Feature-Oriented Specification of Hardware Bus Protocols

by

Paul M. Freitas

A Thesis
Submitted to the Faculty
of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

May 2008

APPROVED:

____

Professor Kathi Fisler, Thesis Advisor

____

Professor Gary Pollice, Thesis Reader

____

Professor Michael Gennert, Head of Department

Abstract

Hardware engineers frequently create formal specification documents as part of the verification process. Doing so is a time-consuming and error-prone process, as the primary documents for communications and standards use a mixture of prose, diagrams and tables. We would like this process to be partially automated, in which the engineer's role would be to refine a machine-generated skeleton of a specification's formal model. We have created a preliminary intermediate language which allows specifications to be captured using formal semantics, and allows an engineer to easily find, understand, and modify critical portions of the specification. We have converted most of ARM's AMBA AHB specification to our language; our representation is able to follow the structure of the original document.

Acknowledgements

This work would not have been possible without the help and support of many people. I would particularly like to thank: Professor Kathi Fisler, who introduced me to this task and guided me through the process even after a later start than usual; Chris King, Danny Yoo, Andrey Sklyar, and Yu Feng, my collaborators in the ISP that led to this project, and whose feedback helped shape many facets of this project; and Professor Gary Pollice, who demonstrated great patience as my thesis reader.

Contents

1 Motivation **6**

2 Background **7**

2.1 Protocol Specifications . . . 7
2.1.1 Timing Diagrams . . . 7
2.1.2 Tables . . . 7
2.1.3 Text . . . 9
2.1.4 Structure . . . 9
2.2 Proposed Workflow . . . 11
2.2.1 Automatic Protocol Skeleton Extraction . . . 11
2.2.2 Completion of Full Intermediate Representation . . . 14
2.2.3 Extraction of Verification and Validation Artifacts . . . 14

3 Representing Timing Diagrams **16**

3.1 Characteristics of Protocol Timing Diagrams . . . 16
3.1.1 Sequence and Synchronization of Signals . . . 16
3.1.2 Compositional Nature of Timing Diagrams . . . 17
3.1.3 Logical Divisions of Bus Protocol Timing Diagrams . . . 18
3.2 Our Method: Blocks . . . 18
3.2.1 Syntax and Informal Semantics . . . 20
3.2.2 Formal Semantics . . . 23
3.2.3 Restrictions on Parallel Sequences . . . 29
3.3 Future Work on Blocks . . . 30
3.3.1 Adding Timing Constraints . . . 30
3.3.2 Glitches and Clean Signals . . . 32

4 Composing Timing Diagram Fragments to Form a Specification **34**

4.1 Composition Requirements for Imitating Specification Structures . . . 34
4.1.1 Extending the Block Structure for Composition . . . 34
4.1.2 Encapsulation of Protocol Features . . . 34
4.1.3 Types of Extension . . . 35
4.1.4 Oblivious Extension . . . 36
4.2 Aspect-Oriented Programming . . . 36
4.3 Our Composition Mechanism, by Example . . . 37
4.3.1 New Language Structures . . . 37
4.3.2 Basic Composition . . . 37
4.3.3 Concurrent Composition . . . 41
4.3.4 More Complicated Examples . . . 42
4.4 Our Primitive PCDs and Advice Types . . . 44
4.4.1 The Difference Between PCDs and ExtendPoints . . . 44

4.4.2 Why Match to Blocks? 45
4.4.3 Primitive Extendpoints 45
4.4.4 Advice Types 47
4.5 Issues Requiring Further Examination 48
4.5.1 Advice Collision and Conflicts 48
4.5.2 Cancelling Scheduled Blocks 49
4.5.3 More Complicated Protocol Features 49
4.6 Other Explored Composition Mechanisms 50
4.6.1 All Blocks 50
4.6.2 Live Sequence Charts 50
4.6.3 State Machine Driven Composition 51

5 Conclusion **53**

A Full Language Example **54**

List of Figures

2.1 Timing diagram for basic read and write transfers in AMBA AHB. 8
2.2 Key to timing diagram conventions in the AMBA document. 8
2.3 A table from the AMBA specification. 9
2.4 Table of contents for the AMBA AHB specification. 10
2.5 Timing diagram for basic read and write transfers in AMBA AHB (shading added). 11
2.6 Timing diagrams showing wait states (Figure (a)) and pipelining (Figure (a)) in AMBA AHB. 12
2.7 Table of transfer types from the AMBA specification. 13
2.8 Transfer types in AMBA AHB. 13

3.1 Timing diagram for basic read and write transfers in AMBA AHB. 17
3.2 2D Regular expression [4] for the AMBA basic transfer shown in Figure 3.1. 18
3.3 Timing diagram for basic read and write transfers in AMBA AHB showing logical blocks. . . 19
3.4 Timing diagram for an undefined-length burst in AMBA AHB showing logical blocks. 19
3.5 Specification for the basic transfer of Figure 3.1 in our language. 20
3.6 Grammar for our block language. 22
3.7 A table from the AMBA specification. 23
3.8 The **TransType** tabletype defined in our language. 23
3.9 A block defining a clock cycle, and its corresponding RTD. 24
3.10 A block defining the address phase of a transfer, and its corresponding RTD. 24
3.11 A block defining the data phase of a transfer, and its corresponding RTD. 25
3.12 A block defining a simple transfer, and its corresponding RTD. 25
3.13 A block with ambiguous semantics. 30
3.14 Four possible timing diagrams representing the same ambiguous block. 31
3.15 Three blocks with inconsistent semantics. 32

4.1 Table of contents for the AMBA AHB specification. 35
4.2 The slave's retry response and the resultant transfer in AMBA AHB. 36
4.3 First timing diagram in AMBA AHB; shows basic read and write transfers. 38
4.4 Second timing diagram AMBA AHB; adds wait states to the basic transfer. 38
4.5 AMBA AHB basic transfer and waitstates in our language. 39
4.6 Pipelining and multiple transfers in AMBA AHB. 40
4.7 Implementing pipelining in our language without synchblocks. 41
4.8 Implementing pipelining in our language using a synchblock. 42
4.9 Four-beat wrapping burst of transfers from the AMBA AHB specification. 43
4.10 Indeterminate-length burst from the AMBA AHB specification. 44
4.11 Process for concurrent advice using synchblocks. 48
4.12 State machine from the AMBA APB specification. 51
4.13 Example of a Statechart which defines a simple pipelined transfer. 52

List of Tables

4.1 Primitive extendpoints in our language. 46
4.2 Advice types in our language. 47

Chapter 1

Motivation

Verification is a critical part of the Application-Specific Integrated Circuit (ASIC) design process; approximately 70% of the effort spent on the average project is consumed by verification. For this reason, methods which improve the efficiency and accuracy of hardware verification are immensely valuable.

In terms of human effort, the largest task in the verification process is the design and creation of the verification artifacts which drive automated verification. Two prominent types of these are Register Transfer Level (RTL) assertions, and testbenches. Both of these artifacts must be hand-written by engineers based on an informal specification of the device to verify. This process can be thought of as a "translation" from the informal specification document to a formal verification artifact which will accept devices with the desired functionality.

Unfortunately, this translation process is difficult in practice. The difficulty comes from the large difference between informal and formal representations. The most obvious difference between informal specifications and verification artifacts is in language; formal verification artifacts use different syntax with well-defined semantics. This difference is desirable; without it, there would be no point to the translation.

The problem for verification engineers lies in the other differences. Verification artifacts differ greatly from informal specifications in their organization, scope, and emphasis. These make the initial translation process time-consuming and error-prone. Locating errors in the resulting verification artifacts is difficult, and it can be a challenge to re-use verification artifacts for later projects with similar specifications.

The solution to these difficulties, we feel, is not to replace informal specifications with documents in a hardware description language or some similar format with rigidly defined semantics. Such a proposal ignores the value of informality, which facilitates conceptual understanding, modification, and extension of specification documents. It also ignores the reality that all specifications, regardless of format, must come from some type of informal description created in the design process.

Instead, we propose to bridge the gap between the existing styles of specification and verification artifacts with an intermediate representation. For this purpose, we have begun designing a language which embraces aspects of typical informal specifications, but which has well-defined semantics. We seek to allow in our language clearly-defined, machine-readable specifications which are similar in scope, organization, and flexibility to the informal specifications they are based on.

Chapter 2

Background

2.1 Protocol Specifications

As this project focuses on the conversion of informal specifications to formal ones, it is important to understand the features of typical informal specification documents. Both the artifacts that are used to convey the details of the specification and the structure in which they are laid out contribute to the effectiveness of these documents. As an example of a typical protocol specification document, we use the Advanced Microcontroller Bus Architecture (AMBA®) Advanced High-Performance Bus (AHB) interface [2]. This standard is widely used both in practice and as a benchmark in interface specification and validation research.

2.1.1 Timing Diagrams

By far the most prolific and important type of non-text artifact found in the AMBA document is the timing diagram. Figure 2.1 shows the first of these that appears in the AMBA AHB, which demonstrates the protocol for a basic read or write transfer. This timing diagram clearly and concisely communicates the basic operation of the protocol to an engineer. It demonstrates both the sequencing of signals and how they are synchronized across multiple channels.

Each line in the timing diagram represents a single signal. Each of these may be a single-bit signal or a bus. Single-bit signals may take on values of logic HIGH and logic LOW, while buses are a collection of many of these. Buses can be distinguished from single-bit signals in Figure 2.1 because their names are annotated with a bit width, for example "HADDR[31:0]". This means that the HADDR bus is a 32-bit bus.

Since this document uses a number of timing diagrams from the AMBA AHB specification, we will describe briefly the conventions used in the AMBA timing diagrams. Figure 2.2 contains the key presented at the beginning of the AMBA document. In addition to the conventions it presents, the AMBA key also specifies that shaded bus and signal areas represent undefined areas, where the value of the signal is not important and can assume any value.

Timing diagrams are largely used to give concrete examples of the operation of a protocol, but, because of their informality, allow for some abstraction. For example, the unlabeled white polygons in the diagram represent "don't care" values; in these areas, the value of the signal is not important to the example shown in the diagram. In addition, the "Control" portion of the protocol at this point in the specification is not well defined. The specification has not described either the number of bits in the control signal or the content of that signal. These have been replaced by a "Control" placeholder.

2.1.2 Tables

Tables are another important type of artifact found in informal specifications. In many cases, a given signal or set of signals has some small number of values it can take on. Sometimes the operation of the protocol depends on which of those values are held on that set of signals. This is often the case for bus control signals.

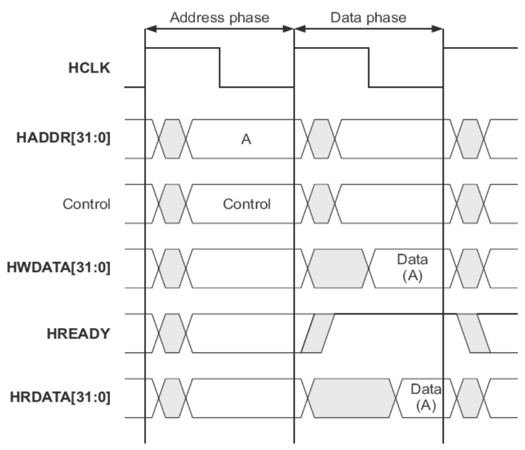


Figure 2.1: Timing diagram for basic read and write transfers in AMBA AHB.

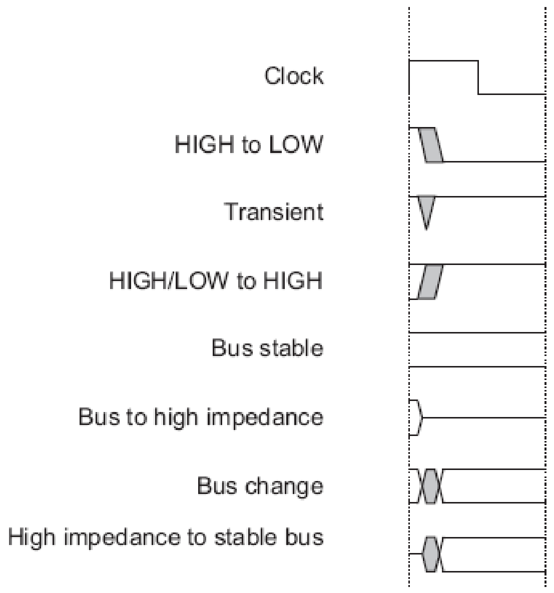
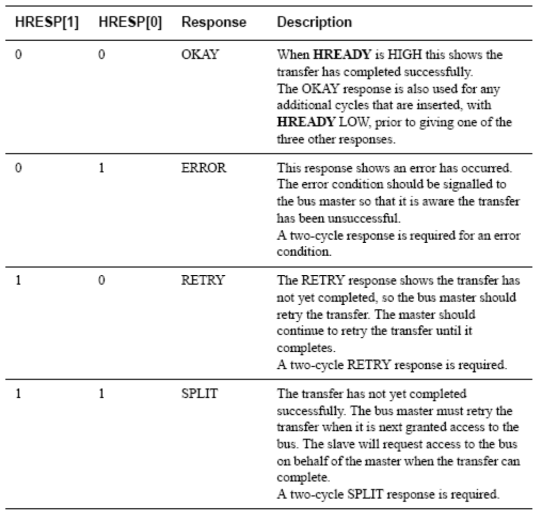


Figure 2.2: Key to timing diagram conventions in the AMBA document.



HRESP[1]	HRESP[0]	Response	Description
0	0	OKAY	When **HREADY** is HIGH this shows the transfer has completed successfully. The OKAY response is also used for any additional cycles that are inserted, with **HREADY** LOW, prior to giving one of the three other responses.
0	1	ERROR	This response shows an error has occurred. The error condition should be signalled to the bus master so that it is aware the transfer has been unsuccessful. A two-cycle response is required for an error condition.
1	0	RETRY	The RETRY response shows the transfer has not yet completed, so the bus master should retry the transfer. The master should continue to retry the transfer until it completes. A two-cycle RETRY response is required.
1	1	SPLIT	The transfer has not yet completed successfully. The bus master must retry the transfer when it is next granted access to the bus. The slave will request access to the bus on behalf of the master when the transfer can complete. A two-cycle SPLIT response is required.

Figure 2.3: A table from the AMBA specification.

Usually, each possible value has a specific meaning and effect on the operation of the protocol. In these cases, a table such as the one in Figure 2.3 will describe all the possible values of the signal, and its meaning and usage. Tables are largely used to summarize a set of possible modes for a given feature in the protocol.

2.1.3 Text

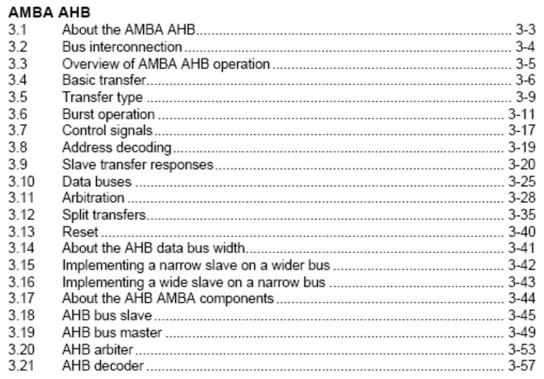
The above artifacts and others which are less widely used are always accompanied in an informal specification by text. Text in an informal specification fills many roles: it introduces features and provides motivation, it summarizes sections, describes causality in the protocol, and it provides information on which devices are responsible for which signals. In terms of describing the functioning of the protocol, however, the most important role of text is in describing nuances and resolving ambiguities that are introduced by other artifacts. All of the other artifacts that appear in the specifications have severe limitations on what they may express. Text is able to clarify areas which they have difficulty describing.

2.1.4 Structure

The final feature of informal specifications we will discuss is their structure. There are two aspects of this that we will discuss. The first is the way in which a protocol is divided into pieces to be described. The second is the way in which the artifacts described above are used together in an informal specification document to describe each of the pieces.

Figure 2.4 shows the table of contents entry for the chapter in the AMBA specification describing the AMBA Advanced High-performance Bus (AHB). The first three sections (3.1-3.3) of that chapter provide an overview of the bus and the devices connected to it. Then the sections on the bus functionality begin. 3.4 introduces the basic transfer and how wait states and pipelining affect it. 3.5 introduces the different types of transfers, including single transfers, burst transfers, placeholder transfers when the master is busy, and idle transfers when there is no data to be transmitted. These type of sections continue until section 3.14. 3.14-3.16 describe some details for implementation of the AHB, and the final five sections describe the devices that may be connected to the AHB.

For our purposes, we care most about the sections that describe the bus functionality (3.4-3.13). These sections describe the bus protocol incrementally, beginning with its most basic operation. Each subsequent


AMBA AHB

3.1 About the AMBA AHB 3-3
3.2 Bus interconnection 3-4
3.3 Overview of AMBA AHB operation 3-5
3.4 Basic transfer 3-6
3.5 Transfer type 3-9
3.6 Burst operation 3-11
3.7 Control signals 3-17
3.8 Address decoding 3-19
3.9 Slave transfer responses 3-20
3.10 Data buses 3-25
3.11 Arbitration 3-28
3.12 Split transfers 3-35
3.13 Reset 3-40
3.14 About the AHB data bus width 3-41
3.15 Implementing a narrow slave on a wider bus 3-42
3.16 Implementing a wide slave on a narrow bus 3-43
3.17 About the AHB AMBA components 3-44
3.18 AHB bus slave 3-45
3.19 AHB bus master 3-49
3.20 AHB arbiter 3-53
3.21 AHB decoder 3-57

Figure 2.4: Table of contents for the AMBA AHB specification.

section then extends or modifies the protocol as introduced in the previous sections in order to add a new piece of functionality.

To show how the artifacts described above are used in one of these sections to convey the material, we will walk through the beginning of the "Basic Transfer" section in the AMBA document (section 3.4). Section 2.1.1 began this process, describing how the timing diagram for the basic transfer (Figure 2.1 is used in that section. That diagram is repeated in Figure 2.5, with shading that we will describe later.

Figure 2.6 shows two more timing diagrams, in the order they are presented in the AMBA document. Sequences of timing diagrams such as this are common in specifications. Even without the accompanying text explaining the nuances of these diagrams a lot of information can be gathered from the sequence. Each new diagram builds on the previous ones by adding detail or functionality.

Figure 2.6a shows that in some cases, the data phase of a transfer may be extended to span multiple clock cycles. The areas in the diagram with solid outlines are identical to the portions shaded the same way in the basic transfer shown in Figure 2.5. The portion of the diagram surrounded by dotted lines (beginning in the third clock cycle) shows the part that has been added in this step.

