

# HOPP Driver Generator Documentation

Thomas Fischer

December 17, 2012

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>User-Documentation</b>	<b>3</b>
2.1	Command Line Interface . . . . .	3
2.2	Resulting Artifacts . . . . .	3
2.2.1	Board Part . . . . .	4
2.2.2	Client Part . . . . .	4
<b>3</b>	<b>Developer-Documentation</b>	<b>5</b>
3.1	Overview . . . . .	5
3.1.1	JFlex & Cup . . . . .	5
3.1.2	Katja . . . . .	5
3.2	Generation Backend . . . . .	5
3.2.1	CModel . . . . .	5
3.2.2	Unparser . . . . .	7
3.3	Transformation . . . . .	7

# Chapter 1

## Introduction

This document is intended to give an overview over the HOPP Driver Generator for both, users and developers. Users can find an easy to understand explanation of the generator as well as usage examples in Chapter 2. Developers and other interested readers can find details about the architecture and implementation of the generator in Chapter 3. Here be an introduction to the system [1].

The HOPP Driver Generator is intended to provide embedded system developers with a simple, convenient interface to communicate with their designed hardware components.

## Chapter 2

# User-Documentation

### 2.1 Command Line Interface

The HOPP Driver Generator can be called using the command line interface (CLI). While the tool generates C/C++ code, it is written in Java and therefore requires an installation of Java 6 or above. The CLI offers several parameters to further configure the run of the generator.

Additionally, the `.mhs` file of the system is required by the generator. So a complete call looks like `java -jar driverGenerator.jar [OPTIONS] <.mhs file>`.

### 2.2 Resulting Artifacts

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.

-d	-dest	Specifies the destination directory. All files will be generated into the specified directory. If none is specified, the current working directory will be used instead.
	-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.

### **2.2.1 Board Part**

For the board, only two relevant files are generated, a header and a plain c source file. Both 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.

### **2.2.2 Client Part**

The client side currently also consists only of a header and a plain c source file. These files can be imported in any client side projects and be used to wrap communication. Currently, it is only possible to set the LEDs of the board.

## Chapter 3

# Developer-Documentation

This section is addressed to future developers of the Driver Generator and describe the architecture and implementation of the generator itself.

### 3.1 Overview

#### 3.1.1 JFlex & Cup

This can be kept short with a reference to the JFlex/Cup documentation

#### 3.1.2 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<sup>1</sup>. 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 3.2.1.

### 3.2 Generation Backend

This section is used to introduce the C generation backend. A reference to the third Katja report might be useful.

#### 3.2.1 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

---

<sup>1</sup>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>

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 ( String name, MDefinitions defs, MStructs structs, MEnums
        enums, MAttributes attributes, MMethods methods, MClasses
        classes )
MClass ( 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, each file has a name. A file 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. 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 ( String name, String value)
MStruct ( MModifiers modifiers, String name, MAttributes attributes
        )
MEnum ( 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.

```
MAttribute ( 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.

```
MMethod ( MModifiers modifiers, MReturnType returnType, String name
        , MParameters parameter, MCode body )
MParameter ( MParamType refType, MAnyType type, String name )
MReturnType = MAnyType | MVoid() | MNone()
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" or "struct student". 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`.

### 3.2.2 Unparser

Explain the unparser of the model. Shouldn't be TOO involved.

**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.

## 3.3 Transformation

Explain the transformation of the source file into the c model. Maybe also explain the generated code??



# Bibliography

- [1] Fischer, T.: Data Binding for Schemata with Integrity Constraints and Atomic Procedures - A Generative Approach for Object-Oriented Languages. Master's thesis, University of Kaiserslautern (2012)