HOPP Driver Generator Documentation

Thomas Fischer

February 22, 2013

Contents

1 Introduction **2**
- 1.1 Origins and Goals . . . 2
- 1.2 Terminology . . . 3

2 Getting Started **4**
- 2.1 Setup . . . 4
- 2.2 Process . . . 5
  - 2.2.1 Building the Generator . . . 5
  - 2.2.2 Running the Generator . . . 5
  - 2.2.3 Executing your Application . . . 7

3 Driver Description **8**
- 3.1 Host Side . . . 9
  - 3.1.1 Component . . . 9
  - 3.1.2 Port . . . 10
  - 3.1.3 Queues . . . 10
  - 3.1.4 I/O Threads . . . 10
  - 3.1.5 Communication medium . . . 11
- 3.2 Board Side . . . 12
- 3.3 Protocol . . . 13
  - 3.3.1 State transitions . . . 13
  - 3.3.2 Control Flow Graph . . . 15
  - 3.3.3 Sequence charts . . . 16

4 Generator **25**
- 4.1 Used Libraries . . . 25
  - 4.1.1 JFlex & CUP . . . 25
  - 4.1.2 Katja . . . 25
- 4.2 Generation Process . . . 26
- 4.3 CModel . . . 26
- 4.4 Unparser . . . 28

Chapter 1

Introduction

This document is intended to give an overview over the HOPP Driver Generator for both, users and developers. While the first chapters are more addressed towards users of the generator, the later ones are more addressed towards developers.

The HOPP Driver Generator itself is intended to provide embedded system developers with a simple, convenient interface to communicate with their designed hardware components. To our knowledge, such an interface for communication with VHDL components has not been done so far, though extensive research exists concerning efficient communication between PCs and FPGAs over different communication channels, mainly Ethernet [3, 1, 2].

Our interface has been prototyped with a `C++` frontend communicating with a Xilinx Virtex 6 ML-605 FPGA over Ethernet. The design enables easy extension to support other frontend languages, FPGA boards or transport media.

1.1 Origins and Goals

The HOPP Driver Generator originally was and currently is developed by the Software Technology Group of the University of Kaiserslautern in cooperation with the Microelectronic Design Research Group of the University of Kaiserslautern.

The design goals of this project are:

- Probably better filled out by the EIT department ;)
- An easy-to-use driver and driver generator
- Well-documented api
- Modularity and extensibility
- Meaningful error description

1.2 Terminology

This section explains the terminology used in this document and the project in general.

Driver The complete software product is called the *driver*. It enables software-side communication with the hardware platform.

Board / board-side The driver is split in two parts, one of which has to be uploaded and executed to the hardware platform itself. This part of the driver is referred to as board-side driver. Sometimes, the terms *server* and *server-side* might be used instead.

Host / host-side In contrast to the board-side driver, the host-side driver is the part of the driver which is located on the communicating computer. This part contains the actual API, embedded developers will work with. Sometimes, the terms *client* and *client-side* might be used instead.

Chapter 2

Getting Started

The purpose of this chapter is to explain how to properly build and use the generator. This part contains all steps required in order to run the generator and acquire drivers for the specified board.

2.1 Setup

In the following, the tools required to build and execute the driver generator are introduced. Please note, that all tools have to be executable from the command line. This requires (for example) Windows users to adjust their `PATH` variable.

Java The generator is implemented in Java due to tool support and the environment of the Software Technology Group. Consequently, a JDK version 6 or above is required.

Gradle Gradle[1] is a build tool, similar to *Maven* or *Ant* (like *Make*, but - for the most part - easier to manage within larger projects). It supports the dependency management from Maven while retaining the flexibility of Ant. Plugins required by the build process are automatically downloaded by Gradle.

Mercurial Mercurial[2] is a distributed versioning tool, comparable with *GIT*, *Bazar* or (to some degree) *Subversion*. The sources of the driver generator are located in a mercurial repository. If you acquired the sources (and this document) through other means mercurial is not required.

Doxygen Doxygen[3] is used for generation of a Java API-like html description of the driver API. While this generation is not required for the driver, it is highly recommended for easier integration of the generated driver.

[1] available at `http://www.gradle.org/`
[2] available at `http://mercurial.selenic.com/`
[3] available at `http://www.doxygen.org/`

C/C++ Compiler Since the host side driver is written in C++, it is also required to have a C/C++ compiler present.

Xilinx Toolsuite In order to generate an `.elf` file that can is used to program the FPGA, the Xilinx toolsuite is required. This includes ISE for generating IPCores out of VHDL files, XPS for composing these and EDK for actually generating the file for the defined hardware platform.

We try to avoid to require the user to actually design anything in the Xilinx suite, but only use it for synthesising the board hard- and software. Both, `.mhs` file and all source files for the EDK, are created by the driver generator.

It is furthermore desirable to skip user-interfaction with the EDK completely. This would require generation of the board support package and external call of Xilinx' compiler.

2.2 Process

If the tools described in Section 2.1 are correctly installed, the following steps should provide you with a working version of the driver generator.

2.2.1 Building the Generator

First of all, the project has to be checked out. As of now, the project is only available at the mercurial repository of the Softech Group of the University of Kaiserslautern.[4]

After checking out the project, all sources have to be compiled and packaged. This can be done using the gradle build tool. The command to build a jar package is `gradle jar`. Simply type it in a shell in the project root repository, where the file `build.gradle` is located. Running the build file will generate an executable jar package under the path `<project>/build/libs`. Navigate there and try running it using the command `java -jar <name of the jar package>`.[5]

2.2.2 Running the Generator

The HOPP Driver Generator can be called using the command line interface (CLI). The CLI offers several parameters to further configure the run of the generator, listed in Table 2.1.

Additionally, the `.mhs` file of the system is required by the generator. So a complete call looks like the following line:

```
java -jar driverGenerator.jar [OPTIONS] <.mhs file>.
```

[4] The repository is located at `https://softech.informatik.uni-kl.de/hg/ag/hopp/`. Please note, that authorization is required.

[5] This should result in the usage help and an error, since no .mhs file has been specified.

`-s`	`--server`	Specifies the server backend. This specifies, for which target platform the driver should be generated. The default setting is a Virtex 6 ML 605 FPGA (currently, no other options are present).
	`--serverDir`	Specifies the destination of the server directory. All files for the server (i.e. files used to generate the board side driver) will be generated into the specified directory. If none is specified, a directory "server" will be generated inside the current working directory.
`-c`	`--client`	Specifies the client backend. This specifies, in which language the client software should be generated. The default setting is a C++ frontend (currently, no other options are present).
	`--clientDir`	Specifies the destination of the client directory. All client for the server will be generated into the specified directory. If none is specified, a directory "client" will be generated inside the current working directory.
`-v`	`--verbose`	The driver generator will print out additional console output. This makes it easier to track errors in the driver generator or the used source file.
`-d`	`--debug`	The driver will be generated with additional debug console output. This makes it easier to track errors in larger test cases.
`-h`	`--help`	Lists all CLI parameters and a short explanation. The generator will abort after parsing this parameter and not generate anything.
	`--mac`	used to set the mac address of a possible Ethernet interface of the board. Notation: XX:XX:XX:XX:XX:XX, where each X marks a hexadecimal number (allowing lower as well as upper cases).
	`--ip`	used to set the ip address of a possible Ethernet interface of the board. Notation: X.X.X.X, where each X marks a decimal number ranging from 0 to 255. The same notation is required for the following two parameters.
	`--mask`	used to set the network mask of a possible Ethernet interface of the board.
	`--gw`	used to set the standard gateway of a possible Ethernet interface of the board.
	`--port`	used to set the communication port of a possible Ethernet interface of the board. Note that these five parameters are only contemporary and will be replaced by the new board description language (I hope - together with the debug parameter. It may be used for actually debugging the generator then ...).

