Processor Memory System Verification using DOGReL: a language for specifying End-to-End properties

Daryl Stewart*, David Gilday*, Daniel Nevill*, Thomas Roberts*

*ARM Ltd.
110 Fulbourn Road
Cambridge, Cambridgeshire, GB-CB1 9NJ
Email: *firstname.lastname*@arm.com
http://www.arm.com/

Abstract—**The memory subsystem of a microprocessor is responsible for scheduling memory accesses as efficiently as possible, hiding latency costs from the Data Processing Unit and, ideally, minimizing power costs. For ARM processors, requirements on ordering and completion of memory accesses from the ARM Architecture and relevant bus protocols specify an envelope of acceptable behavior that must be maintained by the schedule. Consequently, any given sequence of instructions can be satisfied by a range of different schedules. Verifying the correctness of the memory subsystem is therefore a complex task, often on the critical path to release.**

Inspired by earlier work on specifying end-to-end properties we developed a language of Directed Observer Graphs (DOGReL) capable of expressing implementation independent properties at a level of abstraction between architecture and micro-architecture that specify the behavior of the memory subsystem. These properties are then translated into System Verilog monitors which can be applied for bug-hunting at the RTL level using commercial Model Checking tools. During a recent micro-controller design project at ARM these properties proved highly valuable in detecting bugs earlier in the design cycle than simulation. The process of writing an unambiguous specification also exposed early design flaws and provided clearer documentation of the required behaviors for engineers.

Index Terms—**Formal models, interfaces, Register-Transfer-Level (RTL) implementation, verification.**

I. Introduction

The use of commercial model checking tools to aid the verification of hardware designs is becoming more commonplace in industry. Typically, properties are either created manually or generated automatically, associated with RTL using SystemVerilog Assertions (SVA) [1] or the Property Specification Language (PSL) [2] and checked both in simulation and in a model checker. In the case of automatic generation, the properties are either of some template form targeting a specific area of functionality, e.g. checks for undriven wires which influence observable behavior, or they are simple properties extracted from some analysis of the RTL or existing simulation runs. The manually created properties are often akin to a C-programmer's inclusion of 'assert (p)' although as designers become more appreciative of the power of formal tools they will venture into multi-cycle properties and more complex checks. Cover properties are also considered valuable for generating waveforms. A testbench must be programmed with the appropriate stimulus to exercise the desired behavior, whereas a cover expresses the desired behavior directly. This makes the cover more resilient in the face of implementation changes during design development.

Proving even these simple properties on commercial processor designs is beyond the power of current model checkers, so a divide and conquer approach is used, known as 'unit level formal' at ARM. This is an assume-guarantee based methodology which requires a set of properties, or *interface specifications*, to be written about the signals at the interfaces between subunits of the design.

In [3],[4] we proposed a 'chain of reasoning', with verification at various levels of abstraction which are undertaken in the most suitable *style* at each level, but with a suitable *meaning* for combining the methods, with the benefits described by Clarke, Wing, et al. [5]. This framework would allow guarantees at the system or software level to be verified against the RTL implementation as well as unambiguous specifications at each level.

In this paper we present the results of recent work, undertaken alongside the development of a new microcontroller codenamed Pelican with the goal of raising the level of complexity of properties which can be expressed.

A. Pelican Deep Formal

In previous work [6] on the ARM® Cortex®-A5, Stuart Hoad successfully created an abstract model of the Snoop Control Unit[1] (SCU) in a style similar to that of [7] and proved a single end-to-end property on the model about the ordering of transactions through the SCU. This abstract model included under-specified (non-deterministic) behavior in the form of assumptions. The RTL implementation was too large for the property to be proved directly, but assumptions underlying the

[1]The SCU maintains data cache coherency between multiple cores.

model were proved as assertions on the RTL implementation to provide a refinement based argument that the property also held on the RTL. The original goal of the Pelican Deep Formal (PDF) project was to reproduce this on a larger scale for an entire *first level memory system* (L1 MS).

The L1 MS is responsible for scheduling memory accesses required by the processor's *Data Processing Unit* (DPU) at a variety of memory interfaces including standard AMBA® AHB™ and AXI™ interfaces [8], and Pelican specific *Tightly-Coupled Memory* (TCM) interfaces. Features such as exclusive memory access, write-merging, memory barriers, Error Checking and Correction (ECC) and other error handling, caching and forwarding are implemented in the L1 MS. The abstract model would be based on Pelican's micro-architectural specification and a complete set of end-to-end properties covering all memory transactions would be proven against the model. Since the feasibility of these tasks had been demonstrated on Cortex®-A5 on a small scale, it was assumed that the task of creating a larger micro-architectural model and the refinement proofs would be the most challenging tasks.

However, it soon became apparent that the description of the required end-to-end properties was more difficult than anticipated. In particular, the properties included concepts such as universally quantified transaction labels (section III-A1), non-deterministic choice of transactions (section III-A2), predicted values (section III-A3), transaction lifetimes (section III-C) and memory commitment ordering (section III-D). As we identified each of these needs, and struggled to express them using SVA, we found ourselves drawing progressively more formalized diagrams and eventually we converged on a definition of *Directed Observer Graphs* (DOGs), which are the main subject of this paper.

With each new concept we also devised a translation into RTL capable of identifying the corresponding behavior, and so we were able to apply the DOGs directly to the RTL at a very early point in Pelican's design. The implementation of these formal monitors comprises: a reusable interface *abstractor* (ABS) per interface, which is responsible for interpreting signal-level activity as corresponding transaction-level events; and a *monitor* (MON) per DOG, which selects and verifies events provided by the ABSs. Although this translation was performed by hand during PDF, we have subsequently written a tool which compiles a textual description of DOGs written in *DOG Representation Language* (DOGReL) to RTL and SVAs. Throughout section III we describe the features of DOGs in terms of DOGReL. We show that DOGReL is a general purpose language in the sense that the user can define an ABS/MON interface which expresses any kind of transaction events, rather than being restricted to memory-system descriptions.

