High-Level Specification and Automatic Generation of IP Interface Monitors

by

Márcio T. Oliveira

B.S., Federal University of Minas Gerais, Brazil, 1999.

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

August 2003

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
Vancouver, Canada

Date AUG 28TH, 2003

Abstract

A central problem in functional verification is to check that a circuit block is producing correct outputs while enforcing that the environment is providing legal inputs. To attack this problem, several researchers have proposed monitor-based methodologies, which offer many benefits. This thesis presents a novel, high-level specification style for these monitors, along with a linear-size, linear-time translation algorithm into monitor circuits. The specification style naturally fits the complex, but well-specified interfaces used between IP blocks in systems-on-chip. To demonstrate the advantage of our specification style, we have specified monitors for various versions of the Sonics OCP protocol as well as the AMBA AHB protocol, and have developed a prototype tool that automatically translates specifications into Verilog or VHDL monitor circuits.

Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank Alan Hu for all he has done for me, from providing academic guidance to providing food and money. I have been very fortunate to have him as my advisor. Even after I have left UBC he held weekly meetings with me until I finished the writing of this thesis. I also would like express my gratitute to Resve Saleh for agreeing on being the second reader of my thesis. At last, I would like to thank the ISD students for providing such a good work environment, and specially to Xiushan Feng for helping me with the practical aspects of submitting my thesis while I was away from Vancouver.

MÁRCIO T. OLIVEIRA

The University of British Columbia
August 2003

To my parents and Daniela

Chapter 1

Introduction

1.1 Motivation

Standard design practice is block-based — the design task is carved into small pieces to be tackled by an individual designer or a small team. In the past, block boundaries and interfaces have been casually negotiated through face-to-face discussions among the designers. This informal negotiation does not scale with the push for higher productivity and complexity. In addition, we would like to reuse pre-designed and pre-verified IP blocks — either designed previously in-house or purchased from third-party IP suppliers. As a result, the trend is towards designing with large, complex blocks with well-defined functionality and interfaces.

This trend generates two complementary verification problems: how to verify that a block behaves properly *in its intended environment* without having to model and verify the rest of the system, and how to verify that a system behaves properly without having to instantiate all blocks and flatten the design. Current verification practice for the first problem is to create (by hand) an abstracted environment model for formal verification or a testbench for simulation-based verification. Current practice for the second problem is to create by hand abstract models of the blocks and the system, or else to attempt to verify the whole system and suffer from state explosion in formal verification or slow simulation

speeds and poor coverage during system-level simulation. In either case, this practice is labor-intensive, error-prone, and results in time-consuming false error reports (if the models are too flexible) or missed bugs (if the models are too strict).

Several groups have proposed interface-monitor-based methodologies (e.g., [10, 16, 9]) to address this problem. The common theme is to create a monitor circuit that watches the interface between a block and the rest of the system and flags any violations of the interface protocol. The key insight, empirically confirmed in several case studies, is that designing a passive, declarative monitor is easier than designing an active stub to model the environment. Furthermore, because the monitor may be symmetric between the block and the rest of the system, the same monitor can be used to verify both the block with the system abstracted as well as the system with the blocks abstracted, thereby supporting a compositional/hierarchical verification style. The monitor also provides a precise documentation of the interface, on which formal sanity checks can be applied. Finally, it is possible to convert a monitor circuit automatically into a testbench (stimulus generator) for simulation-based verification [20]. The advantages of a monitor-based methodology are compelling.

Unfortunately, although impressive monitors have been built [15], creating a monitor for a complex protocol is a challenging task because all properties must be extracted from the English written document. This process may lead to incomplete or incorrect specifications, since it isn't straightforward to determine the full behavior of the interface by looking at a set of properties that may or may not depend on each other.

1.2 Project

This work introduces a high-level specification style designed explicitly to simplify specification of interface monitors. Our goal is to provide an extremely easy way to generate monitors for common interface idioms. With numerous emerging standards for system-on-chip interconnect, the need for a simple, concise, and readable way to specify interface protocols is clear. Being able to translate these high-level specifications automatically into monitor circuits allows tapping the power of monitor-based methodologies. By using our

specification style, IP suppliers will be able to formally verify that their cores conform to an interface protocol as well as supply a monitor for that protocol that is both easily human-readable and directly usable by verification tools.

1.3 Contributions

The most obvious contribution of our research is the demonstration that regular expressions work very well for specifying IP interface monitors. That statement, however, is actually false. Existing specification styles based on regular expressions do rather poorly as soon as the interface protocol becomes as complex as typical system-on-chip interconnect protocols. However, by introducing two novel extensions — storage variables and a pipelining operator — we have created a specification style that does work very well for interface monitors. This new high-level specification style can be used to describe the full behavior of the interface, making it easier to write, read and modify than a specification written as a set of properties. Since the full behavior of the interface is specified, it is straightforward to check it against the English document for completeness. The properties are implied from the model during the translation to a monitor circuit. The new extensions require a new algorithm for translating specifications into monitor circuits. We have implemented this algorithm in a prototype tool that translates specifications into monitor circuits in Verilog or VHDL. Finally, we have demonstrated the usefulness of our specification style by developing monitors for two standards for system-on-chip interconnect: large portions of ARM's AMBA AHB high-performance bus protocol [2] and several versions of OCP (Open Core Protocol) originated by Sonics [17].

Chapter 2

Background

2.1 Monitors

A monitor is a circuit that watches the inputs and outputs of two or more connected blocks and flags any protocol violation (see Figure 2.1). Even though the idea is very simple, a monitor is a very powerful verification tool.

A well-written monitor is a complete and unambiguous specification of the interface behavior. It can also be used to constrain the inputs of a block, allowing it to be verified before it is connect to any other block. This approach is especially useful for formal verification where blocks are verified separately because the tools available as of today can not handle large designs.

A practical, industrial-strength verification methodology can be built on extensive use of monitors [5]. Using monitor circuits to encapsulate properties to be checked is an established idea. Many companies specialize in writing and selling monitors of standard interfaces like PCI, AGP, or AMBA. These are usually written in an HDL language like Verilog or VHDL.

Our work was directly motivated by the elegance and power of monitor-based approaches to interface specification [10, 16, 9]. The emphasis of those efforts was mainly on the value of this way of thinking; little emphasis is on the specification language. Our focus

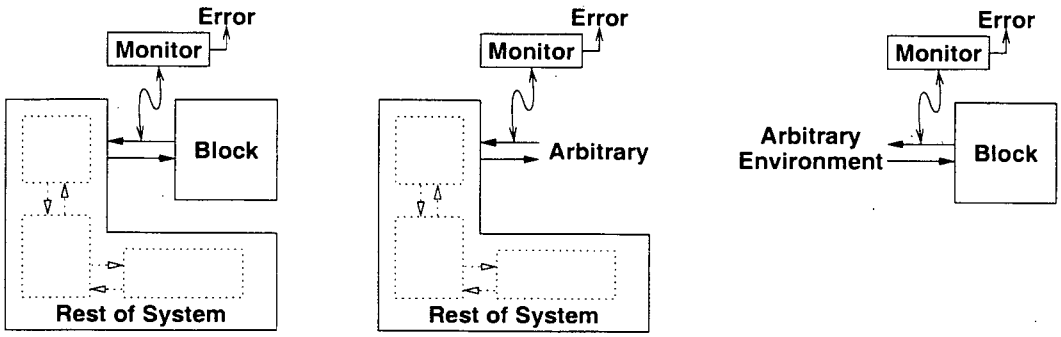


Figure 2.1: A monitor circuit watches the interface and flags any violations of the protocol. The block and system can be formally verified separately. The monitor can also be converted into a simulation testbench.

is on the specification language; we seek to harness those results by providing a shortcut to specifying monitors.

For example, in the style of [16], a monitor is specified as a set of independent properties in the form *antecedent* $\Rightarrow$ *consequent*. An example is show below:

```
previous(request) && !ack => request
previous(!request) => !ack
```

The first formula guarantees that if `request` is high than it must stay high until `ack` is received. The second formula guarantees that `ack` can only be high if `request` was high on the last cycle. Even though this style seems simple, which is also an advantage, it is possible to describe complex monitors using it. Also, by adding a few restrictions on the way the formulas are written, other advantages such as being able to blame the block that was responsible for causing an error, can be obtained (See [16].). One of the disadvantages of this style is that the properties are not written at the transaction level, but at a lower level. Many properties may be needed to describe a transaction, which makes the process of understanding the specification harder.

2.2 Extended Regular Expressions

Since a monitor is a finite state automaton, a possible idea is to use regular expressions to represent it. Regular expressions describe regular languages (languages that can be recognized by a deterministic finite automaton). Every regular expression describes a regular language and every regular language can be described by a regular expression. A regular expression can be composed by the empty string ε, atomic symbols (letters of the language alphabet) and three operations: union, concatenation, and Kleene star.

Given a regular expression r, let $\mathcal{L}(r)$ denote the language (set of strings) recognized by r. We define $\mathcal{L}$ as follows:

Base Case: If a is a letter of the alphabet, then $\mathcal{L}(a) = \{a\}$

Union: If r_1 and r_2 are regular expressions, then $\mathcal{L}(r_1 \cup r_2) = \mathcal{L}(r_2) \cup \mathcal{L}(r_2)$

Concatenation: If r_1 and r_2 are regular expressions, then $\mathcal{L}(r_1, r_2) =$ the set of all strings $\omega = \omega_1, \omega_2$ such that $\omega_1 \in \mathcal{L}(r_1)$ and $\omega_2 \in \mathcal{L}(r_2)$

Kleene star: If r is a regular expression, then $\mathcal{L}(r^*) = \{\varepsilon\} \cup \bigcup_{i \geq 1} \mathcal{L}(r^i)$ where $\mathcal{L}(r^i) = \mathcal{L}(\underbrace{r, \cdots, r}_{i \text{ times}})$

The most direct influence on our work is earlier work from the synthesis community: Production-Based Specification (PBS) [14]. This work uses an extended regular expression language to specify state machines, which are synthesized in polynomial time into circuits, never explicitly building a deterministic finite-state machine and thereby avoiding a potential blowup. Production-based specification has proven to be particularly well-suited to synthesizing protocol state machines, and hence was a natural starting point for our research.

PBS extends regular expressions by adding new operators, like the exception operator, and most important of all, by allowing sub-expressions to be annotated with VHDL code. Every time a sub-expression is matched, the associated VHDL code is executed. This specification style allows the description of complex protocols without the necessity of explicitly describing the control logic associated with it.

In PBS, the regular expression is written in a production-based style, hence the name Production-Based Specification. Productions are nothing but a name given to a sub-expression, which can be used in other productions. Recursion is not allowed, so the productions can be collapsed into a single regular expression. Recursion would allow specifying non-regular languages. The following is an example extracted from [13]:

```
Count -> Valid | Invalid;
Valid -> ONE* Low* ONE*;
{
    IS_LEGAL <= '1';
}
Invalid -> ONE* ZERO+ ONE+ ZERO;
{
    IS_LEGAL <= '0';
    COUNT := 0;
}
Low -> ZERO; {COUNT := COUNT + 1;}
```

This machine counts the number of consecutive zeros in a bit stream. The VHDL code shown below (also extracted from [13]) shows a possible VHDL implementation for this machine.

```
process
begin
wait until CLK'event and CLK = '1';
if (SEEN_TRAILING and DATA = '0') then
    IS_LEGAL <= '0';
    COUNT <= 0;
elsif (SEEN_ZERO and DATA = '1') then
    SEEN_TRAILING := TRUE;
```

```
elsif (DATA = '0') then
    SEEN_ZERO <= '1';
    COUNT := COUNT + 1;
end if;
end process;
```

A specification in PBS is usually very compact if compared with the HDL description of the same protocol, making it very easy to read, write, and modify. Also, productions are an intuitive way to break the description into smaller functional parts.

Because of all these characteristics, PBS seemed a natural starting point for our monitor specification language. But as soon as we began writing monitors using PBS, we found two major problems: pipelined protocols and memory storage. Most complex protocols have some sort of pipelined behavior, and we were not able to describe it easily with PBS. The problem is that the specification is done cycle-by-cycle, but pipelining is best understood as overlapping computations. The other problem is that sometimes we need to check for behavior such as "data should hold until ack" and even though it is possible to describe this using PBS, it would be necessary to write a production for every possible value of the data. Hence, some sort of memory storage is necessary.

2.3 Industrial Specification Languages

Many specification languages have been created recently, such as IBM's Sugar [4], Intel's ForSpec [3], Fujitisu's and Hitachi's CWL [8], and Synopsys' OpenVera Assertions [19]. These could be used to specify monitors, and are therefore related to our specification style. This section will briefly survey these languages and highlight their advantages and disadvantages.

IBM's Sugar is a property specification language that was initially based on the branching-time temporal logic CTL [6] and regular expressions. A branching-time temporal logic reasons about all the possible paths in the model. If a state has many successor states,

then the logic will consider all next states. A CTL formula is composed of path quantifiers and temporal operators. The path quantifiers are A ("all paths") and E ("there exists a path"). The temporal operators are G ("always"), X ("next"), F ("eventually"), and U ("until"). Every temporal operator must be immediately preceded by a path quantifier. The following is a CTL formula for the property "every request must be followed by an ack":

```
AG(request -> AF(ack))
```

This formula can be translated to English as, for *all paths* from the initial state it is *globally* true that for every state where `request` is true then for *all paths* from these states `ack` will *eventually* be true.

Even though CTL is a powerful temporal logic and its verification algorithms can be implemented very efficiently, it may be very cumbersome to write the formulas to express some properties. Sugar's basic idea is to use regular expressions and other syntactic elements to facilitate the writing of complex temporal formulas. The example below shows a Sugar formula and the corresponding CTL formula that represents the property that a `request` must be followed by an `ack` within four clock cycles:

```
Sugar: AG(request -> next_event_f(clk)[1..4](ack))
CTL:   AG(request -> AX(ack) || AX(AX(ack)) || AX(AX(AX(ack)))
                     || AX(AX(AX(AX(ack)))))
```

The operator `next_event_f` is one of the Sugar language extensions. It doesn't add any expressive power to the language but it helps making the task of writing formulas easier. It's easy to see that if the formula involved a larger number of clock cycles it would be very painful to write it in CTL.