Table 2.1: Summary of currently possible CLI parameters

`-p --project`	Specifies the project backend. This specifies, in which tool the hardware design should take place. Currently, only XPS in version 14.1 can be selected.

Table 2.2: Summary of currently possible options for the Virtex6 server backend

The output of the generator consists of two groups of files. The first group contains all files that make up the board side of the driver, which have to be compiled to an `.elf` file by the Xilinx SDK. The second group are files for the client side, that can be used to wrap communication with the board and its components.

In addition to the required sources, documentation for both host- and board-side sources is generated, using doxygen.

2.2.3 Executing your Application

Now the regular design process from Xilinx can be continued. The next step would be writing a host application that uses the generated host-side driver and API. After programming your FPGA with the `.elf` file generated using the board-side driver, your program should be able to communicate with VHDL components on the FPGA through this API.

Chapter 3

Driver Description

This section should provide a conceptual overview of the driver parts. For pure users of the driver generator, only parts of the client side are really relevant. Developers might also be interested in the server part.

For a more detailed documentation of the code and provided methods, Javadoc style comments are provided, which can be transformed into an html or tex representation similar to the Java API specification using doxygen (see Chapter 2 for details on how to enable documentation generation).

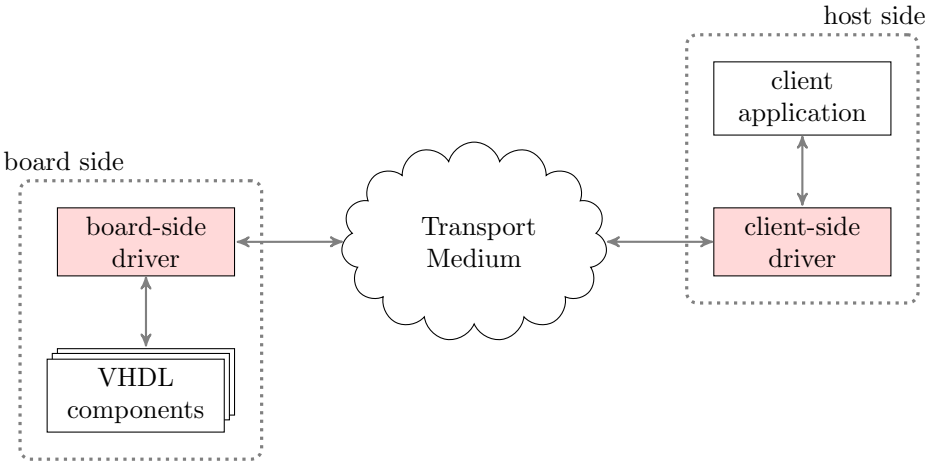

Figure 3.1: A high-level view of the data flow from a client application to vhdl components on the board through the generated driver

The overall architecture of the driver is depicted by Figure 3.1. Data is sent from an embedding client application to the client side driver. This driver communicates over some transport medium with the board side driver, which in turn distributes the received data to corresponding VHDL components on the FPGA. Results are sent back through the same chain.

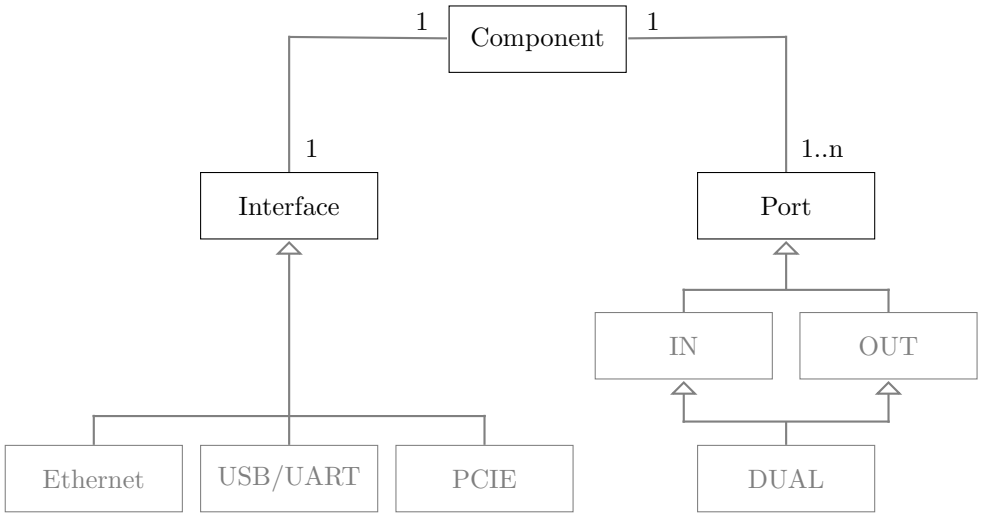

Figure 3.2: Architecture of the host part

3.1 Host Side

The client side consists of several cpp sources and headers files. These files can be imported in any client side project and be used to wrap communication with components on the board.

The architecture of the host software is depicted in Figure 3.2. The core concept of the host software is the *component*. Such a component is a designed hardware unit, which can have several *ports*, over which data can be sent to or received from the component. Hidden from the user, a component also has exactly one *interface*, which handles actual communication.

3.1.1 Component

A component is a piece of designed hardware. It has multiple ports over which communication can take place, i.e. data is sent from or to the component, esp. the control ports *clock* and *reset*. Usually components receive data, process it and send back some results, though possibly on another port.

The host driver contains an abstract generic component, describing components in general. User-defined components have to be described in VHDL for embedded system design with the Xilinx toolsuite. Such VHDL definitions can be referenced in the board description language provided by the HOPP Driver Generator. For each user-defined core, a new subclass is created, which contains the specified ports. For each instance of a core on the board, an object of the cores subclass together with all its ports is instantiated in the driver respectively. Communication with the boards components happens through these objects.

All user-defined cores and instances of these cores can be found in components.h

GPIO compoments These components are specialised, predefined I/O components. Currently, three of these components are supported:

- LEDs
- Switches
- Buttons

Communication to these components is not handled via ports. Instead, it is possible to directly read or write the state of the component. LEDs can only be written to, switches and buttons can only be read from.

