layout of a Sockeye file. The node declarations in a Sockeye file describe a single decoding net. Parts of a decoding net can be packaged into reusable modules (see Section 3.2). With imports (see Section 3.4) modules can also be put into separate files.

In the following sections we use EBNF to describe the Sockeye syntax. Terminals are **bold** while non-terminals are *italic*. The non-terminals *iden*, *letter*, *decimal* and *hexadecimal* correspond to the ones defined in Chapter 2.

The examples are all taken from the Texas Instruments OMAP4460 SoC used on the PandaboardES[1]. A more complete specification of the SoC is located in `SOURCE/socs/omap44xx.soc`.

3.1 Basic Syntax

This section describes the basic syntax for Sockeye. It closely corresponds to the concrete syntax described in [1]. If there are important syntactic or semantic differences it is stated explicitly in the description of the respective syntactical construct.

3.1.1 Node Declarations

A node declaration contains one or more identifiers and the node specification. The order in which the nodes are declared does not matter.

Syntax

$$net_s = \left\{ iden \textbf{ is } node_s \;\middle|\; iden \left\{ \textbf{, } iden \right\} \textbf{ are } node_s \right\}$$

[1] The technical reference manual can be found here.

Example

```
SDRAM is ...

UART1,
UART2 are ...
```

3.1.2 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. The reserved address blocks are only relevant in conjunction with overlays and are used to exclude some addresses from the overlay. The overlay is specified as a node identifier and a number of address bits. The overlay will span addresses from `0x0` to $\texttt{0x2}^{\texttt{bits}} - 1$.

Syntax

$$
\begin{aligned}
\mathit{node}_s &= \left[\ \mathit{type}\ \right] \left[\mathit{accept}\ \right] \left[\ \mathit{map}\ \right] \left[\ \mathit{reserved}\ \right] \left[\ \mathit{overlay}\ \right] \\
\mathit{accept} &= \textbf{accept [}\ \left\{\ \mathit{block}_s\ \right\}\ \textbf{]} \\
\mathit{map} &= \textbf{map [}\ \left\{\ \mathit{map}_s\ \right\}\ \textbf{]} \\
\mathit{reserved} &= \textbf{reserved [}\ \left\{\ \mathit{block}_s\ \right\}\ \textbf{]} \\
\mathit{overlay} &= \textbf{over}\ \mathit{iden/decimal}
\end{aligned}
$$

Example

```
SDRAM is accept [...]
L3 is map [...]
CORETEXA9_SS_Interconnect is reserved [...] over L3/32
```

3.1.3 Node Type

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 CPU core. The `device` type specifies that the accepted addresses are device registers while the `memory` type is for memory nodes like RAM or ROM.

Syntax

$$\mathit{type} = \textbf{core} \mid \textbf{device} \mid \textbf{memory}$$

Example

```
CORETEXA9_1 is core map [...]
UART3 is device accept [...]
SDRAM is memory accept [...]
```

3.1.4 Addresses

Addresses are specified as hexadecimal literals.

Syntax

$$addr = hexadecimal$$

3.1.5 Block Specifications

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`.

Syntax

$$block_s = addr \left[\text{-}\ addr \;\middle|\; \text{/}decimal \right]$$

Example

```
UART1 is accept [0x0-0xFFF]
UART3 is accept [0x0/12]    // same as 0x0-0xFFF
IF_A9_0 is accept [0x44]      // same as 0x44-0x44
```

3.1.6 Map Specification

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`. Note that this is different from the concrete syntax described in [1] where in this case the base address of the source block is used. This was changed due to the mapping to `0x0` being used more often in practice. Multiple translation targets can be specified by giving a comma-separated list of targets.

