Towards an Aspect Language for Bus Protocols

Kathi Fisler, Paul Freitas, Dan Bjorge

WPI Dept of Computer Science
Worcester, MA 01609
kfisler@cs.wpi.edu

Abstract

Hardware-level protocol specifications provide an interesting case study for aspect-oriented programming. Bus protocols are defined around events and values that hold between events. Variables within the protocol synchronize around events. Aspects must alter both events and values on multiple variables while maintaining synchronization. In effect, these are two-dimensional aspects, crosscutting both variables and time. This paper explores and contrasts two styles of aspects for capturing such protocols, using a widely-used bus protocol as a running example. Our main contribution lies in raising questions about how AOP can support domains with highly synchronized, two-dimensional aspects.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords bus protocols, synchronization in aspects, aspects for functional reactive programming

1. Introduction

Protocol-specification documents are often organized as a series of behavioral extensions to a simple core protocol. Formal models of protocols, however, rarely share this structure. This complicates tracing elements of the formal model back to the specification, and makes it hard to quickly get a formal model of only a subset of a protocol's features. An AOP framework for protocols could aid in both problems.

Protocols are typically described as sequences of events on multiple signals. Events on one signal may trigger responses on another. Events and responses may need to be synchronized, or at least ordered, across signals. Hardware-level protocols require extensive synchronization across signals; many such protocols are more readily understood as two-dimensional sequences of *windows* with synchronized endpoints over multiple signals. This tight synchronization implies that extending the behavior of one signal may implicitly extend another. As far as we know, the AOP literature does not discuss extending systems with this tight, interdependent synchronization across multiple signals.

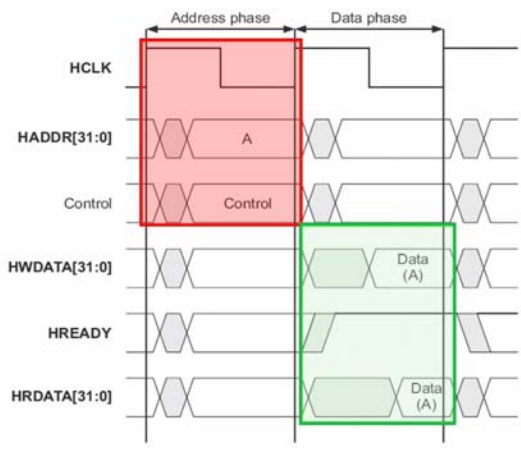

Figure 1. A basic transfer protocol (© ARM)

This paper uses a real, heavily-used, high-performance bus protocol (Section 2) to explore two approaches to creating aspect-like mechanisms for this domain. One makes synchronization windows the atomic unit of composition and advising (Section 3); the other takes window-bounding events as atomic (Section 4). Each effort leaves several open questions (Section 5) about how to provide aspect-like mechanisms for two-dimensional extensions.

2. A Sample Protocol and its Components

Bus protocols coordinate data transfers between devices. The AMBA™ protocols are among the most commonly used in modern 32-bit processors, including those embedded in many smartphones. This paper uses the AHB variant of AMBA-2 [1].[1] The full specification runs 230 pages and covers three protocol variations with different performance targets (the differences are beyond the scope of this work).

[1] Unannotated figures reproduced with permission of ARM Limited.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DSAL'12, March 27, 2012, Potsdam, Germany.
Copyright © 2012 ACM 978-1-4503-1128-1/12/03. . . $10.00

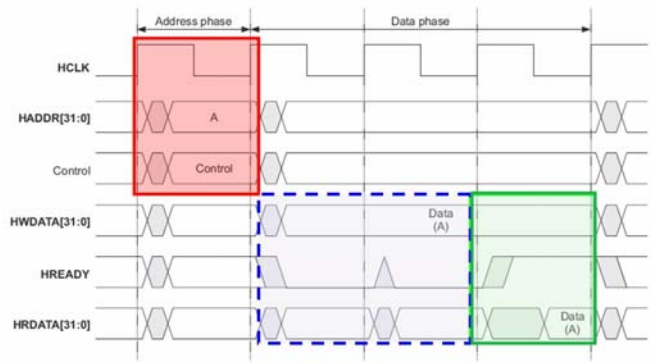

Figure 2. Adding waitstates to the basic protocol (© ARM)