3.1.2 Port

A port marks an AXI stream interface used to send data to or receive data from components. A port is always assigned to a single component, but a component can have multiple ports. Ports can be receiving ports, sending ports or bi-directional ports. They can have an arbitrary bitwidth, yet the bitwidth supported by the microblaze as well as the arm processor is limited to 32-bit. Also, the total number of ports is restricted in the microblaze / arm. We allow an arbitrary bitwidth and port number to retain generality of the driver for other platforms, but support arbitrary bidwidth translation and port number restriction to hide complexity from the user. The same applies for encoding format, i.e. the order of bits.

3.1.3 Queues

Writing data to these ports does not directly result in a message being sent to the board. Instead, written values are stored in a client-side queue for each port. This is done for the following reasons.

First of all, the client-side buffer is a method of *flow control*. The board has only restricted resources and should not be flooded with more data than it can process. Consequently, data is stored on the client, if the board buffers have been filled and no more data can be received.

Additionally, the client buffer offers *congestion control*, since several small values can be packaged together in one message. This results in a smaller amount of messages being sent over the medium.

Similar queues exist for out-going ports. A read on such a port object does not read directly from the board, but from the local queue.

3.1.4 I/O Threads

The queues described above are filled asynchronously with two separate threads, a writing and a reading thread. High-level functionality of theses threads is

rather self-explaining. The writing thread loops over all input queues and writes existing values on the medium. The reading thread does the opposite, i.e. listens to the medium and acts according to received messages (see Section 3.3 for details about messages the server can send).

Both threads have to be started before using the API using `startup()` and should be shut down afterwards using `shutdown()`.

3.1.5 Communication medium

The communication medium is hidden from the user. Each driver is bound to a single communication medium, which wraps lower-level communication (essentially transport layer and below) between host and board driver. The communication medium abstracts from the actually used technology and provides a homogeneous api for the I/O threads. Network interface specific initialisation is generated as well and is not required by the user (other than annotating configuration details in the board description).

Currently, three communication mediums are envisioned:

- **Ethernet Lite**, which is already implemented
- **USB/UART**, which is considered as second interface and
- **PCI Express**, which will not be implemented in the initial driver generator.

Ethernet Lite Communication over Ethernet is based on the lightweight IP stack, originally developed by Adam Dunkels[1].

It is possible to implement more efficient Ethernet communication using dedicated VHDL components instead of running the lwip stack on the general purpose CPU. Such a component could easily be integrated in form of a new Communication Interface. An example for a dedicated VHDL communication component together with a client API is described in [1, 2].

Describe implementation of general interface methods in Ethernet as well as challenges and problems this interface poses.

PCI Express According to [1], there exists a Xilinx wrapper for PCIE communication.

[1]The lwip stack is documented with a wiki available at `http://lwip.wikia.com/wiki/LwIP_Wiki`

3.2 Board Side

The board-side driver consists of several files that can roughly be split into three groups: files handling communication with the host-side driver, files handling communication with board components and setup files.

All these files have to be imported in a new Xilinx SDK project started from an existing XPS project. The `.elf` file generated from this project then has to be uploaded to the target board.

Medium The medium folder contains a source and header file for the medium the board is attached to. In this file, medium-specific setup is performed and listening to incoming messages is handled. An incoming message is passed to the protocol interpreter, which distributes the content of the message in accordance to its header.

Components The components folder contains code for component-specific setup and communication between the boards general purpose cpu and designed IP cores. Code for controlling gpio components will also be generated in this folder.

Initialisation Files for initialisation are generated directly in the `src` folder of the project. Among these files is the `main.c`, which calls the initialisation procedures of the medium and all components and starts the scheduling loop. This loop performs three kinds of operations.

First, it checks for incoming messages and process their contents. Most of the time, this includes storing values in the in-going microblaze queue and acknowledging them. More details about message handling can be found in Section 3.3.

The loop also shifts messages from in-going microblaze queues to in-going hardware queues and vice versa from out-going hardware queues to out-going microblaze queues.

Finally, the loop writes values from out-going microblaze queues on the medium.

3.3 Protocol

This section covers the transmission protocol between client and server application. Note that the protocol is base on the assumptions made above about client and server, specifically the used buffers and client I/O threads allowing parallelism with sending and receiving messages.

3.3.1 State transitions

Note: These calculi denote state transitions of a DFA. The precondition marks several conditions and an in-going message. The postcondition may contain an out-going message. Used messages are:

- **wrt** An application writing values to the driver
- **snd** Send values for a component to board
- **ack** Receive ack for values from board
- **poll** Receive poll from board

In addition, several functions have been used to simplify the calclui.

- **size(q)** The size of a value queue q
- **drop(q, i)** Drops the first i values of a queue q
- **first(q, i)** Obtain the first i values of a queue q. Returns a smaller array, if size(q) $<$ i

The state is denoted by an array of value queues q and an array of transit values s. Both arrays are sized by the number of existing board components. The queues store values that should be received by the board, the transit values store the number of values currently in transit, i.e. that have been sent but not yet been acknowledged by the board.

$$\textsc{Write First}\ \frac{q[a] = \emptyset \qquad s[a] = 0 \qquad wrt(a, v)}{q[a] = v \qquad s[a] = size(first(v, n)) \qquad snd(a, first(v, n))}$$

Initially, the queue is empty and nothing has been sent. If a message is written in this state, store it in the queue, transmit the first n values and store the number of sent values.

$$\textsc{Write More}\ \frac{q[a] = v \qquad wrt(a, v')}{q[a] = v \oplus v'}$$

If more values are received (i.e. while values are in transit), append the new values to the queue.

$$\textsc{Ack All}\ \frac{q[a] = v \qquad s[a] = i \qquad ack(a, i)}{q[a] = drop(v, i) \qquad s[a] = size(first(v, n)) \qquad snd(a, first(v, n))}$$

If all sent values are acknowledged, but the queue contains more values, remove the acknowledged values, send the next n values and set the send counter accordingly. If the queue is empty (i.e. no more data to send), `first` evaluates to the empty list. Do not send anything in this case.

$$\textsc{Ack Some}\ \frac{q[a] = v \qquad s[a] = i \qquad ack(a, j) \qquad j{<}i}{q[a] = drop(v, j) \qquad s[a] = 0}$$

If not all sent values have been acknowledged, clear the acknowledged ones, reset the send counter and wait for polls.

$$\textsc{Poll}\ \frac{q[a] = v \qquad s[a] = 0 \qquad poll(a)}{q[a] = v \qquad s[a]size(first(v, n)) \qquad snd(a, first(v, n))}$$

If a poll is received, send the first n values for the polled component and set the send counter. If the queue doesn't contain any values, `first` evaluates to the empty list. Do not send anything in this case.

$$\textsc{Poll Transit}\ \frac{s[a] = i \qquad i{>}0 \qquad poll(a)}{s[a] = i}$$

Ignore polls for components, which have unacknowledged data in transit. This catches the (unlikely) case of the last sent package (i.e. client queue is empty after sending this package) filling the server queue completely. The server will acknowledge all but will poll after the first value has been consumed.