As with much formal verification work, benefit was derived from the act of creating an unambiguous, executable specification, including the discovery of bugs and ambiguities without running a formal tool. Despite the novelty of the methodology, the use of DOGs in *bug-hunting* began before Pelican's alpha milestone, at the same time that testbenches were coming online. This allowed us to compare PDF with unit and top-level simulation. We present results in more detail in section V.

The benefits realized are compelling enough that the methodology is being adopted on further processor designs. In section VI we describe our plans for extending DOGs to deal with coherent memory systems and to continue research into usable micro-architectural modeling techniques.

II. L1 Memory System

A processor's Memory Subsystem (MS) may comprise elements such as: *Load Store Unit* (LSU); data and instruction caches (D$ and I$); STore Buffers (STB) for merging and forwarding write data; ECC logic; a Memory Protection Unit (MPU); TCM; and external memory interfaces with associated read and write buffers managed by a Bus Interface Unit (BIU). The ARM Instruction Set Architecture (ISA) includes features such as barriers for enforcing strong ordering between memory accesses and load/store exclusive instructions, which allow implementation of semaphores and require an *exclusive monitor* within the memory system.

The LSU of the memory system is responsible for receiving and fulfilling requests from the core's pipelined DPU, in accordance with the memory ordering requirements of the ARM ISA. These two components are strongly connected: in particular, the Pelican DPU and LSU implementations have matching pipeline structures, and transactions progress along the two in lockstep.

Our approach was to focus on the behavior guaranteed by the micro-architecture to the architecture. This is the contract undertaken by the L1 MS to ensure that interactions between the DPU and the memory interfaces are architecturally valid. The micro-architecture is defined in an English text document, and is couched in terms of buffers and queues manipulating 'reads', 'writes', 'stores', 'loads', 'barriers' etc. Previous researchers [9] have described the importance of observing the beginning and end of overlapping transactions and our properties tend to be designed to confirm that, once a transaction at the LSU/DPU interface ends, the appropriate activity has been observed at a memory interface or elsewhere to justify the outcome.

Identifying a particular transaction in order to track its progress, for example through the LSU pipeline, is not particularly problematic. However, correlating with events observed at another point in the system requires some way of labeling events with the transaction which triggered them. In order to solve this problem we chose not to modify the design by adding extra labeling fields: instead we use the data values within the transactions to deduce the correlations. For example, by constraining traces such that exactly one transaction initiated on the LSU/DPU interface makes reference to a specific data value, we can observe transactions leaving the L1 MS with the same data value and deduce a connection. Our properties therefore refer to free variables which we call *oracles* which are described further in section III-A1. This technique has the advantage of working in a black box mode, but it also means that we cannot distinguish transactions with

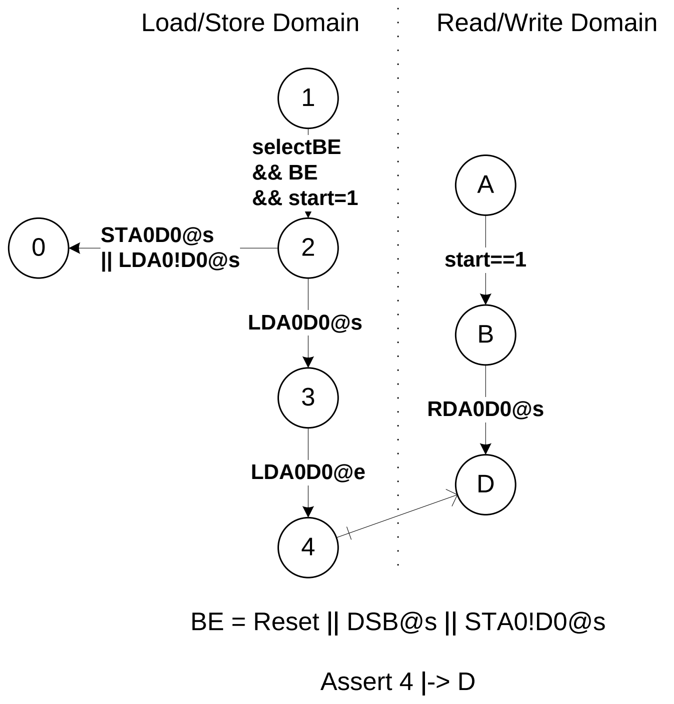

Fig. 1: Uniprocessor Load DOG

matching data. Pragmatically this requires us to check only for *Data Independent Bugs* (DIBs); an example of this is given in section IV.

Consider also the requirement to identify the start of a transaction using the data it observes. For a load request from the DPU this data is not available, and will not be, until a read or other forwarding completes at some later time. We therefore make use of the prediction mechanism described in section III-A3 to identify such events.

As for determining the ordering of memory events, we must deal with the fact that the order in which transactions are dispatched by the BIU to an external memory interconnect may not be the order in which the transactions are received by memory - the interconnect is at liberty to reorder memory accesses within certain limits such that weak memory ordering is maintained. We therefore devised a notation to identify the *commit order* of transactions which is described in section III-D.

III. Directed Observer Graphs

In this section we will use the DOG in Fig. 1 which expresses the *Uniprocessor Load* property to illustrate various features of DOGReL and DOGs.

Note that: a DOG consists of one or more finite state machines (FSMs); each transition is labeled with an *event expression*; and there is an implication arrow between some states, e.g. `4` and `D` in Fig. 1. The implication arrow represents an assertion which must not be violated - in Fig. 1 this assertion is also stated textually. Event expressions may have

Pattern	Matches
Ai	$attr = Ai$
A#	$\exists i.attr = Ai$
A?	True

TABLE I: Oracle Patterns for observable events

observable events, *selectors*, *defined events*, *assignments* or *conditional expressions* as subexpressions. These are covered in the rest of this section. The FSMs are not clock-synchronous: they may execute as many transitions as their event expressions allow, and if it is possible for transitions to be interleaved in such a way that an assertion fails then that trace is a legitimate Counter-EXample (CEX).