Syntax

$$map_s = block_s \textbf{ to } iden \left[\textbf{at } addr\right] \left\{, iden \left[\textbf{at } addr\right]\right\}$$

Example

```
/* Translate 0x54000000-0x0x54FFFFFF
 * to L3_EMU at 0x54000000-0x0x54FFFFFF:
 */
L3 is map [0x54000000/24 to L3_EMU at 0x54000000]

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

/* Multiple translation targets, interrupt vector 0x2 is translated to
 * - SPIMap at 0xC
 * - NVIC at 0x12:
 * /
SDMA is map [0x2 to SPIMap at 0xC, NVIC at 0x12]
```

3.2 Modules

A module encapsulates a decoding net which can be integrated into a larger decoding net. A module instantiation always creates a namespace inside the current one. Normally nodes can only be referenced by nodes in the same namespace. To properly integrate a module into a larger decoding net we need a mechanism to connect the module to the enclosing decoding net. This is done via ports. There are input ports, which create a connection into the module and output ports which create connections from the module to outside nodes. A port has always a width, specified as the number of address bits the port spans. All declared input ports must have a corresponding node declaration in the module body.

When a module is instantiated a list of port mappings can be specified. An input port mapping creates a node outside the module that overlays the node inside the module. An output port mapping creates a node inside the module that overlays the node outside the module. Not all ports a module declares have to be mapped. Not mapping an input port simply means there is no connection to the corresponding node inside the module. For unmapped output ports an empty node inside the module will be generated, acting as a dead end for address resolution.

Additionally a module can be parametrized. It will then be a module template that only becomes a fully defined module when instantiated with concrete arguments. Module parameters are typed and the Sockeye compiler checks that they are used in a type safe way. There are two types of parameters: address parameters and natural parameters. Address parameters allow to parametrize addresses in node specifications. Natural parameters are used in combination with interval template identifiers (see Section 3.3). Parameters can also be passed as arguments to module template instantiations in the module body.

3.2.1 Module Declarations

A module declaration starts with the keyword **`module`** and a unique module name. If the module is a template, a list of typed parameters can be specified. The module body is enclosed in curly braces and starts with the port definitions. The rest of the body are node declarations and module instantiations. If the module has address parameters the name of the parameter can be used wherever in the body an address is expected.

Syntax

$$
\begin{aligned}
\mathit{param\_type} &= \textbf{addr} \mid \textbf{nat} \\
\mathit{parameter} &= \mathit{param\_type}\ \mathit{iden} \\
\mathit{param\_list} &= \textbf{(}\left[\ \mathit{parameter}\left\{\textbf{,}\ \mathit{parameter}\right\}\ \right]\textbf{)} \\
\mathit{input\_port} &= \textbf{input}\ \mathit{iden/decimal}\left\{\textbf{,}\ \mathit{iden/decimal}\right\} \\
\mathit{output\_port} &= \textbf{output}\ \mathit{iden/decimal}\left\{\textbf{,}\ \mathit{iden/decimal}\right\} \\
\mathit{body}_s &= \left\{\ \mathit{net}_s \mid \mathit{mod\_inst}_s\ \right\} \\
\mathit{mod\_decl}_s &= \textbf{module}\ \mathit{iden}\left[\mathit{param\_list}\right]\ \textbf{\{}\ \left\{\mathit{input\_port} \mid \mathit{output\_port}\right\}\ \mathit{body}_s\ \textbf{\}}
\end{aligned}
$$

Example For some examples of module declarations see Listing 3.1.

3.2.2 Module Instantiations

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 to instantiate the module has to be given followed by an optional list of port mappings.

Syntax

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

Example

```
/* Instantiate module 'CortexA9-Subsystem' in namespace 'CortexA9_SS' */
CortexA9-Subsystem as CortexA9_SS

/* Pass arguments to module template e.g. to instantiate a
 * CortexA9 MPCore module with
 * - 2 cores
 * - 0x48240000 as the base of the private memory region
 */
CortexA9-MPCore(2, 0x48240000) as MPU

/* Declare port mappings:
 * - map 'CORTEXA9_1' to input port 'CPU_1'
 * - map output port 'Interconnect' to 'L3'
 */
CortexA9-Subsystem as CortexA9_SS with
    CORTEXA9_1 > CPU_1
    L3 < Interconnect
```

3.3 Templated Identifiers

Templated identifiers allow to declare multiple nodes and ports at once and instantiate a module multiple times at once. There are two forms of templated identifiers:

Interval template The template contains one or several intervals. The identifier will be instantiated for all possible combinations of values in the intervals. Index variables can optionally be named so they can be referenced later.