3.3.2 Control Flow Graph

Host-side Driver

In addition, to the control flow graph provided in Section 3.3.2, the host-side driver state is defined with two variables `q` and `s`. `q` denotes the host queue, while `s` stores the number of values in transit, i.e. those, that have been sent but not yet acknowledged. Finally, the value `n` represents the size of the board-side queue (which is defined by the user in the board specification file and therefore statically known by the generated driver).

Several variables occur in the control flow graph:

- **v** Values of some sort (e.g. an array of integers)
- **i** An unsigned (i.e. positive) integer number, smaller than `n`

In addition, several functions on parts of the state are introduced, to simplify the automaton.

- **size()** The number of values currently held by q
- **empty()** `true`, if `size() == 0`, `false` otherwise
- **drop(i)** Drops the first i values of q
- **first(i)** Obtain the first i values of q. Returns a smaller array, if size() $<$ i

Used messages are:

- **wrt(v)** An application writing values to the driver. Appends `v` to `q`.
- **snd(v)** Send values for a component to board. Sets `s` to `size(v)`.
- **ack(i)** Receive ack for values from board. Receiving an `ack` resets `s` and also drops the values from `q`, i.e. `drop(i)`.
- **poll** Receive poll from board.
- **notify** Notifies the application of successful sending of all queued up messages.

Note, that this graph only addresses a single component. In a scenario with multiple components, an individual state exists for each component. A surrounding scheduler allows only a single automaton to make a transition.

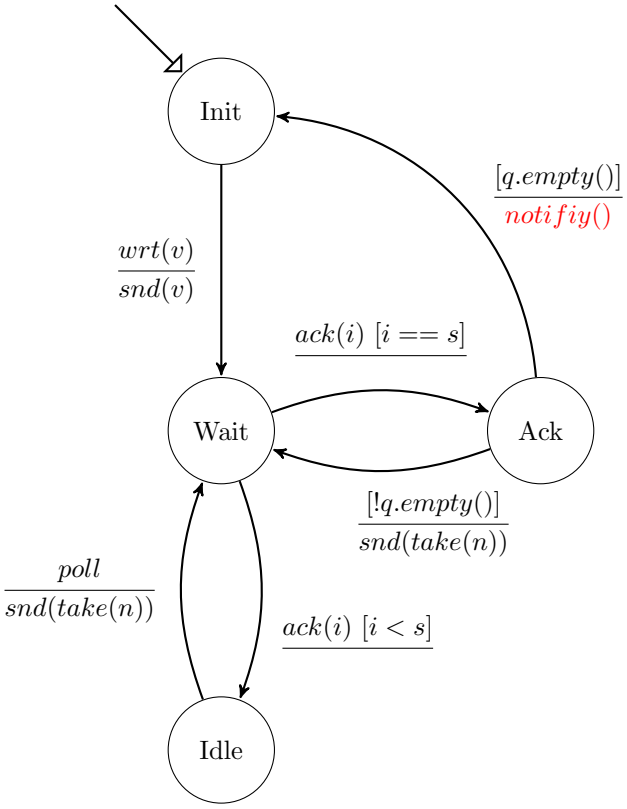

Figure 3.3: Control flow graph of the host-side driver

3.3.3 Sequence charts

Some sequence charts to visualise the envisioned process. For example, Figure 3.14 shows in detail, how a non-blocking write is handled. First of all, the values are stored client-side. The application receives a pointer to a `write` object, which represents the current state of the scheduled write, i.e. how many values have been received by the component on the server.