In the data phase shown in this diagram the signal HREADY takes a logic LOW value instead of the HIGH value shown in the basic transfer. The data phase in this diagram is extended by two clock cycles until HREADY returns to a high value, at which point the transfer continues as normal. The surrounding text explains that the slave device drives HREADY low in this example to tell the bus master it needs more time to perform the request. Such an interaction is called a "wait state". The text also explains that wait states may be repeated until the slave device signals the master to continue by driving HREADY high.

The diagram in Figure 2.6b demonstrates a sequence of transfers. It shows how transfers in sequence are pipelined: each transfers address phase occurs concurrently with the previous transfer's data phase. This diagram also demonstrates an interesting effect that appears when the previous diagram's wait states are combined with this one's pipelining. When the data phase of one transfer is extended by the slave device, the address phase of the next transfer is extended as well. Note that this conflicts with the original stated parameters of the address phase, which the text previously stated lasted one clock cycle.

The next artifact that appears in the specification is the table shown in Figure 2.7. This introduces the four types of transfers possible in the AMBA AHB, as shown in the timing diagram in Figure 2.8. Notice in the timing diagram that the HTRANS[1:0] signal has taken the place of the previously underspecified control block. The text associated with the diagram describes each of the five transfers in the diagram.

Timing diagrams throughout the AMBA document are presented in similar contexts. This example demonstrates three points about specification documents:

1. The specification is introduced in small pieces; first the basic operation is demonstrated, then the document adds functionality to it. This is done either by introducing new features or by refining

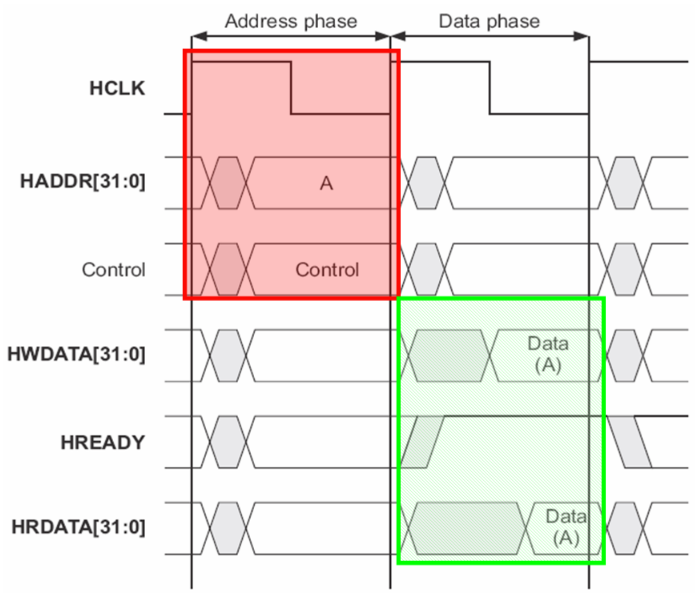


Figure 2.5: Timing diagram for basic read and write transfers in AMBA AHB (shading added).

previous examples.

2. A large portion of the specification can be learned by examining the sequence of timing diagrams alone. Each of these demonstrates a fragment of the protocol that builds on pieces previously shown in other diagrams. New fragments can be identified by eliminating those portions of the diagram which are the same as in previous diagrams.

3. The text surrounding the diagrams conveys clarification, motivation, and causality details. These are essential particularly for conveying the operation of the protocol to a human, and a correct model of the system cannot be constructed without the information they contain.

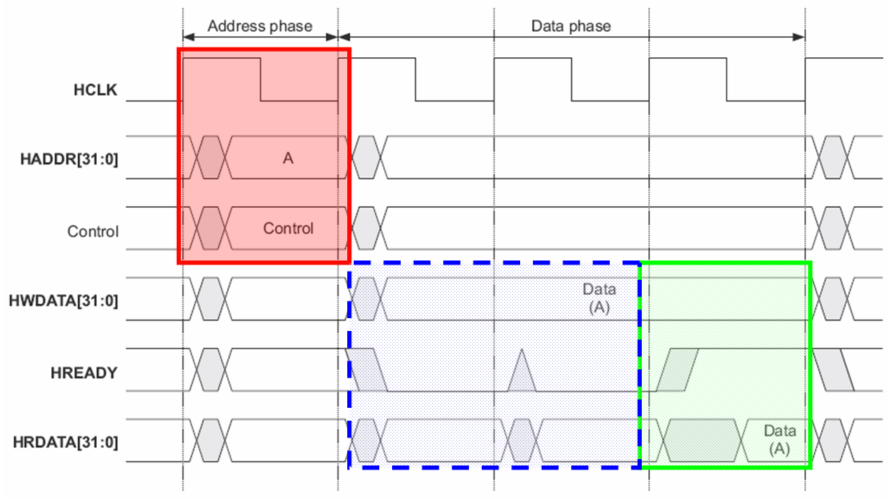
2.2 Proposed Workflow

The current method for creating formal verification artifacts requires a significant amount of human effort. Verification engineers must interpret and understand the informal specification of a design. Then, they have to use that understanding to create each formal artifact by hand.

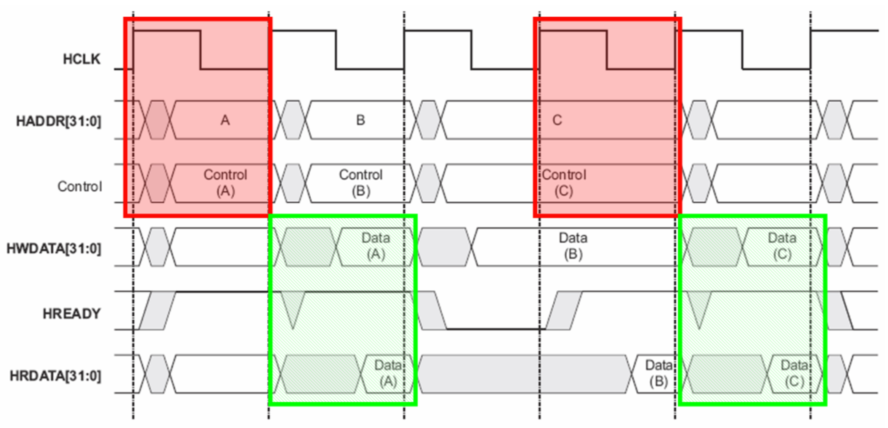
The long-term goal of this project is to partially automate that process. We imagine a verification process with three distinct stages. In the first, a tool would examine the informal specification and extract a skeleton formal specification from it. In the second, verification engineers would compare that skeleton to the informal specification, refining and filling in sections to create a full formal specification. We will refer to this as both the "formal specification" and "intermediate representation". In the third and final stage, a set of software tools would extract from that formal specification the actual verification artifacts. Each of these stages will be described in further detail below.

2.2.1 Automatic Protocol Skeleton Extraction

In Section 2.1, we showed the typical artifacts and structure found in an informal specification. The way in which certain artifacts are used and the document is structured leads us to believe that a machine

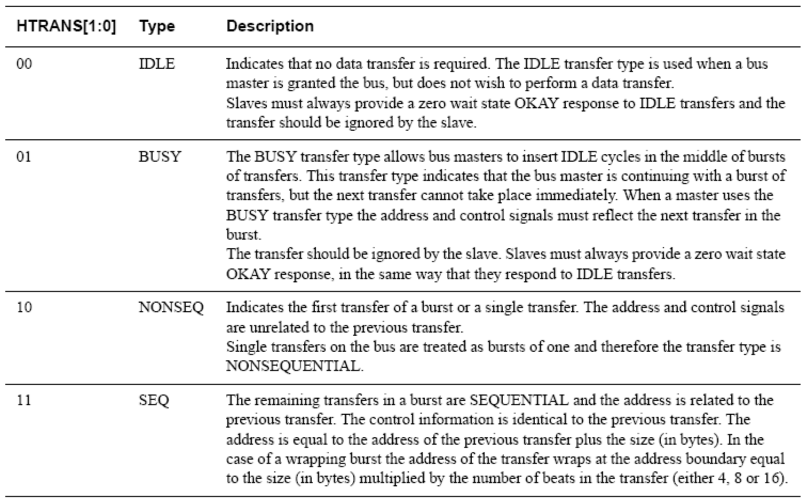


(a) Waitstates



(b) Multiple transfers and Pipelining

Figure 2.6: Timing diagrams showing wait states (Figure (a)) and pipelining (Figure (a)) in AMBA AHB.



HTRANS[1:0]	Type	Description
00	IDLE	Indicates that no data transfer is required. The IDLE transfer type is used when a bus master is granted the bus, but does not wish to perform a data transfer. Slaves must always provide a zero wait state OKAY response to IDLE transfers and the transfer should be ignored by the slave.
01	BUSY	The BUSY transfer type allows bus masters to insert IDLE cycles in the middle of bursts of transfers. This transfer type indicates that the bus master is continuing with a burst of transfers, but the next transfer cannot take place immediately. When a master uses the BUSY transfer type the address and control signals must reflect the next transfer in the burst. The transfer should be ignored by the slave. Slaves must always provide a zero wait state OKAY response, in the same way that they respond to IDLE transfers.
10	NONSEQ	Indicates the first transfer of a burst or a single transfer. The address and control signals are unrelated to the previous transfer. Single transfers on the bus are treated as bursts of one and therefore the transfer type is NONSEQUENTIAL.
11	SEQ	The remaining transfers in a burst are SEQUENTIAL and the address is related to the previous transfer. The control information is identical to the previous transfer. The address is equal to the address of the previous transfer plus the size (in bytes). In the case of a wrapping burst the address of the transfer wraps at the address boundary equal to the size (in bytes) multiplied by the number of beats in the transfer (either 4, 8 or 16).

Figure 2.7: Table of transfer types from the AMBA specification.

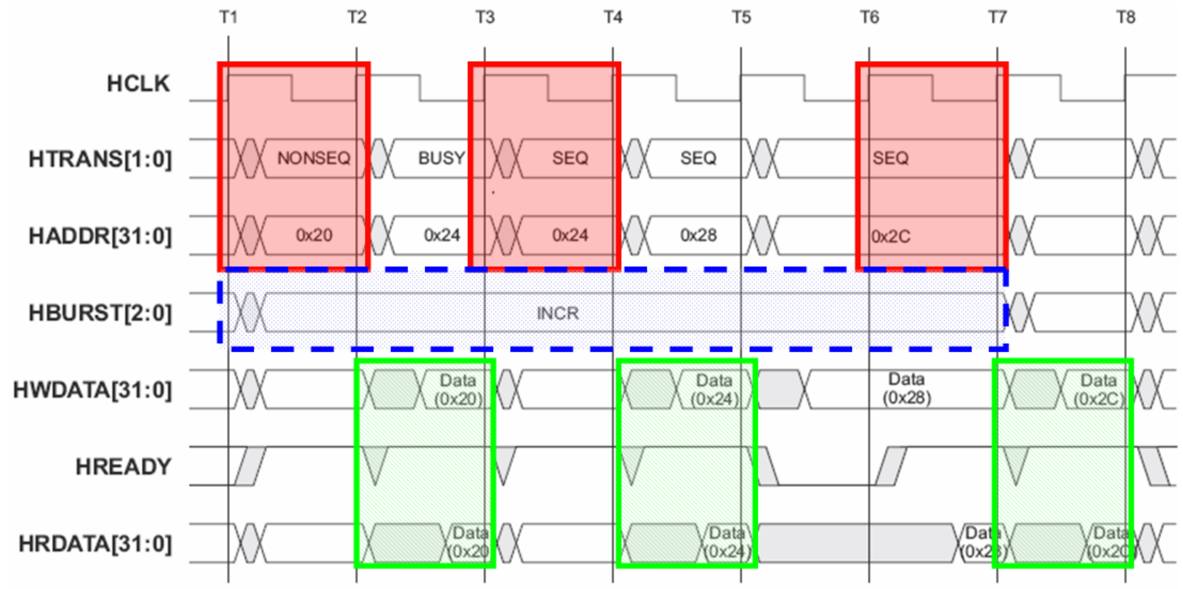


Figure 2.8: Transfer types in AMBA AHB.

could mine quite a bit of information from the specification document. In particular, we believe that by examining sequences of timing diagrams, such as the one in Section 2.1.4, a software tool could extract a lot of information about the functionality of a protocol.

We envision that this kind of information could be used by that same tool to construct the "skeleton" of a formal specification. The tool could begin creating the formal specification, based only on the information it knows. A sophisticated tool could even mark in the skeleton those places where it detected ambiguities or missing information.

With an intermediate representation for the formal specification such as the one we are proposing, this specification skeleton could be constructed to closely mirror the informal specification. For example, the skeleton for the AMBA specification could be organized like the sections in the informal document. The tool could place the information gathered in each section of the informal specification in the corresponding section of the formal specification.

We do not presume that such a tool would be able to extract a complete and fully accurate formal specification. Such a task would require far more sophisticated natural language processing and reasoning capabilities than the current state of the art would allow. However, we feel that automating even a small portion of this process should reduce the burden on the verification engineers.

2.2.2 Completion of Full Intermediate Representation

Since the output of the skeleton extraction tool would not be a complete formal specification, in the second stage engineers would have to fill in the gaps and repair inconsistencies in the formal specification skeleton. In this stage, verification engineers would systematically compare the informal specification document to the formal specification. Any missing information would be filled in, any ambiguities would be resolved, and any errors would be fixed. This process would be greatly facilitated by two things: a very detailed skeleton, and an intermediate representation which facilitated comparison to the original informal specification.

Filling in the protocol skeleton would be greatly facilitated by two kinds of detail in the first stage's output. First, and most obviously, the more accurate information the skeleton extraction tool is able to gather, the better. Over time, we imagine that this tool would get more sophisticated. While initially it would probably only be able to extract information from certain sequences of timing diagrams, over time it might be able to handle a wider variety of artifacts, and do some limited text processing to make its output more accurate.

The second key detail is information the tool does *not* know. This can be at least as valuable as the first kind. With reliable and accurate information about ambiguous parts of the skeleton or pieces that are missing, the engineer's job in the second stage would be much easier.

A high level of detail is important, but it has the potential to obscure the connection to the original informal specification. For this reason, the formal specification should be structured in such a way that it is easy to compare to the informal specification. This means it must be laid out in a similar manner, and matching sections between the informal and formal specifications must contain similar information. This means that any details provided in any section of the formal specification must be understandable given the information provided in the matching section of the informal document.

One final thing to note about the second stage: while we present it here as a separate process that takes place after the first stage, in practice it might prove more efficient and effective if it were done as part of the first stage. At run-time the skeleton extractor could notify a human of each ambiguity or hole in the skeleton being created. That person could then opt to fill in that portion of the skeleton immediately. Providing that kind of information to the skeleton extractor might make it more effective on later parts of the informal specification.

2.2.3 Extraction of Verification and Validation Artifacts

The final result of the second stage would be a full formal specification of the protocol contained in the informal document. Such a representation would unambiguously define the functioning of the protocol. It should be possible to create a set of tools which each extract verification artifacts from that formal specification. We imagine a given verification team would have a set of such tools that matched each of the verification methods they used. For example, a given team might use a tool to extract testbenches and a tool to extract RTL assertions on each of the formal specifications they created.

While the ability to extract these artifacts directly is a huge benefit of the intermediate representation, there is another benefit of that representation in this stage and beyond. Until a verification team is reasonably confident in their verification artifacts, each failed test must be examine to determine if it is a problem with the Device Under Test (DUT) or the verification artifact. In many cases, early failures occur because the verification artifact does not accurately reflect the functional description in the informal specification. If a team were following this proposed workflow, this comparison could be conducted as a comparison between the full formal specification and the informal specification.

Chapter 3

Representing Timing Diagrams

Since timing diagrams are such an important part of informal specifications, it is imperative that any other representation of those specifications provide an adequate means of representing timing diagrams. The basic building blocks of a specification in our language are structures we call “blocks”, which represent small pieces of timing diagrams. These can be composed to represent the type of timing diagrams that are common in protocol specifications. In designing the syntax and semantics of our blocks, we have considered the characteristics of the timing diagrams found in protocol specifications.

3.1 Characteristics of Protocol Timing Diagrams

We studied timing diagrams as they appear in protocol specifications. Through this process, we identified three characteristics of timing diagrams that we feel must be supported by any timing diagram representation. These characteristics are: their representation of both sequence and synchronization of signals, their compositional nature, and the way in which they are divided into logical blocks. In the following sections, we will examine each of these as they appear in the timing diagrams of the AMBA specification [2].

3.1.1 Sequence and Synchronization of Signals

One key aspect of timing diagrams is the type of information they are able to represent. Each timing diagram in a specification represents a single example of an interaction that is allowed by the protocol. It shows a sequence of values on each of a set of signals.

One way of thinking about a timing diagram’s set of signals is as parallel lines of execution. With such a representation, it becomes easy to talk about the sequence of values on a single signal; the value of a signal can simply be represented as a function from a point in time to a value. This technique works whether the signal in question is a single-bit signal or a bus, where the value of a set of related signals can be encoded as an integer or character.

However, timing diagrams show a set of such signals because the interaction between parallel signals is just as important. The values on each signal must be synchronized with those on other signals. For example, in the basic transfer shown in Figure 3.1, the “A” value on the HADDR signal must be synchronized with the "Control" value on Control and one cycle of HCLK. In addition, the end of the value “A” on HADDR must be synchronized with changes on all three of HWDATA, HREADY, and HRDATA.

Adding synchronization information to a “parallel execution” type of timing diagram model requires a difficult tradeoff between the granularity of control available and the syntactic overhead in describing the synchronization. We feel that other work which proposes alternative representations for timing diagrams falls too far on either end of this spectrum. This will be discussed in the rest of this section.

One proposed solution is to divide a timing diagram into equal-length time quanta. Within each quantum, the values on all signals must be held constant, but at the border of each slice, they may change. Oliveira

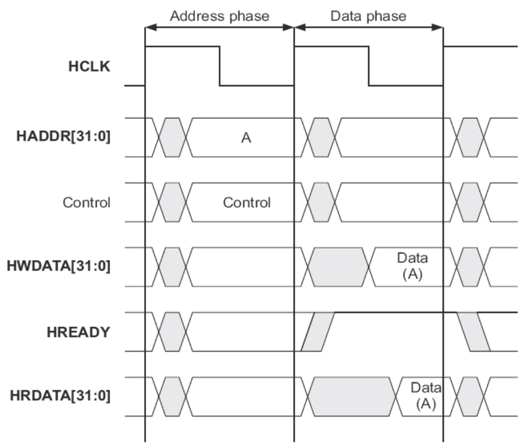


Figure 3.1: Timing diagram for basic read and write transfers in AMBA AHB.

and Hu use this technique to synchronize their language [8]. Each "character" in their regular expression language represents a value on a set of signals that is held for one clock cycle. This means that signal values which persist for any period shorter than a clock cycle, such as the "unstable" (shaded) value on the HRDATA signal in Figure 3.1, cannot be represented in their language.