Simple template Simple templates work very similar to interval templates. The only difference is, that a simple template can only reference index variables declared in the same context. It can not contain intervals to declare new index variables.

Interval templates can be used in identifiers of node declarations (to declare multiple nodes), port declarations (to declare multiple ports) and namespace identifiers of module instantiations (to instantiate a module multiple times). The scope of index variables is the corresponding syntactic construct the interval template was used in. 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 but not module parameter or index variable names.

An important thing to note is that the limits of an interval can reference module parameters of type `nat`. This allows module parameters to control how many ports or nodes are instantiated from a certain interval template.

Syntax

$$
\begin{aligned}
var &= iden \\
limit &= decimal \mid iden \\
interval &= [limit..limit] \\
interval\_templ_s &= iden\{var\ \textbf{in}\ interval\}\left[iden \mid templ\_iden_s \mid for\_iden_s\right] \\
simple\_templ_s &= iden\{var\}\left[iden \mid templ\_iden_s\right]
\end{aligned}
$$

Example

```
/* Declare similar nodes
 * Note that interval templates in node declarations
 * always require the usage of 'are'
 */
GPTIMER_{[1..5]} are device accept [0x0/12]

/* Use the index in the node definition */
GPTIMER_ALIAS_{i in [1..5]} is map [0x100/12 to GPTIMER_{i}]

/* Declare similar module ports
 * (possibly depending on module parameters)
 */
module module CortexA9-MPCore(nat cores, addr periphbase) {
    input CPU_{[1..cores]}
    ...
}

/* Instantiate module multiple times
 * and use index variable in port mappings
 */
CortexA9-Core as Core_{c in [1..2]} with
    CPU_{c} > CPU
```

3.4 Imports

Imports allow a specification to be split across several files. They also allow the reuse of modules. Imports have to be specified at the very top of a Sockeye file. An import will cause the compiler to load all modules from `<import_path>.soc` Nodes declared outside of modules will not be loaded. The compiler will first look for files in the current directory and then check the directories passed with the `-i` option in the order they were given.

Syntax

$$
import_s = \textbf{import}\left\{ letter \mid / \right\}
$$

Example

```
/* Invoked with 'sockeye -i imports -i modules' the following
 * will cause the compiler to look for the files
 * 1. ./cortex/cortexA9.soc
 * 2. imports/cortex/cortexA9.soc
 * 3. modules/cortex/cortexA9.soc
 * and import all modules from the first one that exists.
 */
import cortex/cortexA9
```

3.5 Sockeye Files

A sockeye file consists of imports, module declarations and the specification body (node declarations and module instantiations).

Syntax

$$sockeye_s = \{\ import_s\ \}\ \{\ mod\_decl_s\ \}\ \{\ net_s \mid mod\_inst_s\ \}$$

Example Listing 3.1 shows an example Sockeye specification.

```
module CortexA9-Core(addr periphbase) {
    input CPU/32
    output SCU/8
    output L2/32

    PERIPHBASE is map [
            0x0000-0x00FC to SCU
            0x0600/8 to Private_Timers
        ]

    CPU is map [
            periphbase/13 to PERIPHBASE
        ]
        over L2/32

    Private_Timers is device accept [0x0/8]
}

module CortexA9-MPCore(nat cores, addr periphbase) {
    input CPU_{[1..cores]}/32
    output L2/32

    SCU is device accept [0x0-0xFC]

    CortexA9-Core(periphbase) as Core_{c in [1..cores]} with
        CPU_{c} > CPU
        SCU < SCU
        L2 < L2
}

UART{[1..3]} are device accept [0x0/12]
SDRAM is memory accept [0x0/30]

L3 is map [
    0x48020000/12 to UART3
    0x4806A000/12 to UART1
    0x4806C000/12 to UART2
    0x80000000/30 to SDRAM
]

CortexA9-MPCore(2, 0x48240000) as CORTEXA9_SS with
    CORTEXA9_{c in [1..2]} > CPU_{c}
    L3 < L2
```

Listing 3.1: Example Sockeye specification

Chapter 4

Checks on the AST

The compiler performs the transformation from parsed Sockeye to decoding nets in three steps:

1. Type checker
2. Instantiator
3. Net builder

In each of the steps some checks are performed. The type checker checks that all referenced symbols (modules, parameters and index variables) are defined and ensures module template parameter type safety. The instantiator instantiates module and identifier templates and ensures that each identifier is declared only once. The net builder instantiates the modules and ensures referential integrity in the generated decoding net. It also transforms overlays to translation sets. The checks are detailed in the following sections.