Intel's ForSpec is also a property specification language, but it is based on Linear Temporal Logic (LTL) [11]. In a linear-time logic, a formula reasons about one possible computation (one path) of the model. There are some CTL formulas that cannot be expressed in LTL; conversely there are some LTL formulas that cannot be expressed in CTL. In theory, LTL implementation complexity is higher than CTL's. The following is an LTL formula that cannot be expressed in CTL:

```
FG p
```

The meaning of this formula is that p will *eventually always* be true.

Neither LTL nor CTL can express certain ω-regular properties. In order to solve this problem, ForSpec extends LTL with *regular events*, which are a sequence of events represented by a regular expression. The following is an example of a ForSpec formula extracted from [3]:

```
Globally !(request,(!ack & !request)*,!ack & request)
```

This formula represents the property that if a request is made, another request can not be made until an acknowledgment is received.

ForSpec also adds other facilities such as *time windows* and constructs to model multiple clocks and resets. See [3] for more details.

Recently, Sugar was chosen as the standard property specification language by the Accellera [1] organization. In order to satisfy Accellera requirements, a linear temporal logic was added to Sugar, making it very similar to ForSpec with respect to expressive power.

Sugar and ForSpec are related to our specification style in the sense that both use regular expressions to facilitate the writing of temporal expressions. The fundamental difference is that Sugar and ForSpec are designed to specify properties, whereas our aim is to easily and compactly specify entire interface protocols. Therefore, we provide constructs to partition the interface protocol into functional units. We also provide support for common idioms, such as pipelining, which are not directly supported by either Sugar or ForSpec.

The preceding specification languages evolved from temporal logic; others, such as Synopsys' OpenVera Assertions (OVA), evolved from testbench/simulation languages. OVA's syntax is similar to Verilog, constructs like if-then-else and for loops can be used to facilitate the writing of assertions. The basic construct of the language is a temporal sequence. These sequences can be used to verify coverage and also to generate biased testbenches. Two interesting aspects of the language, which are very similar to aspects in

our specification style, are that sequences can be composed as if they were productions, and data can be stored during any time in the sequence to be used later. Below is an OVA sequence example:

```
if (enable) then
    (request #1 !ack #1 ack)
```

The sequence `(request #1 !ack #1 ack)` will only begin to be verified if the pre-condition `enable` is satisfied. The property describes the sequence in which `request` must be followed by `!ack`, which must be followed by `ack`. In this example, the sequence extends for three cycles.

The key differences between OVA and our work is that OVA was not designed to be fully synthesizable, which means that a formula written in OVA may or may not be synthesizable. OVA also does have direct support for pipelining.

The language that is most similar to our specification style is Fujitisu's and Hitachi's Component Wrapper Language (CWL)[1]. CWL is an interface specification language based on regular expressions. The idea is to use regular expressions to write a generalized version of a waveform, which is called a transaction. These transactions can be composed to specify the full behavior of the interface. As in our specification style, pipelining is automatically supported by using special operators to compose transactions. CWL and our work evolved in parallel and neither group was aware of the other. We published first in an international conference, at which time, we were told of the forthcoming release of the CWL Specification [8]. Their current tool [7] that translates CWL to Verilog does not support the pipelining operators described in the language. The strong similarities between the two independent efforts suggests that the specification style clearly captures fundamentally useful concepts.

[1]CWL evolved from the specification language OwL [18], which supports productions but not pipelining.

Chapter 3

High-Level Monitor Specification

3.1 Specification Style

In this section, we introduce our specification style, which addresses the limitations described in Chapter 2. We call our style PREMiS, which stands for Pipelined Regular Expression Monitor Specification. In this section, we present our style informally by example, and in Section 3.2, we define the formal semantics. Appendix A is the language reference manual.

We introduce the specification style incrementally, starting with regular expressions, and then introducing productions, storage variables, and pipelining. Examples taken from the AMBA AHB protocol will illustrate the concepts. We will try to provide enough information for readers unfamiliar with AHB to understand the examples.

Fundamentally, a regular expression specifies a language, which is just a set of strings, which are sequences of characters, which are drawn from some alphabet. Analogously, we start at the very beginning with the alphabet of our specification style. The interface between a block and the rest of the system consists of a bunch of wires: some are inputs to the block; some are outputs. For example, an AHB slave device interfaces to the system via several wires, such as `HADDR[31:0]`, a 32-bit address input; `HRDATA[31:0]`, a 32-bit data output; `HWRITE`, `HTRANS[1:0]`, `HSIZE[2:0]`, `HBURST[2:0]`, which are control sig-

nals describing the type and size of a transfer; and HREADY, HREADYOUT, and HRESP[1:0], which are hand-shaking and response signals. All of these wires are inputs to the monitor, which passively watches their values. Accordingly, the fundamental building-block of our specifications is an assignment of values to the wires on the interface at a given clock cycle.

For convenience, we allow the user to specify any Boolean formula on the interface wires. For example, the AHB protocol defines encodings for the different transfer and response types, so we allow the user to specify:

```
define idle   = !HTRANS[0] & !HTRANS[1];
define busy   =  HTRANS[0] & !HTRANS[1];
define nonseq = !HTRANS[0] &  HTRANS[1];
define seq    =  HTRANS[0] &  HTRANS[1];

define okay  = !HRESP[0] & !HRESP[1];
define error =  HRESP[0] & !HRESP[1];
define retry = !HRESP[0] &  HRESP[1];
define split =  HRESP[0] &  HRESP[1];
```

Any Boolean formula on the interface wires (and `defined` formulas) is a *primitive expression.*

Given primitive expressions, we can define regular expressions recursively in the usual manner. Any primitive expression is a regular expression. A regular expression concatenated to another regular expression is a regular expression. We use a comma as our concatenation operator. For example, the AHB specification defines different response codes for a slave to signal to the master:

```
wait_state -> !HREADY & okay;
okay_resp  ->  HREADY & okay;
error_resp -> (!HREADY & error) , (HREADY & error);
retry_resp -> (!HREADY & retry) , (HREADY & retry);
```

```
split_resp -> (!HREADY & split) , (HREADY & split);
```

The error, retry, and split responses take two cycles: the first with HREADY low, the second with HREADY high. The choice (denoted by ||) between regular expressions is a regular expression. For example, to specify that a transfer can be one of four different kinds, we can write:

```
transfer -> idle_trans || busy_trans ||
            nonseq_trans || seq_trans;
```

Finally, we have the Kleene closure to denote repetition:

```
resp -> wait_state* , (okay_resp ||
        error_resp || split_resp || retry_resp);
```

The above expression specifies the response phase to be any number of wait states, followed by one of the response types.

For notational convenience, we use *productions* as they were defined in production-based specifications [14]. We have actually been using productions already in the preceding paragraph. The symbol to the left of the -> operator is defined to be an abbreviation for the regular expression on the right-hand side. To guarantee that specifications correspond to finite-state machines, productions cannot be recursive.

The above definitions are the same as earlier regular-expression specification styles and appear to be convenient for describing protocols. The behavior of an AHB slave device, for example, is simply a sequence of transfers or idle periods:

```
slave -> (slave_idle || transfer)*
```

where slave_idle means HREADY is low or the slave is not selected, and transfer is defined as above. Describing the full details of typical IP block interface protocols, however, quickly reveals the limitations of a pure regular-expression specification style.

The first major obstacle is persistent storage of information. In AHB, for example, a slave device can reply with a *split* response, indicating that it needs a long time to complete the request. The interface monitor for a split-capable slave should remember the ID

numbers of all masters who have splits pending, to ensure that a slave does not signal completion of a split transaction that has not happened. Encoding such information with regular expressions is possible, but painful: for *every possible value of the saved information*, the user must write a slightly modified version of every production that is affected. Instead, we propose *storage variables* as a simple alternative. The user can declare finite-state variables as part of the specification. At any point in a regular expression, values can be assigned to the storage variables. The values of the storage variables are available in any Boolean formula defining a primitive expression. The monitor for an AHB slave could have a 16-bit storage variable, with one bit for each possible master to indicate whether it has had a request that has been split. Whenever the slave issues a split, the corresponding bit is set; whenever the slave issues a split completion, the corresponding bit is checked.

The other major obstacle is pipelining. Almost all high-performance interfaces are pipelined to some degree. Most formal specifications describe the cycle-by-cycle behavior of an interface, but unfortunately, pipelining is extremely hard to specify (or understand) at the cycle-by-cycle level. Trying to specify pipelining via regular expressions or any other cycle-by-cycle style requires the user to entangle all the possible parallel behaviors by hand, resulting in a difficult, error-prone specification process and an unreadable specification. Instead, pipelining is most naturally understood as an operation that overlaps sequential operations (Figure 3.1). In the AHB protocol, the arbitration phase, address (request) phase, and data (response) phases are all pipelined. The official AMBA specification document [2] describes these phases sequentially in English, and then presents timing diagrams to attempt to show how they entangle in pipelined operation. Our solution is to provide an explicit pipelining operator, similar to the concatenation operator. For example, for an AHB slave monitor, a transfer has an address phase followed by a response phase:

```
idle_trans -> (idle & HSEL & HREADY) , okay_resp;
busy_trans -> (busy & HSEL & HREADY) , okay_resp;
nonseq_trans -> (nonseq & HSEL & HREADY) , resp;
seq_trans -> (seq & HSEL & HREADY) , resp;
```

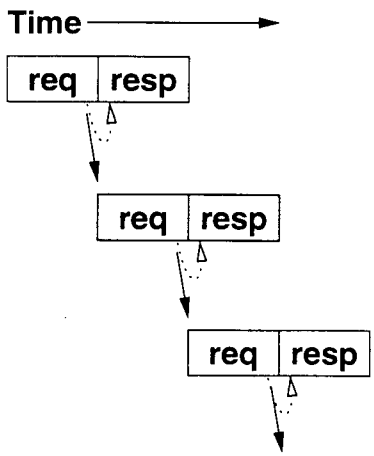


Figure 3.1: This figure shows multiple pipelined transactions, where each transaction has a request phase and a response phase: `(req@resp)*`. Our pipeline operator marks the point where the next computation overlaps the current one. At that point, we fork a new "thread" to complete the current transaction (dotted arrow), while the current thread continues with the rest of the regular expression, if any (solid arrow).

(`HSEL` indicates this slave is selected; `HREADY` is the handshake that indicates the address phase is complete.) However, the address and response phases are pipelined, so that the response phase of one transfer occurs at the same time as the address phase of the next transfer. In our specification style, we simply replace the concatenation operator with the pipeline operator `@`:

```
idle_trans -> (idle & HSEL & HREADY) @ okay_resp;
busy_trans -> (busy & HSEL & HREADY) @ okay_resp;
nonseq_trans -> (nonseq & HSEL & HREADY) @ resp;
seq_trans -> (seq & HSEL & HREADY) @ resp;
```

The semantics of the pipeline operator are that the thread of control forks into two sub-threads when the pipeline operator is encountered: one sub-thread continues with the regular expression as if the right-hand operand of the pipeline operator did not exist, the other sub-thread focuses only on the right-hand operand, ignoring the rest of the regular expression. The thread accepts a string only if both sub-threads accept (Figure 3.1). Multistage pipelines are easily specified as `(a @ (b @ (c @ ...)))`.

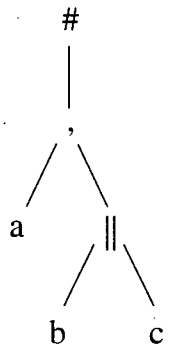


Figure 3.2: Parse tree for the PREMiS expression "a , (b || c)"

3.2 Formal Semantics

In this section, we formally define if a given string is accepted or rejected by a PREMiS expression.

For mathematical convenience, we restrict the PREMiS expression such that the sub-expression controlled by a Kleene star cannot accept the empty string. Regular expressions can be normalized to obey this restriction, as was described in [12].

Given a PREMiS expression, there exists a unique parse tree that represents it. The parse tree consists of nodes and edges connecting these nodes. Each internal node is labeled by one of the possible operators (choice "`||`", Kleene star "`*`", pipelining "`@`", concatenation "`,`"). The leaves are the letters of the alphabet. Figure 3.2 shows the parse tree for the PREMiS expression "`a , (b || c)`". The "#" symbol is used to mark the root of the tree.

We give a unique number to each node by traversing the parse tree in a depth first order. See Figure 3.3. We can then define the function $t(n)$, which returns the operator associated with the node numbered by n. If the node is a leaf, this function returns the symbol T. We also define the function $l(n)$, that when applied to the node n, returns the letter of the alphabet represented by this node if the node is a leaf and is undefined otherwise.