A. Undriven Variables

In SystemVerilog, undriven variables can be conveniently used to implement universally quantified variables. DOGs make use of these in three distinct ways, described in this section.

A DOG may also annotate a transition with variable assignments which are executed when a transition is taken. These variables may be atoms of an event expression, allowing us to synchronise transitions in different FSMs: see for example the use of the variable `start` in Fig 1.

1) Oracles: Oracles are universally quantified variables which are used to allow the formal checkers a free choice of value for some of an event's attributes. The `address` oracle declaration in Fig. 2 creates the *oracle symbol* `A` and two oracle variables, `A0` and `A1`. The patterns usable in observable events and the values they match are shown in table I. A pattern may also be negated by prefixing with '!'. Note that !A? is not used, since it is always false.

In SystemVerilog, oracles are implemented as *pseudo-constants*. That is, they are undriven but are subject to an assumption `$stable(oracle)` which ensures that whatever value is chosen, it is constant throughout the trace.

2) Event Selection: We allow transition events to include a *selector* which is arbitrarily true or false at any point in time. This allows a DOG to express a requirement for an arbitrary instance of an event, as opposed to the next occurring event, i.e. a non-deterministic choice can be made about whether to take a transition. For example, due to the inclusion of the selector `selectBE`, the first transition in the Load/Store domain of Fig. 1 may occur when a `Reset`, `DSB @s` or `ST A0 !D0 @s` can be observed, or we may choose to stay in state 1 and wait for another `BE` to come along.

In SVA, selectors can be implemented by declaring an undriven wire, which allows the formal tool to use any value to justify a CEX.

3) Prediction Mechanism: The prediction mechanism is not used directly in DOGs, but is used in ABS implementations so that the ABS may speculatively report transactions before they are observed in the system. By constraining out incorrect speculation, we ensure that only correctly speculated transactions are reported in counter-examples.

group_decl	::=	“attribute” size? attribute_name (“is” “(” attribute_id_list “)”)? “;”
	|	“oracle” size oracle_name “is” oracle_symbol “of” attribute_name “;”
	|	“event” event_name “is” event_symbol (“with” event_component_list)? “;”
	|	“alias” event_name “with” alias_component_list “as” symbol
event_component	::=	oracle_name
	|	attribute_name
	|	“[” event_component_list “]”
alias_component	::=	event_actual
	|	“[” alias_component_list “]”
observable_event	::=	“COMPLETE”
	|	event_symbol event_actual* start_end ordering?
event_actual	::=	oracle_pattern
	|	attribute_id
	|	“!” event_actual
start_end	::=	“@s” | “@e”
ordering	::=	“*”
	|	“*+”
	|	“*!+”
	|	“*-”
	|	“*!-”

TABLE II: Syntax for Event Declaration and Use

```
pack memsys32 {
  group globals {
    attribute [1] boolean;
    attribute [31:0] addressTy;
    attribute [7:0] dataTy;
    attribute [1:0] cacheTy is (NC, WBWA, WT, WBnWA);
    attribute [2] exclusiveTy is (EXOK, EXNOK, NEX);
    ...
    oracle [2] address is A of addressTy;
    oracle [2] data is D of dataTy;
    oracle [2] inner is CI of cacheTy;
    ...
  }

  group lsu {
    open globals;

    event load_event is LD
      with address, data, [exclusiveTy], [inner];
    alias load_event // good load event
      with address, data, !EXNOK, [inner] as LDg;
    alias load_event  // good load exclusive event
      with address, data, !NEX, [inner] as LDgEX;
    alias load_event // load event matching oracle 0
      with address, data, [exclusiveTy], CI0 as LDg_or0;

    event store_event is ST
      with address, data, [exclusiveTy], [inner];
    alias store_event  // good store event
      with address, data, !EXNOK, [inner] as STg;
    alias store_event // good store exclusive event
      with address, data, !NEX, [inner] as STgEX;
    alias store_event // store event matching oracle 0
      with address, data, [exclusiveTy], CI0 as ST_or0;

    event systemBarrier is DSB;
  }
}
```

Fig. 2: Group LSU written in DOGReL

A prediction mechanism can be implemented in an ABS by leaving the predicted attribute of the event undriven. Its arbitrary value is preserved in whatever data structure represents the transaction, until it reaches a point where the actual value is available in the system. As mentioned in section II we must speculate data values for reads at the LSU/DPU interface, for which confirmation occurs at the LSU pipeline stage where the data is returned from the L1 MS. At this point, known as the *point of confirmation*, an SVA assumption that ‘the predicted value is equal to the observed value’ ensures that any CEX which can be reached in which the prediction is wrong is constrained out of the model check.

A prediction mechanism is also used to determine whether or not a transaction will complete. For example, the DPU may choose to cancel a previously requested load, or the ECC unit may trigger a retry, in which case the original transaction should not be observed as an `LD` since it does not execute as a load. Instead, the ABS may predict the failure and present a `LDkill` or `LDretry` event (not shown in Fig. 2) to the MON.

One caveat of this method is that the final assertion of the DOG must not be evaluated until every ABS involved indicates that all the predicted values in observed events have reached their points of confirmation. If this is not done then a false CEX with a bogus predicted value may be returned.

B. Observable Events

DOGReL allows the user to declare *event types*, which have a unique name, an *event symbol* and a list of attributes that are part of the transaction. Each observable event consists of an event symbol followed by a pattern and is terminated with a start event (`@s`) or end event (`@e`) symbol. The pattern must correspond to the attributes which are acceptable for the given type of event.

Table II shows the two syntactic forms of an observable event. The observable event COMPLETE matches when an end event has occurred for every observed event. The second form allows the specification of a pattern based on a particular event (or alias) symbol. Any event reported by the ABS must include matching values of oracles and attributes as specified by event_actuals as well as the start_end and ordering requirements, see sections III-C and III-D. The permitted oracle_patterns are given in table I.