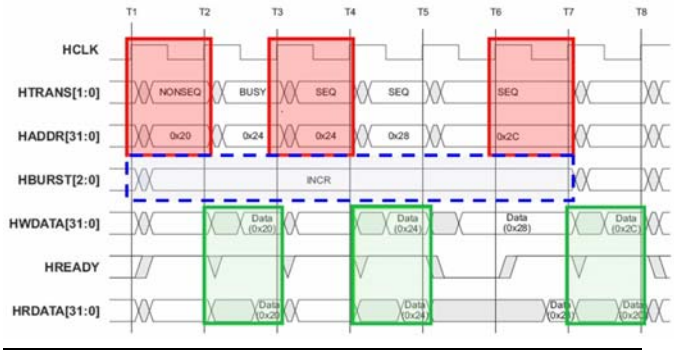

Figure 3. A sequence of burst transfers (© ARM)

Base Protocol Figure 1 shows the most basic AMBA-2 data transfer (read or write). The `HCLK` signal is a standard system clock. `HADDR` carries the address for the transfer. `Control` specifies various parameters for the transfer that are irrelevant for this discussion. `HWDATA` and `HRDATA` contain the data to be written to or read from memory, respectively (only one of these would be active in any one instance of this protocol). `HREADY` indicates when the device is ready for the next transfer request to be initiated. The `[31:0]` annotation on a signal name indicates a 32-bit variable; un-annotated signals carry only single-bit values (1 and 0, corresponding to true and false, respectively). Shared names (such as "A") indicate identical data on different signals. Unshaded, unlabeled polygons represent periods in which the signal must hold a steady value, but the exact value is irrelevant. The small shaded polygons indicate periods of instability in signals. This hardware issue is not relevant to this paper.

We have annotated the diagram with two boxes, one enclosing the events and signals for the address phase and another for the data phase (the phases are labeled at the top of the diagram). Box edges capture synchronization points across signals: all signals active in the address phase, for example, hold their values over the same time window.

Protocol Extensions Figure 2 illustrates the base protocol execution extended with new behavior. If the device handling the data transfer is busy, it may not respond with the data immediately after the address has been issued. In this case, the protocol inserts a sequence of *wait states* (enclosed in dashes) into the data phase. During a wait state, the `HREADY` signal remains low (indicating that the device is not ready to start a new transfer); if the requested transfer was for a write, the data value must hold during the wait states (until the device is ready to grab the data). The boxes illustrate that waitstates correspond to a cohesive block that gets inserted into the data phase of the basic transfer protocol.

Figure 3 shows an extension to pipelined protocol executions to handle so-called *burst* transfers. Bursts consist of several consecutive requests to adjacent memory addresses (as seen in the consecutive address values on `HADDR`). The dashed box shows the newly-added behavior: the `HBURST` signal, whose value spans several pipelined instances of the basic protocol. We will ignore the `HTRANS` signal, which comes from an extension not described in this paper.

Language Requirements These examples illustrate several kinds of extensions that arise in bus protocols:

1. Extend existing phases, effectively changing their endpoints (wait states, Figure 2)
2. Add new phases into existing protocols (not shown).
3. Add new signals that may span existing windows (burst controls, Figure 3)
4. Execute windows from different protocol instances in parallel (pipelining, Figure 3)

3. A Window-Centered Approach

The diagrams in Section 2 suggest viewing protocols as sequences of windows, with aspects inserting new windows into existing sequences. We have fully formalized such a language [6], including joinpoints, pointcuts, and modules for individual extensions. Due to space limitations, we present only the key ideas here.

3.1 The Basic Protocol

A *block* defines a window of behavior over a set of signals. Atomic blocks assign a value to a signal. Compound blocks compose other blocks both sequentially and in parallel. When blocks are composed in parallel, the events at each edge of the blocks are synchronized across all their signals; events internal to the blocks are unordered. The following compound block sequentially composes a block for a basic address phase (`haddrctrl`) with one for a basic data phase (`data`) to define the transfer from Figure 1.

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

3.2 Aspects over Blocks

Pointcut specifications and advice for this language are also defined in terms of blocks. We illustrate our language using the waitstates extension from Figure 2.