A parse tree configuration can be defined by picking one of the edges of the tree (active edge) and giving a direction for this edge. It is easy to see that there are $2n$ unique configurations for every tree, where n is the number of edges. Figure 3.3 shows all possible

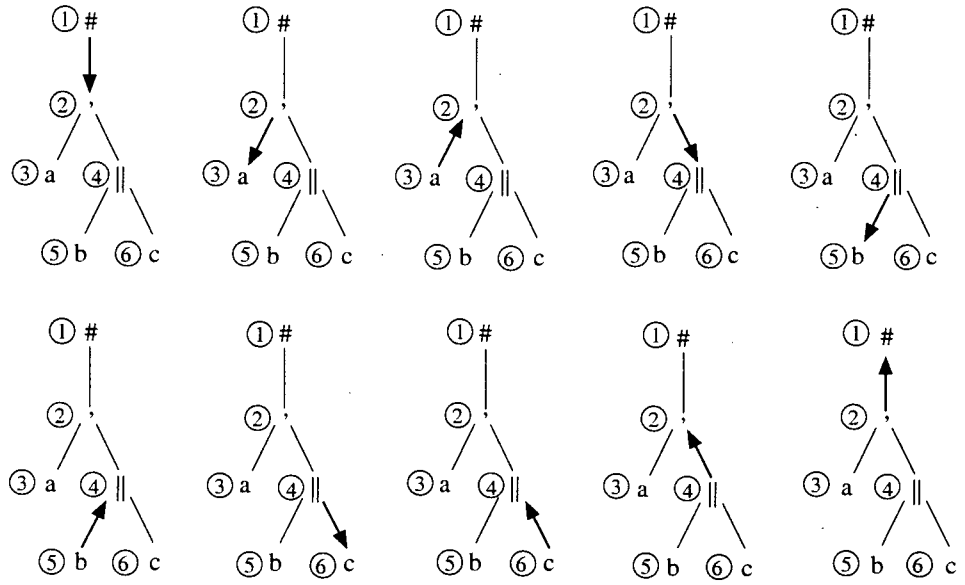


Figure 3.3: All possible configurations for the parse tree for the PREMiS expression "a , (b || c)". Nodes are numbered according to a depth-first traversal. Configurations are also ordered according to a depth-first traversal. In the figure, the order is top row, left-to-right, then bottom row, left-to-right.

configurations for the parse tree representing the PREMiS expression "`a , (b || c)`".

A configuration can be described in two ways. First, we can use the node to where the active edge points and give the direction of the edge according to this node. The following are the four ways to describe this direction: from above, from below (for nodes with one child), from left (for nodes with 2 children), and from right (for nodes with 2 children). Second, we can use the node from where the active edge points and give the direction of the edge according to this node. The following are the four ways to describe this direction: to above, to below (for nodes with one child), to left (for nodes with 2 children), and to right (for nodes with 2 children). The first configuration in figure 3.3 can be described as (node 1, to below) or as (node 2, from above). It is easy to see that the two configurations are equivalent.

We now define a function $V : \mathit{configuration} \times \mathit{string} \longrightarrow \mathcal{B}$, that determines whether a string will be accepted, starting from a configuration:

Definition 1 *Given a configuration c and a string σ, define the function $V(c,\sigma)$ as follows.*

If ε denotes the empty string:

$$V(c,\varepsilon) = \begin{cases} 1 & \text{if } c = (n, from\_below) \text{ and } t(n) = \# \\ 0 & \text{if } c = (n, from\_above) \text{ and } t(n) = T \\ 1 & \text{if } c = (n, from\_above) \text{ and } t(n) = * \\ 1 & \text{if } c = (n, from\_below) \text{ and } t(n) = * \\ V((n,to\_left),\varepsilon) \bigvee V((n,to\_right),\varepsilon) & \text{if } c = (n, from\_above) \text{ and } t(n) = \| \\ V((n,to\_above),\varepsilon) & \text{if } c = (n, from\_left) \text{ and } t(n) = \| \\ V((n,to\_above),\varepsilon) & \text{if } c = (n, from\_right) \text{ and } t(n) = \| \\ V((n,to\_left),\varepsilon) & \text{if } c = (n, from\_above) \text{ and } t(n) = , \\ V((n,to\_right),\varepsilon) & \text{if } c = (n, from\_left) \text{ and } t(n) = , \\ V((n,to\_above),\varepsilon) & \text{if } c = (n, from\_right) \text{ and } t(n) = , \\ V((n,to\_left),\varepsilon) & \text{if } c = (n, from\_above) \text{ and } t(n) = @ \\ V((n,to\_right),\varepsilon) \bigwedge V((n,to\_above),\varepsilon) & \text{if } c = (n, from\_left) \text{ and } t(n) = @ \\ 1 & \text{if } c = (n, from\_right) \text{ and } t(n) = @ \end{cases}$$

If $\sigma = ax$, where a is a letter of the alphabet and x is a string:

$$V(c,ax) = \begin{cases} 0 & \text{if } c = (n, from\_below) \text{ and } t(n) = \# \\ 0 & \text{if } c = (n, from\_above) \text{ and } t(n) = T \\ & \quad \text{and } l(n) \neq a \\ V((n,to\_above),x) & \text{if } c = (n, from\_above) \text{ and } t(n) = T \\ & \quad \text{and } l(n) = a \\ V((n,to\_above),ax) \bigvee & \text{if } c = (n, from\_above) \text{ and } t(n) = * \\ \quad V((n,to\_below),ax) & \\ V((n,to\_above),ax) \bigvee & \text{if } c = (n, from\_below) \text{ and } t(n) = * \\ \quad V((n,to\_below),ax) & \\ V((n,to\_left),ax) \bigvee & \text{if } c = (n, from\_above) \text{ and } t(n) = \| \\ \quad V((n,to\_right),ax) & \\ V((n,to\_above),ax) & \text{if } c = (n, from\_left) \text{ and } t(n) = \| \\ V((n,to\_above),ax) & \text{if } c = (n, from\_right) \text{ and } t(n) = \| \\ V((n,to\_left),ax) & \text{if } c = (n, from\_above) \text{ and } t(n) = , \\ V((n,to\_right),ax) & \text{if } c = (n, from\_left) \text{ and } t(n) = , \\ V((n,to\_above),ax) & \text{if } c = (n, from\_right) \text{ and } t(n) = , \\ V((n,to\_left),ax) & \text{if } c = (n, from\_above) \text{ and } t(n) = @ \\ V((n,to\_right),ax) \bigwedge & \text{if } c = (n, from\_left) \text{ and } t(n) = @ \\ \quad V((n,to\_above),ax) & \\ 1 & \text{if } c = (n, from\_right) \text{ and } t(n) = @ \end{cases}$$

We argue that V is a well-defined function by defining an ordering on pairs of configurations and strings. Pairs with shorter strings come later in the ordering. For pairs with same length strings, the order is determined by the configurations. We order configurations based on the location and direction of the active edge. For any node n, the configuration $(n, from\_above)$ comes before all configurations in which the active edge is below node n, and those configurations come before (n, to_above). Similarly, $(n, from\_left)$ comes before (n, to_right). Intuitively, configurations are ordered according to the order and direction of a

depth-first traversal visiting each edge of the parse tree. Figure 3.3 shows the configurations in this order. It is easy to see that all rules either shorten a string or generate a configuration later in the ordering, except the rule "$(n, from\_below), type(n) = *$". Since we do not allow sub-expressions controlled by a Kleene Star to accept the empty string, every time we reach this configuration, it is guaranteed that the string will be shorter.

3.3 Specification Style Restrictions

We impose three restrictions on our specification style: (1) the sub-expression within a Kleene star cannot accept the empty string, (2) every choice must be deterministic, and (3) a pipeline thread can not start again until it finishes its previous computation. In this section, we describe how to statically detect if a PREMiS sub-expression controlled by a Kleene star accepts the empty string, how to avoid non-determinism, and how to handle pipeline re-entrance.

3.3.1 Empty Strings and Kleene Stars

In [12], a function Λ that detects if a regular expression accepts the empty string is described. If E and F are regular expressions and a is a terminal, Λ is defined as:

$$\Lambda(a) = false \quad \Lambda(E^*) = true$$

$$\Lambda(E \| F) = \Lambda(E) \vee \Lambda(F) \quad \Lambda(E, F) = \Lambda(E) \wedge \Lambda(F)$$

In order to handle the pipeline operator, we take the parse tree for the PREMiS expression and delete the right-hand-operand edges of all pipeline operators, resulting in several, disjoint (sub)-trees. We then remove all pipeline operators by connecting its left-hand-operand to the immediately preceding operator. For each sub-expression controlled by a Kleene star, in all trees, we apply the function Λ. If the result is *true* for any of the sub-expressions, then this expression is not a valid PREMiS expression.

Also in [12], a function to normalize regular expressions to avoid this case is presented. We were not able to find an easy way to modify this function to handle the pipeline

operator, and we are not even sure that such a function exists. In practice, this restriction has not been a problem for us.

3.3.2 Non-determinism

Non-determinism may occur when the choice operator or the Kleene star operator is used. The example below shows a PREMiS expression where it is not possible to know which sub-expression is being matched if the first letter of the string being parsed is an a:

```
(a , b) || (a , c)
```

The following example shows how a Kleene star may generate non-determinism:

```
a* , (a , b)
```

If an a is the first letter of the string being parsed, then it may represent the a in the sub-expression a* or the a in the sub-expression (a , b).

In order to be able to automatically generate efficient monitors, we do not allow non-determinism to occur in our specification style.

To avoid non-determinism, we impose the restriction that whenever a configuration generates two possible next configurations (except for the pipeline operator), one of these configurations must not accept the string consisting of the first letter of the string being parsed. The configurations that may cause non-determinism are: active edge pointing to a choice operator from above, or active edge pointing to a Kleene star.

3.3.3 Pipeline Re-entrance

We define pipeline re-entrance to be when the right-hand sub-expression of a pipeline operator has not finished yet and a new activation occurs. In the example below, pipeline re-entrance can not be avoided (for any valid string with length greater than two):

```
p -> (a @ (b , c))*;
```

In many practical cases, it is desirable to be able to write such expressions. A good example is the AMBA AHB slave, where the pipelined response for a request can take an unlimited amount of time. The following example is based on the AMBA AHB slave:

```
slave -> (idle || transfer)*;
idle -> !HREADY;
transfer -> (HREADY) @ response;
response -> (!HREADY & a_okay)* , (HREADY & a_okay);
```

At first glance, re-entrance may occur, since the response phase may take an unlimited amount of time to finish: `(!HREADY & a_okay)*`. However, since a new request must wait until the previous response is finished (the synchronization is done using the `HREADY` signal), it will never occur.

We do not impose any pipeline re-entrance restriction statically on the parse tree, but we enforce this rule dynamically in the monitor circuit. Thus, we obtain a higher degree of freedom during the specification writing, but we still enforce this rule during the execution of the monitor.

Chapter 4

Translation into Monitor Circuits

4.1 Translation Algorithm

The translation process starts by macro-expanding all productions, since the productions cannot be recursive, resulting in a single (extended) regular expression for the monitor. In theory, this expansion can produce an exponential size blow-up, but in practice, this is often not a problem. The translation from an extended regular expression to circuits can best be understood as recursively building a circuit for each sub-expression, so the structure of the circuit exactly matches the structure of the regular expression. The circuit passes activation signals from sub-circuit to sub-circuit, corresponding to possible parses of the input string by the regular expression. We will elaborate on this construction below.

Our translation is similar to previous work in efficiently converting regular expressions into circuits [14, 12]. The key differences of our algorithm are building a monitor circuit, rather than a recognizer circuit, handling storage variables, and handling pipelining.

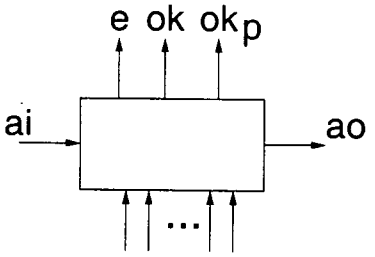
The first difference is that we are interested in monitoring the on-going behavior of an interface, rather than recognizing a regular language, which was the focus of previous work. A recognizer asserts its "OK" output only when the input sequence is a string in the language of the regular expression. A monitor, on the other hand, asserts its OK output as long as the sequence seen so far has not done anything not permitted by the regular

expression. Accordingly, our logic that tracks the correspondence between the interface and the regular expression (the activation signals) is essentially the same as previous work, but the logic to generate the OK signal is completely different.

Pipelining is the most difficult difference. Intuitively, we will create a single thread for each pipeline stage, and the circuit for each thread behaves roughly like previous translations of regular expressions into circuits. The monitor is satisfied only if all active threads are satisfied. Additional bookkeeping is required to track the exact status of each thread.

More precisely, take the (macro-expanded) parse tree for the monitor's regular expression and delete the right-hand-operand edges of all pipeline operators, resulting in several, disjoint parse (sub-)trees. Our restrictions on the specifications (deterministic choice and single thread per pipeline stage) guarantee that each sub-tree will support exactly one thread. Each thread i will generate a thread enable output $tenable_i$ and a thread OK output tok_i. The monitor is satisfied as long as for all threads, $tenable_i \Rightarrow tok_i$. (We use $\Rightarrow$ to denote logical implication.)

Each regular (sub-)expression is converted into a circuit that can read all storage variables and interface wires. The circuit also has an activate-in input a_i, an activate-out output a_o, a circuit-enabled output e, an OK output ok, and an "OK-plus" output ok_p:



Intuitively, the activate signals indicate where a thread is in the regular expression, the enabled signal e indicates if this sub-circuit is enabled (is trying to match the interface signals), and the OK signal indicates that the sub-circuit is enabled and agrees with the current values on the interface wires. The OK-plus signal is a technical detail needed to handle the possibility of recognizing the empty string with a Kleene star; intuitively, it indicates that the sub-circuit is OK at this point even if all stars (zero or more repetitions)

became pluses (one or more repetitions).

Given an extended regular expression, the circuit is build inductively as described in sections 4.1.1– 4.1.7.