A simple solution to this is to divide time into smaller quanta. However, such a solution does not solve another drawback of this scheme: in many cases, it makes the resulting specification too rigid. For example, the AMBA specification does not explicitly specify the duration of the unstable value on the HRDATA signal; it simply specifies that HRDATA will be unstable at the beginning of the Data phase, but will be stable at the end of that clock cycle. An actual implementation of the AMBA specification would need to explicitly define the maximum length of this unstable region, but the protocol specification should not have to be so concrete. Quantization of time in this manner does not give fine enough control over synchronization because it loses an abstraction of time that timing diagrams provide.

Fisler [4] proposed a solution closer to the other end of the spectrum. Her language uses regular expressions similar to Oliveira and Hu's for specifying the sequence of values on a signal, but does not specify the relative duration of those signals. Synchronization between parallel signals is accomplished through separate synchronization annotations. This is true even for synchronizing events with the clock.

Unfortunately, though this approach does allow much more control over synchronization, it makes specification very complicated. Even very simple diagrams such as the basic AMBA transfer (Figure 3.1) require many language constructs to represent. For example, the AMBA basic transfer is (mostly) defined by the "2D regular expression" in Figure 3.2.

There is an even larger difficulty than the sheer amount of text required for such a simple case. The separation of the sequencing and synchronization aspects of timing diagrams makes it difficult to develop an intuitive understanding of the example in Figure 3.2. This solution preserves the granularity of control and the abstraction operations inherent in graphical timing diagrams, but it loses the succinctness and intuitive expression of timing diagrams that Oliveira and Hu's work preserves.

3.1.2 Compositional Nature of Timing Diagrams

The second major feature of timing diagrams that we considered is the way in which multiple diagrams can be composed together into larger ones. This feature can be seen by comparing the timing diagram for the basic transfer in AMBA (Figure 3.3) to a timing diagram for a burst (Figure 3.4). The solid outlined areas of the burst diagram show the portions of that diagram which match the basic transfer diagram. The areas

HADDR $= sd\,;\, Addr_1 \,@\, sd^*$ [* allows multiple transfers]
HWRITE $= sd\,;\, L \,@\, sd^*$
HRDATA $= sd^*\,;\, Data_1 \,@\, sd$
HREADY $= sd\,;\, H\,;\, L\,;\, H \,@\, sd$

Event(HADDR_{T1}, HWRITE_{T1}, HREADY_{T1})
Event(HADDR_{T2}, HWRITE_{T2}, HREADY_{T2})
Event(HRDATA_{T2}, HREADY_{T4})
Event($\text{HRDATA}_{T1-first}$, $\text{HREADY}_{T3-first}$)
Order(HRDATA_{T1}, $\text{HREADY}_{T3-last}$)

ClockSignal(CLK)
GlitchTrans(HREADY_{T1})
Constrain($\text{HADDR}_{T1-first}$, $\text{HADDR}_{T1-last}$, $\text{CLK} = H$)
[similar constraints for remaining glitch regions]

Figure 3.2: 2D Regular expression [4] for the AMBA basic transfer shown in Figure 3.1.

outlined with dotted lines mark new pieces, each of which could have been represented as a stand-alone timing diagram.

3.1.3 Logical Divisions of Bus Protocol Timing Diagrams

The final feature of timing diagrams that informed the design of our timing diagram representation is the way in which they can be divided into logical blocks. Certain sets of signals within a timing diagram will often be synchronized with each other. This allows a timing diagram to be partitioned into rectangular sections, each of which contains a synchronized sequence of values on each of the signals within. Examples of these logical blocks are marked on Figure 3.3 and Figure 3.4. In the basic transfer diagram, the authors of the specification have even labeled the logical blocks for us: one is referred to as the "Address phase", and the other as the "Data phase".

Logical blocks have very clearly synchronized beginnings and endings; however, within the block synchronization is less important. For example, the “Data phase” in the basic transfer diagram (Figure 3.3) very clearly begins and ends at the same time as the beginning and end of the second clock cycle, but the values of the HWDATA and HRDATA signals during that phase are not as clearly defined. The exact point at which they become stable is not synchronized with any events shown in the diagram.

It should be noted that we have observed this logical block feature in all of the bus protocol specifications we have examined. We do not, however, presume that it is present in all applications for which timing diagrams are used. This may mean that some other representation for timing diagrams may be more appropriate than ours in other domains.

3.2 Our Method: Blocks

We have taken a slightly different approach to the problem of representing timing diagrams from most of the established literature. We have chosen to focus on the logical block structure we have found in bus protocol timing diagrams, and have built a language which allows recursive subdivision of timing diagrams into these types of logical blocks.

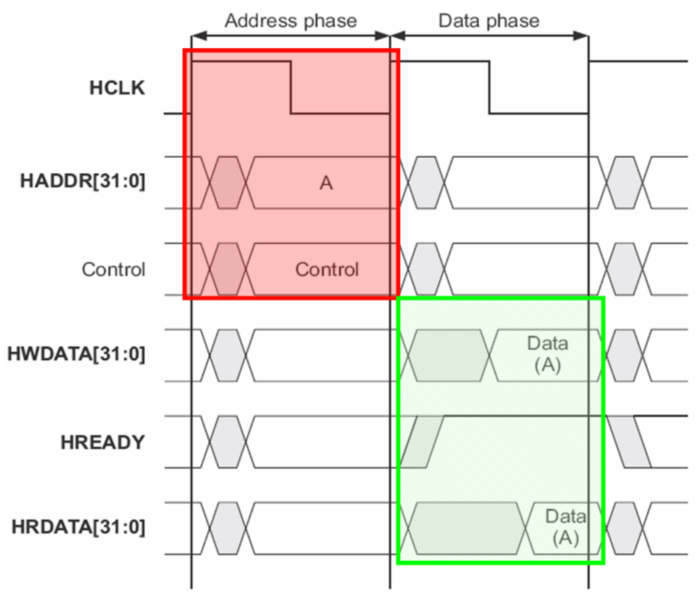


Figure 3.3: Timing diagram for basic read and write transfers in AMBA AHB. Shading added to show logical blocks.

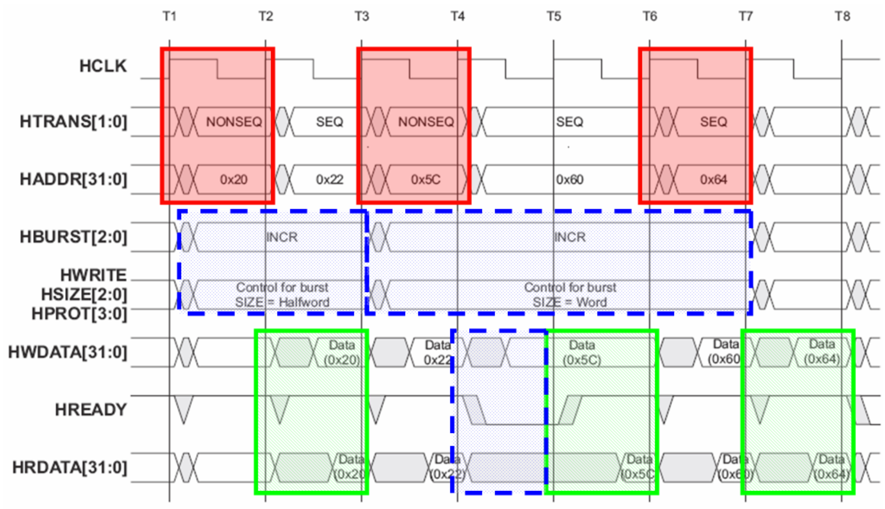


Figure 3.4: Timing diagram for an undefined-length burst in AMBA AHB. Shading added to show logical blocks.

```
//basic clock cycle
block clock() {
	CLK = 1, CLK = 0;
}

//address/control
block haddrctrl(value addr[31:0],block control) {
	clock;
	HADDR[31:0] = addr[31:0];
	control;
}

//data
block data(value wdata[31:0],value rdata[31:0]) {
	clock;
	HREADY = 1;
	HWDATA[31:0] = unstable as wdsetup, HWDATA[31:0] = wdata[31:0];
	HRDATA[31:0] = unstable as rdsetup, HRDATA[31:0] = rdata[31:0];
}

//simple transfer
block simpletrans(value addr[31:0],value wdata[31:0],signal
rdata[31:0],block control) {
	haddrctrl(addr[31:0],control),data(wdate[31:0],rdata[31:0]);
}
```

Figure 3.5: Specification for the basic transfer of Figure 3.1 in our language.

3.2.1 Syntax and Informal Semantics

The fundamental syntactic structure for defining fragments of timing diagrams is what we refer to as a *block*. A few examples of these blocks can be seen in Figure 3.5, which contains the representation in our language of the basic transfer shown previously in Figure 3.1. A block consists of the keyword **block**, followed by an identifier for the block and a list of arguments. The body of the block is defined between curly braces, and consists of a number of *sequences*, each of which is terminated by a semicolon. Each sequence consists of one or more blocks, separated by commas.

Conceptually, our blocks correspond exactly to the logical blocks described in Section 3.1.3. The clock block corresponds to a single clock cycle in Figure 3.1. Similarly, the haddrctrl block corresponds to the address phase, and the data block corresponds to the data phase. The simpletrans block ties them all together, defining a sequence in which there is an address phase and then a data phase.

Like the logical blocks described above as part of timing diagrams, the beginning and end points of our blocks are very clearly defined. However, all events in the middle of a block can occur at any time between those two points. In terms of the syntactic structures in Figure 3.5, this means that all sequences within a block must begin simultaneously and end simultaneously. Each sequence within a block describe a series of sub-blocks which occur sequentially. Each block in a sequence begins immediately after the block before it ends, but the length of each of the blocks in the sequence is not defined.

To more clearly define the semantics of blocks, we will examine some of the blocks in Figure 3.5 in more detail. First, the clock block is the simplest of the example blocks. It contains only one sequence, and defines only one signal. It simply defines a clock cycle, in which the CLK signal is first high, and then low. It says nothing about the duration of the high or low value, nor does it describe the relative lengths of either portion of the clock cycle. We will describe later how this type of information might be added to this block description.

The next block, the haddrctrl block, is a bit more complicated. The first two sequences are rather simple; the previously defined clock block is used to ensure every event in the haddrctrl block occurs within one clock cycle, and the sequence HADDR[31:0] = addr[31:0] specifies that for the duration of that clock cycle the HADDR signal should have the value "addr". "addr" is a signal value which is an argument to the block. Exposing this value argument outside of the haddr block will allow later blocks which use the haddrctrl block to constrain its behavior. Note that the statement "HADDR[31:0] = addr[31:0]" is what we refer to as an *atomic block*. This is the simplest form a block can take–one value held on one signal.

The "[31:0]" notation next to both HADDR and addr specify the bit width of each value. Both HADDR and ADDR are 32-bit values, with indices 0-31 referring to each bit. This notation is used both to specify the bit width of a value argument to a block, and to assign portions of values to different signals. In some protocols, a value argument may be split between multiple signals. In such a case, the specification in our language might have an atomic block of the form "WDATA[8:0] = HADDRDATA[31:24]". We have seen a need for this in protocols with multiplexed buses, such as the multiplexed address/data bus in Intel's 8088 specification [6].

The only remaining piece in the haddrctrl block is the sequence consisting entirely of the block "control", which is an argument to the haddrctrl block. Looking at the basic transfer diagram in Figure 3.1, we can see that the "Control" part of that diagram looks a little different: its name is not in bold, and it is not followed by a bit width in brackets like the other signals. The surrounding text tells us that is because "Control" is not a signal; it is a placeholder for a set of signals which will be defined later. The representation for this in our language is a block named "control" which is passed into the haddrctrl block as an argument.

The data block, which corresponds to the data phase in the basic transfer, introduces only a few new concepts. These can all be seen in the third and fourth sequences in that block. The first new thing is the use of the keyword **unstable**. This is a special value that can be assigned to a signal whose value may fluctuate before taking on a distinct value. This is represented in the graphical diagram as a shaded polygonal area in the middle of a signal's waveform.

One might wonder why we have chosen to explicitly specify the unstable areas in the data phase of the basic transfer when there are a number of other unstable regions shown in the diagram (for example, before and after the "A" value on the HADDR signal). This is because the unstable regions before the "Data(A)" values in the data phase are functionally different than the other unstable regions in the diagram; this is suggested in the diagram by the relative length of those unstable regions. The other unstable regions are what are refered to as *glitches*: momentary fluctuations in a signal as it changes from one value to another. The unstable regions at the beginning of the data phase, however, are not glitches; they are the *setup periods* for the read and write data on the bus. Because of the delays involved when accessing data from most devices (such as RAM chips), the data for a transfer is unlikely to be available at the very beginning of the data phase. Therefore, the value of the HWDATA and HRDATA signals will fluctuate as the data is retrieved.

Glitches are relatively unimportant events; because of the complicated logic driving most signals and the time it takes for an electrical signal to propogate through that logic (*propogation delay*), most signals will glitch when their values change. A signal may even glitch when its values do not change, if the internal state of the device driving that signal changes. It is usually more important in a specification to explicitly state when a signal will not glitch, as such a signal (referred to as *clean*) is not the typical case. Ensuring a clean signal typically requires extra components in the design of the circuit to smooth fluctuations.

Setup times are different. They are based on the performance of the device driving the data on the bus. Specifications usually impose very strict time constraints on the length of these periods; a data signal which is unstable past the allowed setup time may be unstable when that signal's value is read. Such an occurence can lead to extremely unpredictable system behavior.

Because protocol designers often add timing constraints to setup periods, we have chosen to name both of the setup periods in the data block. This is by using the keyword **as** after the definition of those atomic blocks. These names are intended to make it easier to add timing constraints to those blocks at a later point.

The final block in the example, called simpletrans, demonstrates how higher-level blocks may be composed.

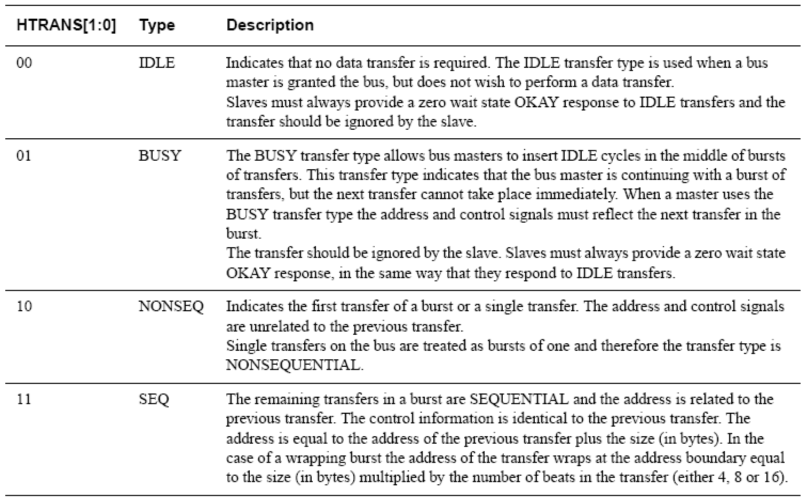
$ex ::= bl\ ex \mid tt\ ex \mid \epsilon$	(*expressions*)
$bk ::= \textbf{block}\ id\ (al)\ \{\ sq\ \}$	(*block definition*)
$al ::= ar, al \mid \epsilon$	(*argument list*)
$ar ::= \textbf{block}\ id \mid \textbf{value}\ si \mid id\ id$	(*argument declaration*)
$sq ::= bl;$	(*sequences*)
$bl ::= be,\ bl \mid \epsilon$	(*block list*)
$be ::= nb \mid id \mid ib$	(*block expression*)
$nb ::= id(nl)$	(*named block*)
$nl ::= na, nl \mid \epsilon$	(*named block argument list*)
$na ::= id \mid lv$	(*named block argument*)
$ib ::= si = v \mid si = v\ \textbf{as}\ id$	(*inline block*)
$si ::= id \mid id[n : n]$	(*signal identifier*)
$v ::= si \mid n$	(*values*)
$tt ::= \textbf{tabletype}\ si\ \{el\}$	(*tabletype definition*)
$el ::= ed,\ el \mid \epsilon$	(*tabletype enumerated value list*)
$ed ::= id = n$	(*tabletype enumerated value*)
$lv ::= \textbf{unstable} \mid \textbf{dc} \mid n \mid id$	(*logical values*)
$n \in \mathbf{N}$	(*numbers*)
$id \in \{a - zA - z\}\{a - zA - Z0 - 9\_\}*$	(*identifiers*)

Figure 3.6: Grammar for our block language.

Syntactically, it is very simple, but it exposes some of the advantages of our block syntax. The simpletrans block defines the whole basic transfer in Figure 3.1, which consists of an address phase followed by a data phase. In a block, this behavior can all be defined within one sequence, even though the address and data phases do not contain all of the same signals. This means that all sequences in the address phase end immediately before the beginning of all the sequences in the data phase. Any signals which are not defined in one of the two blocks have unconstrained values for that portion of the sequence. The semantics of our blocks allow us to very intuitively define the basic transfer as an address phase followed by a data phase.

The grammar for our language is shown in Figure 3.6. There are a few things represented in the grammar which are not described above. The first of these is the *tabletype definition* (*tt*) expression. These are used to reproduce tables like those found in informal specifications, which define a set of possible values for a signal and the use or meaning for each. An example of this is shown in Figure 3.7, and the associated tabletype in our language is shown in Figure 3.8. In our language, these function like enumerated types in C++ or Java. In blocks, they are treated just as signals with the specified bit width, but they may be assigned the constant values defined in their associated *enumerated value list*. In the grammar, the $ar \rightarrow id\ id$ production for argument delcarations is used for declaring arguments of a defined tabletype.

