Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces

Mattias O'Nils, Johnny Öberg, Axel Jantsch
Royal Institute of Technology
Department of Electronics, Electronic System Design Lab.
Electrum 229, S-164 40 Kista, Sweden
Email: {mattias,johnny,axel}@ele.kth.se

Abstract
ProGram, a grammar based communication protocol description language, is used for architectural independent modelling of device drivers and bus interfaces for mixed hardware/software systems. The specification of the protocol is separated from the description of processor bus interfaces and operating system device driver interfaces, which ensures a high efficiency in device driver development and maintenance. A synthesis method for device drivers is presented together with results on modelling and implementation efficiency for both device drivers and bus interfaces.

1 Introduction
Embedded IPs integrated in large systems on a single chip, make interface design and validation to a dominating design problem. We address the problem of interfacing devices in embedded hardware/software systems. As elaborated by Tuggle in [9], for a given device, a device driver must be written for every operating system it will be used with. For hierarchical device drivers like SCSI systems, the number of device drivers needed for a device is the product of the number of supported operating systems and the number of supported host adapters. If we also add different implementation styles of a device driver, the number of possible implementations will be the product above, multiplied with the number of implementation alternatives. One quickly realizes, that verifying all these device drivers is a huge task. To solve this, we propose techniques for architecture independent modelling and refinement of hardware/software interfaces (device drivers) as well as for the hardware/hardware interfaces (bus interfaces).

Figure 1 identifies what has to be modelled to interface a device to an off-the-shelf processor. For device drivers one needs to declare the I/O to both the device and to the application software from where the device driver is called. The general structure is common, a device driver is composed of several functions for initialization, access of data and interrupt handling, see figure 1. To describe the functionality of the device driver functions, the specification language has to capture all the behaviors defined in figure 1, field c.

To our knowledge, there exists no architecture and implementation independent approach for device driver modelling. For the bus interface process, which translates the bus protocols of the two devices, exists several approaches to generate bus interfaces from architecture independent descriptions [2,3,4,7,8]. Our approach extends ProGram [10] to model both device drivers and bus interfaces. ProGram is a data communication protocol description language that has been proven efficient for the modelling and synthesis of telecommunication protocols [10].

The rest of the paper is organized in the following way. The next section goes through the related work and compares it to our approach. Then we present the ProGram language and propose extensions to model both device drivers and bus interfaces. The fourth section discusses synthesizing techniques for both device drivers and bus interfaces. Finally, we conclude the paper with results on modelling and implementation efficiency.

2 Related Work
The main focus in literature on device drivers is on writing implementations [9] or providing tools for library handling [5]. Chou et al. [1] models and generates software device drivers, but not in the context of an operating system as we do, but rather to emulate the bus interface in software. In our approach we provide the possibility to specify both the bus interfaces and device drivers in one architecture independent language, ProGram.

Bus-level protocols for components are usually described graphically by means of timing diagrams in a data
sheet. For this reason, [1] and [8] have developed specification languages based on timing diagrams. A problem with these approaches is that it is not possible to specify high level behaviour. Lin et al. model the interface with extended STG (Signal Transition Graph) [7] which is based on Petri nets. With this modelling technique they can model asynchrony and events. On the extended-STG model asynchronous synthesis is performed. This approach enables the modelling of more complex behaviours. [3] and [4] use chrony and events. On the extended-STG model presented by other research groups are modelled more efficiently. Approaches described above and the extended version of ProGram et al. use a program state machine (PSM) to specify hardware description languages to model bus interfaces, asynchronous synthesis is performed. This approach enables interfaces but in all examples they use the HDL capability of ProGram which is better suited than SEQ. For synchronous interfaces, ProGram is still comparable with the SEQ model. For asynchronous interfaces, ProGram is better suited than SEQ.

Table 1 shows a comparison between some of the approaches described above and the extended version of ProGram presented in this paper. The table shows that interfaces presented by other research groups are modelled more efficiently in ProGram. The example for SEQ in row 4 is an asynchronous circuit, which is the primary target for SEQ but not for ProGram, but the ProGram model is still comparable with the SEQ model. For synchronous interfaces, ProGram is better suited than SEQ.

### Table 1. Comparison of Lines of Code (LoC).

<table>
<thead>
<tr>
<th>Reference model</th>
<th>ProGram</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDL (VHDL), List 1 in [3]</td>
<td>20</td>
<td>5:4</td>
</tr>
<tr>
<td>Extended TD, fig 1 in [8]</td>
<td>18*</td>
<td>N.a.</td>
</tr>
<tr>
<td>PSM, Fig. 8.10 in [4]</td>
<td>21</td>
<td>1:2</td>
</tr>
<tr>
<td>Timing diagram (SEQ), [2]</td>
<td>6</td>
<td>1:2</td>
</tr>
<tr>
<td>Petri nets (E-STG), [7]</td>
<td>11</td>
<td>1:2</td>
</tr>
</tbody>
</table>

* Number of conditions specified graphically

### 3 Interface modelling with ProGram

ProGram (Protocol Grammar), the proposed language for interface description, is inspired by YACC [6] with respect to its notations but has been designed to handle hardware synthesis. Specifications in ProGram deal with sequences of allowed events instead of states and state transitions as it would in a FSM model.