To help improve readability of DOGs, further event symbols for commonly used observable events may be declared using the `alias` declaration. These aliases may require a different subset of the event’s attributes: any attributes which are not required must be given a default value in the alias declaration. Attributes which are optional for the main event symbol may also be specified as mandatory for an alias event symbol.

To illustrate this, consider Fig. 2 which shows simplified declarations for event types named `load_event` and `store_event` which can be observed on the LSU/DPU interface.[2] The `load_event` declaration enables the use of

[2]Not to be confused with read and write events which occur at memory interfaces.

observable events starting with the event symbol `LD` followed by two required patterns which match `address` and `data` oracles, and two optional patterns which match `exclusiveTy` values (which denote ARM load/store exclusive instructions) and the `inner`[3] oracle (which denotes cacheability of the transaction). Where an oracle is specified, an observable event may have either a pattern from table I which matches particular oracles or a value from the oracle's attribute type. This means that the pattern matching the `inner` oracle may be an oracle pattern such as `CI0`, or a value of `cacheTy` such as `NC`.

For example, the definition of `load_event` allows the use of the observable event:
`LD A0 D0 @s`
which matches the 'start `load_event`' with `address==A0 && data==D0` and any `exclusiveTy` or `inner`. This can be seen in Fig. 1 on the transition from state 2 to state 3, hence that transition can only be taken when the ABS observes such an event in the system.

Another instance of an observable load event is:
`LD !A0 D# !EXNOK NC @s`
which matches the 'start `load_event`' with `address!=A0 && (data==D0 || data==D1) && exclusiveTy!=EXNOK && inner==NC`.

This can be written more concisely using the `LDg` alias: `LDg !A0 D# NC @s`.

The `exclusiveTy` is used by the ABS to describe whether the transaction is a load/store exclusive, and also whether it succeeded in being executed as an exclusive access (non-exclusive accesses are `NEX`). Although it is unknown when the transaction begins whether it will succeed or not, a prediction mechanism (see section III-A3) can be used to signal EXOK or EXNOK for success or failure respectively in the start event.

As a further aid to conciseness, we permit the declaration of a symbol as equivalent to an event expression. An example is shown in Fig. 1 with the definition of symbol `BE`.

C. Transaction Lifetimes

A particular transaction may be present in a system for a period of time, rather than occurring instantaneously. We distinguish between the event denoting the start of a transaction and the event for its end by suffixing the observable event with `@s` or `@e` respectively.

Returning to Fig. 1 we can see that the transition from state 2 to state 3 occurs when a `load_event` for address `A0` which fetches data `D0` starts. When the FSM makes this transition, the `load_event` responsible is said to become an *observed event* and cannot cause further transitions. This distinction is necessary to permit correct observation of the underlying transaction's end for the transition from state 3 to 4. That is, no transaction's end may match `LD A0 D0 @e` other than the end of a transaction for which an observed event exists. In particular this means that if a second `LD A0 D0` transaction occurs while the DOG is in state 3 then, since there is no transition which observes it, its completion cannot justify the transition to state 4.

[3]The ARM architectures allow cacheability to be defined independently for Inner and Outer cacheability locations in multiprocessing systems.

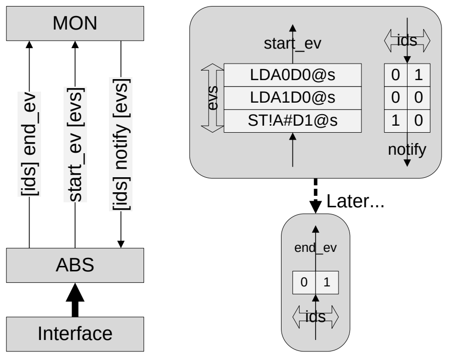

Fig. 3: Mechanism for implementing @s/@e signaling. Wires are annotated with SystemVerilog sizes.

Figure 3 shows a way to implement start and end signaling in SVA implementations. Every observable event in a DOG is given a unique *event index* number. A onehotzero vector[4] `notify` is signaled by the MON to the ABS for each channel on which the ABS can present observable events to the MON. In the example, three events can be indicated by the ABS, and the MON requires two event indices, suggesting that the DOG will only ever have two outstanding observed events. One bit of `notify` is reserved for each event index and by setting a bit to true, the MON indicates to the ABS that the observable event in that channel has been observed. The `notify` is propagated along the ABS pipeline with the transactions in progress. When a transaction completes, the `notify` value associated with it is returned to the MON as `end_ev` and the MON can identify the observable event which just ended by inspecting the appropriate bits. The `end_ev` shown in Fig.3 indicates a `LD A0 D0 @e`.

D. Star Ordering

One pragmatic issue with specifying the expected behavior of a weak memory system is that the order in which read and write events are dispatched from the processor may not be the order in which they commit at their destination. DOGReL includes notation for expressing the required order of commitment between two events.

[4]A onehotzero vector may either have one of its bits set or be zero.

Star Form	Permitted orders	MON matches
*+	α	`isAfter`
*!+	β or γ	`!isAfter`
*-	β	`isBefore`
*!-	α or γ	`!isBefore`

(a) Star Forms

Order	`isBefore`	`isAfter`
α	0	1
β	1	0
γ	0	0
$\alpha\beta$	y	$\neg y$
$\alpha\gamma$	0	y
$\beta\gamma$	y	0
$\alpha\beta\gamma$	$\neg y \wedge \neg w$	$\neg y \wedge w$

(b) ABS Ordering Signals

TABLE III: Ordering Possibilities

When defining ordering of events we consider the ordering to be relative to a particular observer, i.e. the ordering specified by a DOG is that observed by the subsystem to which the DOG is applied. In the case of our memory system DOGs, this is considered to be the L1 MS. This distinction is important because the order in which events are observed to commit may be different for different observers in the wider system. We define observed order according to the states which may be observed and we consider three possible commit orderings for transactions: α, β and γ ordering. Given two events, A and B, $A\alpha B$ is true if it is possible for A to be observed to commit before B, $A\beta B$ is true if it is possible for B to be observed to commit before A and $A\gamma B$ if it is impossible to observe either A or B committing before the other (i.e. they are atomic events).