Another syntactic element in the grammar not shown in the above examples is the value "**dc**", which may be assigned to a signal. This is used to define a portion of a signal whose value doesn't matter, but is not unstable. We expect this to be used infrequently; typically exposing such an area as a value argument to a block accomplishes the same thing. However, we have used "**dc**" for one case: a busy transfer in the AMBA specification. These are exactly like normal transfers, but the control signals are different, and the address and data don't matter. "**dc**" is used in this case to turn the already defined simple transfer into a



HTRANS[1:0]	Type	Description
00	IDLE	Indicates that no data transfer is required. The IDLE transfer type is used when a bus master is granted the bus, but does not wish to perform a data transfer. Slaves must always provide a zero wait state OKAY response to IDLE transfers and the transfer should be ignored by the slave.
01	BUSY	The BUSY transfer type allows bus masters to insert IDLE cycles in the middle of bursts of transfers. This transfer type indicates that the bus master is continuing with a burst of transfers, but the next transfer cannot take place immediately. When a master uses the BUSY transfer type the address and control signals must reflect the next transfer in the burst. The transfer should be ignored by the slave. Slaves must always provide a zero wait state OKAY response, in the same way that they respond to IDLE transfers.
10	NONSEQ	Indicates the first transfer of a burst or a single transfer. The address and control signals are unrelated to the previous transfer. Single transfers on the bus are treated as bursts of one and therefore the transfer type is NONSEQUENTIAL.
11	SEQ	The remaining transfers in a burst are SEQUENTIAL and the address is related to the previous transfer. The control information is identical to the previous transfer. The address is equal to the address of the previous transfer plus the size (in bytes). In the case of a wrapping burst the address of the transfer wraps at the address boundary equal to the size (in bytes) multiplied by the number of beats in the transfer (either 4, 8 or 16).

Figure 3.7: A table from the AMBA specification.

```
tabletype TransType[1:0] {IDLE = 00b,BUSY = 01b,NONSEQ = 10b,SEQ = 11b};
```

Figure 3.8: The **TransType** tabletype defined in our language. Corresponds to the table in the AMBA specification shown in Figure 3.7.

busy transfer. This can be seen on line 74 in Appendix A.

One element which is not represented in either the grammar or the examples above is the way in which values may be represented. The grammar states that values may be any natural number ($n \in \mathbf{N}$), but it does not state the notation for those numbers. We allow values to be written in decimal (ex: 49), binary (ex: 101101b), hexadecimal (ex: 31x), or as characters, where the value is equal to the character's 8-bit ASCII code (ex: '1'). These forms of notation are interchangable wherever the non-terminal n appears.

3.2.2 Formal Semantics

For defining the formal semantics of our block language, we use *Regular Timing Diagrams* (RTDs), as proposed by Amla et al. [1]. RTDs provide a way of expressing both synchronous and asynchronous timing diagrams with precise semantics. Amla et al. also provide a technique for converting RTDs into ∀FA automata and for using them in model checking. As automata are standard representations for validation and verification, this semantics gives us a good foundation for using blocks with mainstream verification techniques.

An RTD is a triple (WF, SD, CD), where:

- WF is a finite set of waveforms, each of which is a function from points on that waveform to values ($A : [0, n) \mapsto SV$). Values (SV) may be 0, 1, or X, where X denotes a "don't care", or variable, value. Points on a waveform are ordered, but have no other timing or value constraints. The initial point of some waveform A is always $(A, 0)$, and the endpoint is always $(A, length(A) - 1)$.
- SD is a set of *sequential dependencies* on the points of WF. A sequential dependency denotes a constraint on the amount of time between two points, which may be in different waveforms. Each of these is specified as $(A, i) \xrightarrow{[a,b\rangle} (B, j)$, where $a \in \mathbf{N}, b \in \mathbf{N} \cup \{\infty\}, 1 \leq a$ and $a \leq b$.

```
//basic clock cycle
block clock() {
      CLK = 1, CLK = 0;
}
```

(*clock block*)

$WF : \{CLK\}$

$\quad CLK : 0 \mapsto 1, 1 \mapsto 0, 2 \mapsto X$

$SD : \{\}$

$CD : \{\{(CLK, 0)\}, \{(CLK, 2)\}\}$

Figure 3.9: A block defining a clock cycle, and its corresponding RTD.

```
//address/control
block addr (value addr[31:0]) {
      clock;
      HADDR[31:0] = addr[31:0];
}
```

(*addr block*)

$WF : \{CLK, HADDR\}$

$\quad CLK : 0 \mapsto 1, 1 \mapsto 0, 2 \mapsto X$

$\quad HADDR : 0 \mapsto addr, 1 \mapsto X$

$SD : \{\}$

$CD : \{\{(CLK, 0), (HADDR, 0)\}, \{(CLK, 2), (HADDR, 1)\}\}$

Figure 3.10: A block defining the address phase of a transfer, and its corresponding RTD. Uses the clock block from Figure 3.9.

- CD is a set of *concurrent dependencies*. Each of these is a set of points which occur at the same time. Each concurrent dependency's set of points must be disjoint from that of all other concurrent dependencies in a given RTD. Each point may occur in only one concurrent dependency. The set of initial and final points of each waveform are predefined concurrent dependencies.[1]

Figures 3.9-3.12 show an example of expressions in our language translated into RTDs. We will describe the algorithm for doing this in the next few paragraphs, and the pseudocode for the algorithm can be found in Algorithm 3.

Since blocks are recursively composed of other blocks, the translation algorithm is recursive. Therefore, we begin our discussion of the algorithm by examining the base case for recursion, the atomic block (for example, "CLK = 0"). The RTD for this kind of block is very simple because its timing diagram is simple. The RTD has one waveform, with one point at which that waveform has the specified value (in this case, 0). In order to represent the fact that the value is held for some period of time, and to make later unification of sequences of blocks easier, we add a second point at which the waveform takes on a value of X. The final requirement to make this a complete RTD is to add the concurrent and sequential dependencies. While there are no sequential dependencies, RTDs are required to always have two concurrent dependencies, one at the beginning of all waveforms, and one at the end of all waveforms. Each of the two points is the sole point in one of those two concurrent dependencies.

We will now incrementally expand the complexity of this example to reveal the entire translation process.

```
//data
block data(value wdata[31:0]) {
       clock;
       HREADY = 1;
       HWDATA[31:0] = unstable as wdsetup, HWDATA[31:0] = wdata[31:0];
}
```

$$\begin{aligned}
&(\textit{data block})\\
&WF: \{CLK, HREADY, HWDATA\}\\
&\quad CLK: 0 \mapsto 1, 1 \mapsto 0, 2 \mapsto X\\
&\quad HREADY: 0 \mapsto 1, 1 \mapsto X\\
&\quad HWDATA: 0 \mapsto X, 1 \mapsto \mathit{data}, 2 \mapsto X\\
&SD: \{\}\\
&CD: \{\{(CLK, 0), (HREADY, 0), (HWDATA, 0)\}, \{(CLK, 2), (HREADY, 1), (HWDATA, 2)\}\}
\end{aligned}$$

Figure 3.11: A block defining the data phase of a transfer, and its corresponding RTD. Uses the clock block from Figure 3.9.

```
//simple transfer
block simpletrans(value addr[31:0],value wdata[31:0],signal
rdata[31:0]) {
       haddrctrl(addr[31:0]),data(wdate[31:0]);
}
```

$$\begin{aligned}
&(\textit{simpletrans block})\\
&WF: \{CLK, HADDR, HREADY, HWDATA\}\\
&\quad CLK: 0 \mapsto 1, 1 \mapsto 0, 2 \mapsto 1, 3 \mapsto 0, 4 \mapsto X\\
&\quad HADDR: 0 \mapsto \mathit{addr}, 1 \mapsto X, 2 \mapsto X\\
&\quad HREADY: 0 \mapsto X, 1 \mapsto 1, 2 \mapsto X\\
&\quad HWDATA: 0 \mapsto X, 1 \mapsto \mathit{data}, 2 \mapsto X\\
&SD: \{\}\\
&CD: \{\{(CLK, 0), (HADDR, 0), (HREADY, 0), (HWDATA, 0)\},\\
&\quad \{(CLK, 2), (HADDR, 1), (HREADY, 1), (HWDATA, 1)\},\\
&\quad \{(CLK, 4), (HADDR, 2), (HREADY, 2), (HWDATA, 2)\}\}
\end{aligned}$$

Figure 3.12: A block defining a simple transfer, and its corresponding RTD. Builds on the blocks in Figures 3.9-3.11.

Algorithm 1 Utility functions used by ConvertToRTD

```
function ChangePoint(waveform $W$, point $p$, value $v$) : waveform
    $F \leftarrow W$
    $F(p) \leftarrow v$
    return $F$
end function

function AddPoint(waveform $W$, value $v$) : waveform
    return $W \cup \{|W| \mapsto v\}$
end function

function GetSignals(block $b$) : {signal}
    $R \leftarrow \{\}$
    for all sequences $s$ in $b$ do
        for all blocks $c$ in $s$ do
            if $c$ is an atomic block, of the form $g = v$ then
                $R \leftarrow R \cup \{g\}$
            else
                $R \leftarrow R \cup$ GetSignals($c$)
            end if
        end for
    end for
    return $R$
end function
```

The next most simple case is a block which has only one sequence, which contains only atomic blocks referring to one signal. An example of this is the clock block in Figure 3.12. This case is almost equivalent to the concatenation of two atomic blocks; again, each atomic block in the sequence corresponds to a single point in the RTD's waveform. Since in this case multiple points are already being defined on that waveform, the X-transitions at the end of every sub-block can be removed. However, the final X-transition is still necessary. Initial and final concurrent dependencies are added to this example in the same way as in the previous one.

Adding concurrent sequences adds very little complexity to the resulting RTD. To do this, we need only to add additional waveforms for each additional signal, and add the beginning and end points of each of those waveforms to the initial and final concurrent dependencies. However, it should be noted that in order for the semantics to be unambiguous, no signal may appear in two parallel sequences. This is certainly a limitation of this translation algorithm, but it seems to also be a limitation of the block model. This limitation is discussed further in Section 3.2.3. So far, however, we have not encountered a case where it has obstructed our representation of a protocol specification in blocks.

The next step in increasing the complexity of the examples we consider is adding more complex sub-blocks. The data block shown in Figure 3.12 is an example of such a block. In order to understand how nested blocks are incorporated into a larger block's RTD, consider the case where a block is listed in a sequence with atomic blocks on both sides. Intuitively, we would like to inject the RTD semantics of the nested block into the RTD of theblock that contains it. With a few small modifications, that is exactly what can be done.

The nested block is first converted into a standalone RTD. All of its points must then be shifted to account for the points added for the previous blocks in the sequence. This process must be applied not only in the definition of the nested block's waveform, but also in its concurrent and sequential dependencies. Once this process is done, the nested block's sequential dependencies can be added as-is to the containing block's sequential dependencies. The concurrent dependencies can be similarly added to the containing block's, but any overlapping dependencies must be merged together (no point may occur in multiple concurrent

Algorithm 2 Appending one RTD to another

```
function AppendRTD(RTD $R1 = (W1, S1, C1)$, RTD $R2 = (W2, S2, C2)$) : RTD
    Precondition: $domain(W2) \subseteq domain(W1)$
    $newW \leftarrow W1$
    $newS \leftarrow S1$
    $newC \leftarrow C1$
    for all $w \in dom(W2)$ do
        $newW(w) \leftarrow$ ChangePoint($(newW(w), |newW(w)| - 1, W2(w)(0))$)
        for all $i \in dom(W2(w)), i > 0$ do
            $newW(w) \leftarrow$ AddPoint($newW(w), W2(w)(i)$)
        end for
    end for

    for all $sd = ((x, i) \xrightarrow{[a,b\rangle} (y, j)) \in C2$ do
        $k \leftarrow i + length(W1(x)) - 1$
        $l \leftarrow j + length(W1(y)) - 1$
        $newS \leftarrow newS \cup \{((x, k) \xrightarrow{[a,b\rangle} (y, l))\}$
    end for
    $newS \leftarrow W1 \cup newS$

    for all $cd \in C2$ do
        $newcd \leftarrow cd$
        for all $(x, i) \in cd$ do
            $newcd \leftarrow newcd \cup \{(x, i + length(W1(x)) - 1)\}$
        end for
        for all $c \in newC$ do                    ▷ merge CDs referring to the same point
            if $newcd \cap c \neq \{\}$ then
                $newC \leftarrow newC - \{c\}$
                $newcd \leftarrow newcd \cup c$
            end if
        end for
        $newC \leftarrow newC \cup \{newcd\}$
    end for
    return $(newW, S1 \cup newS, C1 \cup newC)$
end function
```

Algorithm 3 Translating a block into an RTD

```
1: function ConvertToRTD(block $block$) : RTD
2:     $W \leftarrow \{\}$                                                    ▷ Initialize
3:     $S \leftarrow \{\}$
4:     $C \leftarrow \{\}$
5:     for all $g \in$ GetSignals($block$) do
6:         $W \leftarrow W \cup \{s \mapsto \{0 \mapsto X\}\}$
7:     end for

8:     if $b$ is an atomic block, of the form $g = v$ then
9:         $W(g) \leftarrow$ ChangePoint($W(g), 0, v$)
10:        $W(g) \leftarrow$ AddPoint($W(g), X$)
11:    else

12:        $finalsigs \leftarrow \{\}$
13:        for all sequences $s \in block$ do                          ▷ Main RTD Construction Loop
14:            $p \leftarrow$ null
15:            $sW \leftarrow \{\}$
16:            $sS \leftarrow \{\}$
17:            $sC \leftarrow \{\}$
18:            for all blocks $b \in s$, in order do
19:                $R \leftarrow$ ConvertToRTD($b$)
20:                $(sW, sS, sC) \leftarrow$ AppendRTD($(sW, sS, sC), R$)
21:                $p \leftarrow b$
22:            end for
23:            $finalsigs \leftarrow finalsigs \cup$ GetSignals($p$)
24:            $W \leftarrow W \cup sW$
25:            $S \leftarrow S \cup sS$
26:            $C \leftarrow C \cup sC$
27:        end for
28:        for all $g \in$ GetSignals($block$) $- finalsigs$ do
29:            $W(g) \leftarrow$ AddPoint($W(g), X$)
30:        end for
31:    end if

32:    $bcd \leftarrow \{\}$                                          ▷ add concurrent dependencies
33:    $ecd \leftarrow \{\}$
34:    for all $g \in$GetSignals(block) do
35:        for all $cd \in C$ do
36:            if $(g, |W(g)| - 1) \in cd$ then
37:                $ecd \leftarrow ecd \cup \{cd\}$
38:                $C \leftarrow C - \{cd\}$
39:            end if
40:            if $(g, 0) \in cd$ then
41:                $bcd \leftarrow bcd \cup \{cd\}$
42:                $C \leftarrow C - \{cd\}$
43:            end if
44:        end for
45:    end for
46:    $C \leftarrow C \cup \{bcd, ecd\}$
47:    return $(W, S, C)$
48: end function
```

dependencies). In addition, the initial concurrent dependency of the block must be merged with the final one of the previous block in the sequence. The algorithm for the whole process can be seen in Algorithm 2.

The only remaining problem is the issue of areas in the sequence for which certain signals are undefined. A block in a sequence may specify signals which no other blocks in the sequence mention, but the containing block must correctly specify that signal's behavior for the entire duration of the block. Therefore, whenever a signal is mentioned in the current block that is not mentioned in the previous block, an additional point must be added to that block's waveform with a value of X to represent the unspecified period.

The ConvertToRTD function in Algorithm 3 contains the overall process for converting a block to an RTD. This function takes a block (either atomic or non-atomic) as input and returns an RTD (a triple (WF, SC, CD), described above). The function consists of three distinct phases: initialization (lines 2 - 7), construction of the waveforms (lines 8 - 31), and unification and creation of the concurrent dependencies (lines 32 - 46). Currently, blocks do not reaquire the creation of sequential dependencies, so every block's RTD has an empty set for SD.

The initialization phase is very simple. The three sets that will become the RTD's WF, SD and CD (W, S, and C, respectively) are all initialized to empty sets. Then, a waveform for each signal that is mentioned in the input block or its sub-blocks is added to W. Each of these initial waveforms contains only one point with value X. Throughout the function, the terminating X-transition will be maintained on each of the waveforms in W as new points are added.

The waveform construction phase is the most complicated portion of the algorithm. All of the waveforms are constructed incrementally by examining each sequence block-by-block. The only exception is the case where *block* is an atomic block. This is the base case for ConvertToRTD's recursion, and is handled by lines 8 - 10. An atomic block's waveform has two points, and concurrent dependencies at the beginning and end.

The case for more complicated blocks is on lines 12 - 31. The loop on line 13 iterates through each sequence in the block (order does not matter). The nested loop on line 18 iterates through each sub-block in a given sequence in order, incrementally constructing an RTD for the sequence. The code within that loop converts each sub-block into an RTD, and concatenates it to the RTD being constructed. At the end of the sequence loop, on lines 23 - 26, all of the signals that appear in the final block of s are recorded, and the RTD for s is added to the RTD in progress for the entire block.

The only thing left to mention about the non-atomic block case is that any signal which is not specified in the last block in a sequence must have an additional X-transfer. If this is not explicitly added, the RTD will not correctly represent that unspecified period at the end of the block. The variable $finalsigs$ is used to keep track of all the signals specified at the end of each sequence, and the loop at line 28 adds the X-transfers for any signals which are specified somewhere in *block*, but are not specified at its end.

Only one thing remains at the end of ConvertToRTD: creating the initial and final concurrent depencencies. This process seems relatively simple, but there is a constraint on the concurrent dependencies of an RTD; each point in the RTD appear in at most one concurrent dependency. The loop on lines 32-46 handles this task, adding each signal's initial and final points to the block's initial and final concurrent dependencies.

It was mentioned above that each sub-block of *block* is converted to an RTD and then appended to the working RTD; however, we did not adequately describe the how appending is done. The code for that function is in Algorithm 2. The process is relatively simple; the primary change that needs to be done is shifting all of the points in the second RTD to be after the existing waveforms in the first RTD. This must be repeated for the waveform, sequential dependency, and concurrent dependency portions of the RTD.

3.2.3 Restrictions on Parallel Sequences

Section 3.2.2 mentioned briefly that no pair of parallel sequences may contain blocks which assign values to the same signal. If this rule is not enforced, it complicates the simple semantics of blocks, and can lead to blocks with ambiguous or inconsistent semantics.

If we consider the case where two sequences assign values to the same signal, the second sequence read in must unify its assignments with the first signal's. This can be done in two ways. The second sequence may fit its assignments into undefined segments of the existing waveform, or it may unify its assignments with

```
block ambiguous() {
      SIG1 = 1A, SIG2 = 1B, SIG1 = 1C;
      SIG3 = 2A, SIG2 = 2B, SIG3 = 2C;
}
```

Figure 3.13: A block with ambiguous semantics.

matching areas in the existing waveform. Both options create a partial ordering on events across multiple sequences, which come from new cross-sequence dependencies. The syntax of blocks is intended to make it easy to see the ordering within a sequence; ordering across sequences could be very difficult to identify.