4.1.1 Base Case

If the expression is a primitive expression, build the combinational logic to evaluate the Boolean formula for the primitive expression. Let f denote the output of this formula. The enable output e is equal to the activate input a_i. Both ok and ok_p are set to $a_i \wedge f$. The activate-out signal is $a_i \wedge f$ delayed by one clock signal (one flip-flop in the circuit).

$$\begin{aligned} e &= a_i \\ ok &= a_i \wedge f \\ ok_p &= a_i \wedge f \\ a_o &= delay(a_i \wedge f) \end{aligned}$$

4.1.2 Choice Operator

If X and Y are regular expressions with corresponding circuit translations, then build the circuit for $X \mid\mid Y$ from the circuits for X and Y as follows: (Denote the signals for X's circuit with $[X]$, similar for Y. See Figure 4.1.)

$$\begin{aligned} a_i[X] &= a_i \\ a_i[Y] &= a_i \\ e &= e[X] \vee e[Y] \\ ok &= ok[X] \vee ok[Y] \\ ok_p &= ok_p[X] \vee ok_p[Y] \end{aligned}$$

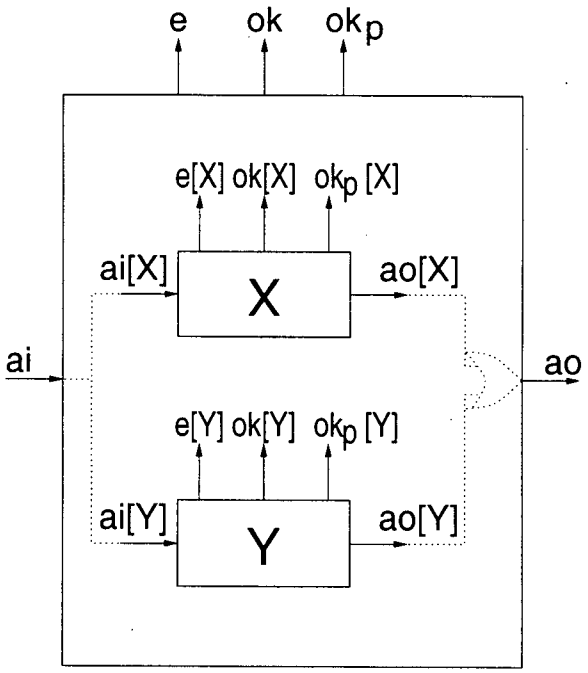


Figure 4.1: Circuits are built recursively from the circuits for their sub-expressions. The dotted lines show the construction for the activation signals for the choice operator $X||Y$.

$$a_o = a_o[X] \vee a_o[Y]$$

4.1.3 Sequence Operator

Similarly, build the circuit for X , Y as follows:

$$\begin{aligned}
a_i[X] &= a_i \\
a_i[Y] &= a_o[X] \\
e &= e[X] \vee e[Y] \\
ok &= (e[X] \Rightarrow ok[X]) \wedge (e[Y] \Rightarrow ok[Y]) \wedge \\
&\quad (ok[X] \vee ok[Y]) \\
ok_p &= ok_p[X] \vee ok_p[Y] \\
a_o &= a_o[Y]
\end{aligned}$$

The first two equations connect the activation signals so that X goes first, and then activates Y in sequence. The e and ok_p constructions are intuitive — the circuit as a whole is enabled or "OK-plus" if either sub-circuit is enabled or OK-plus. The extra clauses for ok are needed because X or Y might consist of a Kleene star, and the construction for the Kleene star is always OK as soon as the circuit is activated, regardless of the values on the interface wires (because the star allows matching zero copies of the repeating expression). The extra clauses prevent these empty-match OK signals from propagating erroneously.

4.1.4 Pipeline Operator

For X @ Y, all of X's signals are connected to the corresponding signals for the circuit for (X @ Y), since the current thread ignores Y. In addition, a new thread for Y gets activated when X completes:

$$
\begin{aligned}
a_i[X] &= a_i \\
e &= e[X] \\
ok &= ok[X] \\
ok_p &= ok_p[X] \\
a_o &= a_o[X] \\
a_i[Y] &= a_o[X]
\end{aligned}
$$

4.1.5 Kleene Star

Build the circuit for $X*$ as follows:

$$
\begin{aligned}
a_i[X] &= a_i \vee a_o[X] \\
e &= e[X] \vee a_o[X] \\
ok &= ok[X] \vee a_i \vee a_o[X] \\
ok_p &= ok_p[X] \\
a_o &= !ok_p \wedge (a_i \vee a_o[X])
\end{aligned}
$$

Because of the repetition, the circuit self-activates, so the $a_o[X]$ signal appears in several formulas. The Kleene star accepts the empty string, so the a_i signal appears combinationally in the equations for ok and a_o (as well as indirectly in e for the first cycle of X's activation). Here, we see the use of the ok_p signal: the activate output is disabled if X is truly matching the interface (rather than vacuously matching because of a Kleene star). The deterministic choice restriction prevents the case where a_o should be true at the same time as $ok_p[X]$.

4.1.6 Storage Variables

Storages variables are translated into memory elements that can be read or written at any time during the parsing of the input string. The memory elements can be read in the same way as the inputs to the monitor can be read. Writes occur through actions.

Each action is associated with a node of the parse tree. Actions are activated when the a_o signal of the sub-expression to which the action is associated is asserted. We do not allow actions to be connected to a node representing the pipeline operator, because the semantics of that is not well defined.

The changes to the storage variables can be seen at the same time that the action is activated. Since many actions can be active at the same time, it is possible for a storage variable to be set more than once at the same time. In this case, we use the order imposed on the parse tree nodes (See Section 3.2.) to define which assignment to a storage variable will take place. If two or more conflicting assignments can take place at the same time, the one associated with the action connected to the node with the highest number will be executed.

4.1.7 Monitor Circuit

In Section 3.3, we imposed three restrictions to allow efficiently building a monitor: (1) no empty strings within Kleene stars, (2) deterministic choice, and (3) one thread per pipeline stage. The first two are imposed statically on the regular expression. To enforce the third restriction, we augment our monitor to generate a pipeline-violation error. Intuitively, for each pipeline operator X @ Y, trying to activate Y when it is already running generates a pipeline-violation error. A complication, however, is that the signals from the already running thread and the new activation can interfere. The easiest way around this complication is to generate three versions of every signal in the construction described in sections 4.1.1– 4.1.7: the regular version as described already; a primed version, which ignores any new activation; and a double-primed version, which tracks only the first cycle of a new activation. The formulas for the primed signals are identical to the ones above, with primed signal names

replacing unprimed signal names, except for the initial thread activate signal $a_i'[Y] = \text{false}$ instead of $a_i[Y] = a_o[X]$ (cf. Section 4.1.4), and at all base case circuits (cf. Section 4.1.1), the outputs a_o' are driven by the same flip-flop (unprimed) that drives the corresponding a_o. By disabling $a_i'[Y]$, the primed version of the thread ignores new activations, but indicates if the thread is still enabled. The formulas for the double-primed signals are also identical to the regular signals, except with double-primed signal names, and in the base case, a_o'' is always false rather than driven by the flip-flop. In other words, the double-primed version sees only the initial activation of a thread, and not any subsequent cycles. (Considerable redundancy could be eliminated, but this construction is easy to explain and implement.) A pipeline-violation error occurs whenever a new activation occurs while the thread is still running: $a_i[Y] \wedge ok_p'[Y]$. The thread enable and ok signals are defined as $tenable = e[Y]$ and $tok = (e'[Y] \Rightarrow ok'[Y]) \wedge (e''[Y] \Rightarrow ok''[Y])$. Intuitively, if an existing activation is enabled, it must be ok, and if a new activation is enabled it must also be ok.

The top-level monitor *ok* signal is represented by $\bigwedge_{i=1}^{n}(tenable_i \Rightarrow tok_i)$ where n is the number of threads in the circuit.

4.2 Complexity Analysis

In this section, we present the complexity analysis of the translation algorithm presented in the previous section. In order to facilitate the analysis, we will divide it into four parts: productions flattening, PREMiS operators, storage variables, and the top-level monitor circuitry.

Productions Flattening: the translation process starts by macro-expanding all productions and all *multiple concatenation* "^" operators. A multiple concatenation operator allows a sub-expression to be automatically concatenated n times, where n is a constant. The sub-expression controlled by a multiple concatenation operator is replaced by the same expression, concatenated n times.

In theory, macro-expanding productions and multiple concatenation operators can

produce an exponential size blow-up. For example, the specification below:

```
p1 -> p2, p2;
p2 -> p3, p3;
    ...
pn -> b, b;
```

produces the PREMiS expression `(b, b, ..., b, b)` which contains 2^n operators, where n is the number of operators in the original specification. The multiple concatenation operator produces a pseudo-polynomial blow-up. The length of the expanded expression is linear in the *value* of n, but exponential in the length of the original expression, because the value of a number is exponential in the number of digits needed to represent it. For example, the PREMiS expression `(a,b)^3` is replaced by `(a,b),(a,b),(a,b)`.

PREMiS Operators: for each of the operators in sections 4.1.1– 4.1.5, a constant amount of circuitry is generated. It is also easy to see that the circuitry generation takes a constant time for each operator.

Storage Variables: storage variables do not add any size complexity to the circuitry generated, except for a storage element for each variable. The activation signals for the actions are the same signals constructed during the translation of the PREMiS operators. The logic to resolve multiple assignments is a simple priority circuit with size linear in the number of concurrent assignments.

Top-Level Monitor Circuitry: the circuitry for the *tenable* and *tok* signals and for the pipeline re-entrance detection can be accounted for in the cost of the pipeline operator. For each pipeline operator, we replicate its circuitry twice. These circuits differ only on the base case and on the thread activation signals (cf. section 4.1.7), making them the same size as the original circuitry.

The top-level monitor circuitry $\bigwedge_{i=1}^{n}(tenable_i \Rightarrow tok_i)$ depends only on the number n of pipeline operators in the PREMiS expression. Thus, this construction is linear in size and time with respect to the number of pipeline operators present in the PREMiS expression.

Since all the steps necessary to build a monitor circuit from a macro-expanded PREMiS expression generate a constant amount of circuitry, the translation complexity is linear with respect to the number of operators and storage variables in the expression. In theory, macro-expanding the productions can produce an exponential size blow-up, but in practice this has not been a problem.

Chapter 5

Examples

This chapter describes in detail the specification of the AMBA AHB master and slave, and the Sonics OCP master and slave. The results obtained from the specification and the translation into monitor circuits are also shown.

5.1 ARM AMBA AHB Bus

The ARM Advanced Microcontroller Bus Architecture (AMBA) is a set of three System-on-Chip buses: Advanced Peripheral Bus (APB), Advanced System Bus (ASB), and Advanced High-performance Bus (AHB). APB is a simple, low-performance bus designed for low-bandwidth peripherals like keypads. It can be connected to AHB or ASB through a bridge. ASB is a high-performance bus which can be used to connect high-bandwidth modules like processors and on-chip memories. AHB is a new generation bus that supports pipelined operations for improved performance. Figure 5.1 shows a possible configuration of a system using AHB as the main bus.

The AMBA AHB specification describes the following components:

- **Master**: able to initiate read or write transfers.
- **Slave**: responds to transfers indicating to the master the success, failure or waiting of the data.

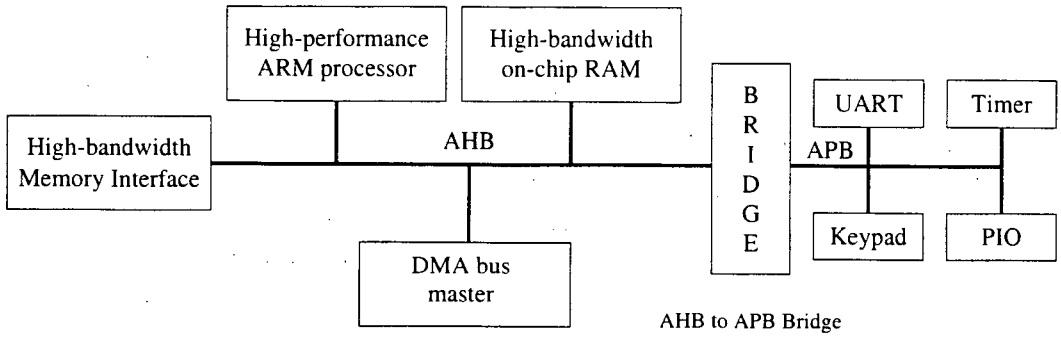


Figure 5.1: System using AHB as the main bus and APB to connect the peripherals. Figure extracted from [2].

- **Arbiter**: responsible for granting the masters access to the bus. Only one master at a time is allowed to initiate transfers. The specification does not describe any arbitration algorithm, allowing the designer to choose it according to the application requirements.
- **Decoder**: selects the active slave given the transfer address set by the active master.

Figure 5.2 shows an example of an AHB configuration.