For an α-ordering ($A\alpha B$) this means that if an observer makes a series of observations they may observe three states in the order:

1) Neither event has committed
2) Only event A has committed
3) Both events have committed

For a β-ordering ($A\beta B$) the second state observable would be ‘only event B has committed’. For a γ-ordering the second state is not observable.

Furthermore, two events may have a combination of possible commit orderings, in which case the observable states may be either of the possible orderings. Consider an $\alpha\gamma$-ordering for which an observer may be unable to distinguish which of the orderings happened, since if ‘only event A has committed’ is not observed, that doesn’t mean it didn’t happen. However, this ordering provides no guarantee that the two events will be committed non-atomically.

In PDF, these orderings are used to describe the commit ordering guaranteed by transactions dispatched on the AMBA AXI interface. Generally, AXI transactions which overlap temporally are not guaranteed to commit in any order (they are $\alpha\beta$-ordered) and ordering can only be relied on if one transaction is signaled as completed by the interconnect before the other is requested. For reads with identical *IDs* however, ordering is guaranteed according to the order of requests - they are α ordered [8].

In practice, not all possible ordering need to be expressed in DOGReL. To specify the permitted orderings we use a star notation suffix for observable events which is capable of expressing four ordering requirements as shown in table IIIa. This can be thought of as a pattern which will match a specific set of orderings. Firstly, the intended A event is annotated with a single star and is referred to as the the *lone-star event*. Only one lone-star event should be observable on any path in the DOG, and no other star form may appear earlier.

The B event has one of the star forms in table IIIa and B can only be observed by the DOG if the ABS indicates that it is correctly ordered with respect to the *lone-star event*. The implementation of this in SVA is achieved with a `notify_order` signal from MON to ABS for each channel to indicate the lone-star event A, and a pair of Booleans,

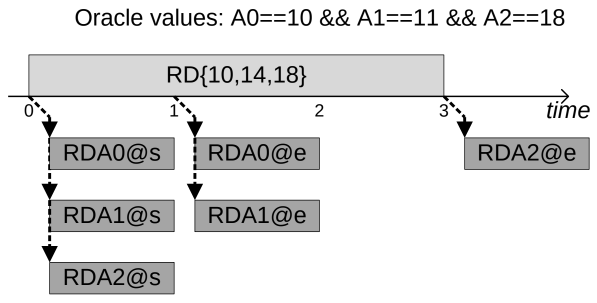

Fig. 4: Multiple events from a single transaction, including RDA0γRDA1.

which accompany each event from the ABS named `isAfter` and `isBefore`. When the ABS detects B, it decides on the applicable ordering of A and B from the first column of table IIIb, and ensures the Booleans for B are driven with values consistent with the second column. The variables y and w are existentially quantified so an $\alpha\beta$ transaction may drive {`isBefore, isAfter`} as {0, 1} or {1, 0} and an $\alpha\beta\gamma$ transaction may drive {`isBefore, isAfter`} as {0, 0}, {0, 1} or {1, 0}. In the SVA implementation there are appropriate undriven variables which allow the model checker to make a free choice. This means a property requiring an α ordering (i.e. it has a *+ event leading to success which requires `isAfter==1`) can produce a CEX if the event is $\alpha\beta$ since the model checker is free to choose the ABS to indicate {`isBefore, isAfter`} as {1, 0}. The same event is also capable of failing to satisfy a *- star form if necessary by choosing to indicate {`isBefore, isAfter`} as {0, 1}.

Ordering of observed events is complicated by two implementation details. Firstly, a single memory transaction may access more than one address on the same clock cycle. Secondly, it is possible for a single *transaction* to be observable as more than one *event*. Fig. 4 illustrates both of these issues with a read transaction accessing three 32 bit words at memory locations 10, 14 and 18 which is initiated at time=0. If each memory location holds a single byte, then we see that the four bytes from 10 through 13 are returned atomically at time=1, with successive 32 bit words returning at time=2 and time=3. Assuming the oracles A0 and A1 have the values shown, we can observe three start events at time=0, two end events at time=1 and one at time=3 as shown.

Consider a property requiring `RD A0 @s` α `RD A2 @s`. Upon observing the `RD A0 @s`, the MON identifies it as the lone star event to the ABS. It subsequently observes the `RD A2 @s` and requires the ABS to indicate that this `isAfter` the lone star event. Since both events are observed at the same simulation time, and since the ABS cannot indicate timing `isAfter`/`isBefore` information before it knows the lone star event’s identity, there is a possibility in Verilog semantics that the MON could observe the `RD A2 @s` before the ABS can provide the correct `isAfter` value - if it is implemented as combinational logic.

To avoid this problem, we implement a prediction mechanism in the ABS for the lone star event. This makes it seem as if the ABS knows what the MON will observe before the MON itself does. Remember however, that we are simply constraining out CEXs where the ABS and MON disagree on the observation of A.

In section III-C we stated that once an event is responsible for a transition in the MON FSM, it is an "observed event" and cannot cause further transitions. For memory interface events such as those in Fig. 4 we require an event per oracle address so that observing the event as `RD A0 @s` does not preclude observing it as `RD A2 @s` - further discussion is given in section IV-A along with the DOGReL syntax for declaring this.

IV. The Uniprocessor Load-Store Property

The *Uniprocessor Load-Store* DOG of Fig. 5 is one of the more complicated properties from PDF and illustrates many subtleties about how the DOGs specify required behavior. This DOG enforces correct memory interface activity for a Load followed by a Store to the same address, with no intervening access to that address, as observed by a uniprocessor executing the instructions. Namely that any Read corresponding to the Load should not occur after the Write corresponding to the Store.