Even worse, these new dependencies would allow the definition of blocks with ambiguous or even inconsistent semantics. An example of an ambiguous block is shown in Figure 3.13, and some of the possible timing diagrams for that block are shown in Figure 3.14. Because the beginning and end points of the middle blocks of each of the sequences in the ambiguous block are not clearly defined (because of the semantics of blocks), the order of the "1B" and "2B" values on SIG2 is ambiguous. When these two values are equal, they may even overlap, as shown in Figures 3.14(c) and (d).

A block can have inconsistent semantics when two sequences assign conflicting values to the same signal at the same time. A simple example of this is shown in Figure 3.15a. Figure 3.15b shows a slightly more complicated example, where the inconsistency comes from conflicting assignments to the SIG2 signal at the end of the block. While all blocks could be checked for consistency at compile time, Figure 3.15c demonstrates that the inconsistency in a block can be complicated even further. These examples show only simple blocks; in some cases, an inconsistency might only arise in the case of a very specific composition. This would make unification an even more difficult task.

We enforce a restriction on parallel sequences to avoid all these ambiguity and inconsistency pitfalls. The restriction is that the sets of signals that parallel sequences assign values to must be disjoint. In the examples we have studied, however, this restriction has not been very limiting.

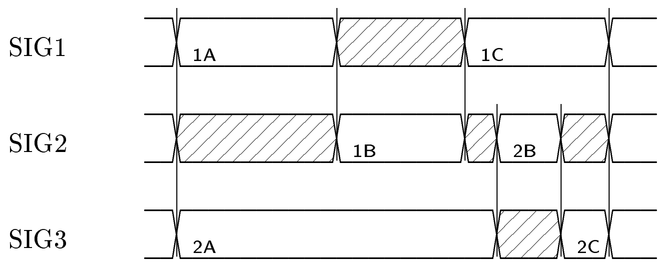
3.3 Future Work on Blocks

So far, we have been able to test the block model on many different examples, and have found that it supports representing protocol specifications very well. We have refined the block model somewhat through this process, and identified some possible extentions. Two of these were representing timing constraints in blocks and representing glitches. While we have identified mechanisms for both of these which seem to work well, we have not yet been able to investigate them enough to unequivocally recommend their use.

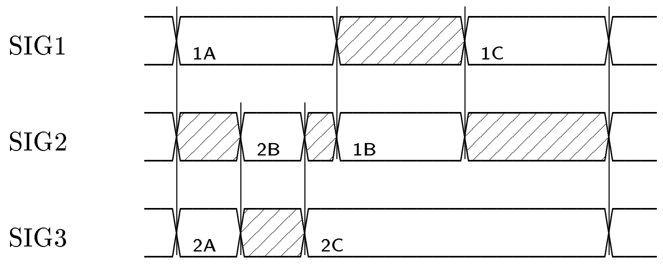
3.3.1 Adding Timing Constraints

Protocol specifications usually describe the logical functioning of the protocol in a way that is independent of the technology used to implement the protocol. This means that protocol specifications rarely give concrete timing constraints for the events in the protocol. Concrete implementations of those specifications, however, must give such consraints. For this reason, we would like our block model to allow timing constraints without requiring them, and allow them to be added to an existing block specification. We have identified a way of specifying timing constraints which seems to satisfy these requirements. However, our efforts so far have been focused on the AMBA specification, which does not provide timing constraints, so we do not feel we have conducted an adequate investigation of this technique's merits.

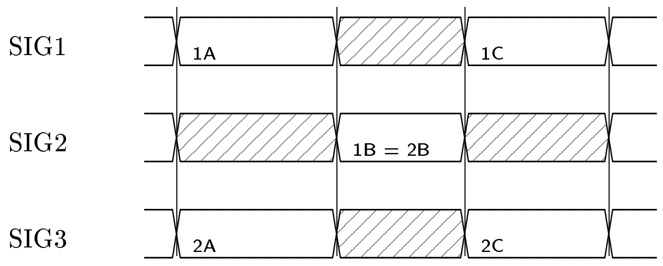
Our technique takes advantage of the fact that blocks have clearly defined beginnings and ends, and that everything within the block must occur between those two points. Our proposed technique adds a new type of block, which contains only a timing constraint. By using such a block as part of a sequence, a delay between two events can be specified. By using a timing block in parallel with another block, one can specify



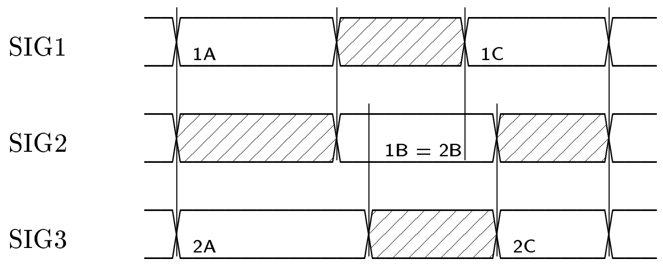
(a) Possibility 1



(b) Possibility 2



(c) Possibility 3



(d) Possibility 4

Figure 3.14: Four possible timing diagrams representing the block in Figure 3.13. (c) and (d) might be possible when 1B=2B.

```
block inconsistent1() {
      SIG1 = 1A;
      SIG1 = 2A;
}
```

(a) A simple inconsistent block

```
block inconsistent2() {
      SIG1 = 1A, SIG2 = 1B;
      SIG3 = 2A, SIG3 = 2B, SIG2 = 2C;
}
```

(b) A more complicated example

```
block inconsistent3() {
      X = 1A, Y = 1B, X = 1C;
      Y = 2A, X = 2B, X = 2C, Y = 2D;
      Z = 3A, Y = 3B, Z = 3C;
}
```

(c) An even more complicated example

Figure 3.15: Three blocks with inconsistent semantics.

the duration of that block. We have not decided exactly what types of timing constraints should be allowed; however, it seems that it would be necessary to allow ranges of times.

A good example of this would be fixing the length of the clock cycle in a clock block. By replacing the atomic blocks with more complicated blocks that contain both a timing constraint and the change in clock value, both of the segments of the clock block could have their durations fixed. While such an approach seems to require designing the clock block at the same time as specifying the timing constraints, the composition operators shown in Chapter 4 would allow this kind of timing constraint to be added after the fact.

3.3.2 Glitches and Clean Signals

Section 3.2.1 describes briefly the role of *glitches* in any digital circuit. Most sufficiently complicated digital circuits display glitching behavior when their internal state or inputs change; these fluctuations in output are a result of delays in the time it takes for a signal to propogate through the circuit. While in some applications glitches can be disasterous, most synchronous protocols and circuits allow them because they only read signal values on clock edges.

In the domain of bus protocols (which are almost always synchronous), glitches are usually the default case whenever the internal state of a device driving the bus changes. This means that any change in the values on the bus is usually preceded by a glitch (notice the glitches before every change in value in Figure 3.4). In addition, any signal which depends on the dynamic state of some component, may also exhibit a glitch even when its value does not change (as is the case for the HREADY signal in Figure 3.4.

In our work with the AMBA specification, we have not explicitly included glitches in our language's representation of the protocol. However, we have noticed that the places in which the specification shows a possible glitch always correspond to the boundary of a block in our language, and that those occasions where glitches do *not* occur on a block boundary are almost always special cases (such as the clock signal).

For this reason, it seems as though the semantics of our block language should allow glitches at any block boundary, unless otherwise specified. We have not yet decided on a syntax for representing boundaries which do not have glitches, but there are three options we have considered. The first is that a signal could be specified as *clean*, signifying that it is not allowed to glitch at any point. Second, an individual block could have annotations added to specify that none of the signals within it glitch at its boundaries. Finally, an individual boundary between two blocks (a comma in a sequence, or the beginning or end of a sequence) could be marked as non-glitching. We have not yet examined these options in depth, and for that reason, we ignore glitches for this version of the language.

Chapter 4

Composing Timing Diagram Fragments to Form a Specification

While the block language we have created seems to support specification of protocol timing diagrams very nicely, it doesn't do much towards our stated goal of allowing intermediate representations that are structurally similar to the specifications they represent. To achieve that goal, we need ways to compose our timing diagram blocks into full protocol representations. We have designed a mechanism based on existing work on Aspect Oriented Programming (AOP) for doing this that both allows the types of composition needed for this task, and which preserves the synchronization and sequencing characteristics of our block language.

4.1 Composition Requirements for Imitating Specification Structures

Composition of blocks needs to support the operations that are required to compose block fragments into full definitions for bus protocols. We have identified four features that are necessary to allow specification in our language to follow the structure of informal specifications. Each of these features is described in the following sections.

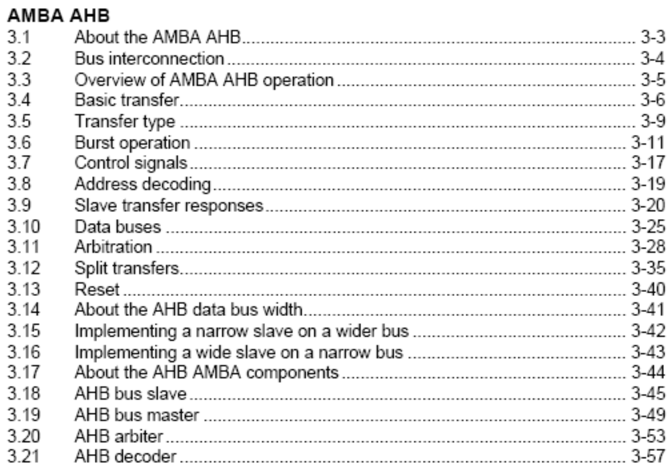
4.1.1 Extending the Block Structure for Composition

The first requirement we set for our composition language was that it should preserve the underlying block structure of the fragments being composed. Higher-level protocol fragments should be able to be broken down into blocks, in the same way blocks representing timing diagrams can be. This feature is important because it allows an engineer working with the language to use the intuition they have developed for the block language when working with the composition language. Also, the block model seems to fit this task very nicely at lower levels, and this allows that model to be extended to higher levels.

Another reason for extending the block structure into the composition language is that it allows our composition operators to be used on both block fragments and composed fragments. If composed fragments did not adhere to the block structure, there would have to be special composition operators to create larger composed fragments. If the composition operators preserve that structure, however, they can be re-used at all levels of the specification.

4.1.2 Encapsulation of Protocol Features

The second requirement for our specification language is that it should allow features of the protocol to be encapsulated into cohesive fragments. In informal specifications, each feature is introduced in its entirety in


AMBA AHB

3.1 About the AMBA AHB 3-3
3.2 Bus interconnection 3-4
3.3 Overview of AMBA AHB operation 3-5
3.4 Basic transfer 3-6
3.5 Transfer type 3-9
3.6 Burst operation 3-11
3.7 Control signals 3-17
3.8 Address decoding 3-19
3.9 Slave transfer responses 3-20
3.10 Data buses 3-25
3.11 Arbitration 3-28
3.12 Split transfers 3-35
3.13 Reset 3-40
3.14 About the AHB data bus width 3-41
3.15 Implementing a narrow slave on a wider bus 3-42
3.16 Implementing a wide slave on a narrow bus 3-43
3.17 About the AHB AMBA components 3-44
3.18 AHB bus slave 3-45
3.19 AHB bus master 3-49
3.20 AHB arbiter 3-53
3.21 AHB decoder 3-57

Figure 4.1: Table of contents for the AMBA AHB specification.

a single section of the document. This can be seen by examining the table of contents for the AMBA AHB specification, shown in Figure 4.1. Sections 3.4 - 3.13 each introduce a new, previously unmentioned, feature to the protocol specification.

We do not use a precise definition for the term “feature”. Features in the AMBA specification range from adding previously unspecified signals to existing diagrams to defining completely new transfer modes. Regardless of the scope of the feature, however, it must be possible in our language for its specification to be encapsulated in a single area, just as it is encapsulated within a single section of the informal specification document. This must be true even if the feature requires modifications to multiple other features. In the Aspect-Oriented community, such a feature would be referred to as a *cross-cutting concern*.

Another incentive for tight encapsulation of features in a specification is the common practice of creating “product lines” of devices that each implement a different subset of a design’s more advanced features. For example, the AMBA AHB specification includes descriptions of burst transfers, split transfers, and retries that a designer might not want to support in a simple device. A design team might want to create a product line of devices, in which the simplest (and cheapest) models implement only the basic functionality of the protocol specification, and the more sophisticated models allow all of the more advanced protocol features to be used. Ideally, a single specification for the protocol should be able to describe the entire product line; a verification engineer should be able to ignore optional features during verification without modifying the specification document.

4.1.3 Types of Extension

The third feature required in our composition language is support for all the types of composition that protocol specifications use. While the need for this is relatively obvious, identifying the operations required is not a trivial task. So far, we have identified four basic ways that protocol specifications compose timing diagram fragments to add features:

1. Extension of existing fragments (wait states, Figure 4.4)

2. Addition of previously unspecified signals (burst control signals, Figure 4.9, and basic control signals, Figure 4.3)

3. Sequencing fragments with overlap (pipelining, Figure 4.6), repetition of fragments (wait states, Figure 4.4)

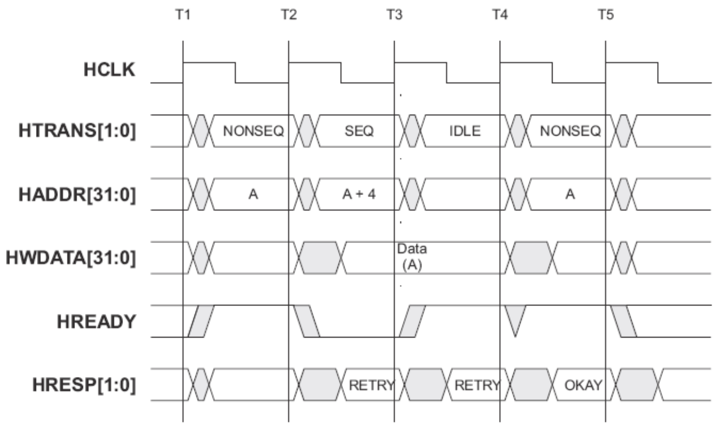


Figure 4.2: The slave's retry response and the resultant transfer in AMBA AHB.

4. Insertion of fragments (retry response, Figure 4.2)

4.1.4 Oblivious Extension

Fragments that are *oblivious* to extension is the final requirement for our composition language. In informal specification documents, all of the extension operations described above are done without explicit modification points in the more basic sections. For example, the basic transfer diagram shown in Figure 4.3 does not show the burst control signals, even though those are defined for that transfer. They are added later. This allows the basic cases to stay simple; they don't get cluttered with the requirements for more complicated features.

4.2 Aspect-Oriented Programming

The issues described above have very striking parallels to the motivations for Aspect-Oriented Programming (AOP). The most obvious of these are encapsulation of cross-cutting concerns, base code that is oblivious to modification, and base code that is functional before modification. Because of these similarities, we based our compositional operations on AOP, as represented in the widely-deployed AspectJ language [7].

AspectJ is based on Java, and it adds three new syntactic constructs to the language: *pointcut designators*, *advice*, and *aspects*. Together, these three constructs allow new code to be inserted and new functionality to be added to basic Java applications without modifying the original Java source code. We will examine each of these constructs briefly, and discuss how they and AspectJ relate to the stated requirements above.

Pointcut designators (PCDs) are the constructs in AspectJ that allow modification points to be picked out of existing code. AspectJ contains a set of primitive PCDs that can be combined using boolean operators to very precisely define sets of *join points* in the call graph of a program where new functionality should be added. These can be at the calling or return of a method, getting or setting a field, or a shift of control flow into an object. PCDs allow a programmer to define extension points within an existing codebase without requiring explicit hooks for modification within the base code. This is exactly like our stated requirment of extension of oblivious base code.

Advice is the actual code that can be applied at join points picked out by PCDs. AspectJ advice may add code before, after, or around the code being called at the join point. Advice may even modify the code being called. These types of modifications are very similar to our desired types of extension.

Aspects are the modules that encapsulate related PCDs and advice. They are rather like classes in Object-Oriented Programming in that they are intended to encapsulate the code implementing a cross-cutting concern into a single modular block. In our language, we wish to also allow this kind of encapsulation and modularity.

The final thing to point out about AspectJ is that when the cross-cutting concerns in aspects are correctly encapsulated, the base Java code that the aspects modify can be fully functional. A similar mechanism would allow a single document in our language to describe a whole product line of implementations; feature modules could be selectively removed from the full specification to define simpler devices.

4.3 Our Composition Mechanism, by Example

To introduce the composition portion of our language, we will demonstrate its features incrementally by walking through some examples from the AMBA-2 specification [2]. First, however, we will describe some of the basic syntax that is necessary for any composition language fragment. The AMBA-2 examples that will be shown after that will demonstrate in detail how our composition language allows simple composition, and composition with concurrent signals. Finally, we will briefly examine how the concepts presented in those sections extend to more complicated examples.

4.3.1 New Language Structures

We begin our introduction of the composition language by describing the new syntactic structures that the compositon language adds to the block language. These three new structures, *extendpoints*, *extendrules*, and *features*, closely correspond to Pointcut Designators, Advice, and Aspects in AspectJ.

Our language's analogue to a Pointcut Designator is called an *extendpoint*. We have chosen to give these a slightly different name because while they are similar to PCDs in that they identify sets of points in the base code to modify, the points they pick out are slightly different from the join points identified by PCDs. This will be elaborated more in later sections.

Just as PCDs are closely related to extendpoints, our language has *extendrules* as a replacement for advice. Our extendrules take an extendpoint and a block, and define how the block is used to modify the points referred to by the extendpoint.

Aspects in AspectJ are very similar to our *features*. Features in our language are used to define modular blocks of functionality, and are intended to correspond to a section in the informal specification document. Features contain blocks, extendpoints, and extendrules. Together these either define a basic feature of a protocol, or how to modify existing functionality to add a new feature. Features may depend on other features in a partial ordering, and it is intended that features may be removed from the specification in our language to create one instance of a device in a product line.

4.3.2 Basic Composition

Our first example will serve to show very basic composition of two specification fragments in our language. Figures 4.3 and 4.4 show the first two timing diagrams in the AMBA AHB specification. These diagrams and the text around them together define the basic AHB transfer (Figure 4.3) and wait states (Figure 4.4). The basic transfer is the fundamental fragment on which all other features in the AHB specification are based, and wait states are a slight extension to the basic transfer that allows a bus slave to delay its response to a bus request.