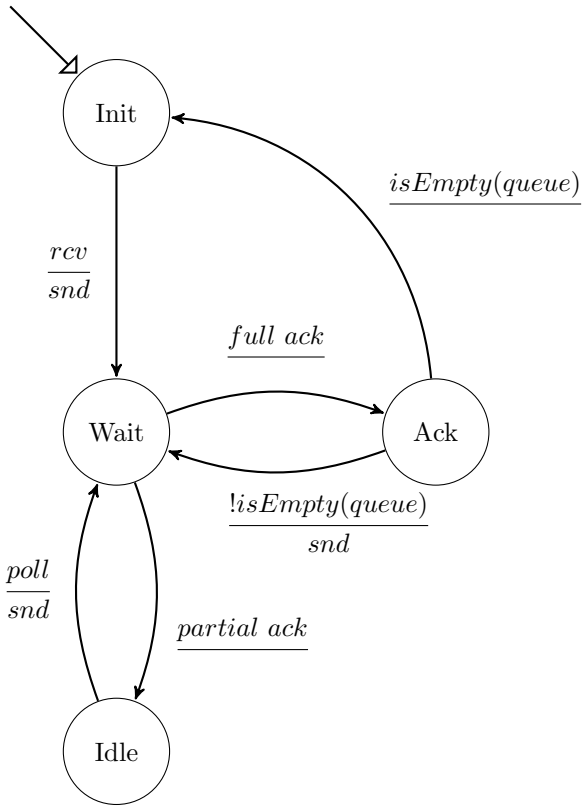

Figure 3.4: Simplified control flow graph of the host-side driver

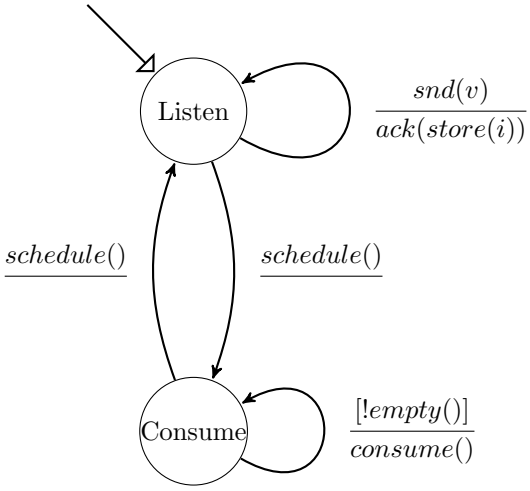

Figure 3.5: Control flow graph of the board-side driver

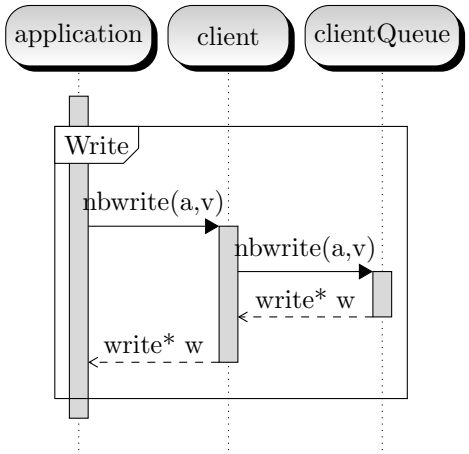

Figure 3.6: An application writing values through the client-side driver api

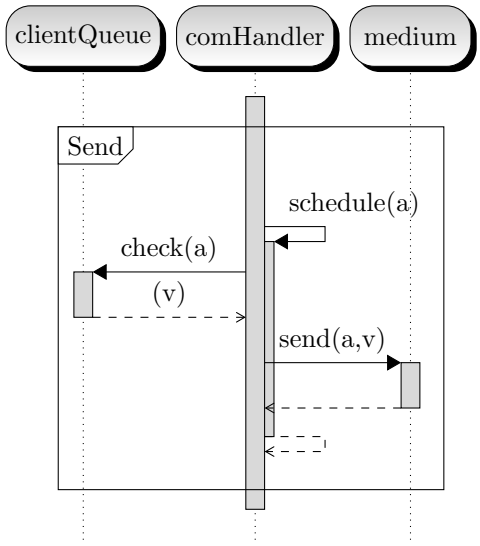

Figure 3.7: A client sending data

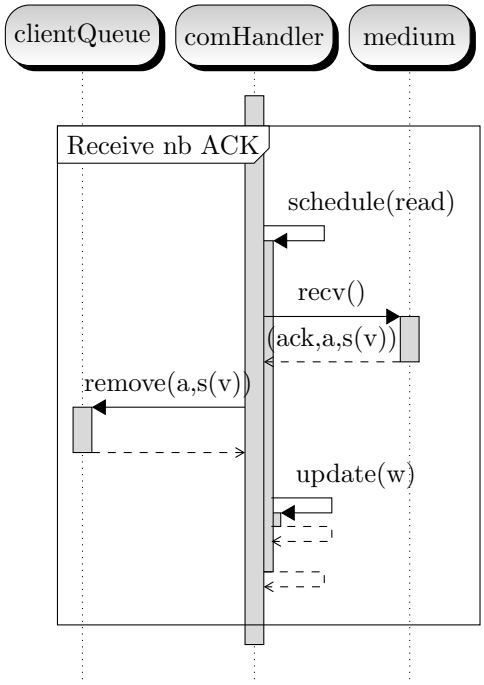

Figure 3.8: The client-side driver receiving an acknowledgement from the server

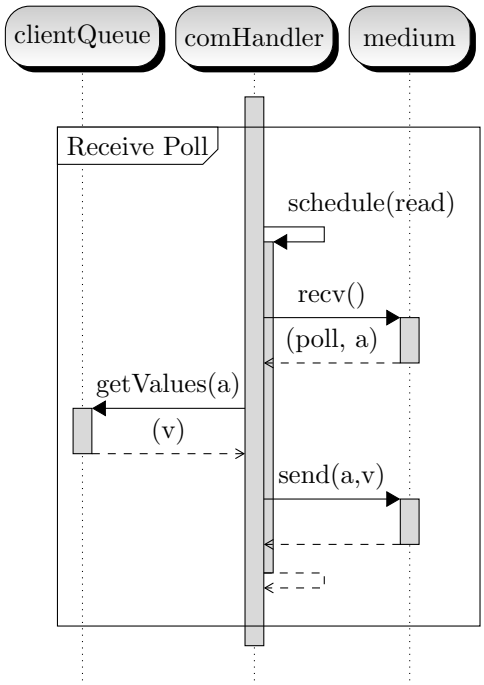

Figure 3.9: The client-side driver receiving a data request from the server

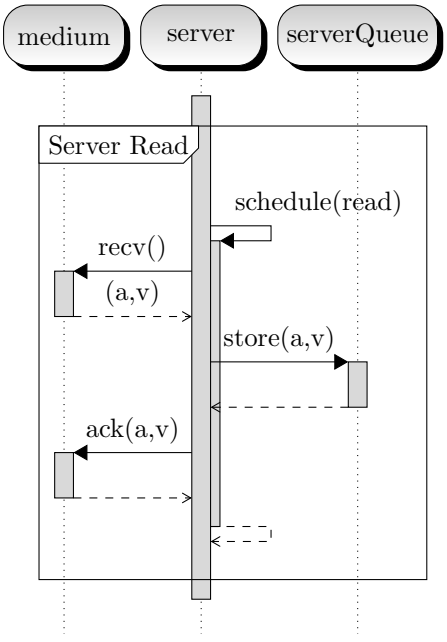

Figure 3.10: The server-side driver receiving a data package from the client

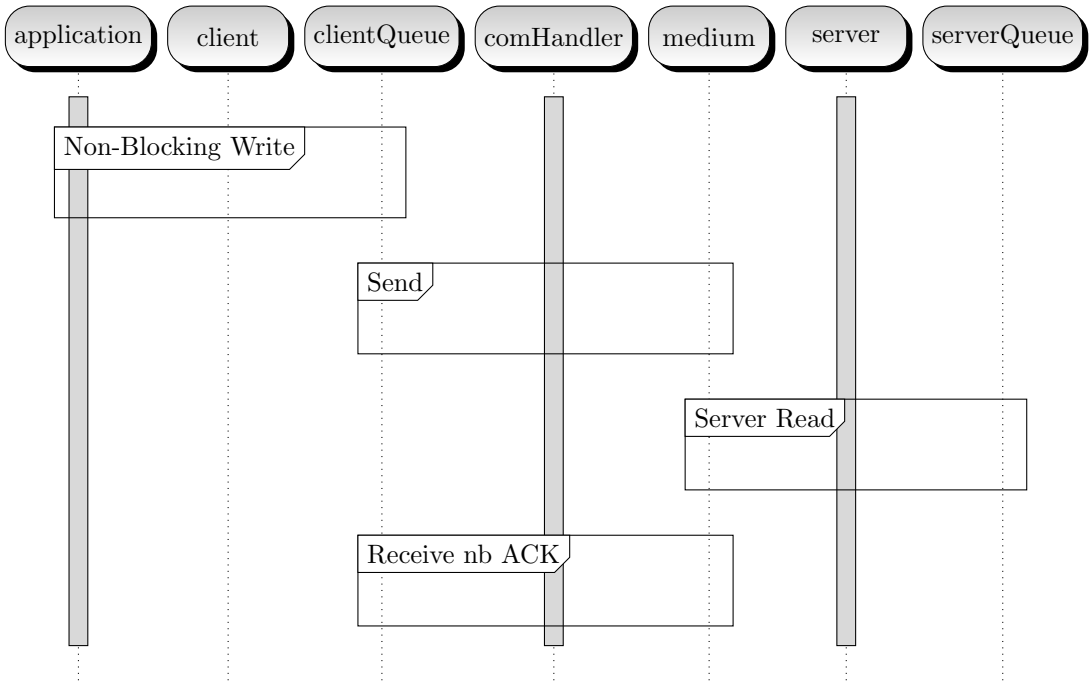

Figure 3.11: Composed example of a non-blocking write using the building blocks defined before.