Joinpoints Advice inserts new blocks around, or inserts new data into, existing blocks. Accordingly, pointcuts identify blocks, rather than points within blocks. Patterns over block names and parameter values identify blocks; given a pattern, the pointcut indicates whether to select blocks that either match the pattern or precede, follow, or are nested within a block that matches the pattern.

Advice Advice blocks may compose sequentially (as in waitstates) or concurrently (as in bursts) with existing blocks. Sequential compositions raise a question: after composition, should the new block be considered part of the advised block (within its start and end points)? Capturing this variation yields a total of five advice types: *append* and *prepend* for sequential composition that augments the advised block, *before* and *after* for sequential composition adjacent to the advised block, and *concurrent* for parallel composition with the advised block (the new and advised blocks will form a new block with synchronized start and end points).

Aspects To support extensions that introduce several pieces of advice, our language provides a construct that encapsulated related advice into a single aspect. Additional details are not relevant to this paper.

3.3 Limitation of the Blocks-based Approach

Blocks attempt to structure increments of protocol behavior and where increments compose to form full protocols. They nicely capture individual protocol instances, but not the conditions that trigger different protocol instances. In the basic transfer (Figure 1), for example, only one of `HWDATA` and `HRDATA` (the data for a write or read transfer, respectively) would actually hold relevant data in a single protocol instance. A read/write flag in the `Control` signal logic indicates the transfer type. Other AMBA-2 features abound with similar control decisions.

Using blocks as joinpoints demands that any potential extension point in a protocol be encapsulated in a block. This leads to a deep nesting hierarchy of blocks. While this hierarchy nicely captures the interplay of sequential and parallel composition within a protocol, it also exposes complexity. For example, if advice needs to add parameters to a block, all blocks from the root of the hierarchy to the advised block must be updated.

These observations taken together suggest exploring finer-grained approaches to aspectualizing bus protocols that also account for control logic.

4. A Finer-Grained Approach Using FRP

Functional Reactive Programming (FRP) is a declarative, stream-based, dataflow-inspired, event-driven programming model. Section 4.1 argues by example that FRP seems a natural fit for capturing bus protocols: events capture synchronization points, while other standard operators capture window contents between events. FRP also supports basic control operators. Aspects for FRP have not been studied, however. Our question then is whether we could aspectualize FRP to capture the design increments of bus protocols. We used the FrTime [3] FRP language for this project.

4.1 Windows and Boundaries in FRP

Streams of values or events are the atomic units of FRP programs. In a bus protocol, each clock transition would yield an event, as would each change in value on the `HADDR` signal (indicating that a new transfer should begin). Streams of events are called *event streams*. Streams of values defined based on events are called *behaviors*. The contents of the data signals in a bus protocols are examples of behaviors.

Operators in FRP languages combine streams, filter datapoints on streams, or build new streams based on other behaviors or event streams. New address phases begin, for example, when a rising event on a clock stream coincides with a change-of-value event on the `HADDR` stream. To define window contents, we use a FrTime operator called `hold` that creates a behavior out of an event stream: whenever the argument stream to `hold` has an event, the resulting behavior has the value of that event until the next event on the argument stream. The streams `val-events` and `hold(val-events)` in the following diagram illustrate this operator:

```
            events e1      e2    e3      e2
        val-events a       b     c       b
  hold(val-events) a a a a b b b c c c c b
```

Windows in protocols typically have different values from those of their triggering events; this is evident when the same triggering event defines a window over several signals. Capturing AMBA-2 therefore requires translating a triggering event to desired window values. FrTime's `map-e` operation, analogous to a map operation in many programming languages, translates each event to a new event while maintaining the event-stream structure. The previous diagram also illustrates a stream `events` that is mapped into the stream `val-events` (whose values are subsequently held to define window contents).

Combining `hold` and `map-e` in the spirit of the previous diagram yield a template for defining an individual bus-protocol signal using FrTime:

```
(define <SIGNAME>
  (hold
    (map-e (lambda (event)
             (cond [(eq? event <BOUNDARY1>) <VAL1>]
                   [(eq? event <BOUNDARY2>) <VAL2>]
                   ...))
           (merge-e <BOUNDARY1> <BOUNDARY2> ...))))
```

This template references streams of events that initiate new protocol windows (`<BOUNDARYx>`). The template merges

these individual events into a single event stream (using `merge-e`). The function argument to `map-e` (the `lambda` expression) translates each boundary event into its corresponding value event (`VALx`). Applying `hold` to these value events yields the desired window contents.