The basic transfer feature (basictrans) is nearly identical to the blocks presented in Figure 3.5. The only differences are two additions: the enclosing basictrans feature, and (on line 28) the statement "`toplevel is {simpletrans}`". The `feature` statement defines these four blocks as a single encapsulated feature of the standard. The `toplevel` statement specifies that any sequence of signals that conforms to the specification will be based on a sequence of simpletrans blocks. The `toplevel` statement does not, however, prevent the modification of those simpletrans blocks by other features.

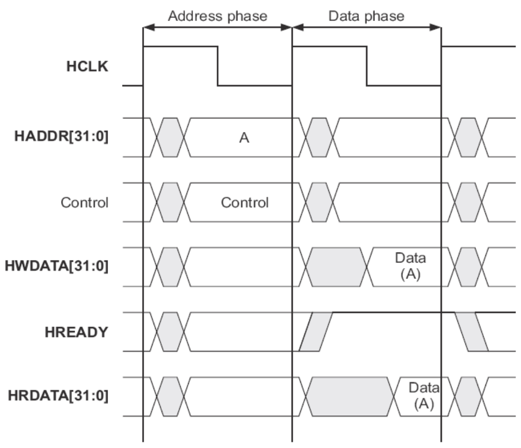


Figure 4.3: First timing diagram in AMBA AHB; shows basic read and write transfers.

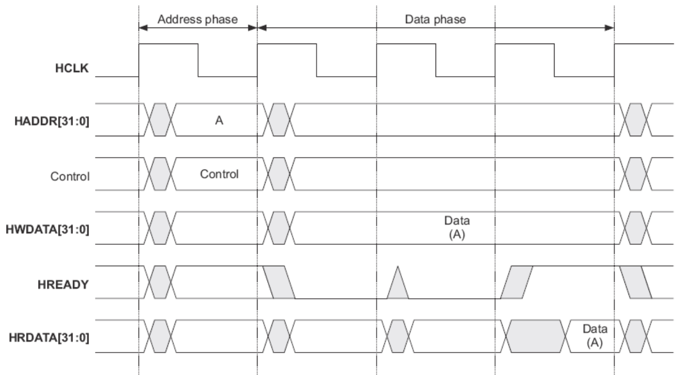


Figure 4.4: Second timing diagram AMBA AHB; adds wait states to the basic transfer.

```
feature basictrans {

        //basic clock cycle
        block clock() {
                CLK = 1, CLK = 0;
        }

        //address/control
        block haddrctrl(value addr[31:0],block control) {
                clock;
                HADDR[31:0] = addr[31:0];
                control;
        }

        //data
        block data(value wdata[31:0], value rdata[31:0]) {
                clock;
                HREADY = 1;
                HWDATA[31:0] = UNSTABLE as wdsetup, HWDATA[31:0] = wdata[31:0];
                HRDATA[31:0] = UNSTABLE as rdsetup, HRDATA[31:0] = rdata[31:0];
        }

        //simple transfer
        block simpletrans(value addr[31:0], value wdata[31:0], value rdata[31:0], block control) {
                haddrctrl(addr[31:0],control),data(wdate[31:0],rdata[31:0]);
        }

        toplevel is {simpletrans};
}

feature waitstate requires basictrans {
        extendpoint waitsplit(value wdata[31:0], value rdata[31:0])
                                = data(wdata, rdata) && in(simpletrans);

        block waitstate(value data[31:0]) {
                clock;
                HREADY = 0;
                HWDATA[31:0] = data[31:0];
        }

        extendrule: prepend waitsplit(wdata, rdata) waitstate(wdata) allowed;
}
```

Figure 4.5: AMBA AHB basic transfer and waitstates in our language.

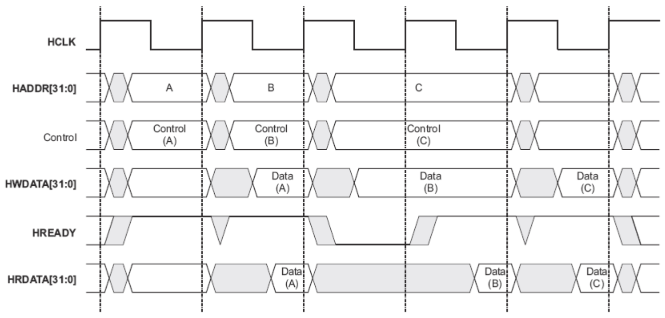


Figure 4.6: Multiple transfers in AMBA AHB. Shows pipelining of sequential transfers.

The feature after it introduces our composition operators, using them to add waitstates to the basic transfer defined in the basictrans feature. The waitstate feature definition delares that it "`requires`" basictrans. Intuitively, this means that in order for the wait states to be used in a specification, basic transfers must be used. Semantically, it allows the waitstate feature to refer to any blocks or extendpoints within the basictrans feature.

The waitstate feature contains three pieces: an extendpoint, a block, and an extendrule. All three of these are required to use our aspect-like composition language. The extendpoint declaration defines a name (waitsplit), arguments (wdata and rdata), and a boolean expression of PCDs which defines the set of join points that belong to this extendpoint. The waitsplit extendpoint occurs at the data phase of a simple transfer. The extendpoint's arguments will give any extendrule that uses it access to that data.

The waitstate block within the waitstate feature defines a wait state lasting a single clock cycle. This matches both the second and third clock cycles in Figure 4.4. This block is very similar to the data block in the basictrans feature, but HREADY is 0 and HRDATA is left unspecified.

The extendrule at the end of the waitstate feature describes how the waitstate block is to be added to the basic transfer. Line 41 is responsible for this specification, and it contains four pieces. First is the keyword `prepend`, which states what kind of advice is to be added. Second is the extendpoint at which to add the advice (in this case, waitsplit). Third is the block which this extendrule will insert (waitstate). The final piece is the (optional) keyword `allowed`, which means that this extendrule is not required to be applied in all situations where it could be.

That extendpoint uses `prepend` advice, which is not available in AspectJ. This type of advice adds the specified block before the indicated extendpoint; in this case, it adds a wait state before the data phase of the basic transfer. That behavior could also be achieved with `before` advice. However, the `prepend` advice also modifies the data phase of the basic transfer; the new wait state actually becomes part of the previously specified data block. Any other extendpoints that refer to the same data block will obliviously refer to the new combined data and wait state block.

The `prepend` advice also has another effect. Normally, each extendrule may only be applied at any given extendpoint once. In this case, however, the addition of a wait state to the data block actually changes the waitsplit extendpoint. The data block referred to in that extendpoint has changed because of the addition of the wait state, so the new combined data and wait state block is a new match for the waitsplit extendpoint. The two primitive PCDs that make up the waitsplit extendpoint will always match the data phase of a basic transfer, regardless of the number of added wait states. Therefore, the extendrule in this feature could be recursively and infinitely applied. In this case, the `allowed` statement makes it possible to end this recursion.

```
feature pipeline requires basictrans {
        extendpoint datapoint(data d, simpletrans s1, simpletrans s2)
                = d && in(s1) && next(s2);

        //override haddrctrl                                                      5
        //have to do this-the basictrans version of haddrctrl explicitly makes it
                //a single clock cycle
        block haddrctrl(value addr[31:0], block control) {
                HADDR[31:0] = addr[31:0];
                control;                                                          10
        }

        extendrule: concurrent pipesplit(d, s1, s2) s2.haddrctrl 1;
        extendrule: after pipesplit(d, s1, s2) s2.data 1;
}                                                                                 15
```

Figure 4.7: Implementing pipelining in our language without synchblocks. Builds on the code in Figure 4.5.

4.3.3 Concurrent Composition

Figure 4.6 demonstrates multiple transfers in a row, with the address phase of each transfer occuring simultaneously with the data phase of the previous one. This is called *pipelining*, and is the next feature that the AMBA AHB introduces. An important part of pipelining in the AMBA specification is the way in which it interacts with wait states; a wait state during the data phase of one transfer extends the address phase of the next transfer. In Figure 4.6, this occurs during the data phase of B and the address phase of C.

Our first attempt to implement pipelining behavior is shown in Figure 4.7. The first part of the pipeline feature looks very similar to the waitstate feature; it also requires basictrans, and the pipesplit extendpoint identifies the same location as the waitsplit extendpoint in waitstate. pipesplit differs from waitsplit in the arguments it makes available to its advice, and that pipesplit refers to the next transfer as well. While waitsplit makes the wdata and rdata arguments to the data block available, pipesplit provides access to the data block, and both the current and next simpletrans blocks. This allows the pipelining extendrules to use those arguments.

The pipelining feature has to override the haddrctrl block, and does so on line 6. This is to allow that block to correctly extend when it is pipelined with a data phase that is extended with wait states. This does not deviate from the specification, which also claims that the address phase only lasts one clock cycle when the basic transfer is introduced, but changes that when demonstrating pipelining. The new haddrctrl block is identical to the original, except it does not contain a clock block. The original's clock block would have conflicted with the multiple clock cycles that are possible in a data phase extended with wait states.

The pipeline feature also contains two extendrules to define how it fits in with the previous features. In this version of pipeline, the address and data phases of the two simpletrans blocks are added to the sequence separately. The haddrctrrl sub-block of the second simpletrans is added concurrently with the first transfer's data block, and the second transfer's data block is added afterward.

While the above example does correctly define a single pair of pipelined transfers, we are not satisfied with the way in which it does so. This method splits the second simpletrans in order to place it back in a different location with the same configuration. The process of pipelining two transfers is an exercise in *synchronization*, not modification and extension. Such a task should be possible without modifying the blocks to be synchronized.

For this reason, we added a new language structure called a synchblock, shown in our final pipeline feature in Figure 4.8. A synchblock has the same syntax as a regular block, with one difference: the restriction that parallel sequences may not refer to the same signal is different. Parallel sequences in a synchblock may not

```
feature pipeline requires basictrans {
        extendpoint pipesplit(data d, simpletrans s1, simpletrans s2)
                = d && in(s1) && next(s2);

        //override haddrctrl
        //have to do this-the basictrans version of haddrctrl explicitly makes it
                //a single clock cycle
        block haddrctrl(value addr[31:0], block control) {
                HADDR[31:0] = addr[31:0];
                control;
        }

        synchblock pipelined(simpletrans s1, simpletrans s2) {
                s1.data 1;
                s2.addr 1;
        }

        extendrule: concurrent pipesplit(d, s1, s2) s2 using pipelined(s1, s2);
}
```

Figure 4.8: Implementing pipelining in our language using a synchblock. Builds on the code in Figure 4.5.

refer to the same signal *as it appears in different blocks.* This is possible because a synchblock only defines how the pieces of multiple blocks are synchronized. As long as the synchblock is not inconsistent with the structure of the blocks it synchronizes, this condition is sufficient to ensure that the synchblock will not cause any inconsistencies in the surrounding blocks.

The pipelined synchblock in the pipeline feature synchronizes the data phase of the current transfer and the address phase of the next transfer. To pull these sub-blocks out of the two simpletrans arguments, we use syntax that has not previously been introduced. "`s1.data 1`" refers to the first data block appearing in the s1 block. If multiple data blocks appeared within the s1 block, later ones could be referred to by using higher numbers (2 for the second, 3 for the third, etc.).

As long as "sub-block" to in this syntax does not refer to a placeholder block such as the control block in basictrans, this technique is guaranteed to always uniquely identify a single sub-block. Each non-placeholder block of a given type will assign values to a concrete signal; within a consistent block these assignments will be fully ordered. Therefore, all the occurences of a given type of sub-block will also be ordered, and referring to those sub-blocks by an integer will uniquely identify a single sub-block.

4.3.4 More Complicated Examples

We will briefly describe two more examples, but in the interest of space they will not be reproduced here. They can, however, be found in Appendix A, within the AMBA AHB specification. Both examples are within the bursts feature, on lines 87-279.

The AMBA AHB specification defines *burst transfers*, which are comprised of sequences of transfers to sequential addresses. These are provided because AHB devices can be optimized to handle bursts more efficiently than equivalent sequences of basic transfers. The AMBA AHB defines both fixed-length bursts, which can be 2, 4, 8, or 16 transfers long, and indeterminate-length bursts, which last until they are terminated by the master.

The basic blocks for fixed-length bursts are not complicated; for example the 4-beat wrapping burst block (burst4wrap) on lines 156-163 in Appendix A should be mostly understandable from the explanations given so far. The timing diagram for this is shown in Figure 4.9. What is worth examining in this example,

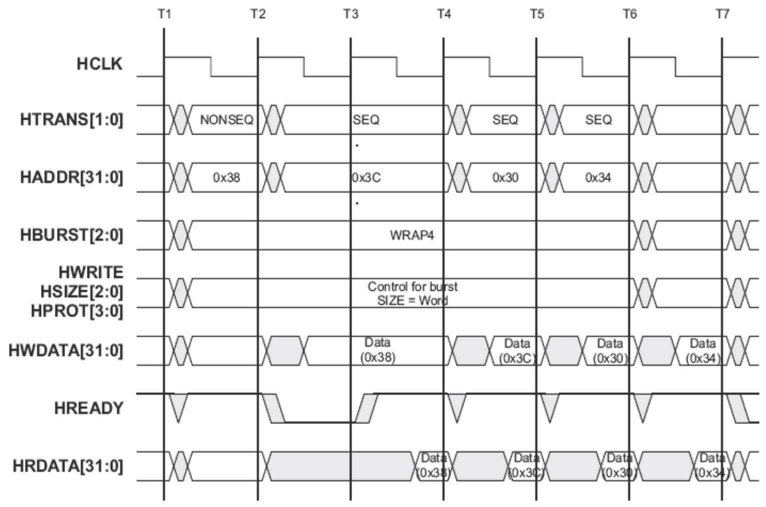


Figure 4.9: Four-beat wrapping burst of transfers from the AMBA AHB specification.

however, is the way in which the address for each sucessive transfer is calculated. The AMBA-2 specification contains an entire page of text (page 3-11) describing the address calculation process for burst transfers. To avoid reproducing the same calculations for each block in a burst, we created a set of functions for doing these calculations. The incrementval, burstbeatval, and burstaddrcalc functions are responsible for these calculations, and should be understandable for anyone familiar with C.

It should be noted that these blocks appear to violate our previously stated restriction on assignment to the same signal in parallel sequences. The second sequence in each of these blocks also refers to the final data block of the first transfer. While by our rules mentioned above this should not be allowed, this is a shortcut we use to simplify these blocks which does not actually violate that rule. By referring to a specific data block within the same block, we have added an extra point of synchronization within the block. This same effect could have been achieved by removing the second sequence from the block and adding new advice to each block to add it back in using a synchblock to achieve the same synchronization.

The final line of the pipeline feature demonstrates how synchblocks are used. They are only used with `concurrent` advice. Use of the `using` keyword with concurrent advice and a synchblock specifies that rather than synchronizing the entire advice block with the entire block being advised, just the portions defined within the synchblock should be synchronized. Any remaining overhang should displace any blocks before or after the block being advised.

The second example shows how indeterminate-length bursts are defined. A sample timing diagram for this kind of burst is in Figure 4.10. The first important step in specifying this kind of burst is enabling a possibly infinite chain of transfers. To do this while allowing the burst to end at any point requires a recursively applied optional (`allowed`) extendrule which adds a transfer every time it is applied. That extendrule and its corresponding extendpoint are on lines 250 - 254 in Appendix A.

One would expect rhe extendpoint for this extendrule to be fairly simple, considering it only picks out the end of the current sequence of transfers and adds a new one. It is so complicated because it also must retrieve the address of the last transfer so that the address for the new transfer being added can be calculated. This means that it has to specifically retrieve the last transfer in the current burst.

There is one more facet of creating the indeterminate length burst transfer which is suprisingly difficult: adding the control signals. The specification defines two burst control signals, HBURST and HSIZE, which must be held constant from the address phase of the first transfer in the burst to the address phase of the last transfer in the burst (in the data phase of the last transfer, these switch to the control signals for the

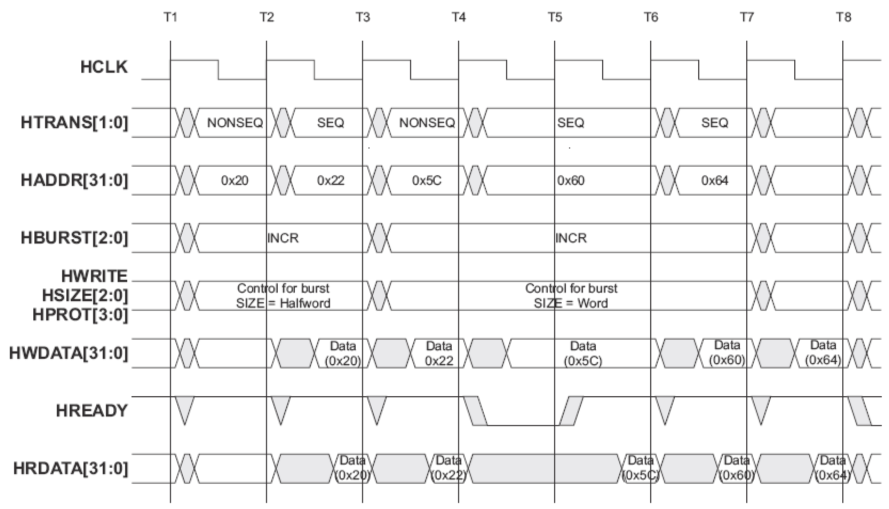


Figure 4.10: Indeterminate-length burst from the AMBA AHB specification.

next pipelined transfer). There are two ways we saw to approach this.

The simpler of the two (not shown in Appendix A) is to define concurrent advice on each address phase in the burst with the specified control signals. While this method would correctly set the control signals for the correct duration, it would not work if we adopted the glitch semantics described in Section 3.3.2. The specification requires that the control signals be held without glitches for the full duration of the burst, but splitting those signals into separately applied advice would allow them to glitch using those glitch semantics.

The second method, and the one we chose to use, uses a synchblock to synchronize the control signals with the burst. The difficulty for this method, however, is in eliminating the data phase of the final transfer from the block synchronized with the control signals. The semantics for synchblocks and picking out sub-blocks from their parents that was shown in the previous section would allow any data block at a constant offset from the beginning of the burst to be eliminated, but not the final data block. Unfortunately, since this is a burst of abitrary length, we cannot use the numerical constant notation used earlier to pick out this data block. For this reason, we added the `last` keyword, which allows sub-blocks to be picked out from the end of a block.

4.4 Our Primitive PCDs and Advice Types

We mentioned earlier that the extendpoints in our language were different from AspectJ's Join Points and Pointcut Designators. This section will describe that difference and why we made our language that way; then it will describe each of the basic extendpoints and advice that make our language's composition operators possible.