ProGram as presented in [10] needs extensions to be able to model device drivers and bus interfaces, since ProGram in [10] targets protocol specification for hardware implementation. Figure 2 highlights the parts of an interface between a processor and a device we want to model with ProGram. The rest of this section is devoted to describing how device driver/bus interface functionality is captured with ProGram, at the end the ProGram language is described.

#### 3.1 Modelling of device drivers

The nature of a device driver is to assign and access registers/memories according to predetermined control flow, waiting for events to occur and/or waiting for some specified time interval to elapse. Hence, a device driver is characterized with assignments, control flow, event handling rather than data flow and computations, these characteristics fit very well to the spirit of ProGram. However to model device drivers with ProGram, we first have to analyse behaviours that are not captured with ProGram as specified in [10].

**Port declarations.** A device driver has two types of ports: the first port type is the communication with the device (i.e. declaration of device registers/memories) and the second type of port is for the communication with the application software. Both these types are captured with the ordinary port declaration in ProGram. The identification of where a port is connected, is done with the two key words: hw (device register) and sw (port to application software). For the port declared as hw, an address relative to the base address of the device has to be specified.

The only data type allowed in ProGram is unsigned bit vectors, this is not a problem since most of the functionality of a device driver is to control and move data of data types that can be derived from a bit vector type. Internal arrays are declared as memory in ProGram, the same key word is used to identify array signals (pointers) in the port declaration, i.e. memory. These arrays are treated in the same way as memories with the layout specified by the type in the port declaration.

**Time.** Real time has to be handled in ProGram in order to capture the full functionality of device drivers. Since within a device driver it can be necessarily to wait for some time interval instead of waiting for events, for this the key word wait (time) is used. Another use of time is to specify timing constraints for certain wait loops, i.e. define the wait's timeout to prevent dead-locks.

### Table 2. Device driver modelling examples.

<table>
<thead>
<tr>
<th>Examples of items in figure 1</th>
<th>ProGram</th>
</tr>
</thead>
<tbody>
<tr>
<td>a Declaration of device driver &amp; component entity</td>
<td>%start read(); start_handler(intSignal)</td>
</tr>
<tr>
<td>b Declaration of in parameter from SW (16-bit) and declaration of out parameter to SW (8-bit, size 4)</td>
<td>%input data [16 sw]; %inputOutData [8 sw memory 4]</td>
</tr>
<tr>
<td>c Describing functionality (wait on event and assignment)</td>
<td>readData(PORT); wait (others (data=RD));</td>
</tr>
<tr>
<td>e Waiting for an interrupt to occur</td>
<td>waitForInt; others waitForInt</td>
</tr>
<tr>
<td>f Declaration of a write only device register (8-bit) with relative address 2, and declaration of read/write device memory (8-bit, size 16)</td>
<td>%output FCR [bit8 hw 2] %input buff [bit8 hw 0 memory 16]</td>
</tr>
</tbody>
</table>

**Start conditions and interrupt.** A start condition for hardware is evaluated as soon as the signals within the start condition change (i.e. same as a sensitivity list in VHDL), but since the concept of software and hardware are different the start conditions for device drivers are handled differently. Instead of start conditions that are triggered on signal changes, a start condition is triggered when it is called (i.e. like a function call). This means that the condition fields for device driver rules are empty. Each start condition will be a component of the device driver, as described in figure 1. For interrupt handlers the start condition is modelled as a hard-
ware condition, i.e. sensitive to a signal (interrupt signal).

The keyword interrupt is used to indicate that one branch of a production rule is entered only if an interrupt condition occurs. If an interrupt is activated the other branches are aborted, see Table 2. For example, this construct can be used to model a wait for an external event.

3.2 Modelling of bus interfaces

Since specification of a bus interface is to describe the conversion of two protocols (see figure 2) and [10] defines ProGram for specification of protocols, only small additions to ProGram are necessary to enable bus interface modelling.

**FIGURE 4. Interface between a single port processor and a two port memory.**

**Combinatorial.** All reactions to ProGram grammar rules in bus interfaces are synchronized with a clock and thus the out ports and ways be latched. In some cases this will lead to one extra clock cycle latency. As example consider an interface between a processor and a two port memory (figure 4). To model this interface, the address and data lines have only to be by passed through a buffer. To model this, we use a construct where combinatorial values can be assigned to signals (transparent).