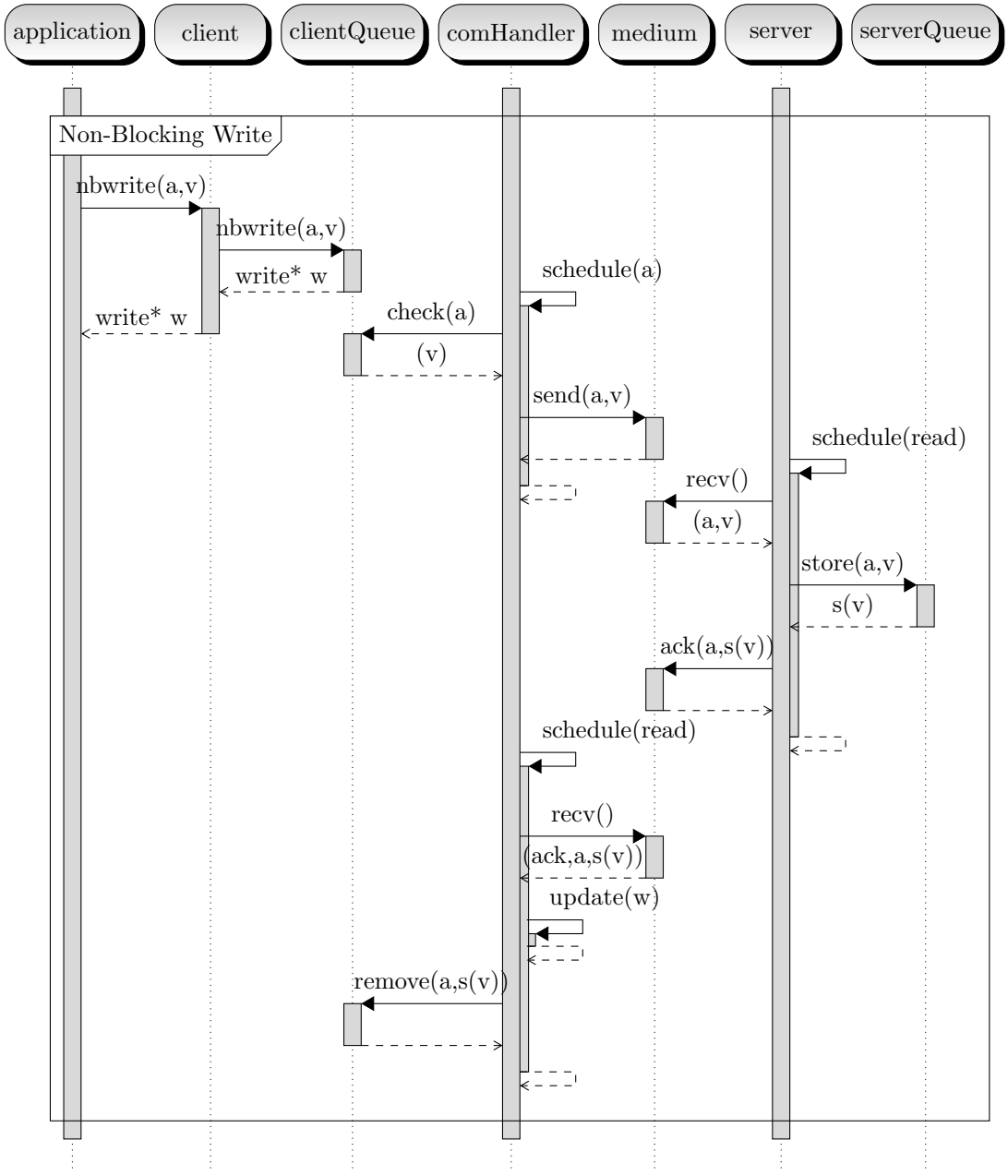

Figure 3.12: A sequence diagram for a non-blocking write. Note, that non-blocking only means, the client does not wait for the target **component** to receive the message, but still waits for the client queue to receive the values.

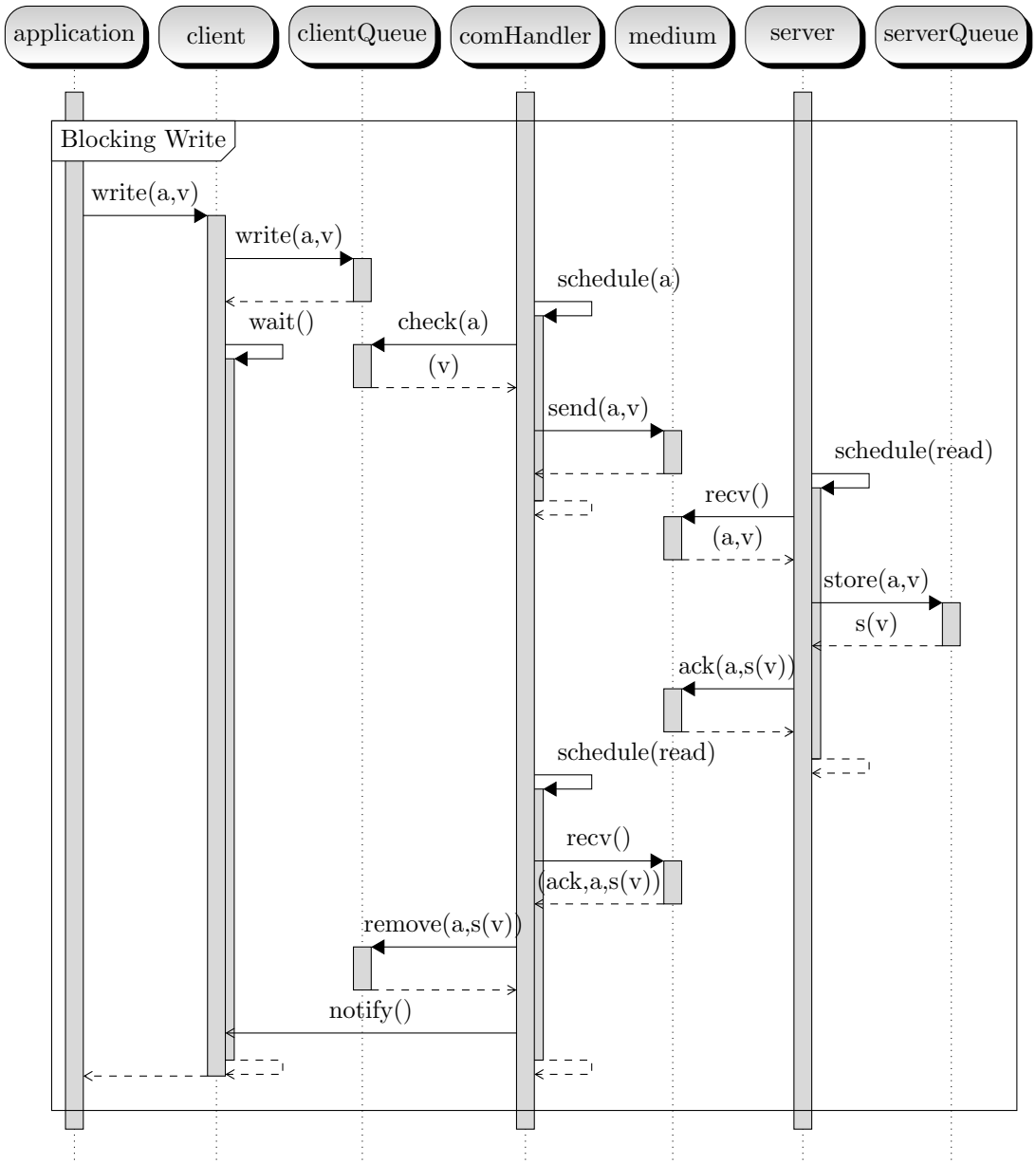

Figure 3.13: A sequence diagram for a blocking write. Note, that only the application is blocked, not the communication handler, which may process non-blocking writes sent before the blocking write **after** the non-blocking write.