Throughout the DOG, we associate the Load/Read activity with `D0` and the Store/Write with `D1`. In several places we exclude the possibility of any other transaction observing these data values by transitions to state 0, which provide *vacuous exits* that cause the antecedent to remain unsatisfied. This is justified because we only check for DIBs: any bug that requires the same data value to occur in two distinct transactions is a data-dependent bug and of no interest to us. We now describe the transitions. Each of the clauses L1-L7 and M1-M7 shows the DOGReL code for the condition on a transition, followed by a description of its purpose.

In the following discussion it is important to remember that the order in which *transactions commit* is distinct from the order in which *events* are *observed*.

L1: `1->0: ST A0 D1 @s`
Exit if we observe a store of `D1`. This ensures `D1` is fresh w.r.t. all writes since there is no preceding `ST A0 D1` that the `WR A0 D1` could be attributed to.

L2: `1->2: selectBE && (Reset || DSB @s || ST A0 !D# @s) && start=1`
Pick either
- (i) a store that can be neither forwarded from (`ST A0 D0 @s`) nor responsible for the `WR A0 D1 @s`
- (ii) or some other standard boundary event.

and update the synchronization variable `start`.

L3: `2->0: ST A0 DX @s || BE`
Ensure no further ST or BE occurs, between L2 and the load of L4.

L4: `2->3: LD A0 D0 @s`
Observe the first `LD A0 D0 @s` after the `BE`. An arbitrary number of loads of `!D0` may occur before it.

L5: `3->0: BE || LD A0 DX @s || ST A0 !D1 @s`
This ensures that we have selected:
- (i) a `D0` which allows us to identify the last `LD A0` before the `ST A0` of L6.
- (ii) a `D1` s.t. we capture the first store after L4.

Note `ST A0 !D1 @s` is redundant wrt our `BE`, but is included for clarity.

L6: `3->4: ST A0 D1 @s`
Observe the first store (see L5ii) after the `LD A0` of L4.

L7: `4->5: COMPLETE`
`COMPLETE` is observed once an @e event has occurred for all start events which were observed in transitions leading to the current state. In particular, when `@e` events for L6 and L4 have occurred in any order.

In the Read-Write domain we need the memory system to exhibit the behaviour 'the write does not precede the read'. We also know that there must be a read, since we excluded a possibility of `D0` being forwarded from the memory system; however, there may be no write, since a store may complete without memory being written if the memory system buffers the write in the STB.

M1: `A->B: start==1`
Synchronize with transition L2 via `start`.

M2: `B->C: WR A0 D1 @s*`
If we observe a `WR` which corresponds to the `ST` it becomes the lone-star event. Unless we observe a `RD` which commits earlier, this is a failure state.

M3: `C->F: RD A0 D0 @s*-`
We observe a `RD` which commits before the `WR*` and reach an antecedent satisfying state, F.

M4: `B->D: RD A0 D0 @s*`
If we observe a `RD` which justifies the `LD` it becomes the lone-star event. If no `WR` is observed from state `D` this is still enough to satisfy the property.

M5: `D->E: WR A0 D1 @s*!+`
We move to state `E` if a `WR` occurs which is committed at the same time or earlier than the `RD` of M4. We refer to such a write as the *forbidden write*.

M6: `E->F: RD A0 D0 @s`
If we have observed a forbidden write, but a second `RD` occurs then it may have committed:
- (i) before the `RD*` and we cannot tell whether it preceded the forbidden write, in which case the write would no longer be forbidden
- (ii) at the same time as or after the `RD*`, in which case the write is still forbidden

The transition to state `F` is obviously correct for M6i but for M6ii we justify moving to `F` with DIBs as follows. Note that one of the reads is associated with our `LD` and the other is a speculative preload (which the architecture permits). Now consider that if the speculative preload has any effect on the `LD` then, by DIBs, it must do so without