**I/O.** To describe the data port connected to the processor in figure 4, we need to declare an inout port (k inout). For an inout port the drivers for a signal has to be controlled, switching off the signal with help of assigning the constant 0 to a signal as in VHDL.

**Hierarchy.** To model the whole interface in figure 4 with ProGram, we need constructs for declaring (k component) and instantiating components, i.e. hierarchy constructs. Components are instantiated as production rules.

3.3 Specification of ProGram

**Interface declaration.** The interface section starts with a component declaration which is the first part of the hierarchy construct (second is the instantiation), i.e.

```
%component 68toMem
```

The second part of the interface section declares ports for a component and the internal signals used for inter process communication within the component:

```
%input enable bit
%internal synch bit
%output dataOut [bit132 sw
%output RBR [bit32] hw Ox000
```

An input declaration consists of the name, the width and for hardware it can also have a bit rate constraint for each signal. For device driver modelling an indication from where a signal originates: sw, is used for communication with software, we indicate that it is a register in the device with a specified relative address. The interface section ends with start rules as in YACC. These start rules work as process declarations that define the signals used for the start condition. Hence, ProGram can specify concurrent processes in contrast to YACC:

```
%start transferStart(request)
%start start (enable)
```

**Tokens and memories.** A Token is a pattern of bits read from an input stream or written to an output stream and can be viewed as constant. Reading and writing tokens are the primary events. Memories are specified as:

```
memory con_mem [[bit]8] con_stat
```

The size of the memory is given by the number in brackets as the number of address lines connected. Then the rule that specifies the memory field layout is given.

**Unlatched assignments.** Unlatched assignments (used to describe unlatched behaviour) are equivalent with concurrent assignment statements in VHDL, e.g.

```
transparent outValue = (value AND enable)
```

**Actions and grammar rules.** Actions in the grammar specify assignment of values to signals. Expressions to compute these values may be put directly in the assignments or may be associated with some symbols in the action values section. The assignments can then simply refer to these symbols. The expressions allow concatenation and conditionals in addition to the usual arithmetic and logic operations. The operands can be constants, signals, other action value symbols or bit patterns recognized by grammar symbols. A grammar rule consists of a grammar symbol that serves as a rule identifier and a list of alternatives. Each alternative is a sequence of non-terminal symbols, terminals and actions. Passing the new signal stream as a parameter to the subtree of productions does a redirection of the input stream. Actions are enclosed in curly brackets:

```
grammar_rules (sig): symbol_1 grammar_rule |
... |
| symbol_n (action_value_section);
```

An example of a grammar rule with two alternatives selected with the two symbols 0 and 1 follows below. 0 transferStart, is a non-terminal symbol which recursively calls itself. The other alternative, first has an action value and is ended by a termination of the rule.

```
transferStart[reqt] 0; transferStart
| 1 (dataOut = dataIn,)
```

4 Synthesis techniques

4.1 Device driver synthesis

In the following we focus on the device driver synthesis to C code for a single threaded software architectures.

**Port implementation.** As mentioned in the section on ProGram, there exist two different types of ports for device drivers: ports to the device and ports to application software.

Ports to the device are translated to a pointer, with the pointer address set to the sum of the device base address and the port address (port address the product of the relative address form the port declaration and the byte width of the data bus).

Ports to the application program, are translated to pointers to objects in the function parameter list, the pointer address is set by the caller of the device driver. In ports from the application program, are translated to objects in the function parameter list.

**Handling of start conditions.** As discussed in the ProGram section, start conditions without signals are treated as functions and with signals in the start condition are treated as interrupt handlers. Both are implemented as functions, but detailed implementation for software architectures of the interrupt handler is left for future work.

**Code generation.** The generation of C code is based on the algorithms for hardware synthesis presented in [10], i.e. a state machine is synthesized from the ProGram code. Where each start condition results in a state machine clocked with a clock signal specified in the start condition. When the state machine is in idle state, it waits for the start condition to become true. This does not efficiently translate to software execution, where the start condition is translated into a
function and the state machine is not clocked, i.e. the time between state transitions are not fixed.

```
mergeDelayStates(start_symbol);
begin
  for i = 1 to Number_of_States loop
    if Number_of_Predecessors{number of states}[i] = 0 AND Number_of_Successors{number of states}[i] = 1
      mergeStates{number of states}[i],{number of states}[i+1];
    end if;
  end loop;
end method;
```

**Figure 5.** The merge delay states algorithm.

A synthesis procedure optimized for software is derived from the synthesis procedure presented in [10]. To optimize this for software we have to remove output synchronization states (delay states), these are not necessary in software since there is not a fixed time delay between state transitions. The delay state are merged into super states where the code for a sequence of delay states are merged into one state, but making sure the execution order for the code is kept. The algorithm for merging of delay states is shown in figure 5.