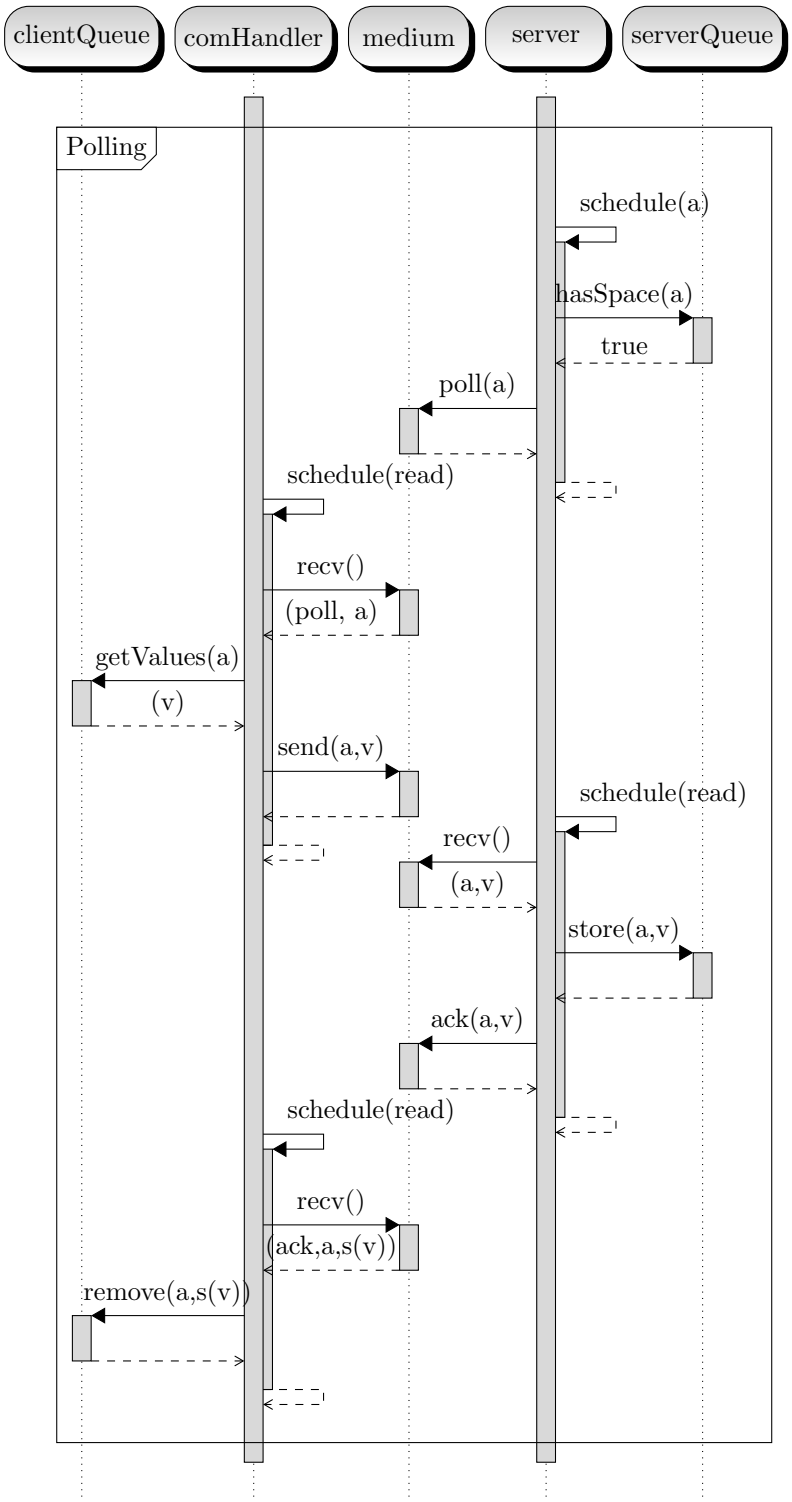

Figure 3.14: A polling scenario. The server scheduler reaches component a and registers free space in the queue (that was formerly full). He then polls values for component a. The poll triggers sending of data at the client, which is later received in a read phase of the server.

Chapter 4

Generator

This chapter is used to explain the code generator itself. This information is intended for future developers of the driver generator.A reference to the third Katja report might be useful.

4.1 Used Libraries

This section gives a short overview over the used libraries and explains what for and why they are used. These libraries are included in the repository and build jar file and require no further user interaction. Still, as the generator code depends on them, they are introduced here for future developers.

4.1.1 JFlex & CUP

This can be kept short with a reference to the JFlex/Cup documentation Flex is a scanner generator, CUP a parser generator. Both together are currently used to parse `.mhs` files into abstract syntax trees. In future versions, they will be used to parse the DSL of the driver generator.

4.1.2 Katja

The driver generator uses the Katja tool, developed by the Software Technology Group of the University of Kaiserslautern. This tool generates several data types[1]. To be more specific, it provides the AST build up by the CUP parser as well as a model of the `C/C++` language described in detail in Section 4.3.

[1]Since these data types will be described using their Katja specifications, it is strongly recommended to read through the Katja specification provided in form of three technical reports at `https://softech.informatik.uni-kl.de/Homepage/Katja`

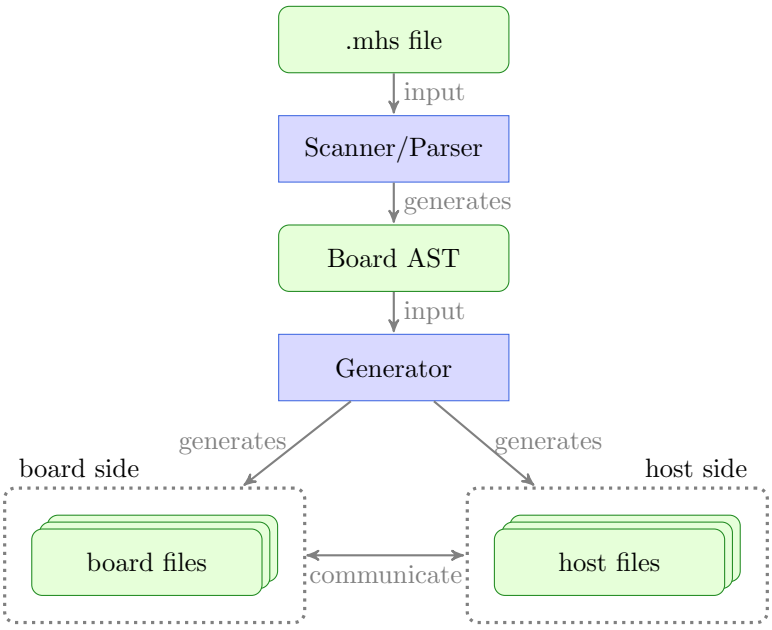

Figure 4.1: A rough sketch of the translation process so far

4.2 Generation Process

The overall process of the driver generator is described by Figure 4.1. The source `.mhs` file describing the system board design is first translated into an internal representation of the board, i.e. an abstract syntax tree. This AST is used as input for the generator, which in turn outputs models of all required header- and source files. These models are translated into files by the `C/C++` unparser.