The template code suggests that this approach synchronizes signals that share boundary events. Thus, two signals that are synchronized would share the same `merge-e` arguments. Signals that are partly synchronized would share some, but not all, `merge-e` arguments. Having window boundaries easily identifiable in the code enables us to reason about some relationships between signals.

The following expression uses the template to capture `HADDR` from the basic protocol (Figure 1). The expression assumes an event stream `clock-rise` defining clock transitions and a function `gen-addr` that generates the memory address to put on the bus:

```
(define haddr
   (hold
     (map-e (lambda (event)
              (cond [(eq? event true) (gen-addr)]))
            (merge-e clock-rise))))
```

4.2 Aspects in FrTime

To explore joinpoints and advice in FRP, we consider how the waitstate and burst extensions would alter the definition of `haddr`. Under the waitstates extension, a new address phase begins on a rising clock when the `HREADY` signal has value `true`. There are two candidate places to add the `HREADY` constraint to the current definition of `haddr`:

1. In the generation of window-bounding events (within the call to `merge-e`).
2. In the test position of the `cond` statement that checks the event to decide whether to generate a new address.

The first approach maintains a clear separation between window boundaries (in `merge-e`) and window contents (in the `cond`). This in turn makes it easier to see which signals are synchronized within the overall protocol. Under waitstates, the `merge-e` expression in `haddr` would look like:

```
(merge-e (filter-e
           (lambda (e) (value-now hready))
           clock-rise))
```

Adding burst transfers (Figure 3) alters the values within windows, rather than the triggering events. This modification must happen within the `cond`, at the point where `gen-addr` is called. Whether a new address should be generated is determined by the value on a new `hburst` signal (not shown). The revised `cond` clause within `haddr` would be:

```
(cond [(eq? event true)
       (cond [(eq? hburst 'incr) next-addr]
             [else (gen-addr e)])])
```

where `next-addr` contains consecutive memory addresses.

4.3 The Notion of Advice in FrTime

Our limited example raises some interesting questions about the nature of advice in an FRP-based approach. Modifying window boundaries may involve adding, deleting, or modifying existing boundary events. Semantically, the FrTime operators `merge-e`, `filter-e`, and `map-e` perform these transformations (where `filter-e` removes events that fail a given predicate, analogous to the standard `filter` operator in functional languages). This suggests that advice on window boundaries should take the form of a transformation function that is applied to the current event stream.

Modifying window contents may involve adding or deleting entire `cond` clauses or modifying the answer components of existing clauses. For protocols, we have not seen the need to modify the `cond` test conditions (which implies that the content of an event, rather than its existence, changed). Adding clauses is straightforward as events are distinct (which makes the order of the clauses irrelevant within the `cond`). Advice that modifies the values within `cond` answers is trickier because the value expressions can be arbitrary code with minimal structure to guide identification of joinpoints. In our work on AMBA-2 to date, we have found that aspects alter the entire existing value within a window; this allows advice to be a function that transforms the value, rather than a syntactic operation on the code that computes the value. Whether this approach suffices for a large range of protocols remains open for future work.

There are two key insights here. First, traditional advice types such as *before* and *after* are not relevant to modeling bus protocols with FRP. Ordering arises among events; FRP operators such as `merge-e` manage ordering automatically. This enables a much simpler model of advice as a transformation function on event streams. Second, advising events and window contents are separate activities over code with different structure. It therefore makes more sense to design separate advising languages for these two components, while retaining a way to use both languages to define a single aspect for a particular extension.

5. Lessons Learned

This paper explores the nature of aspect languages for capturing bus protocols, a domain characterized by deeply nested sequential and parallel compositions with synchronized boundaries across signals. We explored the tradeoffs between two high-level approaches: one that explicitly models windows synchronized across multiple signals (blocks), and one that makes synchronization implicit around common events on individual signals (FRP). Several issues inform the future design of languages for this domain:

Adding windows Advising a protocol with a new window over multiple signals is conceptually simpler when windows are explicit (as with blocks). If signals are independent (as in FRP), the aspect must separately advise every signal involved in the window. A language built atop FRP should pro-

vide domain-specific constructs for advising multiple signals with a single command to support this extension.