Exit transitions (correspond to completion of grammar rules) in the state machine are extracted to generate return-from-function. The state machine in C is built with goto statements since these are executed fast on standard processors.

### 4.2 Bus interface synthesis

In [10] it has been shown, that ProGram can effectively be used for HW interface synthesis. In correspondence to our extensions to the ProGram language, we handle transparent signal assignment and hierarchy with a direct translation to the corresponding VHDL constructs, i.e. concurrent signal assignment and VHDL components since these constructs are well supported in logic synthesis tools.

### 5 Results

Methods for software generation from ProGram descriptions modelling device drivers have been presented. These methods are specific for single threaded software. A comparison of ProGram and C for different devices is presented in table 3, where it is shown that code generated from ProGram is comparable to hand written C code, with respect to size and speed. The software generation has to be further investigated in the future, to find how different software architectures like for instance real-time kernels will affect the code generation and how this should be modelled.

### 6 Conclusion

We have shown how a grammar based communication protocol description language, ProGram, can be used for description of both bus interfaces and device drivers. We have compared ProGram to different bus interface modelling techniques presented by other research groups, and shown that ProGram results in compact descriptions, see table 1, and effective implementations (see tables 3 and 4).

In addition, our approach addresses the maintenance problem of device drivers by significantly reducing the number of entities that must be maintained. Assume a company must support device drivers for 4 different operating systems on 8 variants of embedded processors and each device driver can be implemented in 3 different ways depending on performance and cost requirements. Furthermore we assume 4 protocols which differ in the detailed control and communication flow. Currently the company has to develop 8-3-4 = 384 device drivers and it has to develop 8-3-4 = 96 new device drivers when an operating system is added. In our approach we separate the description of micro processor, operating system and protocol from each other and cover different implementation alternatives by the code generation tool based on user directives. Thus, we have only 4 descriptions for the operating system, 8 for the micro processor and 4 protocol descriptions in ProGram. Consequently, we have only 4+8+4 = 16 independent entities to maintain each of which is significantly smaller than a fully coded device driver. Moreover, some of these entities can be used also in the hardware design of the system, further increasing the development and maintenance efficiency.

### 7 References


### Table 4. Comparison of ProGram and VHDL

<table>
<thead>
<tr>
<th>Interface</th>
<th>Lines of Code</th>
<th>Gate count</th>
<th>Critical path</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC68040 &lt;= &gt; RAM</td>
<td>32/1</td>
<td>32/2</td>
<td>3/2</td>
</tr>
<tr>
<td>MC68040 &lt;= &gt; 4-ph</td>
<td>18/46</td>
<td>54/69</td>
<td>5/5</td>
</tr>
<tr>
<td>2-phase &lt;= &gt; 2-ph</td>
<td>9/52</td>
<td>72/54</td>
<td>5/5</td>
</tr>
<tr>
<td>2-phase &lt;= &gt; 4-ph</td>
<td>9/52</td>
<td>56/31</td>
<td>4/3</td>
</tr>
<tr>
<td>MC68040 &lt;= &gt; YM3623</td>
<td>21/70</td>
<td>1061/710</td>
<td>13/11</td>
</tr>
</tbody>
</table>

a. ProGram/VHDL

b. Generated VHDL code/hand written VHDL code for LSI 10k

---

TABLE 3. Comparison of device drivers specified with ProGram and C.

<table>
<thead>
<tr>
<th>Device driver</th>
<th>Lines of code</th>
<th>Code size</th>
<th>Code efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX650, DAC</td>
<td>6/11</td>
<td>0/0</td>
<td>0.90</td>
</tr>
<tr>
<td>6800, ALCA</td>
<td>17/17</td>
<td>1.33</td>
<td>1.54</td>
</tr>
<tr>
<td>TL16C552, DUART</td>
<td>12/222</td>
<td>1.13</td>
<td>0.98</td>
</tr>
<tr>
<td>MAX197, ADC</td>
<td>5/11</td>
<td>0.86</td>
<td>0.98</td>
</tr>
<tr>
<td>YM 3623</td>
<td>9/12</td>
<td>1.50</td>
<td>1.42</td>
</tr>
</tbody>
</table>

a. ProGram/C

b. Lines of assembler code for HP (lisp) processor, generated = hand written.
c. exec_time (Generated code) = exec_time (Hand written C).

do. Optimization techniques for bus interfaces are also discussed. In table 4 bus interfaces described in VHDL and in ProGram are compared with respect to lines necessary to describe the functionality and the synthesis result. This comparison shows that the result from synthesis is comparable to a hand written VHDL code even though the description is about one third of the VHDL description.