4.1 Type Checker

4.1.1 Duplicate Modules

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

4.1.2 Duplicate Parameters

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

4.1.3 Duplicate Index Variables

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

4.1.4 Undefined Modules

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

4.1.5 Undefined Parameters

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

4.1.6 Undefined Index Variables

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

4.1.7 Parameter Type Mismatch

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

4.1.8 Argument Count Mismatch

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

4.1.9 Argument Type Mismatch

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

4.2 Instantiator

4.2.1 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.

4.2.2 Duplicate Namespaces

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

4.2.3 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.

4.2.4 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.

4.2.5 Duplicate Port Mapping

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

4.3 Net Builder

4.3.1 Mapping to Undefined Port

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

4.3.2 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.

Chapter 5

Prolog Mapping for Sockeye

The Sockeye compiler generates ECL^{i}PS^e-Prolog[1] to be used within the SKB. A decoding net is expressed by the predicate `node/2`. The first argument to the predicate is the node identifier and the second one the node specification.

Node identifiers are represented as a functor `node_id/2`. The first argument is the node's name, represented as an atom, and the second one is the (possibly nested) namespace it is in. The namespace is represented as a list of atoms where the head is the innermost namespace component.

Node specifications are represented by a Prolog functor `node_spec/3`. The arguments to the functor are the node type, the list of accepted addresses and the list of translated addresses. The overlay is translated to address mappings and added to the list of translated addresses during compilation.

The node type is one of four atoms: `core`, `device`, `memory` or `other`. The accepted addresses are a list of address blocks where each block is represented through the functor `block/2` with the start and end addresses as arguments. The translated addresses are a list of mappings to other nodes, represented by the functor `map/3` where the first argument is the translated address block, the second one is the target node's identifier and the third one is the base address for the mapping in the target node.

There is a predicate clause for `node/2` for every node specified.

The code is generated using ECL^{i}PS^e's structure notation. This enables more readable and concise notation when accessing specific arguments of the functors.

Listing 5.1 shows the generated Prolog code for the Sockeye example in Listing 3.1.