4.4.1 The Difference Between PCDs and ExtendPoints

The major difference between our extendpoints and AspectJ's PCDs is that while AspectJ's PCDs pick out sets of *points* in the runtime call graph of a program, our extendpoints pick out sets of *blocks* in the block representation of the protocol. There are two pieces of this that need to be more clearly defined. First, what is the "block representation of the protocol", and second, what does it mean for an extendpoint to "pick out" blocks in that representation?

An AspectJ program listing serves to define the runtime behavior of that program, and the runtime call graph is an abstraction of that behavior. The generation of the program's behavior can be thought of as a two phase process. In the first phase, the basic runtime call graph is generated from the main function of the program and all of the code called from it. In the second phase, each PCD picks out a set of nodes, and any advice that applies to them is used to augment or modify that basic call graph. The end result is the full call graph of the program.

A listing in our language describes something different: a description of all the possible behavior allowed by a given protocol specification. To simplify this, we look at a particular instance of that behavior, or a "run" of the protocol as might be required to be generated or validated by a test bench. Such a run could be represented by the logical blocks that it is made up of.

We can also imagine the generation of such a run as a two-phase process. In the first phase, a sequence of "toplevel" blocks are laid out. In the second phase, each extendpoint picks out a set of blocks, and its associated extendrules are used to modify those blocks, and to insert new ones. The final result is a single instance of an allowed protocol run represented in blocks. This is what we refer to as the block representation of the protocol.

Both our extendpoints and AspectJ's PCDs are collections of criteria connected by and, or and not (`&&`, `||`, and `!`). Each PCD matches a set of points in the runtime call graph which satisfy its criteria. It is said to "pick out" those join points. Extendpoints in our language work in the same way, except that blocks are picked out by this process. Note that the extendpoint need not pick out a named block; it may also pick out a sequence of blocks, an atomic block, or a set of contiguous blocks with synchronized beginning and end points.

4.4.2 Why Match to Blocks?

We decided to have our extendpoints match blocks rather than points in a run of the protocol for two reasons. First, our language places far more emphasis on concurrency and synchronization than AspectJ, and block extendpoints support these far more naturally than point PCDs. Second, block extendpoints more naturally support the type of extension that the `prepend` and `append` advice in our language allow.

The first reason for block extendpoints is that defining and synchronizing concurrent advice is a lot more natural when the join "points" being picked out are blocks. This is because synchronization in our language is only accomplished in one of two ways: through composition of blocks, or by using synchblocks. We would like to minimize the number of synchblocks that an engineer working with our language would have to write, so we would like to allow synchronization through the former method as often as possible. If single-point PCDs were used in our language, concurrent advice would always require some additional mechanism to define how the advice was to be synchronized.

The second reason for block extendpoints is to support `append` and `prepend` advice. In the examples we have studied and translated into our language, we have found that these types of advice are far more common than `before` and `after` advice. Therefore, we want our composition language to be optimized for using `prepend` and `append` advice. Applying such advice at a point is somewhat ambiguous: which of the adjacent blocks subsume the advice blocks? With block extendpoints, it is the block defined by the extendpoint that subsumes the advice block.

4.4.3 Primitive Extendpoints

Table 4.1 shows all of the primitive extendpoints that we currently support in our language. Each of these will match a set of blocks in a run of the protocol, and that set may be refined by combining these primitive extendpoints with "and", "or", and "not". While a brief description of each is given in the table, we will examine them in more depth in the following sections.

The `block` extendpoint is by far the simplest; it picks out any blocks that match the signature specified in the BlockPattern. It is also very commonly used in the example in Appendix A. It can first be seen on line 33, in the waitstate feature.

`block`(BlockPattern)	Picks out every block matching BlockPattern.
`in`(BlockPattern)	Picks out every block that is enclosed within a block matching BlockPattern.
`next`(BlockPattern)	Picks out every block whose end coincides with the beginning of a block matching BlockPattern.
`prev`(BlockPattern)	Picks out every block whose beginning coincides with the end of a block matching BlockPattern.
`if`(BooleanExpression)	Picks out every block for which BooleanExpression evaluates to true. May only access parameters exposed by the enclosing extendpoint.

Table 4.1: Primitive extendpoints in our language.

The `in` extendpoint picks out every block that is contained within a block that matches the signature of its BlockPattern. This includes any blocks which themselves match that BlockPattern. One block b_1 is considered contained within another block b_2 if three conditions are satisfied:

1. The beginning of b_1 occurs at the same time or after the beginning of b_2
2. The end of b_1 occurs at the same time or before the end of b_2
3. The signals of b_1 are a subset of the signals of b_2.

The `prev` and `next` primitive extendpoints are very closely related. They each allow specification of extendpoints based on the blocks that appear sequentially before and after them. For each block picked out by the BlockPattern, there is a set of blocks that match the `next` primitive extendpoint. Each block in this set has the property that its end point is concurrent with the starting point of a block that matches the BlockPattern. For example, in Figure 4.6, there are three blocks which match the primitive extendpoint "`next(data(C,C))`":

1. The data block for transfer B
2. The address block for transfer C
3. The entire basic transfer block B

The endpoints of all three of these blocks coincide with the beginning of the data block for transfer C. The `prev` primitive extendpoint functions in the same way, but matches blocks which occur immediately *after* a block that matches the BlockPattern.

All but the `if` primitive extendpoint use a BlockPattern to refer to a set of blocks. At this point, we have not settled on all of the possibilities for that language construct. AspectJ allows a wide variety in its equivalents (MethodPattern, FieldPattern, ConstructorPattern, etc.), including wild card characters and argument conditions. We feel that the requirements for the BlockPattern in our language need to be examined in more depth; at this moment we have only a few examples to go by. In all our examples, we have not needed more than the name of a named block, and optionally its arguments. We allow wild cards in place of arguments which do not matter. Another option for BlockPatterns which we have not explored is specification of a block based on attributes of its beginning and end points.

The `if` primitive extendpoint is different from the others. While the other extendpoints are all based on the relative position of an extendpoint to a particular block, the `if` extendpoint allows specification of an extendpoint by characteristics of data. The argument to the `if` primitive extendpoint is any boolean expression which depends only on values exposed by the enclosing (non-primitive) extendpoint. It matches any block for which that boolean expression evaluates to true.

`before`	Adds the advice block immediately before the extendpoint.
`after`	Adds the advice block immediately after the extendpoint.
`append`	Like `before`, except all `block` and `in` primitive extendpoints that are part of this extendpoint's expression are extended to include the advice block.
`prepend`	Like `after`, except all `block` and `in` primitive extendpoints that are part of this extendpoint's expression are extended to include the advice block.
`concurrent`	Adds the advice block in parallel with the extendpoint. The beginning and end of the advice block are synchronized with the beginning and the end of the extendpoint, unless a synchblock is used.

Table 4.2: Advice types in our language.

4.4.4 Advice Types

Table 4.2 contains all of the advice types that we currently allow in our language. We have been able to construct all of our examples using only these basic advice types. While the table describes each briefly, the following sections will provide a more in-depth discussion of the semantics of each. For each of these discussions, b will refer to the block being added by the advice, and e will refer to the block picked out by the advice's extendpoint.

`before` and `after` advice are very basic, and are closest to the types of advice available in AspectJ. Since the only difference in their functioning is whether b is added before or after e, we will discuss only `after` advice. `after` advice simply adds b immediately after e, which means that the end point of e coincides with the beginning of b in the resultant block representation. Any blocks which previously began immediately after e are displaced by the new b block. `before` and `after` advice may cause conflicts with `concurrent` advice that uses synchblocks. Each `before` and `after` advice may only be applied once at each extendpoint.

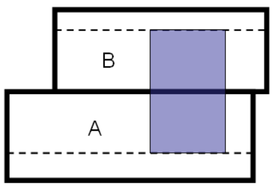
`prepend` and `append` advice are unique to our composition language, and have the same semantics as `before` and `after` advice, respectively, with one difference. Whereas the new block b added due to `before` or `after` advice is considered to be a new standalone block, in the case of `append` and `prepend` advice, the new block is made part of the extendpoint it was added to. This is done by modifying any blocks matching either `in` or `block` primitive extendpoints within the extendpoint that the advice is modifying to include the new block.

For example, when the `waitstate prepend` advice is applied in Figure 4.4, the data block is extended; afterward, its beginning has changed to the beginning of the new wait state block. Because applying `append` or `prepend` advice modifies the blocks that an extendpoint matches, the application of either of these types of advice may allow the same advice to be repeatedly and infinitely applied to the same base block. This infinite application of advice may be avoided by either specifying that the advice is optional, by using the `allowed` keyword, or by using `if`, `prev`, and `next` primitive extendpoints to avoid matching the same block multiple times.

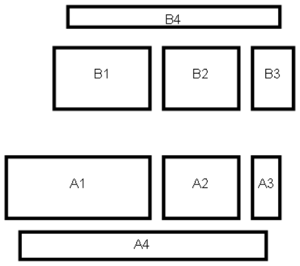
The final type of advice, `concurrent` advice, is also unique to our language. In its simplest form, b, the advice block, is added in parallel with e, the block matching the extendpoint. Together, b and e define a new block, and all of the basic block syntax applies. Their beginning and end points are synchronized to each other, but none of their internal events are. In addition, they may not both modify the same signal, in concordance with the restrictions on signal assignment in parallel sequences.

`concurrent` advice may be used with synchblocks, as in the pipelining example in Figures 4.6 and 4.8. In this case, both b and e are essentially split into four separate blocks. Each is split into:

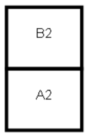
1. a block containing the portion of the sequence occurring strictly before the events in the synchblock
2. a block containing the portion which occurs within the sequence
3. a block containing the portion occuring strictly after the events in the synchblock



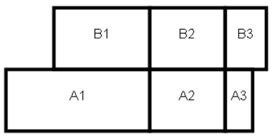
(a) Intended result of concurrent advice with synchblocks; the shaded area represents the portion specified by the synchblock.



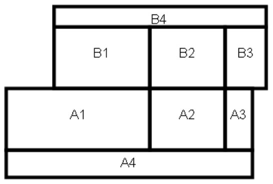
(b) Component blocks split into parts based on the synchblock.



(c) Synchblock parts composed concurrently.



(d) Before and after fragments added.



(e) Non-synchronized signals added, final result.

Figure 4.11: Process for concurrent advice using synchblocks.

4. a block containing events which are unaffected by the synchblock (any signals which are not mentioned in the synchblock)

This splitting, and the process described below, are shown in Figure 4.11.

These four blocks are then added to the run of the protocol in the following way. All four blocks of e remain in their original positions. Block 2 of b is added to the run in the same way as `concurrent` advice to block 2 of e. Blocks 1 and 3 of b are added to the run as `before` and `after` advice, respectively, on block 2 of b. Finally, Block 4 of b is added as `concurrent` advice to the extendpoint defined by blocks 1, 2 and 3 of b. In many cases, a few of these blocks will be empty. This is the case in the pipelining example, where blocks 1 and 4 of b and blocks 3 and 4 of e contain no signals.

4.5 Issues Requiring Further Examination

At this point, our composition language has not been fully fleshed out. There still remain many areas which have not yet been examined in adequate detail. These include: conflicting advice applied at the same point, a mechanism to cancel scheduled blocks, and support for more advanced protocol features. Each of these issues will be discussed in more depth in the following sections.

4.5.1 Advice Collision and Conflicts

A problem that is often discussed in Aspect-Oriented literature is advice collision, which occurs when more than one piece of advice applies to a single join point. It is important to handle such a situation consistently, because in some cases the order in which the advice is applied can matter. While the same problem applies to our language, we have not yet examined it in enough depth to recommend a solution.

AspectJ handles the problem by applying advice in order of its specificity [7]. If we think of the advice being applied in layers around the join point, the least specific advice is applied on the innermost layer, while the most specific is applied outermost. This means that for before and around advice, the most specific advice is applied first, while for after advice the opposite is true.

We have not yet designed such a mechanism for our language because we have not seen many examples which produce conflicting advice. The only rule we have consistently applied so far is that `prepend` and `append` advice are applied before concurrent advice. This is necessary to ensure that concurrent blocks are correctly synchronized. Based on the examples we have examined, we expect that many advice collisions will be avoidable.

4.5.2 Cancelling Scheduled Blocks

In the `bursts` feature on lines 87-279 of the AMBA AHB example in Appendix A, we use a type of advice which does not appear anywhere else in our examples. It is intended to allow the early termination of a burst of transfers, as described in the AHB specification. The extendrule that uses this advice appears on line 275, and it reads: "`extendrule:  before inburst(b) end(b) allowed`". The semantics of this extendrule are intended to be that all sub-blocks of block `b` after the specified point should be removed from the protocol run. In effect, the end of block `b` should be moved to the current point.

We are not satisfied with this mechanism, though it seems as though there may be many more cases where such a mechanism is needed. Except for this `end` advice, no other command in our language allows blocks to be removed from the protocol run. This particular solution has not been adequatly examined, but we suspect that it is inadequate, and that a more sophisticated mechanism will be required to meet this need.

4.5.3 More Complicated Protocol Features

We had hoped to provide an example of the full AMBA AHB specification in our language. However, one of the more sophisticated features of the AMBA specification slowed our progress. The AMBA AHB specification describes a few ways in which a transfer between a given master and slave may be "split". This can occur when a burst is interrupted, or the slave requests a transfer to be repeated or split. In these cases, control of the bus may shift to another master, splitting the transfer in progress. The catch is, when a master regains control of the bus, the specification states that it is obligated to continue any split transfers that are pending.

This requirement is not possible to represent using our language in its current form. We can imagine two mechanisms which could be used to do this. The first would be to provide a way to define advice which reverts to the top-level blocks of a feature. This would allow the splitting of the transfer to be specified as advice that gets inserted in the middle of the transfer, and allows any sequence of top-level blocks to be inserted. The problem with this approach is that the top-level blocks within that advice would have to be restricted; the master device waiting on the split transfer would not be able to make any transfers for the duration of that advice. This becomes even more complicated if multiple masters have split transfers pending.

The second way we can think of to enable split transfers requires two mechanisms. The first is mentioned in the previous section: a way to remove blocks from the protocol run. This would allow the rest of the transfer to be canceled. The second mechanism would enable restoring the transfer, and would take the form of *one-time advice*. This would be a new type of advice which could be dynamically defined and added, and would apply only once. For split transfers, it would take effect the next time the current master gained control of the bus, and would insert blocks to complete the transfer. The one-time advice would be defined when the transfer was split to contain the necessary blocks, and would no longer exist after being applied.

Specification of split transfers is so complicated because its introduction in the AMBA AHB specification brings along with it a previously hidden and very large feature: bus arbitration. Before splitting transfers is mentioned, all of the transfers in the AHB specification are presented as if each AHB bus has one master. Interruption of transfers and split transfers only occur on buses with multiple masters and an arbiter; the section on these in the AHB specification occurs after interruption is introduced, and consumes 11 pages (3-28 to 3-39 in [2]). For comparison, all of the features we have implemented take up 19 pages. This feature is more sophisticated than any of the others we have examined, and depends on the functional description of the bus arbiter device.

Up to this point, we have tried to represent the protocol specification without describing the devices that use it in too much detail. It does not seem possible to present bus arbitration in this manner; therefore, a solution for representing split transfers should be based on a more thourough examination of cases where protocol specifications contain functional descriptions of devices they depend on.

4.6 Other Explored Composition Mechanisms

We chose to base our composition language on Aspect-Oriented programming after examining a few other options. The following sections will examine a few of these, and describe why we decided on Aspects.

4.6.1 All Blocks

The early work we did on composition attempted to use the basic composition available in the block language to construct the entire protocol specification. In order to handle cases where a given sequence was optional or could be repeated, we allowed regular expression-type operators such as `*`. Such a scheme *could* be used to describe entire specifications; this is demonstrated by the fact that all of our aspect-based composition operators can be reduced to blocks.

We learned that blocks are too concrete of a specification mechanism to satisfactorily handle the task of higher-level fragment composition. Representing more complicated protocol features revealed that blocks do not allow sufficient abstraction. They are far better suited to representing simple, small, concrete cases of signals and their values. It is interesting that a very similar criticism has been applied to timing diagrams [4].

A very simple example that demonstrates this is the case of pipelining. Adding this feature to a protocol using only blocks is suprisingly difficult. It is impossible using the syntax of blocks to define the overlap of two simple transfers without re-defining the address and data phases of adjacent transfers to occur within the same block. Such a solution requires that the address and data phases of the *same* transfer be separated into two unrelated blocks, which is unacceptable.

We explored adding operators which would collapse non-conflicting adjacent blocks into each other. This could have allowed the specification of pipelining without breaking up the basic transfer block; such a mechanism would allow the address phase of one transfer to be collapsed into the corresponding undefined region of the previous transfer's data phase. However, all the solutions we examined in this area seemed to either take too much control out of the hands of the engineer or required too much annotation for a simple operation.

4.6.2 Live Sequence Charts

Another method we examined was based on Damm and Harel's work on Live Sequence Charts (LSCs) [3], which extend the Message Sequence Charts (MSCs) in the Unified Modeling Language (UML). We were drawn to this work because MSCs suffer from problems similar to the ones we noticed while pursuing all-block composition (described in the previous section). MSCs are popular artifacts for defining small, concrete examples of an interaction between multiple objects in an Object-Oriented design. Like blocks and timing diagrams, however, MSCs have difficulty extending to more complicated examples. LSCs are an extension to MSCs which are intended to allow specification of an entire system using MSC-like diagrams.

LSCs do this by adding an *activation condition* to each MSC. They also extend MSCs to have both an existential and a universal form. Existential LSCs define a sequence which is *allowed* when the chart's

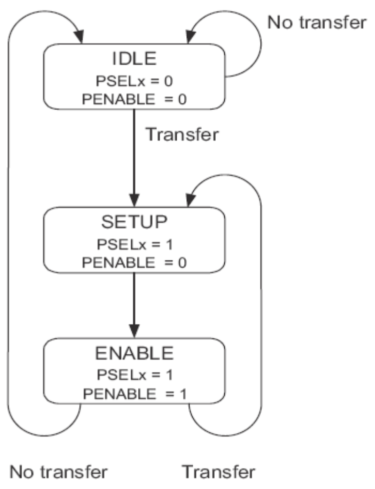


Figure 4.12: State machine from the AMBA APB specification.

activation condition is satisfied, while Universal LSCs define a sequence that *must* be followed whenever the chart's activation condition is satisfied. By composing these together, it is possible to define all of the allowed sequences that may occur in any run of the system.

We were not satisfied with our initial investigation into doing the same thing with blocks. Specifying the correct placement for a block via its activation condition obscured some of the relationships between blocks. In addition, this technique did not provide us with the oblivious extension or product line flexibility that the aspect-based approach provides.