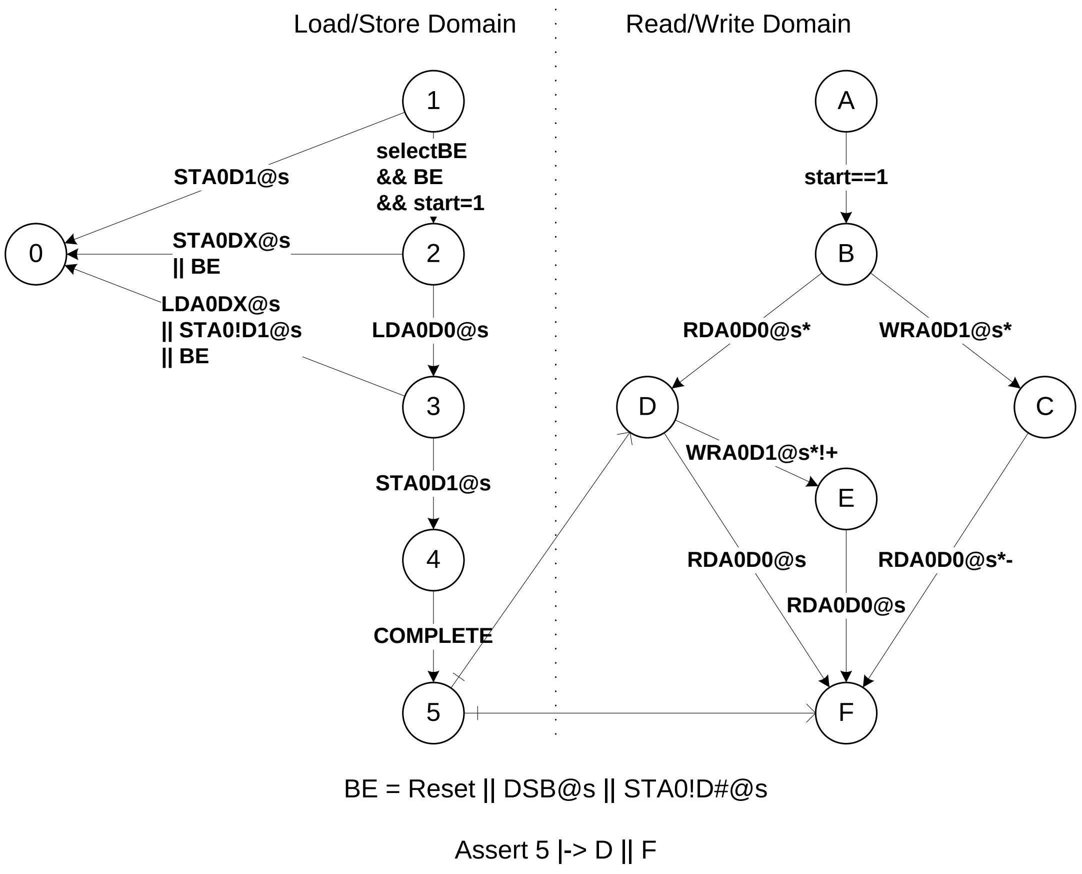

Fig. 5: Uniprocessor Load-Store DOG

regard to its data value, hence there is also a CEX where the preload returns `!D0` and so we can safely allow the trace with `RD A0 D0 @s` to lead to the accepting state `F`.

M7: `D->F: RD A0 D0 @s`
For M6 we justified transitions to state `F` if a second `RD` is observed after the `RD*`. This transition allows for the same scenario but with events observed in a different order, i.e. that one of the `RD`s is speculative and we would be unable to order any `WR` with respect to them.

We refer to the transitions of M6 and M7 as *lone star escapes*, an idiom which often occurs in DOGs to ensure that the lone star event is indeed the only occurrence of that event in order to avoid the kind of ordering confusion noted in M6i. Note that there is no lone star escape for the `WR*`, but we know that the load/store domain FSM prohibits more than one `WR A0` and that the separate *Store-Only* DOG, which ensures every observed write has a corresponding unique store, checks for multiple writes from a single store.

A. Event Descriptors

The DOGs specify the FSMs, assumptions and assertions required to implement the checkers in a design independent way, but each implementation will have specific details about how events may arise from the underlying hardware. Consider a single transaction, a Load of a 32 bit word at the DPU/LSU interface. Since our addresses are at byte level, and data is a single byte, the load can be observed as either `LD A0 D0 @s`, `LD A1 D1 @s` or `LD !A# D0 @s`. An ABS can indicate these oracle matches using a two dimensional array, or *match structure* as in table IV. The table values also match the predicate function which identifies a `LD A# D1 @s`, since at least one of the rows `A0` and `A1` is true in column `D1`.

	!D#	D1	D0
!A#		0	1
A1	0	1	0
A0	0	0	1

TABLE IV: Two Oracle Match Structure

The complete *event descriptor* will have fields describing the type of event and the values of other attributes along with the match structure. The match structure allows a single event descriptor to match a range of observable events, but only once, since if the event is observed as `LD A0 D0 @s` it cannot also be observed as `LD A1 D1 @s`.

An interface may also allow a single transaction to be observed as multiple events. For example, our memory interfaces allow a read transaction to be observed as a separate event for each *possible address* which the DOG may distinguish, namely the address oracle variables `A0` and `A1` as well as the remaining possibility, `!A#`. In DOGReL we declare these ABSs' channels as:

```
abs ahb {
    channel rw_ahbp [1] of read_write
      match data per address;
}
```

The keyword `per` indicates that we wish to be able to observe an event *per* possible address. This results in the channel having an event descriptor for each possibility: in this case an array with three event descriptors. Note that the channel declaration specifies `match data` to ensure that a match structure is created in each event descriptor: in this case a one-dimensional array whose meaning is as in a single row of the match structure in table IV.

The width `[1]` in the declaration means this is a single channel, and so it describes a single transaction. The `start_ev` shown in Fig. 3 is an example of a channel of width three which can describe three transactions: in this case, a superscalar pipeline which can issue three instructions at once.

These channel declarations enable the automatic generation of RTL interfaces, and predicate functions which determine whether the values sent on the channel match a particular observable event. All that remains is for the user to provide the body of the ABS which monitors activity in the DUT and drive the channels appropriately. Note that the DUT may be RTL or some other object, such as a micro-architectural specification.

B. ABS/MON Instantiation

The top-level connections between MONs and ABSs for a system with P DOGs and R memory interfaces and a single DPU/LSU interface is shown in Fig. 6. Each MON is connected to the DPU/LSU ABS, and to R Mem ABSs. The signals shown in Fig. 3 are routed through the multiplexers at the top of Fig. 6 so that only one MON is connected to the ABSs. The choice of MON to connect is made by the pseudo-constant `PROP`.

Note that the MON contains logic which allows it to sample the memory interface channels in a non-deterministic order since no ordering is implied between different memory interfaces. On the DPU/LSU side however, for a multi-channel ABS attached to a superscalar pipeline, whether or not this is in order, the ABS will present events in transaction order, i.e. program order. That is, there is a specific order in which the channels should be sampled. DOGReL includes syntax which allows the DOGs to specify whether an FSM state samples events in deterministic order or not.

V. Results

Over the course of the Pelican project, over fifty DOGs were developed, grouped in families covering *Uniprocessor Load/Store, Exclusive Load/Store, Faults, Routes, Memory Barriers* and *Device/Strongly Ordered Memory Accesses*. Having these available as diagrams greatly helped in development and communication of the properties.

A common question to ask is whether a specification is complete. To fully answer this question, a suitable semantics for the ISA and all the memory interfaces would be required and could then be used to ask whether an implementation satisfying all the DOGs would correctly implement the machine instructions. Some parts of the semantic are available, others are under development, but even with a complete description, such a proof may be prohibitively difficult. The main goal of applying the DOGs to Pelican was to improve bug-hunting during development, hence completeness was not a strong requirement. However, we found it was often apparent that within a particular family of DOGs all reasonable cases were being covered, since consideration of each DOG tends to reveal the scenarios they do not cover, and these are often explicitly deferred to a separate DOG. For example, the relationship between *Load-Store* and *Store-Only* discussed in section IV.