Adding signals spanning a sequence of windows The burst transfer added a new window containing a signal `INCR`. This window synchronizes in parallel with a sequence of blocks for instances of the basic protocol. This sequence is interesting because the number of instances (consecutive addresses) in the burst is specified dynamically within the `Control` signal. Our current blocks-based approach does not support these dynamic conditions. The FRP approach does, but exposes another issue: the triggering event for the first instance in the sequence of burst transfers needs two interpretations: a normal event for starting a transfer (to trigger the appropriate behavior on `haddr` and other signals), and an event designating a burst sequence (for synchronizing with the `INCR` signal). Since FRP leaves windows implicit through events, it would need an explicit mechanism to manages these two interpretations, both within the code and for a human trying to understand the protocol from the code.

Revising Window Contents The FRP approach needs an explicit language for advising block contents, as the contents of an individual block get specified within the code for an individual signal. The block approach, in contrast, avoids this need by taking block contents as a parameter. Relying on parameters to modify block contents, however, can break obliviousness if an extension must modify contents that are not already parameterized: adding a parameter to one block requires adding that parameter to every block containing it in the nesting hierarchy. A block-based protocol language needs scoping rules and mechanisms for less invasive modification to deeply nested block contents.

Overall Assessment Overall, the shortcomings of the FRP language appear more cosmetic, amenable to careful design of a domain-specific language for bus protocols. The shortcomings in the blocks approach feel more systemic: the model is simply too rigid for incremental protocol development. Moving forward, we would like to investigate a blocks-inspired DSL atop FRP that exposes the window-bounding events through joinpoints. Before doing so, we need to figure out how to handle extensions in which one run of a protocol is suspended while another executes. This has proven tricky to capture because the pending transfer resumes only in a specific context, rather than under a general triggering condition. Trace-based aspects [4] might help, though these are generally rare in bus protocols.

6. Related Work

Despite our emphasis on events, our domain does not require event-based aspects (which focus on dynamic, rather than static, joinpoints). Our joinpoints on events primarily support synchronized concurrent behavior across protocol signals. Nùñez and Noyé [7] use coordinating processes to model concurrent execution of aspects in Java. Their advice language is not, however, tailored to the distinctions needed to advise contents of windows within protocols.

Several systems propose multiple aspect languages in the context of the same problem. Bockisch et al. [2] advocate separating the language of events that trigger advice from the language being advised, but do not support advising the event specifications themselves. Tanter and Noyé [8] propose a kernel and architecture for integrating aspect languages for different high-level concerns, whereas our domain requires different languages for fine-grained concerns that interact. Tarr et al.'s work on multi-dimensional separation of concerns [9] supports different abstractions (requirements, object hierarchies, code) that form a single feature. Our domain extends one artifact—the protocol behavior—that has rich constraints in two dimensions: synchronization across signals, and behaviors of individual signals within new features. Each of our blocks and FRP approaches emphasizes one of these two dimensions, which illustrates the dominant decomposition problem in Tarr et al's work.

França et al. [5] model the increments that comprise bus protocols, but do not propose techniques to actually compose protocols from these increments.

References

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

[2] C. Bockisch, S. Malakuti, M. Akşit, and S. Katz. Making aspects natural: Events and composition. In *Proceedings of the International Conference on Aspect-Oriented Software Development, Modularity Visions Track*, 2011.

[3] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, Mar. 2006.

[4] R. Douence, D. L. Botlan, J. Noyé, and M. Südholt. Trace-based aspects. In *Aspect-oriented Software Development*, pages 141–150. Addison-Wesley, 2004.

[5] R. B. França, J. Farines, J.-P. Bodeveix, L. B. Becker, and M. F. Amine. Modeling a bus protocol: An incremental approach. In *Workshop on Real-Time Systems*, 2007.

[6] P. M. Freitas. Feature-oriented specification of hardware bus protocols. Master's thesis, WPI Department of Computer Science, 2008.

[7] A. Núñez and J. Noyé. Baton: A domain-specific language for coordinating concurrent aspects in java. In *Journée Francophone sur le Développement de Logiciels Par Aspects*, 2007.

[8] E. Tanter and J. Noyé. A versatile kernel for multilanguage AOP. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 173–188, 2005.

[9] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software engineering*, ICSE '99, pages 107–119, 1999. ACM.