4.3 CModel

The `C` model provides data types representing a `C/C++` program. Note, that the model is neither complete nor always valid, i.e. not all `C` programs can be described using this model and it is possible to specify a model not translating into valid `C`. Still, the model simplifies the process of code generation. The model is used for generating `C` as well as `C++` code.

```
MFile ( MDocumentation doc, String name, MDefinitions defs,
    MStructs structs, MEnums enums, MAttributes attributes,
    MMethods methods, MClasses classes )

MClass ( MDocumentation doc, MModifiers modifiers, String name,
    MTypes extend, MStructs structs, MEnums enums, MAttributes
    attributes, MMethods methods, MClasses nested )
```

```
MModifier = PRIVATE() | PUBLIC() | CONSTANT() | STATIC() | INLINE()
```

First of all, files can be documented using an `MDocumentation` element. If the documentation is not empty, a `@file` tag is attached to indicate this as a file documentation for doxygen. A file has a name and consists of several definitions, structures, enums, attributes and methods. Files also can contain several classes. These classes again have a name and can contain all these components including other classes. In addition, classes can contain modifiers and inherit components from other classes. Classes also can be documented. The allowed modifiers are private, public, constant, static, and inline. Note, that not all of these modifiers are class modifiers, and several combinations of modifiers are invalid (e.g. private and public). The model relies on the developer to choose modifiers according to the modified program part.

```
MDefinition ( MDocumentation doc, String name, String value)
MStruct ( MDocumentation doc, MModifiers modifiers, String name,
    MAttributes attributes )
MEnum   ( MDocumentation doc, MModifiers modifiers, String name,
    Strings values )
```

Definitions, structs and enums mark rather trivial tuple productions. A definition simply assigns a name to a value. Enums list a number of possible values. All three can be documented.

```
MAttribute ( MDocumentation doc, MModifiers modifiers, MAnyType
    type, String name, MCodeFragment initial )
MCodeFragment ( String part, MIncludes needed )
```

Attributes are similar to definitions, but are typed and may also be left unassigned, using an empty code fragment. The `MIncludes` is required if the type of the attribute is not defined within this c file itself. They also can be documented.

```
MMethod ( MDocumentation doc, MModifiers modifiers, MReturnType
    returnType, String name, MParameters parameter, MCode body )

MReturnType = MAnyType | MVoid() | MNone()

MParameter ( MParamType refType, MAnyType type, String name )
MParamType  = VALUE() | REFERENCE() | CONSTREF()

MCode ( Strings lines, MIncludes needed )
```

Methods have a return type and a list of parameters. The method body is also more complex than a simple code fragment and can consist of several lines, which are not checked any further in this model. The return type can be any `C` type as well as void. For constructors in `C++`, the return type `MNone` is used. Parameters also have a type and name. Furthermore, the mode of parameter passing has to be specified.

```
MAnyType = MType              ( String name )
         | MArrayType         ( MAnyType type, Integer length )
```

```
        | MPointerType      ( MAnyType type )
        | MConstPointerType ( MAnyType type )
```

The type system of the model supports arrays as well as (const) pointers. The basic type is the `MType`, which consists only of a string that has to reference an existing `C` type, e.g. `int`, `struct student` or `enum day`. This type can then be extended using pointer or array types. So `MArrayType(MType("int"), 5)` would mark an integer array of length 5. Note, that these types are nested semantically rather than in the same order as in `C`. Consequently, a point type of a const pointer type of type integer will be translated into `int const ** a`.

```
MDocumentation ( Strings doc, MTags tags )

MTag = PARAM      ( String name, Strings details )
     | RETURN     ( Strings details )
     | THROWS     ( String type, Strings details )
     | DEPRECATED ( Strings details )
     | SEE        ( String see )
     | AUTHOR     ( String name )
     | SINCE      ( String date )
```

For generation of the API, Javadoc style documentation has to be added to the model. A `MDocumentation` element contains documentation for the following block as well as possibly several tags. Tags can be parameter or return value descriptions of a method/procedure, descriptions for exception behaviour, deprecation descriptions or references to other elements of the code. The `JAVADOC_AUTOBRIEF` option is set in the generated doxygen config files, resulting in the first sentence (concluded by a dot and following space or newline) will be used as short description for the documented program part. Since neither the driver generator nor doxygen perform sanity checks, we rely on the user to only introduce meaningful tags for a documentation element. Note further, that the order of tags influences the order of elements in the generated API description.

4.4 Unparser

Different unparsers are used, to translate the `C` model in actual code. For each instance of the model, a header file and a corresponding source file, either `C` or `C++`, has to be generated. These unparsers are called depending on the particular instance of the model. Files intended to be loaded to the board have to be plain `C`, files intended for the client side can also be `C++` files. The unparsing itself is realized using the visitor pattern. Each visit method appends code to a string buffer depending on the visited element.

Header Unparser The header unparser is used for both, unparsing `C` as well as `C++` code. Consequently, it doesn't filter any constructs, but accepts everything specifiable with the model.

This unparser generates only signatures for all methods and only the declarations of attributes and enums. However, the header file will contain all includes referenced within the model.

Plain `C` Unparser Since plain `C` doesn't have any concept of classes, using a model with classes in this unparser will result in exceptions. Otherwise, all components and combinations are accepted. The source file will only import the corresponding header file, no other headers or sourcefiles.

`C++` Unparser For the `C++` unparser, like with the header unparser, every component and combination is allowed. The source file will only import the corresponding header file, no other headers or sourcefiles.

Bibliography

[1] Alachiotis, N., Berger, S.A., Stamatakis, A.: Efficient PC-FPGA Communication over Gigabit Ethernet. In: Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology. pp. 1727–1734. CIT '10, IEEE Computer Society, Washington, DC, USA (2010), http://dx.doi.org/10.1109/CIT.2010.302

[2] Alachiotis, N., Berger, S.A., Stamatakis, A.: A Versatile UDP/IP based PC-FPGA Communication Platform. In: ReConFig 2012 (2012)

[3] Lofgren, A., Lodesten, L., Sjoholm, S., Hansson, H.: An analysis of fpga-based udp/ip stack parallelism for embedded ethernet connectivity. In: NORCHIP Conference, 2005. 23rd. pp. 94 – 97 (nov 2005)