5.1.1 Specification

Slave

The specification starts by declaring the interface wires that are the inputs (`HTRANS`, `HREADY`, `HSEL`) and outputs (`HRESP`) of the slave. The data and address signals are not included because they do not affect the slave behavior.

```
input HTRANS[1:0], HREADY, HSEL;
output HRESP[1:0];
```

Two internal variables are used. `i_split` keeps track of which masters have been split by this slave. `i_master` keeps track of the number of the master performing the transfer.

```
internal i_split[15:0];
```

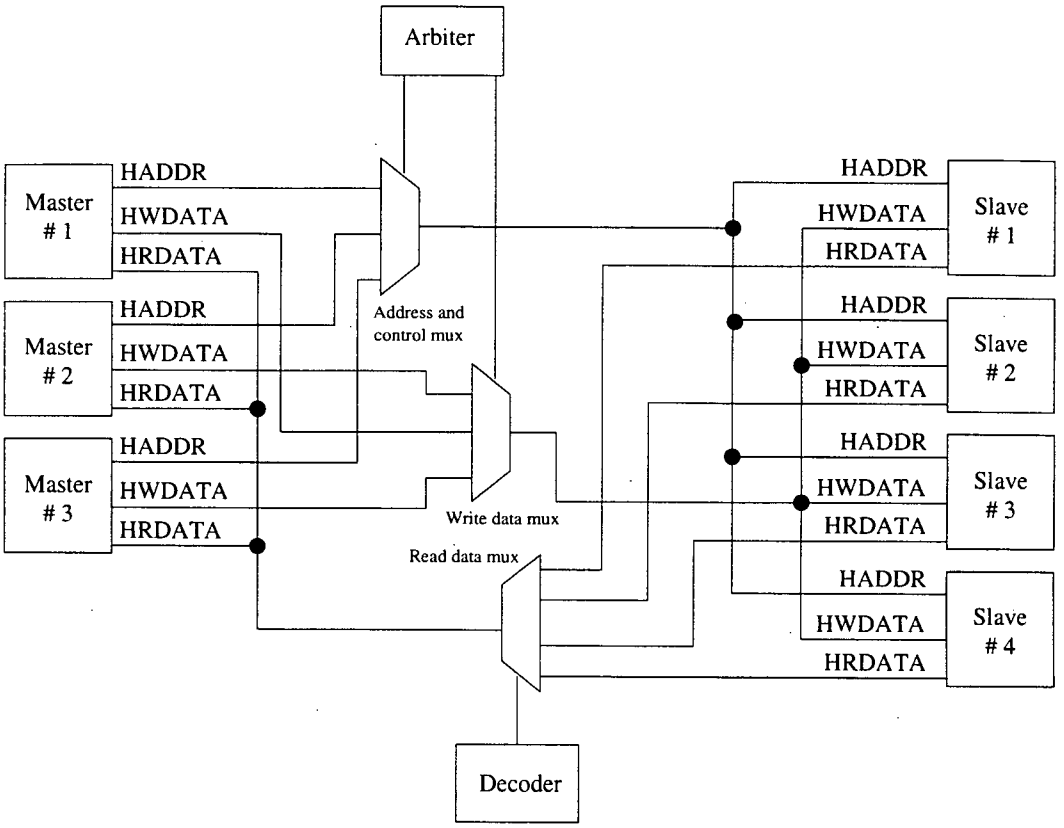


Figure 5.2: AHB configuration consisting of three masters, four slaves, the arbiter, and the decoder. Figure extracted from [2].

```
internal i_master[3:0];
```

The following are abbreviations for the four different transfers a slave can perform and for the four different responses it can issue for every transfer.

```
/*
 * Transfer type
 */
define a_idle   = !HTRANS[0] & !HTRANS[1];
define a_busy   =  HTRANS[0] & !HTRANS[1];
define a_nonseq = !HTRANS[0] &  HTRANS[1];
define a_seq    =  HTRANS[0] &  HTRANS[1];

/*
 * Slave responses
 */
define a_okay  = !HRESP[0] & !HRESP[1];
define a_error =  HRESP[0] & !HRESP[1];
define a_retry = !HRESP[0] &  HRESP[1];
define a_split =  HRESP[0] &  HRESP[1];
```

The full slave specification can be split into seventeen smaller specifications. `slave` specifies the behavior of the slave regarding transfers and responses. The other specifications assure that the slave can only complete ("unsplit") a transaction that had been previously split. The following declaration declares these seventeen monitors which run in parallel.

```
monitor slave,
        unsplit_0, unsplit_1, unsplit_2, unsplit_3,
        unsplit_4, unsplit_5, unsplit_6, unsplit_7,
        unsplit_8, unsplit_9, unsplit_10, unsplit_11,
        unsplit_12, unsplit_13, unsplit_14, unsplit_15;
```

The behavior of the slave can be defined as being idle or performing a transfer.

```
slave -> (idle || transfer)*;
```

If it is in an idle state, it is because it is not selected (another slave is performing a transfer) or it has been selected but the last transfer hasn't finished yet, indicated by HREADY low.

```
idle -> (!HSEL) || (HSEL & !HREADY);
```

A transfer can be an idle transfer, or a busy transfer, or a non-sequential transfer, or a sequential transfer.

```
transfer -> idle_transfer    ||
            busy_transfer    ||
            nonseq_transfer  ||
            seq_transfer;
```

All the four transfers types consist of an address phase (indicated by HSEL & HREADY and the type of the transfer) followed by a response phase. An idle or busy transfer must be followed immediately with an OK response. The sequential and non-sequential transfers may have wait states inserted by the slave during the response phase. The address phase and response phase are pipelined, which means that the slave may be responding to a transfer and at the same time reading the address and control signals for the next transfer, thus the use of the pipeline operator "@". For non-sequential and sequential transfers, the monitor stores the number of the master performing the transfer. This information will be used later if the slave responds with a split response.

```
idle_transfer -> (a_idle & HSEL & HREADY) @ okay_response;
busy_transfer -> (a_busy & HSEL & HREADY) @ okay_response;
nonseq_transfer -> ((a_nonseq & HSEL & HREADY)
                    {i_master <- HMASTER;}) @ response;
seq_transfer -> ((a_seq & HSEL & HREADY)
                 {i_master <- HMASTER;}) @ response;
```

A response may have any number of wait states inserted by the slave before one of the four response types is given.

```
response ->
    wait_state* ,
    (okay_response || error_response ||
     split_response || retry_response);
```

During a wait state the slave must keep HREADY low, and it must set the response wires to a_okay.

```
wait_state -> !HREADY & a_okay;
```

The OK response takes one cycle and is indicated by HREADY high and the response wires set to a_okay. The other three responses take two cycles, the first with HREADY low and the second with HREADY high. The response wires can not have their values changed during these two cycles. For a split response, the monitor uses the internal variable i_split to keep track of the master the slave has split.

```
okay_response  ->  HREADY & a_okay;
error_response -> (!HREADY & a_error) , (HREADY & a_error);
retry_response -> (!HREADY & a_retry) , (HREADY & a_retry);
split_response -> ((!HREADY & a_split) , (HREADY & a_split))
                    {i_split[i_master] <- 1;};
```

If a slave unsplits a master, then it must have given a split response to this master before.

```
unsplit_0 -> (!HSPLIT[0] ||
              (HSPLIT[0] & i_split[0]
               {i_split[0] <- 0;}))*;
unsplit_1 -> (!HSPLIT[1] ||
              (HSPLIT[1] & i_split[1]
               {i_split[1] <- 0;}))*;
```

```
unsplit_2 -> (!HSPLIT[2] ||
              (HSPLIT[2] & i_split[2]
               {i_split[2] <- 0;}))*;
unsplit_3 -> (!HSPLIT[3] ||
              (HSPLIT[3] & i_split[3]
               {i_split[3] <- 0;}))*;
unsplit_4 -> (!HSPLIT[4] ||
              (HSPLIT[4] & i_split[4]
               {i_split[4] <- 0;}))*;
unsplit_5 -> (!HSPLIT[5] ||
              (HSPLIT[5] & i_split[5]
               {i_split[5] <- 0;}))*;
unsplit_6 -> (!HSPLIT[6] ||
              (HSPLIT[6] & i_split[6]
               {i_split[6] <- 0;}))*;
unsplit_7 -> (!HSPLIT[7] ||
              (HSPLIT[7] & i_split[7]
               {i_split[7] <- 0;}))*;
unsplit_8 -> (!HSPLIT[8] ||
              (HSPLIT[8] & i_split[8]
               {i_split[8] <- 0;}))*;
unsplit_9 -> (!HSPLIT[9] ||
              (HSPLIT[9] & i_split[9]
               {i_split[9] <- 0;}))*;
unsplit_10 -> (!HSPLIT[10] ||
               (HSPLIT[10] & i_split[10]
                {i_split[10] <- 0;}))*;
unsplit_11 -> (!HSPLIT[11] ||
```

```
                (HSPLIT[11] & i_split[11]
                 {i_split[11] <- 0;}))*;
unsplit_12 -> (!HSPLIT[12] ||
                (HSPLIT[12] & i_split[12]
                 {i_split[12] <- 0;}))*;
unsplit_13 -> (!HSPLIT[13] ||
                (HSPLIT[13] & i_split[13]
                 {i_split[13] <- 0;}))*;
unsplit_14 -> (!HSPLIT[14] ||
                (HSPLIT[14] & i_split[14]
                 {i_split[14] <- 0;}))*;
unsplit_15 -> (!HSPLIT[15] ||
                (HSPLIT[15] & i_split[15]
                 {i_split[15] <- 0;}))*;
```

Master

The specification starts by declaring the interface wires that are the inputs and outputs of the master.

```
input HGRANT;
input HREADY;
input HCLK;
input HRESP[1:0];
input HRDATA[31:0];
input HRESETn;

output HBUSREQ;
output HLOCK;
output HTRANS[1:0];
```

```
output HADDR[31:0];
output HWRITE;
output HSIZE[2:0];
output HBURST[2:0];
output HPROT[3:0];
output HWDATA[31:0];
```

The following are abbreviations for the four different types of transfers a master can perform, the eight possible burst types, and the four response types a slave can issue.

```
/*
 * Transfer type
 */
define IDLE   = !HTRANS[1] & !HTRANS[0];
define BUSY   = !HTRANS[1] &  HTRANS[0];
define NONSEQ =  HTRANS[1] & !HTRANS[0];
define SEQ    =  HTRANS[1] &  HTRANS[0];

/*
 * Burst type
 */
define SINGLE = !HBURST[2] & !HBURST[1] & !HBURST[0];
define INCR   = !HBURST[2] & !HBURST[1] &  HBURST[0];
define WRAP4  = !HBURST[2] &  HBURST[1] & !HBURST[0];
define INCR4  = !HBURST[2] &  HBURST[1] &  HBURST[0];
define WRAP8  =  HBURST[2] & !HBURST[1] & !HBURST[0];
define INCR8  =  HBURST[2] & !HBURST[1] &  HBURST[0];
define WRAP16 =  HBURST[2] &  HBURST[1] & !HBURST[0];
define INCR16 =  HBURST[2] &  HBURST[1] &  HBURST[0];
```

```
/*
 * Response type
 */
define OKAY  = !HRESP[1] & !HRESP[0];
define ERROR = !HRESP[1] &  HRESP[0];
define RETRY =  HRESP[1] & !HRESP[0];
define SPLIT =  HRESP[1] &  HRESP[0];
```

For convenience, we also use the abbreviations shown below. ARESP indicates if the response given by the slave was an abnormal response (split, retry, or error). MIDDLE is asserted if the master is performing a defined-length burst that hasn't finished yet. CHOLD and AHOLD assure that the current control signals and address signals have not changed since they were last stored. DHOLD indicates that if a master is performing a write transfer, then the data must not change. HOLD enforces CHOLD, AHOLD, and DHOLD at the same time.

```
define ARESP = i_sr | i_error;
define MIDDLE = i_count != i_burst;
define CHOLD = (i_size == HSIZE) & (i_prot == HPROT) &
               (i_write == HWRITE);
define AHOLD = (i_addr == HADDR);
define DHOLD = (i_wdata == HWDATA) | !i_write;
define HOLD  = CHOLD & AHOLD & DHOLD;
```

The following are the internal variables used by the monitor. i_sr indicates that the transfer response was a split or retry response. i_error indicates that the transfer response was an error response. i_count counts the number of transfers already performed in the current burst. i_burst is the length of the current burst being performed by the master. i_incr indicates that the current transfer is part of an undefined-length incremental burst. i_write, i_size, i_prot, i_addr, and i_wdata are used to store the control signals, address signals, and write data signals.

```
internal i_sr = 0;
internal i_error = 0;
internal i_count[3:0] = 0;
internal i_burst[3:0] = 0;
internal i_incr = 0;
internal i_write = 0;
internal i_size[2:0] = 0;
internal i_prot[3:0] = 0;
internal i_addr[31:0] = 0;
internal i_wdata[31:0] = 0;
```

The master can be granted at any time by the arbiter, and for every transfer the master must receive a grant from the arbiter. If the master is granted, then it must perform a transfer in the next cycle. The process of looking at the grant and performing a transfer is pipelined in the sense that at the same time the transfer is in progress, the master must also watch the grant signals to find out if it will have to perform a transfer in the next cycle or not.

```
master ->
    (not_ready || not_granted || (granted @ transfer))*;
```

The master only needs to look at the grant signal if HREADY is high. If HREADY is low, any change to the grant signal will not affect the master.

```
not_ready ->
    !HREADY;
```

If HREADY is high and the grant signal HGRANT is low, the master will not be able to perform a transfer in the next cycle. Since the master has lost ownership of the address and control bus, the internal variables that keep track of this type of information are reinitialized.

```
not_granted ->
    (!HGRANT & HREADY)
```

```
{i_count <- 0; i_burst <- 0; i_write <- 0;
 i_size <- 0; i_prot <- 0; i_addr <- 0;
 i_wdata <- 0;};
```

A grant is indicated by HREADY being high and the grant signal HGRANT being high at the same time.

```
granted ->
    HGRANT & HREADY;
```

There are four types of transfers a master can perform: an idle transfer, a busy transfer, a non-sequential transfer, or a sequential transfer.

```
transfer ->
    idle_transfer    ||
    busy_transfer    ||
    nonseq_transfer  ||
    seq_transfer;
```

Every transfer may be immediately accepted, indicated by HREADY high, or it may be delayed by the slave, which drives HREADY low for a certain number of cycles.

```
idle_transfer ->
    immediate_idle_transfer ||
    wait_idle_transfer;
```

If the idle transfer is immediately accepted, the internal variables are reinitialized. Idle transfers can not occur in the middle of a burst, unless an abnormal response is received. This production does not handle the mandatory idle transfer after an abnormal response, as indicated by ARESP being low. The production abnormal_response handles the case where an abnormal response is received from the slave.

```
immediate_idle_transfer ->
```

```
        (HREADY & IDLE & !MIDDLE & !ARESP)
        {i_count <- 0; i_burst <- 0; i_write <- 0;
         i_size <- 0; i_prot <- 0; i_addr <- 0;
         i_wdata <- 0;};
```

If the idle transfer is being delayed by the slave, indicated by HREADY low, the master may decide to change this transfer to a non-sequential transfer.

```
    wait_idle_transfer ->
         (!HREADY & IDLE & !MIDDLE & !ARESP)+ ,
         (end_idle_transfer || nonseq_transfer);


end_idle_transfer ->
        (HREADY & IDLE & !MIDDLE)
        {i_count <- 0; i_burst <- 0; i_write <- 0;
         i_size <- 0; i_prot <- 0; i_addr <- 0;
         i_wdata <- 0;};
```