[1] http://eclipseclp.org/

```
node{id:node_id{name:'CORTEXA9_1',namespace:[]},spec:node_spec{type:other,accept:[],
    ↪ translate:[map{src_block:block{base:0x0,limit:0xffffffff,props:[]},dest_node:
    ↪ node_id{name:'CPU_1',namespace:['CORTEXA9_SS']},dest_base:0x0,dest_props:[]}]}}.
node{id:node_id{name:'CORTEXA9_2',namespace:[]},spec:node_spec{type:other,accept:[],
    ↪ translate:[map{src_block:block{base:0x0,limit:0xffffffff,props:[]},dest_node:
    ↪ node_id{name:'CPU_2',namespace:['CORTEXA9_SS']},dest_base:0x0,dest_props:[]}]}}.
node{id:node_id{name:'CPU',namespace:['Core_1','CORTEXA9_SS']},spec:node_spec{type:other,
    ↪ accept:[],translate:[map{src_block:block{base:0x48240000,limit:0x48241fff,props
    ↪ :[]},dest_node:node_id{name:'PERIPHBASE',namespace:['Core_1','CORTEXA9_SS']},
    ↪ dest_base:0x0,dest_props:[]},map{src_block:block{base:0x48242000,limit:0xffffffff,
    ↪ props:[]},dest_node:node_id{name:'L2',namespace:['Core_1','CORTEXA9_SS']},dest_base
    ↪ :0x48242000,dest_props:[]},map{src_block:block{base:0x0,limit:0x4823ffff,props:[]},
    ↪ dest_node:node_id{name:'L2',namespace:['Core_1','CORTEXA9_SS']},dest_base:0x0,
    ↪ dest_props:[]}]}}.
node{id:node_id{name:'CPU',namespace:['Core_2','CORTEXA9_SS']},spec:node_spec{type:other,
    ↪ accept:[],translate:[map{src_block:block{base:0x48240000,limit:0x48241fff,props
    ↪ :[]},dest_node:node_id{name:'PERIPHBASE',namespace:['Core_2','CORTEXA9_SS']},
    ↪ dest_base:0x0,dest_props:[]},map{src_block:block{base:0x48242000,limit:0xffffffff,
    ↪ props:[]},dest_node:node_id{name:'L2',namespace:['Core_2','CORTEXA9_SS']},dest_base
    ↪ :0x48242000,dest_props:[]},map{src_block:block{base:0x0,limit:0x4823ffff,props:[]},
    ↪ dest_node:node_id{name:'L2',namespace:['Core_2','CORTEXA9_SS']},dest_base:0x0,
    ↪ dest_props:[]}]}}.
node{id:node_id{name:'CPU_1',namespace:['CORTEXA9_SS']},spec:node_spec{type:other,accept
    ↪ :[],translate:[map{src_block:block{base:0x0,limit:0xffffffff,props:[]},dest_node:
    ↪ node_id{name:'CPU',namespace:['Core_1','CORTEXA9_SS']},dest_base:0x0,dest_props
    ↪ :[]}]}}.
node{id:node_id{name:'CPU_2',namespace:['CORTEXA9_SS']},spec:node_spec{type:other,accept
    ↪ :[],translate:[map{src_block:block{base:0x0,limit:0xffffffff,props:[]},dest_node:
    ↪ node_id{name:'CPU',namespace:['Core_2','CORTEXA9_SS']},dest_base:0x0,dest_props
    ↪ :[]}]}}.
node{id:node_id{name:'L2',namespace:['CORTEXA9_SS']},spec:node_spec{type:other,accept:[],
    ↪ translate:[map{src_block:block{base:0x0,limit:0xffffffff,props:[]},dest_node:
    ↪ node_id{name:'L3',namespace:[]},dest_base:0x0,dest_props:[]}]}}.
node{id:node_id{name:'L2',namespace:['Core_1','CORTEXA9_SS']},spec:node_spec{type:other,
    ↪ accept:[],translate:[map{src_block:block{base:0x0,limit:0xffffffff,props:[]},
    ↪ dest_node:node_id{name:'L2',namespace:['CORTEXA9_SS']},dest_base:0x0,dest_props
    ↪ :[]}]}}.
node{id:node_id{name:'L2',namespace:['Core_2','CORTEXA9_SS']},spec:node_spec{type:other,
    ↪ accept:[],translate:[map{src_block:block{base:0x0,limit:0xffffffff,props:[]},
    ↪ dest_node:node_id{name:'L2',namespace:['CORTEXA9_SS']},dest_base:0x0,dest_props
    ↪ :[]}]}}.
node{id:node_id{name:'L3',namespace:[]},spec:node_spec{type:other,accept:[],translate:[map
    ↪ {src_block:block{base:0x48020000,limit:0x48020fff,props:[]},dest_node:node_id{name:
    ↪ 'UART3',namespace:[]},dest_base:0x0,dest_props:[]},map{src_block:block{base:0
    ↪ x4806a000,limit:0x4806afff,props:[]},dest_node:node_id{name:'UART1',namespace:[]},
    ↪ dest_base:0x0,dest_props:[]},map{src_block:block{base:0x4806c000,limit:0x4806cfff,
    ↪ props:[]},dest_node:node_id{name:'UART2',namespace:[]},dest_base:0x0,dest_props