4.6.3 State Machine Driven Composition

The last composition technique we examined took advantage of an artifact which appears in many hardware specifications: state machines. Figure 4.12 shows one from the AMBA APB (Advanced Peripheral Bus) specification, which shows the states the APB goes through during each transfer on the bus. Because these are widely used in specifications, we spent a considerable amount of time examining the feasibility of using these to compose blocks. We examined two methods for doing so, each of which will be elaborated further.

The first method we examined was representing the high-level operations of the protocol with a state machine, and annotating its transitions with blocks. Every time the state machine made a transition, its corresponding block would be output on the protocol run. Unfortunately, simple state machines such as the one in Figure 4.12 are inadequate for such a model; that kind of simple state machine is unable to represent parallel lines of execution and other more complicated features of a full bus specification.

We therefore attempted to apply this same idea to a more complicated model of state machine. Statecharts, as presented by David Harel [5], are able to handle parallel execution very well. They can express parallel state machines, and use conditions on the transitions to describe synchronization between parallel state machines. In addition, Harel's Statecharts can be composed into larger and more complicated systems very easily, allowing for incremental development of the full protocol specification. Our early exploration revealed that we were able to specify many of the examples that were problems for the other techniques.

Unfortunately, though the StateCharts could represent the things we needed, they tended to obscure the functionality of even simple examples. For example, Figure 4.13 is the Statechart used for defining a basic transfer with wait states. It obscures the relationship between the address and data phases, and the synchronization of the two halves of the Statechart is difficult to see. For this reason, Statecharts were unacceptable.

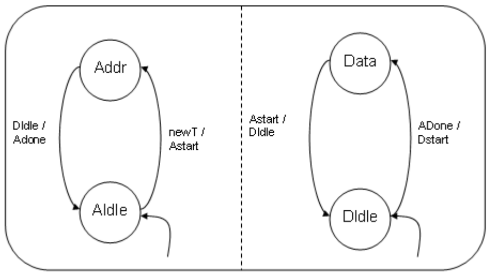


Figure 4.13: Example of a Statechart which defines a simple pipelined transfer.

The other method we tried was an event-based model using FrTime, a scheme package for Functional Reactive Programming. In this model, the state machine constantly cycled, and output events for leaving each state, entering each state, and for each transition. The surrounding code then defined how blocks were laid out based on those signals. Each block was given a condition based on the signals that defined its start point, and a similar condition for its end point. While the state machines used for this were much simpler than the Statecharts, they did not have to represent more sophisticated concurrency and synchronization. Those could be handled by the start and end conditions for each block.

While we were able to generate timing diagrams for the simple transfers in the AMBA APB, it required a lot of work. In addition, the framework seemed rather inflexible; extension would have required careful modification of the fundamental state machine. Another drawback of this approach was that further study of other protocol specifications revealed that state machines were not as commonly used as we thought. Within the AMBA spec the APB state machine is the only time state machines are used to describe signals; all the others are used to describe the functioning of devices. This may be because the APB is not pipelined. In addition, on no occasion did we see a state machine which represented the full capabilities of a protocol. A verification engineer would be required to design such an artifact, and we would like to eliminate that kind of task from the verification workflow.

Chapter 5

Conclusion

This document presented a preliminary language for specifying hardware bus protocols. It is intended to eventually support a new process for hardware verification of these protocols, in which it will be the language for a machine-generated and human-refined intermediate representation. Its role in that process requires it to be human-readable, and to allow specification in a manner similar to the informal specifications which it represents. Our language can be divided into two sub-languages, one which is used for defining small fragments on the scale of individual timing diagrams, and one for composing those fragments together into full specifications.

The fragment-level language we have designed is based on the logical blocks we found common in bus protocol timing diagrams. Like the timing diagrams it is intended to represent, it is particularly good at defining concrete cases of a protocol's functioning. It captures both the sequencing and synchronization aspects of timing diagrams using simple syntax with intuitive semantics. Unlike some other approaches to representing protocol fragments, it can be used to represent asynchronous protocols. The same mechanism that this language uses to create fragments can also be used to compose such fragments to represent arbitrarily large timing diagrams.

While it does not currently support specification of timing constraints, we have proposed a method for doing so which should support all the ways in which that kind of information is used. Our proposal would also allow such information to be easily added after the fact. In addition, our preliminary investigation suggests that it may also naturally support representation of glitches, a feature of timing diagrams which some languages for the same purpose have ignored.

Our fragment language also naturally supports the composition language, which is based on ideas from the Aspect-Oriented Programming community. Aspect-Oriented ideas have given us a composition language which allows oblivious modification of existing code. This allows simple cases in our language to avoid being complicated by more advanced protocol features. In addition, aspects allow our specification to be organized into sections which match those in an informal specification. This is significant because sections in those documents often introduce new features which extend and modify multiple preceding sections.

We have been able to specify most of the AMBA2 AHB specification in this language, including basic transfers, wait states, pipelining, and bursts. That specification, which appears in Appendix A, introduces these features in the same order as the informal specification, and can easily be compared to that specification. We have also used our fragment language to define the PS/2 mouse and keyboard protocol, which is an asynchronous protocol, and the Address/Data bus of Intel's 8088 microprocessor.

The more advanced features in the AMBA2 specification which our language is not yet able to define require more sophisticated language constructs. Further study must be done to investigate similar features in other protocols before such constructs are added to the language.

Once the language is completed, the next step will be to build the tools that support the proposed workflow mentioned in Section 2.2. These include an automatic specification skeleton extractor, and a set of tools to extract verification artifacts from a specification in our language. The latter should be created first, as they will allow the language to be applied to real problems. At the moment, we have not implemented any tools to consume a specification in our language.

Appendix A

Full Language Example

```
feature basictrans {

        //basic clock cycle
        block clock() {
                CLK = 1, CLK = 0;
        }

        //address/control
        block haddrctrl(value addr[31:0],block control) {
                clock;
                HADDR[31:0] = addr[31:0];
                control;
        }

        //data
        block data(value wdata[31:0], value rdata[31:0]) {
                clock;
                HREADY = 1;
                HWDATA[31:0] = UNSTABLE as wdsetup, HWDATA[31:0] = wdata[31:0];
                HRDATA[31:0] = UNSTABLE as rdsetup, HRDATA[31:0] = rdata[31:0];
        }

        //simple transfer
        block simpletrans(value addr[31:0], value wdata[31:0], value rdata[31:0], block control) {
                haddrctrl(addr[31:0],control),data(wdate[31:0],rdata[31:0]);
        }

        toplevel is {simpletrans};
}

feature waitstate requires basictrans {
        extendpoint waitsplit(value wdata[31:0], value rdata[31:0])
                                        = block(data(wdata, rdata)) && in(simpletrans);

        block waitstate(value data[31:0]) {
                clock;
                HREADY = 0;
                HWDATA[31:0] = data[31:0];
        }

        extendrule: prepend waitsplit(wdata, rdata) waitstate(wdata) allowed;
}

feature pipeline requires basictrans {
        extendpoint pipesplit(data d, simpletrans s1, simpletrans s2) = block(d) && in(s1) && next(s2);

        //override haddrctrl
        //have to do this-the basictrans version of haddrctrl explicitly makes it a single clock cycle
        block haddrctrl(value addr[31:0], block control) {
                HADDR[31:0] = addr[31:0];
                control;
        }

        synchblock pipelined(simpletrans s1, simpletrans s2) {
                s1.data 1;
                s2.addr 1;
        }

        extendrule: concurrent pipesplit(d, s1, s2) s2 using pipelined(s1, s2);
}

//up to Figure 3-5
tabletype TransType[1:0]    {IDLE = 00b,BUSY = 01b,NONSEQ = 10b,SEQ = 11b};

feature transfertypes requires basictrans {
        block control(Transtype tt) {
                HTRANS[1:0] = tt;
        }

        block singletrans(value addr[31:0], value wdata[31:0], value rdata[31:0]) {
                simpletrans(addr[31:0], wdata[31:0], rdata[31:0], control(NONSEQ));
        }

        block busy(value addr[31:0]) {
                simpletrans(addr[31:0], dc, dc, control(BUSY));
        }

        toplevel is {singletrans, busy};
}

tabletype BurstType[2:0] {SINGLE = 000b,INCR = 001b,WRAP4 = 010b, INCR4 = 011b,
                                WRAP8 = 100b, INCR8 = 101b, WRAP16 = 110b, INCR16 = 111b};

tabletype BurstSizeType[2:0] {B8 = 000b, B16 = 001b, B32 = 010b, B64 = 011b, B128 = 100b,
                                B256 = 101b, B512 = 110b, B1024 = 111b};

feature bursts requires transfertypes {
        block burstctrl(Transtype tt, BurstType bt) {
                control(tt);
                HBURST[2:0] = bt;
        }

        //single transfer operation
        //note: overrides singletrans in prev feature
        block singletrans(value addr[31:0], value wdata[31:0], value rdata[31:0]) {
                simpletrans(addr[31:0], wdata[31:0], rdata[31:0], burstctrl(NONSEQ,SINGLE));
        }

        function incrementval(BurstSizeType bs) {
                switch(bs) {
                        case B8: return 1;
                                break;
                        case B16: return 2;
                                break;
                        case B32: return 4;
                                break;
                        case B64: return 8;
                                break;
                        case B128: return 16;
                                break;
                        case B256: return 32;
                                break;
                        case B512: return 64;
                                break;
                        case B1024: return 128;
                                break;
                }
        }

        function burstbeatval(burstType bt) {
                switch(bt) {
                        case SINGLE:
                        case INCR: return 1; //doesn't really matter, undefined size burst
                                break;
                        case WRAP4:
                        case INCR4: return 4;
                                break;
                        case WRAP8:
                        case INCR8: return 8;
                                break;
                        case WRAP16:
                        case INCR16: return 16;
                        break;
        }
}

function burstaddrcalc(value iaddr[31:0], BurstType bt, BurstSizeType bs, value transNo[3:0]) {
        if(transNo == 0) {
                return iaddr;
        }
        value nextaddr[31:0] = transNo[3:0] * incrementval(bs) + iaddr[31:0];
        if(bt == WRAP4 || bt == WRAP8 || bt == WRAP16) {
                value burstspace[31:0] = incrementval(bs) * burstbeatval(bt)
                value upperbound[31:0] = ((iaddr[31:0] + burstspace[31:0])
                                / burstspace[31:0]) * burstspace[31:0];
                                //integer division/mult to get nearest boundary
                if(nextaddr[31:0] > upperbound[31:0]) {
                        nextaddr[31:0] = nextaddr[31:0] - burstspace[31:0];
                }
        }

        return nextaddr[31:0];
}

//4 beat wrapping burst
block burst4wrap(value iaddr[31:0], value wd[31:0][4], value rd[31:0][4], BurstSizeType bs) {
        simpletrans(iaddr[31:0], wd[31:0][0], rd[31:0][0], control(NONSEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP4, bs, 1), wd[31:0][1], rd[31:0][1], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP4, bs, 2), wd[31:0][2], rd[31:0][2], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP4, bs, 3), wd[31:0][3], rd[31:0][3], control(SEQ));

        HBURST[2:0] = WRAP4,(this.simpletrans 4).data 1;
}

block burst4incr(value iaddr[31:0],value wd[31:0][4],value rd[31:0][4],BurstSizeType bs) {
        simpletrans(iaddr[31:0], wd[31:0][0], rd[31:0][0], control(NONSEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], INCR4, bs, 1), wd[31:0][1], rd[31:0][1], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], INCR4, bs, 2), wd[31:0][2], rd[31:0][2], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], INCR4, bs, 3), wd[31:0][3], rd[31:0][3], control(SEQ));

        HBURST[2:0] = INCR4,(this.simpletrans 4).data 1;
}

block burst8wrap(value iaddr[31:0], value wd[31:0][8], value rd[31:0][8], BurstSizeType bs) {
        simpletrans(iaddr[31:0], wd[31:0][0], rd[31:0][0], control(NONSEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP8, bs, 1), wd[31:0][1], rd[31:0][1], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP8, bs, 2), wd[31:0][2], rd[31:0][2], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP8, bs, 3), wd[31:0][3], rd[31:0][3], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP8, bs, 4), wd[31:0][4], rd[31:0][4], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP8, bs, 5), wd[31:0][5], rd[31:0][5], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP8, bs, 6), wd[31:0][6], rd[31:0][6], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP8, bs, 7), wd[31:0][7], rd[31:0][7], control(SEQ));

        HBURST[2:0] = WRAP8,(this.simpletrans 8).data 1;
}

block burst8incr(value iaddr[31:0], value wd[31:0][4], value rd[31:0][4], BurstSizeType bs) {
        simpletrans(iaddr[31:0], wd[31:0][0], rd[31:0][0], control(NONSEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], INCR8, bs, 1), wd[31:0][1], rd[31:0][1], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], INCR8, bs, 2), wd[31:0][2], rd[31:0][2], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], INCR8, bs, 3), wd[31:0][3], rd[31:0][3], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], INCR8, bs, 4), wd[31:0][4], rd[31:0][4], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], INCR8, bs, 5), wd[31:0][5], rd[31:0][5], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], INCR8, bs, 6), wd[31:0][6], rd[31:0][6], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], INCR8, bs, 7), wd[31:0][7], rd[31:0][7], control(SEQ));

        HBURST[2:0] = INCR8,(this.simpletrans 8).data 1;
}

block burst16wrap(value iaddr[31:0], value wd[31:0][16], value rd[31:0][16], BurstSizeType bs) {
        simpletrans(iaddr[31:0], wd[31:0][0], rd[31:0][0], control(NONSEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,1), wd[31:0][1], rd[31:0][1], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,2), wd[31:0][2], rd[31:0][2], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,3), wd[31:0][3], rd[31:0][3], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,4), wd[31:0][4], rd[31:0][4], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,5), wd[31:0][5], rd[31:0][5], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,6), wd[31:0][6], rd[31:0][6], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,7), wd[31:0][7], rd[31:0][7], control(SEQ)),
        simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,8), wd[31:0][7], rd[31:0][7], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,9), wd[31:0][7], rd[31:0][7], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,10), wd[31:0][10], rd[31:0][10], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,11), wd[31:0][11], rd[31:0][11], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,12), wd[31:0][12], rd[31:0][12], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,13), wd[31:0][13], rd[31:0][13], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,14), wd[31:0][14], rd[31:0][14], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], WRAP16,bs,15), wd[31:0][15], rd[31:0][15], control(SEQ));

                HBURST[2:0] = WRAP16,(this.simpletrans 16).data 1;
        }

        block burst16incr(value iaddr[31:0], value wd[31:0][4], value rd[31:0][4], BurstSizeType bs) {
                simpletrans(iaddr[31:0], wd[31:0][0], rd[31:0][0], control(NONSEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 1), wd[31:0][1], rd[31:0][1], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 2), wd[31:0][2], rd[31:0][2], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 3), wd[31:0][3], rd[31:0][3], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 4), wd[31:0][4], rd[31:0][4], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 5), wd[31:0][5], rd[31:0][5], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 6), wd[31:0][6], rd[31:0][6], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 7), wd[31:0][7], rd[31:0][7], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 8), wd[31:0][8], rd[31:0][8], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 9), wd[31:0][9], rd[31:0][9], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 10), wd[31:0][10], rd[31:0][10], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 11), wd[31:0][11], rd[31:0][11], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 12), wd[31:0][12], rd[31:0][12], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 13), wd[31:0][13], rd[31:0][13], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 14), wd[31:0][14], rd[31:0][14], control(SEQ)),
                simpletrans(burstaddrcalc(iaddr[31:0], INCR16, bs, 15), wd[31:0][15], rd[31:0][15], control(SEQ));

                HBURST[2:0] = INCR16,(this.simpletrans 16).data 1;
        }

        block burstincr(value iaddr[31:0], value wd[31:0], value rd[31:0], BurstSizeType bs) {
                simpletrans(iaddr[31:0], wd[31:0] ,rd[31:0], control(NONSEQ));
        }

        block burstincradd(value addr[31:0], value wd[31:0], value rd[31:0], BurstSizeType bs) {
                simpletrans(addr[31:0], wd[31:0], rd[31:0], control(SEQ));
        }

        extendpoint incrburst(value prevaddr[31:0], BurstSizeType bs) =
                (block(burstincr(prevaddr[31:0], wd, rd, bs)) && !contains(burstincradd()))
                || (block(burstincradd(prevaddr[31:0], wd, rd, bs)) && !next(burstincradd()));
        extendrule: append incrburst(prevaddr[31:0], bs)
                burstincradd(burstaddrcalc(prevaddr[31:0], INCR, bs, 1), wd[31:0], rd[31:0], bs) allowed;

        block burstincrctrl(BurstSizeType bs) {
                HBURST = INCR;
                HSIZE = bs;
        }

        synchblock burstincrsynch(burstincrctrl bic, burstincr bi) {
                bic, bi.data last;
                bi;
        }

        extendpoint burstincrblock(burstincr bi, BurstSizeType bs) = block(burstincr(*, *, *, bs) bi);
        extendrule: concurrent burstincrblock(bi, bs) burstincrctrl(bs) bic using burstincrsynch(bic, bi);

        //early termination
        extendpoint inburst(block b) = block(simpletrans) && in(b)
                && if(blocktype(b) == burst4wrap || blocktype(b) == burst4incr
                || blocktype(b) == burst8wrap || blocktype(b) == burst8incr
                || blocktype(b) == burst16wrap || blocktype(b) == burst16incr
                || blocktype(b) == burstincr;
        extendrule: before inburst(b) end(b) allowed;

        toplevel is also {burst4wrap, burst4incr, burst8wrap,
                burst8incr, burst16wrap, burst16incr, burstincr};
}
```

Bibliography

[1] Nina Alma, E. Allen Emerson, and Kedar S. Namjoshi. Efficient Decompositional Model Checking for Regular Timing Diagrams. 1703:700, 1999.

[2] ARM Limited. AMBA specification (rev 2.0). http://www.arm.com/products/solutions/amba2overview.html, May 1999.

[3] Werner Damm and David Harel. LSCs:Breathing Life into Message Sequence Charts. *Formal Methods in System Design*.

[4] Kathi Fisler. Two-Dimensional Regular Expressions for Compositional Bus Protocols. *Formal Methods in Computer Aided Design*, pages 154–157, 2007.

[5] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[6] Intel Corporation. 8088 8-Bit HMOS Microprocessor. August 1990.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ.

[8] Marco T. Oliveira and Alan J. Hu. High-level specification and automatic generation of IP interface monitors. pages 129–134, 2002.