A busy transfer may only happen in the middle of a defined-length burst, indicated by MIDDLE, or during an undefined-length incremental burst, indicated by i_incr. The master may decide to change the transfer type to any other type if HREADY is low. Since the busy transfer is performed during a burst, it must not change the control signal values, which were set during the first transfer of the burst. CHOLD guarantees that the control signals remain unchanged. For any transfer, if a split or retry response is received the master must immediately perform an abnormal termination. If a error response is received, the master has the option to perform an abnormal termination, or continue with the current transfer.

```
    busy_transfer ->
         (HREADY & BUSY & CHOLD & (MIDDLE | i_incr)) ||
         ((!HREADY & BUSY & CHOLD & (MIDDLE | i_incr) & !i_sr)+ ,
          ((HREADY & BUSY & CHOLD & (MIDDLE | i_incr) & !i_sr) ||
```

```
    idle_transfer     ||
    seq_transfer      ||
    nonseq_transfer   ||
    abnormal_termination));
```

Sequential transfers can only occur after an initial non-sequential transfer is performed, this condition is indicated by (MIDDLE | i_incr). For defined-length bursts, the master can not perform a sequential transfer if it has already performed the number of transfers defined in the first non-sequential transfer. i_count keeps track of the number of transfers performed in the current burst. MIDDLE is true if i_count is not equal to the burst length (i_burst). The monitor does not enforce the property that a burst must not cross a 1KB boundary, and it also does not enforce any rule regarding the sequence of values allowed in the address signals. Enforcing these rules would require a lot of effort because the current version of the monitor specification language supports only two basic math operators: addition and subtraction.

```
seq_transfer ->
    immediate_seq_transfer ||
    wait_seq_transfer;

immediate_seq_transfer ->
    (((HREADY & SEQ & !i_sr & CHOLD & (MIDDLE | i_incr))
     {i_count <- i_count + 1;}) @ response);

wait_seq_transfer ->
    ((((!HREADY & SEQ & !i_sr & CHOLD & (MIDDLE | i_incr))
     {i_addr <- HADDR; i_wdata <- HWDATA;}) ,
     (!HREADY & SEQ & !i_sr & HOLD)* ,
     ((((HREADY & SEQ  & !i_sr & HOLD)
      {i_count <- i_count + 1;}) @ response) ||
```

```
        abnormal_termination));
```

Every burst must start with a non-sequential transfer. There are eight types of bursts, as shown below.

```
nonseq_transfer ->
    single ||
    incr   ||
    wrap4  ||
    incr4  ||
    wrap8  ||
    incr8  ||
    wrap16 ||
    incr16;

single ->
    immediate_single ||
    wait_single;

incr ->
    immediate_incr ||
    wait_incr;

wrap4 ->
    immediate_wrap4 ||
    wait_wrap4;

incr4 ->
    immediate_incr4 ||
    wait_incr4;
```

```
wrap8 ->
    immediate_wrap8 ||
    wait_wrap8;

incr8 ->
    immediate_incr8 ||
    wait_incr8;

wrap16 ->
    immediate_wrap16 ||
    wait_wrap16;

incr16 ->
    immediate_incr16 ||
    wait_incr16;
```

A single burst consists of only one transfer. For any non-sequential transfer, if the slave delays the transfer by asserting HREADY low, the master must maintain the control and address signals constant until HREADY becomes high. If the master is performing a write transfer, then the write data must also not change. Non-sequential transfers can not happen in the middle of a burst, indicated by MIDDLE.

```
immediate_single ->
    (((HREADY & NONSEQ & SINGLE & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 0;}) @ response);

wait_single ->
    ((!HREADY & NONSEQ & SINGLE & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 0;
```

```
        i_size <- HSIZE; i_prot <- HPROT;
        i_write <- HWRITE; i_addr <- HADDR; i_wdata <- HWDATA;} ,
       (!HREADY & NONSEQ & SINGLE & !i_sr & HOLD)* ,
       ((HREADY & NONSEQ & SINGLE & !i_sr & HOLD) @ response) ||
        abnormal_termination);
```

If a non-sequential undefined incremental burst is performed, the internal variable i_incr is set.

```
immediate_incr ->
    (((HREADY & NONSEQ & INCR & !i_sr & !MIDDLE)
      {i_incr <- 1; i_count <- 0; i_burst <- 0;
       i_size <- HSIZE; i_prot <- HPROT;
       i_write <- HWRITE;})
      @ response);


wait_incr ->
    ((!HREADY & NONSEQ & INCR & !i_sr & !MIDDLE)
     {i_incr <- 1; i_count <- 0; i_burst <- 0;
      i_size <- HSIZE; i_prot <- HPROT;
      i_write <- HWRITE; i_addr <- HADDR; i_wdata <- HWDATA;} ,
     (!HREADY & NONSEQ & INCR & !i_sr & HOLD)* ,
     ((HREADY & NONSEQ & INCR & !i_sr & HOLD) @ response) ||
        abnormal_termination);
```

For defined-length bursts, i_burst indicates the length of the burst, and i_count indicates how many transfers have been performed in the current burst. Since a defined-length non-sequential transfer is always the first transfer in any defined-length burst, i_count is initialized to zero and i_burst is initialized to the length of the burst.

```
immediate_wrap4 ->
```

```
    (((HREADY & NONSEQ & WRAP4 & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 4;
      i_size <- HSIZE; i_prot <- HPROT; i_write <- HWRITE;})
    @ response);

wait_wrap4 ->
    ((!HREADY & NONSEQ & WRAP4 & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 4;
      i_size <- HSIZE; i_prot <- HPROT;
      i_write <- HWRITE; i_addr <- HADDR; i_wdata <- HWDATA;} ,
     (!HREADY & NONSEQ & WRAP4 & !i_sr & HOLD)* ,
     ((HREADY & NONSEQ & WRAP4 & !i_sr & HOLD) @ response) ||
        abnormal_termination);

immediate_incr4 ->
    (((HREADY & NONSEQ & INCR4 & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 4;
      i_size <- HSIZE; i_prot <- HPROT; i_write <- HWRITE;})
    @ response);

wait_incr4 ->
    ((!HREADY & NONSEQ & INCR4 & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 4;
      i_size <- HSIZE; i_prot <- HPROT;
      i_write <- HWRITE; i_addr <- HADDR; i_wdata <- HWDATA;} ,
     (!HREADY & NONSEQ & INCR4 & !i_sr & HOLD)* ,
     ((HREADY & NONSEQ & INCR4 & !i_sr & HOLD) @ response) ||
        abnormal_termination);
```

```
immediate_wrap8 ->
    (((HREADY & NONSEQ & WRAP8 & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 8;
      i_size <- HSIZE; i_prot <- HPROT; i_write <- HWRITE;})
    @ response);

wait_wrap8 ->
    ((!HREADY & NONSEQ & WRAP8 & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 8;
      i_size <- HSIZE; i_prot <- HPROT;
      i_write <- HWRITE; i_addr <- HADDR; i_wdata <- HWDATA;} ,
     (!HREADY & NONSEQ & WRAP8 & !i_sr & HOLD)* ,
     ((HREADY & NONSEQ & WRAP8 & !i_sr & HOLD) @ response) ||
        abnormal_termination);

immediate_incr8 ->
    (((HREADY & NONSEQ & INCR8 & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 8;
      i_size <- HSIZE; i_prot <- HPROT; i_write <- HWRITE;})
    @ response);

wait_incr8 ->
    ((!HREADY & NONSEQ & INCR8 & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 8;
      i_size <- HSIZE; i_prot <- HPROT;
      i_write <- HWRITE; i_addr <- HADDR; i_wdata <- HWDATA;} ,
     (!HREADY & NONSEQ & INCR8 & !i_sr & HOLD)* ,
```

```
     ((HREADY & NONSEQ & INCR8 & !i_sr & HOLD) @ response) ||
        abnormal_termination);

immediate_wrap16 ->
    (((HREADY & NONSEQ & WRAP16 & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 16;
      i_size <- HSIZE; i_prot <- HPROT; i_write <- HWRITE;})
    @ response);

wait_wrap16 ->
    ((!HREADY & NONSEQ & WRAP16 & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 16;
      i_size <- HSIZE; i_prot <- HPROT;
      i_write <- HWRITE; i_addr <- HADDR; i_wdata <- HWDATA;} ,
     (!HREADY & NONSEQ & WRAP16 & !i_sr & HOLD)* ,
     ((HREADY & NONSEQ & WRAP16 & !i_sr & HOLD) @ response) ||
        abnormal_termination);

immediate_incr16 ->
    (((HREADY & NONSEQ & INCR16 & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 16;
      i_size <- HSIZE; i_prot <- HPROT; i_write <- HWRITE;})
    @ response);

wait_incr16 ->
    ((!HREADY & NONSEQ & INCR16 & !i_sr & !MIDDLE)
     {i_incr <- 0; i_count <- 0; i_burst <- 16;
      i_size <- HSIZE; i_prot <- HPROT;
```

```
    i_write <- HWRITE; i_addr <- HADDR; i_wdata <- HWDATA;} ,
   (!HREADY & NONSEQ & INCR16 & !i_sr & HOLD)* ,
   ((HREADY & NONSEQ & INCR16 & !i_sr & HOLD) @ response) ||
      abnormal_termination);
```

During an abnormal termination the master must perform an idle transfer. Since this transfer will cancel any burst being performed, the internal variables are reinitialized.

```
abnormal_termination ->
    (HREADY & IDLE & ARESP)
    {i_count <- 0; i_burst <- 0; i_write <- 0;
     i_size <- 0; i_prot <- 0; i_addr <- 0;
     i_wdata <- 0;};
```

A response may have any number of wait states inserted by the slave before one of the four response types is given.

```
response ->
    wait_state* ,
    (ok_response || error_response ||
     split_response || retry_response);
```

During a wait state the slave must keep HREADY low, and it must set the response wires to OKAY.

```
wait_state ->
    !HREADY & OKAY;
```

The OK response takes one cycle and is indicated by HREADY high and the response wires set to OKAY. The other three responses take two cycles, the first with HREADY low and the second with HREADY high. The response wires can not have their values changed during these two cycles. During the first cycle of the three abnormal response types, the respective internal variable is set to indicate to the current transfer that the previous transfer received an error response (i_error), or a retry response (i_sr), or a split response (i_sr).

```
ok_response ->
    HREADY & OKAY;

error_response ->
    (!HREADY & ERROR) {i_error <- 1;} ,
    (HREADY & ERROR) {i_error <- 0;};

split_response ->
    (!HREADY & SPLIT) {i_sr <- 1;} ,
    (HREADY & SPLIT) {i_sr <- 0;};

retry_response ->
    (!HREADY & RETRY) {i_sr <- 1;} ,
    (HREADY & RETRY) {i_sr <- 0;};
```

5.2 Sonics OCP

Sonics Open Core Protocol (OCP) is actually a very broad, parameterized family of core-centric protocols, spanning an enormous range of performance and cost objectives. The interface monitors shown are for a Basic OCP master and slave with only one transaction in-flight at a time.

In Basic OCP, the master presents a command at the same time as the address, as well as the data if the command is a write. These values must be held constant until the slave accepts the command by asserting `SCmdAccept`, which could happen in the same cycle that the command is presented. Writes are posted (no response required once the slave accepts the command), but read commands have a response phase during which the slave sends data back to the master. The slave can insert zero or more wait states by keeping `SResp` set to the null response. The slave terminates the response phase by setting `SResp` to indicate that the data is valid or an error occurred. `SResp` can go non-null in the same cycle as `SCmdAccept`

is asserted, which can be in the same cycle as the master presents a command.

We restrict ourselves to Basic OCP because in more complex OCP configurations, the master may issue a finite number of requests before any response comes back. Modeling this behavior with PREMiS is not straightforward since this implies that the same pipeline stage (wait for response) may be activated again before the computation of a previous activation has finished. The pipeline operator does not allow re-entrance (See Section 3.3.3.), thus it can not be directly used. It is still possible to specify this behavior by writing multiple monitors (See Section A.1.9.) that communicate using storage variables. Each monitor would be a pipeline stage, and the same stage may have more than one monitor associated with it. In this case, the synchronization logic between multiple pipeline stages must be explicitly written using storages variables. We suggest some possible enhancements to PREMiS to address this problem in Chapter 6.