    ↪ :[]},map{src_block:block{base:0x80000000,limit:0xbfffffff,props:[]},dest_node:
    ↪ node_id{name:'SDRAM',namespace:[]},dest_base:0x0,dest_props:[]}]}}.
node{id:node_id{name:'PERIPHBASE',namespace:['Core_1','CORTEXA9_SS']},spec:node_spec{type:
    ↪ other,accept:[],translate:[map{src_block:block{base:0x0,limit:0xfc,props:[]},
    ↪ dest_node:node_id{name:'SCU',namespace:['Core_1','CORTEXA9_SS']},dest_base:0x0,
    ↪ dest_props:[]},map{src_block:block{base:0x600,limit:0x6ff,props:[]},dest_node:
    ↪ node_id{name:'Private_Timers',namespace:['Core_1','CORTEXA9_SS']},dest_base:0x0,
    ↪ dest_props:[]}]}}.
```

```
node{id:node_id{name:'PERIPHBASE',namespace:['Core_2','CORTEXA9_SS']},spec:node_spec{type:
    ↪ other,accept:[],translate:[map{src_block:block{base:0x0,limit:0xfc,props:[]},
    ↪ dest_node:node_id{name:'SCU',namespace:['Core_2','CORTEXA9_SS']},dest_base:0x0,
    ↪ dest_props:[]},map{src_block:block{base:0x600,limit:0x6ff,props:[]},dest_node:
    ↪ node_id{name:'Private_Timers',namespace:['Core_2','CORTEXA9_SS']},dest_base:0x0,
    ↪ dest_props:[]}]}}.
node{id:node_id{name:'Private_Timers',namespace:['Core_1','CORTEXA9_SS']},spec:node_spec{
    ↪ type:device,accept:[block{base:0x0,limit:0xff,props:[]}],translate:[]}}.
node{id:node_id{name:'Private_Timers',namespace:['Core_2','CORTEXA9_SS']},spec:node_spec{
    ↪ type:device,accept:[block{base:0x0,limit:0xff,props:[]}],translate:[]}}.
node{id:node_id{name:'SCU',namespace:['CORTEXA9_SS']},spec:node_spec{type:device,accept:[
    ↪ block{base:0x0,limit:0xfc,props:[]}],translate:[]}}.
node{id:node_id{name:'SCU',namespace:['Core_1','CORTEXA9_SS']},spec:node_spec{type:other,
    ↪ accept:[],translate:[map{src_block:block{base:0x0,limit:0xff,props:[]},dest_node:
    ↪ node_id{name:'SCU',namespace:['CORTEXA9_SS']},dest_base:0x0,dest_props:[]}]}}.
node{id:node_id{name:'SCU',namespace:['Core_2','CORTEXA9_SS']},spec:node_spec{type:other,
    ↪ accept:[],translate:[map{src_block:block{base:0x0,limit:0xff,props:[]},dest_node:
    ↪ node_id{name:'SCU',namespace:['CORTEXA9_SS']},dest_base:0x0,dest_props:[]}]}}.
node{id:node_id{name:'SDRAM',namespace:[]},spec:node_spec{type:memory,accept:[block{base:0
    ↪ x0,limit:0x3fffffff,props:[]}],translate:[]}}.
node{id:node_id{name:'UART1',namespace:[]},spec:node_spec{type:device,accept:[block{base:0
    ↪ x0,limit:0xfff,props:[]}],translate:[]}}.
node{id:node_id{name:'UART2',namespace:[]},spec:node_spec{type:device,accept:[block{base:0
    ↪ x0,limit:0xfff,props:[]}],translate:[]}}.
node{id:node_id{name:'UART3',namespace:[]},spec:node_spec{type:device,accept:[block{base:0
    ↪ x0,limit:0xfff,props:[]}],translate:[]}}.
```

Listing 5.1: Generated Prolog code

Chapter 6

Compiling Sockeye Files with Hake

SoC descriptions are placed in the directory `SOURCE/socs` with the file extension `soc`. Each top-level Sockeye file has to be added to the list of SoCs in the Hakefile in the same directory. When passed a filename (without extension), the function `sockeye :: String -> HRule` creates a rule to compile the file `SOURCE/socs/<filename>.soc` to `BUILD/sockeyefacts/<filename>.pl`. The rule will also generate `BUILD/sockeyefacts/<filename>.pl.depend` (with the `-d` option of the Sockeye compiler) and include it in the Makefile. This causes `make` to rebuild the file also when imported files are changed. To add a compiled Sockeye specification to the SKB RAM disk, the filename can be added to the `sockeyeFiles` list in the SKBs Hakefile.

References

[1] R. Achermann, L. Humbel, D. Cock, and T. Roscoe. Formalizing memory accesses and interrupts. In *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems*, pages 66–116, 2017.