5.2.1 Specification

Slave

The specification starts by declaring the interface wires that are the inputs (`MAddr`, `MCmd`, `MData`) and outputs (`SCmdAccept`, `SResp`, `SData`) of the slave.

```
input MAddr[31:0], MCmd[2:0], MData[31:0];
output SCmdAccept, SResp[1:0], SData[31:0];
```

The following are abbreviations for the three different transfers a slave can perform and for the three different responses it can issue for every transfer:

```
define null_resp = !SResp[0] & !SResp[1];
define dva_resp  =  SResp[0] & !SResp[1];
define err_resp  =  SResp[0] &  SResp[1];

define cmd_idle      = !MCmd[0] & !MCmd[1] & !MCmd[2];
define cmd_write     =  MCmd[0] & !MCmd[1] & !MCmd[2];
```

```
define cmd_read      = !MCmd[0] &  MCmd[1] & !MCmd[2];
```

The behavior of the slave can defined as being idle or performing a transfer.

```
slave -> (idle || transfer)*;
```

If the master sets the command wires to cmd_idle indicating an idle transfer, then the slave must set SCmdAccept to low and it must respond with a null response.

```
idle -> cmd_idle & !SCmdAccept & null_resp;
```

A transfer can be a write transfer or a read transfer.

```
transfer ->
    write_transfer ||
    read_transfer;
```

During a write transfer the slave must keep SCmdAccept low and set the response to null until it accepts the transfer, indicated by SCmdAccept high and the response wires set to null_resp.

```
write_transfer ->
    (cmd_write & !SCmdAccept & null_resp)* ,
    (cmd_write &  SCmdAccept & null_resp);
```

If the master is performing a read transfer the slave can delay the acceptance of this transfer by any number of cycles by setting SCmdAccept to low and the response wires to null_resp.

```
read_transfer ->
    (cmd_read & !SCmdAccept & null_resp)* ,
    (wait_state_response || instant_response);
```

The slave may accept the transfer and delay the response by setting the response wires to null_resp.

```
wait_state_response ->
    (SCmdAccept & null_resp) ,
    (!SCmdAccept & null_resp)* ,
    response;
```

After delaying the response for a certain number of cycles the slave can issue a data valid response (dva_resp) or an error response (err_resp). SCmdAccept must be kept low.

```
response ->
    (!SCmdAccept & dva_resp) ||
    (!SCmdAccept & err_resp);
```

An instant response consists of accepting the transfer (indicated by SCmdAccept) and at the same time, setting the response wires to a data valid response (dva_resp) or an error response (err_resp).

```
instant_response ->
   (SCmdAccept & dva_resp) ||
   (SCmdAccept & err_resp);
```

Master

The specification starts by declaring the interface wires that are the inputs (SCmdAccept, SResp, SData) and outputs (MAddr, MCmd, MData) of the master.

```
input SCmdAccept, SResp[1:0], SData[31:0];
output MAddr[31:0], MCmd[2:0], MData[31:0];
```

The following are abbreviations for the three different transfers a master can perform and for the three different responses the slave can issue for every transfer.

```
/* Response codes defined in standard. */
/* NULL, Data VAlid, and ERRor */
define null_resp = !SResp[0] & !SResp[1];
```

```
define dva_resp  =  SResp[0] & !SResp[1];
define err_resp  =  SResp[0] &  SResp[1];

/* Commands defined in standard. */
define cmd_idle  = !MCmd[0] & !MCmd[1] & !MCmd[2];
define cmd_write =  MCmd[0] & !MCmd[1] & !MCmd[2];
define cmd_read  = !MCmd[0] &  MCmd[1] & !MCmd[2];
```

The behavior of the master can be defined as being idle or performing a transfer. If it is in an idle state, it drives the command signals to cmd_idle.

```
master -> (cmd_idle || transfer)*;
```

A transfer can be a write transfer or a read transfer.

```
transfer -> write_transfer || read_transfer;
```

During a write transfer the master drives the command signals to cmd_write. It then waits until the slave responds with the successful completion of the transfer by driving SCmdAccept high. Writes are posted (no response required once the slave accepts the command).

```
write_transfer ->
    (cmd_write & !SCmdAccept)*, (cmd_write & SCmdAccept);
```

The read transfer is a bit more complicated, starting with zero or more states waiting for the slave to accept the command, followed by either an instantaneous response or a response with zero or more wait states.

```
read_transfer ->
    (cmd_read & !SCmdAccept)* ,
    (wait_state_resp || instant_resp);
```

A wait state response starts with the slave accepting the transfer (indicated by SCmdAccept high) and setting the response to null_resp. Any number of wait states can be inserted before a final response is given.

```
wait_state_resp ->
    (cmd_read & SCmdAccept & null_resp) ,
    (cmd_idle & null_resp)* , response;
```

During an instant response, the slave drives the response signals to data valid (dva_resp) or error (err_resp) at the same time that it accepts the transfer.

```
instant_resp -> (cmd_read & SCmdAccept & dva_resp) ||
                (cmd_read & SCmdAccept & err_resp);
```

A response can be a data valid response or an error response.

```
response -> (cmd_idle & dva_resp) ||
            (cmd_idle & err_resp);
```

In Basic OCP, the master presents a command at the same time as the address, as well as the data if the command is a write. These values must be held constant until the slave accepts the command by asserting SCmdAccept. For simplicity, the previous specification does not check that the master holds the address and data values constant if the slave does not accept the command immediately. To enforce this requirement, we need only make a few changes to the specification. First, we would declare some storage variables to remember the values of the address and data:

```
internal hold_addr[31:0] = 0;
internal hold_data[31:0] = 0;
```

Next, we modify the transfers so that if the slave does not accept the command immediately, we remember the address (and data if applicable):

```
write_transfer ->
    (cmd_write & SCmdAccept) /* Same cycle accept */
    ||
    (
     (cmd_write & !SCmdAccept)
        /* Store original address and data. */
        { hold_addr <- MAddr; hold_data <- MData; } ,
     (cmd_write & !SCmdAccept &
        (hold_addr == MAddr) & (hold_data == MData)
     ) * ,
     (cmd_write & SCmdAccept &
        (hold_addr == MAddr) & (hold_data == MData))
    );
```

The read transfer production is modified similarly. To enforce the same constraints using regular expressions without storage variables would require separate read and write productions for each possible value of the address and data.

5.3 Results

Table 5.1 shows some statistics for the AHB slave, AHB master, OCP slave, and OCP master. Considering that the AHB English written specification is 43 pages long (including the arbiter, which was not described in this chapter), and it is still ambiguous and incomplete, the results obtained by expressing the same protocol in our specification style are very encouraging.

Monitor	Lines	Productions	Actions	Internal Variables	Flops
AHB Slave	64	29	19	2	292
AHB Master	308	44	28	10	1478
OCP Slave	54	8	0	0	118
OCP Master	52	7	0	0	118

Table 5.1: Results for the AHB slave, AHB master, OCP slave, and OCP master. Lines is the total number of lines, including blank lines and lines with comments only, in the specification. Productions is the number of productions used to specify the interface. Actions reflects the number of nodes in the parse tree that have an action associated with it. Internal variables is the number of declared internal variables, each variable may be one bit or an array of bits. Flops is the number of flops in the generated Verilog monitor.

Chapter 6

Conclusion and Future Work

We have presented a novel, high-level specification style for interface monitors, as well as a linear-size, linear-time translation algorithm into monitor circuits. The specification style naturally fits the interfaces used between IP blocks in systems-on-chip. We hope that these results will facilitate the use and broaden the adoption of monitor-based verification methodologies.

In the short term, we need to improve the translation tool. Our current implementation is rather crude. A revised, more robust tool would be better suited for public distribution. In addition, many optimizations are possible and should be implemented. To access the tool flows of other verification researchers, our tool will have to be able to translate specifications into the lower-level specifications used by other tools. In particular, we do not anticipate difficulties translating from our specifications into the specifications used in the interface-monitor research that inspired this work [10, 16, 9].

The macro-expansion of productions can blow-up the size of the regular expressions. Additional research is needed to see if there are practical ways to avoid this problem, such as by introducing new language features to simplify the expressions or by creating better translations that schedule reuse of circuitry.

Out-of-order completion of transactions is not easily supported by the pipeline operator. Usually, each transaction is *tagged* with a unique identifier that is carried through all

pipeline stages until its completion. This tag allows each pipeline stage to identify which transaction it is currently being processed. Multiple issue of ordered transactions is also not currently supported. For example, one of the possible OCP configurations allows a finite number of transfer requests to be issued before a response is given to any of the transfers. Responses are given in an ordered way, the first response to arrive is the response for the first request issued and so on. Even though it is possible to specify this type of behavior by explicitly creating multiple copies of the same pipeline stage (and using storage variables for the tagged transactions) it would be interesting to add a new pipeline operator or even extend the semantics of the current pipeline operator that would automatically handle these cases. For example, this operator could allow more than one computation thread to be active at a time in the same pipeline stage. In order to be able to translate the specification into a monitor, it would be required that the maximum number of "pending requests" be specified beforehand. Also, storages variables that are only valid inside a given pipeline stage would be necessary to track the tag of each transaction. The translated monitor would have as many copies of the pipeline stage as the maximum number of "pending requests", and it would have to able to schedule the usage of each of these resources.

An important line of future work is to gain experience with this specification style on additional, real interface protocols. Applying the specification style in practice is the only way to validate the adequacy of the specification style, or determine what additional features are needed.

Bibliography

[1] Accellera Home Page. www.accellera.org.

[2] ARM Limited. *AMBA Specification (Rev 2.0).* 13 May 1999.

[3] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. The ForSpec temporal logic: A new temporal property-specification language. In *Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference*, pages 296–311. Springer, 2002. Lecture Notes in Computer Science Number 2280.

[4] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. The temporal logic Sugar. In *Computer-Aided Verification: 13th International Conference*, pages 363–367. Springer, 2001. Lecture Notes in Computer Science Number 2102.

[5] Lionel Bening and Harry Foster. *Principles of Verifiable RTL Design: A Functional Coding Style Supporting Verification Processes in Verilog.* Kluwer Academic Publishers, 2nd edition, 2001.

[6] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer-Verlag, 1981. Lecture Notes in Computer Science Number 131.

[7] Fujitsu Limited, Fujitsu Laboratories Limited. *CWL2HDL User's Manual.* Revision 1.0, 27 Jan 2003.

[8] Hitachi Limited, Fujitsu Laboratories Limited, Fujitsu Limited. *Component Wrapper Language User's Guide (Rev 1.1).* 01 Sep 2002.

[9] M. S. Jahanpour and E. Cerny. Compositional verification of an ATM switch module using interface recognizer/suppliers (IRS). In *International High-Level Design, Validation, and Test Workshop*, pages 71–76. IEEE, 2000.

[10] Matt Kaufmann, Andrew Martin, and Carl Pixley. Design constraints in symbolic model checking. In *Computer-Aided Verification: 10th International Conference*, pages 477–487. Springer, 1998. Lecture Notes in Computer Science Number 1427.

[11] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[12] Pascal Raymond. Recognizing regular expressions by means of dataflow networks. In *23rd International Colloquium on Automata, Languages, and Programming*, pages 336–347. Springer, 1996. Lecture Notes in Computer Science Number 1099.

[13] Andrew Seawright and Forrest Brewer. Synthesis from production-based specifications. In *29th Design Automation Conference*, pages 194–199. ACM/IEEE, 1992.

[14] Andrew Seawright and Forrest Brewer. High-level symbolic construction techniques for high performance sequential synthesis. In *30th Design Automation Conference*, pages 424–428. ACM/IEEE, 1993.

[15] Kanna Shimizu, David L. Dill, and Ching-Tsun Chou. A specification methodology by a collection of compact properties as applied to the Intel Itanium Processor Bus protocol. In *Correct Hardware Design and Verification Methods: 11th IFIP WG 10.5 Advanced Research Working Conference, (CHARME)*, pages 340–354. Springer, 2001. Lecture Notes in Computer Science Number 2144.

[16] Kanna Shimizu, David L. Dill, and Alan J. Hu. Monitor-based formal specification of PCI. In *Formal Methods in Computer-Aided Design*, pages 335–353. Springer, 2000. Lecture Notes in Computer Science Number 1954.

[17] Sonics Incorporated. *Open Core Protocol Specification 1.0.* Document Version 1.2.

[18] Kei Suzuki, Kouji Ara, and Kazuo Yano. OwL: An interface description language for IP reuse. In *Custom Integrated Circuits Conference*, pages 403–406, 1999.

[19] Synopsys Incorporated. *OpenVera Language Reference Manual: Assertions.* Document Version 2.3, April 2003.

[20] Jun Yuan, Kurt Shultz, Carl Pixley, Hillel Miller, and Adnan Aziz. Modeling design constraints and biasing in simulation using BDDs. In *International Conference on Computer-Aided Design*, pages 584–589. IEEE/ACM, 1999.

Appendix A

Pipelined Regular Expression Monitor Compiler Manual

A.1 Introduction to PREMiS

A.1.1 Overview

The Pipelined Regular Expression Monitor Specification (PREMiS) is a high-level specification style designed to facilitate the construction of interface monitors. The PREMiS compiler automatically translates the interface specification into a Verilog or VHDL monitor. This manual describes the PREMiS language and the usage of the compiler.

A.1.2 Identifiers and Reserved Words

An identifier in the PREMiS language is any sequence of alphabet letters, digits, and underscores, where the first character must be an alphabet letter. Identifiers are case-insensitive in order to facilitate the translation to VHDL, which is also case-insensitive. Examples of legal identifiers are: `grant` (same as `GrAnt`), `split_2`, `data32`. Examples of illegal identifiers are: `_grant`, `2_split`, `32data`.

The following is the list of PREMiS reserved words:

```
internal input output in_out define monitor
```

Reserved words cannot be used as identifiers. Their meaning will be explained in detail in the following sections.

A.1.3 Input Signals

The input signals of the monitor, are all the input and output signals of the block being monitored. These signals must be declared at the beginning of the file, and they can be any of the following three types:

- `input`: this signal is an input to the block being monitored.
- `output`: this signal is an output of the block being monitored.
- `in_out`: this signal is an input and an output to the block being monitored.

The actual implementation of the compiler does not make any distinction between these types but in the future we intend to use these different types to be able to blame the block responsible for causing an error.

The syntax for signal declarations is:

signal-type comma-separated-list-of-identifiers;

Arrays can be declared as *identifier[begin:end]*. A few examples of signal declaration taken from the AMBA slave specification are shown below:

```
input HTRANS[1:0], HREADY, HSEL, HMASTER[3:0];
output HRESP[1:0], HSPLIT[15:0];
```

The signals `HTRANS`, `HREADY`, `HSEL`, and `HMASTER` are the slave inputs and signals `HRESP` and `HSPLIT` are the slave outputs.

In order to access an element from an array, square brackets are used to index the specific element. For example, `HMASTER[2]` refers to the third element of array `HMASTER` which has size four.

A.1.4 Storage Variables

Storage variables are used in the PREMiS language to facilitate the monitor specification. They can be seen as a type of memory. The main difference between a signal and a storage variable is that the latter can have values assigned to it according to some conditions in the specification (See Section A.1.10.). Other than that, they can be used in any place a signal is expected.

The declaration of storage variables has almost the same syntax as signal declaration with the difference that they can be initialized. Initialization values are constants (See Section A.1.5.), and in case no initial value is set, the default value is 0 (zero). In the PREMiS compiler input file, the declaration of storage variables is done in the beginning of the file together with signal declarations.

A few examples from the AMBA master specification are shown below:

```
internal i_grant = 0;
internal i_first = 1;
internal i_wdata[31:0] = 0;
```

The storage variables `i_grant` and `i_first` are used by the monitor to keep track of the grant signals and `i_wdata` is used to guarantee that the master will not change the data on the bus during a write transfer.

A.1.5 Constants

Constants in the PREMiS language are unsigned integers represented in base ten. A few examples usage of constants are shown below:

```
internal i_count[2:0] = 0
i_count != 7
i_count <- i_count + 1
```

In the first example the constants are used for declaring the size of the array and to initialize it to 0. The second line shows a constant being used in a comparison expression

(See Section A.1.6.), and in the last line the constant is used in a action expression (See Section A.1.10.) .

A.1.6 Primitive Expressions

There are three types of expressions defined in the PREMiS language: *primitive expressions* (explained here), *extended regular expressions* (explained in section A.1.7), and *action expressions* (explained in section A.1.10).

A *primitive expression* is a formula consisting of signals, storage variables, unary operators, comparison operators and bitwise operators. The precedence of all operators are listed in Table A.1 from highest precedence to lowest precedence.

Precedence	Operators
Highest	!
	== !=
Lowest	| &

Table A.1: Operator precedence.

Signals and Storage Variables

Any monitor input signal or storage variable of size one is a primitive expression. Arrays are not considered primitive expressions but array elements are.

Unary Operators

The only unary operator in the PREMiS language is the *negation* operator which is represented by the symbol "!". This operator can be used with any primitive expression and it cannot be used with arrays. For example: if `HTRANS[1:0]` is an array of size two then `!HTRANS` is illegal but `!HTRANS[0]` is legal.

Comparison Operators

The *equality* operator "==" and the *not equal* operator "!=" are the two comparison operators present in the language.

Only input signals, storage variables and constants can be compared. An array can be compared to another array only if both have the same size and the begin and end positions match. Arrays can also be compared to constants if their size is big enough to hold the constant. As an example, the following code shows the declaration of four arrays A, B, C, D:

```
input A[0:1], B[0:2], C[1:2], D[0:2];
```

The only valid comparison of two arrays is between B and D. Even though arrays A and C have the same size, they cannot be compared because the initial position of A (0) is different of initial position of C (1) and their end positions (1 and 2) are also different. B and D can be compared to any constant which has a value less than 8 and, A and C can be compared to any constant which has a value less than 4. In summary:

B == C is legal

B != A is illegal because of size difference

A == C is illegal because of initial and end positions difference

A != 3 is legal

A == 4 is illegal because A can only represent values from 0 to 3

Bitwise Operators

The two bitwise operators in the language are the *and* operator "&" and the *or* operator "|".

The left-hand-side and right-hand-side of the operators can be any primitive expression (arrays are not primitive expressions). The semantics for both operators is the same semantics used for logic circuits. The following is an example form the AMBA AHB slave specification:

```
!HTRANS[0] & !HTRANS[1] & HSEL & HREADY
```

The previous expression corresponds to the control signals for a slave being selected to perform an idle transfer.

A.1.7 Extended Regular Expressions

A *extended regular expression* is a formula containing primitive expressions and extended regular expression operators. The precedence of all operators are listed in Table A.2 from highest precedence to lowest precedence.

Precedence	Operators
Highest	+
	^
	*
	||
	,
Lowest	@

Table A.2: Extended regular expression operator precedence.

Primitive Expressions

Any primitive expression is an extended regular expression.

Choice

The *choice* operator "||" is used to describe two possible behaviors. At least one of the two sub-expressions must be in a matching state in order for the monitor not to generate an error.

The following is an example from the AMBA AHB slave:

```
transfer -> idle_transfer    ||
            busy_transfer    ||
            nonseq_transfer  ||
```

```
seq_transfer;
```

Each transfer can be one of the four possible types: idle, busy, non-sequential and sequential.

Concatenation

The *concatenation* operator "," concatenates the behavior of two sub-expressions in a time sequence. The monitor watches the left-hand-size and as soon as this sub-expression is over, it starts to watch the right-hand-side. If a mismatch occurs in any of the two sub-expressions the monitor will generate an error.

The following is an example from the AMBA AHB slave:

```
error_response -> (!HREADY & a_error) , (HREADY & a_error);
```

An error response consists of two cycles, the first one with HREADY low and the second one with HREADY high.

Multiple Concatenation

The *multiple concatenation* operator "^" is used to concatenate the behavior of a sub-expressions a given number of times in a time sequence. The actual implementation of the compiler expands the expression into a sequence of concatenations. For example, the following expression:

```
frame_header -> one^5, zero^3;
```

is expanded in the sequence below by the compiler:

```
frame_header -> one, one, one, one, one, zero, zero, zero;
```

Pipeline

The *pipeline* operator "@" makes the monitor watch the left-hand sub-expression and as soon as the parsing of this expression is done, the monitor starts two new threads, the first

will watch the right-hand sub-expression and the second will watch the sub-expression that comes after the pipeline expression. Both threads must not generate an error in order for the monitor to not generate an error.

The following is an example based on the ARM AMBA slave:

```
transfer -> ((a_nonseq & HSEL_1 & HREADY) @ response)*;
```

A transfer consists of an address phase (`a_nonseq & HSEL_1 & HREADY`) followed by a pipelined response phase (`@ response`). As soon as the left-hand-side sub-expression is done, the monitor starts two new threads, the first will watch the `response` sub-expression and the second will watch the `a_nonseq & HSEL_1 & HREADY` sub-expression because of the Kleene star operator. This way we can get the expected pipelined behavior, on every cycle, an address phase and a response phase take place.

Kleene Star

The *Kleene star* operator "*" is used to represent zero or more repetitions of the behavior described by the associated sub-expression.

The following is an example from the AMBA AHB slave:

```
response ->
    wait_state* ,
    (okay_response || error_response ||
     split_response || retry_response);
```

A slave response may include any number of wait states before the actual response is set.

One-or-more

The *one-or-more* operator "+" is used to represent one or more repetitions of the behavior described by the associated sub-expression. The compiler expands the one-or-more expression into a concatenation followed by a Kleene star expression. For example, the following expression:

```
write_sequence -> write+;
```

is expanded into the expression below:

```
write_sequence -> write, write*;
```

Extended Regular Expression Restrictions

In order to be able to automatically build a monitor from the specification, we impose a few restrictions on the extended regular expressions.

First, we require the expression contained within a Kleene star not to accept the empty string. Known constructions can normalize regular expressions to obey this restriction [12], but our implementation does not currently include this step.

Second, we forbid non-deterministic choice: we allow the choice operator, but the choices must be distinguishable within the first clock cycle. In practice, this restriction is not a problem because protocols are typically designed to make it easy to determine immediately what action is occurring.

Finally, we allow at most one thread at a time to execute in a pipeline stage. For example, the expression `(a@(b,c))*` generates an error when the second repetition arrives at the `b` while the first repetition's pipeline sub-thread is still at the `c`. This restriction corresponds to allowing only one transaction at a time to use the hardware resources devoted to a pipeline stage.

A.1.8 Define Statement

The *define* statement is used to declare an abbreviation for a primitive expression. These definitions must be done after the signal and storage variable declaration section and before the *monitor* statement section. An example from the AMBA AHB slave specification is shown below:

```
define idle   = !HTRANS[0] & !HTRANS[1];
define busy   =  HTRANS[0] & !HTRANS[1];
```

```
define nonseq = !HTRANS[0] &  HTRANS[1];
define seq    =  HTRANS[0] &  HTRANS[1];
```

The identifiers on the left-hand-side of the "=" operator are the abbreviation for the four possible transfer types, `idle`, `busy`, `nonsequential`, and `sequential`.

A.1.9 Productions

A production is an abbreviation for an extended regular expression. Its name is an identifier that can be used in other extended regular expressions. In fact, we have already been using productions in the previous examples. In order to be able to obtain a finite state machine, recursive productions are not allowed.

The syntax for a production declaration is:

production-name → *extended-regular-expression;*

The example below shows the usage of productions in the specification of an AMBA AHB master that performs only idle transfers:

```
master_bus -> (idle || idle_transfer)*;
idle -> !i_grant;
idle_transfer ->
     (a_idle & !HREADY & i_grant)* ,
     (a_idle & HREADY & i_grant);
```

`master_bus`, `idle`, and `idle_transfer` are the production names.

The first production in the input file is the top-level production of the monitor. If the file contains the specification of more than one monitor, then the *monitor* statement must be used to indicate which production is the top-level production for each monitor. The example below shows the usage of the *monitor* statement in the AMBA AHB master spec:

```
monitor master_bus, grant;
```

The productions `master_bus` and `grant` are the top-level productions of the two monitors described in the AHB master specification.

A.1.10 Variable Assignment

Storage variables can have values assigned to them in an extended regular expression. The assignment is triggered when the extended regular sub-expression associated with it is matched. The following is an example from the AMBA AHB master:

```
grant -> ((HGRANT_1 & HREADY {i_grant <- 1;})  ||
          (!HGRANT_1 & HREADY {i_grant <- 0;}) ||
          (!HREADY))*;
```

The storage variable `i_grant` is assigned the value 1 every time the sub-expression `HGRANT_1 & HREADY` is matched and it is assigned the value 0 when the sub-expression `!HGRANT_1 & HREADY` is matched. If the sub-expression `!HREADY` is matched, `i_grant` remains unchanged.

The right-hand-side of an assignment is an *action expression*. The action expression can be a constant, a signal/variable, or a signal/variable array. Arithmetic addition "+" and subtraction "-" are also available.

The same storage variable may have different values assigned to it in the same cycle. The result of simultaneous assignments is implementation specific. In the current implementation, the order of assignments is defined by the parse tree of the regular expression, with assignments done in the order of a pre-order traversal of the parse tree.

A.1.11 Input File Format

The PREMiS compiler input is a file consisting of four sections: *Signal/Variable Declaration*, *Define Declaration*, *Monitor Declaration*, and *Productions*. The order of the sections cannot be changed and the *Define Declaration* and *Monitor Declaration* parts are optional. The *Signal/Variable Declaration* section is where all input signals and storage variables are declared. The *Define Declaration* part contains the declaration of all primitive expression abbreviations. If a file describes more than one monitor than the *Monitor Declaration* statement indicates which production is the top-level production for each monitor. The last part

is a list of productions that describes the behavior of the interface being specified. A more detailed explanation of each part is given in the following sections.

A.2 Running the PREMiS Compiler

A.2.1 Command Line Syntax and Options

The PREMiS compiler takes one file as input and it generates one file as output. The input file is the specification of the monitor and the output file is a Verilog or VHDL monitor. The command line syntax is:

premsc [options] <input-file>

where options are:

- `-t <verilog, vhdl>`: Set output language. The default value is `verilog`.
- `-o <output-file>`: Write output to `output-file`. The default value is `stdout`.

A.2.2 Output File

The compiler generates a Verilog file or a VHDL file as output. If the chosen language is Verilog, the file contains one module called `MONITOR`. If VHDL is the chosen language, the file contains one entity called `MONITOR` and one architecture description called `MONITOR_BEHAVIOUR`.

The input signal declaration follows the same order they were declared in the specification. In addition to these signals there are two more inputs and one output to the monitor. The first additional input is the clock signal and the second is the reset signal. The only output is the OK signal which, when high, indicates that there have not been any violations to the protocol yet. These additional signals are declared after the ones declared in the specification, and they follow the order they were presented here.

Appendix B

Language Grammar

Below is the grammar for the PREMiS language. Non-terminal symbols are represented in italics. Boxed items correspond to terminal symbols. The two undefined symbols (identifier and constant) are described in Appendix A. The symbol [ε] denotes the empty string.

spec →

signal-declaration-list

| define-declaration-list

| start

| production-list

signal-declaration-list →

signal-declaration

| signal-declaration-list signal-declaration

signal-declaration →

type-specifier signal-list [;]

type-specifier →

`internal`

| `output`

| `input`

| `in-out`

signal-list →

`ε`

| signal

| signal `,` signal-list

signal →

identifier

| identifier `[` constant `:` constant `]`

| identifier `=` constant

| identifier `[` constant `:` constant `]` `=` constant

define-declaration-list →

`ε`

| define-declaration

| define-declaration-list define-declaration;

define-declaration →

`define` identifier `=` boolean-expression `;`

boolean-expression →

comparison-expression

| identifier

| identifier `[` index-expression `]`

| boolean-expression `&` boolean-expression

| boolean-expression [||] boolean-expression

| [!] boolean-expression

| [(] boolean-expression [)]

comparison-expression →

comparison-term [==] comparison-term

| comparison-term [!=] comparison-term

comparison-term →

identifier

| identifier [[] index-expression []]

| constant

index-expression →

identifier

| constant

production-expression →

boolean-expression

| production-expression [|||] production-expression

| production-expression [,] production-expression

| production-expression [^] constant

| production-expression [*]

| production-expression [+]

| production-expression [@] production-expression

| [(] production-expression [)]

| production-expression [{] action-list [}]

start →

ε

| `monitor` start-list `;`

start-list →

identifier

| identifier `,` start-list

production-list →

production

| production production-list

production →

identifier `->` production-expression `;`

action-list →

action

| action action-list

action →

identifier `<-` action-expression `;`

| identifier `[` index-expression `]` `<-` action-expression `;`

action-expression →

constant

| identifier

| identifier `[` index-expression `]`

| action-expression `-` action-expression

| action-expression `+` action-expression