Despite the methodology being created at the same time as the RTL implementation work, we managed to attach our first properties before the alpha milestone of the project, the first point at which it was reasonable to expect enough functionality from the RTL for the results to be meaningful and the target for traditional testbenches to come online. We also found our first bug at that point in a failed load/store exclusive scenario.

As RTL development proceeded, PDF continued to find bugs at a competitive rate. Overall, for the design units covered by PDF, simulation found 62% of bugs filed, PDF found 24% and other formal techniques found the remaining 14%. Compared to simulation, PDF used less human resource and at a cost of up to a third of the *compute time per bug*.

We also identified a number of bugs which could not have been found by the early unit level testbenches, since they required full-system simulation for which more complete functionality of the RTL is required. Furthermore, Pelican is the first implementation of its particular architecture, and the act of writing unambiguous specifications forced closer review of both the architecture and the micro-architecture, causing several corrections to be made.

Several bugs were found in interface specifications. We believe this is because, although simulation uses the interface specifications to check generated stimulus, the PDF properties use them to generate stimulus, so under-constraints, which often cause CEXs, are quickly found. The DPU/LSU interface was particularly complex and could not be explored by simulation until top-level test, but was fundamental to the operation of PDF. In fact, a serious rewrite of part of this interface was triggered by a DOG bug-find.

The original goal of PDF was to explore *bug absence* proofs using micro-architectural models and refinement to RTL. Although this was not achieved for the L1 MS, we applied the DOG methodology to a simple bridge component and managed to complete a single proof of an end-to-end

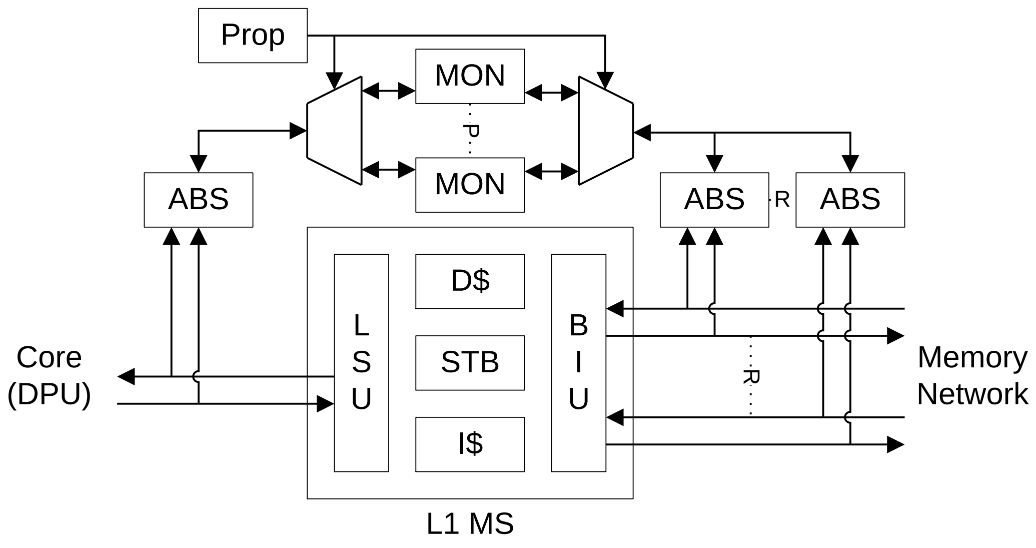

Fig. 6: MON/ABS Arrangement for Multiple DOGs

property. This involved creating invariants which mapped DOG states to RTL states and we are hopeful that the lessons learned from this will help develop a methodology for larger systems.

VI. Further Work

The results for Pelican have been compelling enough that we intend to apply the DOGs to future core memory system projects. Since the DOGs represent requirements of the architecture on the micro-architecture, they are applicable to any processor with similar architecture. One future challenge is to create DOGs for coherent components which we believe can be achieved with a divide and conquer approach.

One of the original goals of PDF to develop a refinement proof between RTL and a verified micro-architectural model was not achieved. We intend to resume our original task of devising a notation which improves specification of such models in the same way that DOGs improve the writing of end-to-end properties.

VII. Conclusion

The Pelican Deep Formal project created a new methodology for describing end-to-end properties using the visual notation of Directed Observer Graphs which can be converted into monitors for use in model checkers. The notation allows the use of non-determinism and prediction of values to simplify the DOGs, and includes features for expressing: pattern matching to attributes of transactions; the lifetime of a transaction; assumptions about the environment; universally quantified variables; and ordering requirements between observable results. Although we focused on transactions found in a processor memory system, the language in which properties are described, DOGReL, allows for user defined transactions. The properties we created proved valuable in discovering bugs from an early point in RTL development, including bugs outside the range of simulation to find, and bugs in the design specification.

References

[1] *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*, IEEE std 1800-2012 ed., Feb 2013.

[2] *Standard for Property Specification Language (PSL)*, Iec 62531:2012(e) (IEEE std 1850-2010) ed., June 2012.

[3] D. Stewart, “Formal for everyone - challenges in achievable multicore design and verification.” in *FMCAD*, G. Cabodi and S. Singh, Eds. IEEE, 2012.

[4] D. Stewart, D. Gilday, D. Nevill, and T. Roberts, “Pelican deep formal,” in *Jasper Architects’ Forum Meeting*, October 2013.

[5] E. M. Clarke, J. M. Wing *et al.*, “Formal methods: State of the art and future directions,” *ACM Comput. Surv.*, vol. 28, no. 4, pp. 626–643, Dec. 1996. [Online]. Available: http://doi.acm.org/10.1145/242223.242257

[6] D. Stewart and S. Hoad, “Applications of formal micro-architectural specifications,” in *Jasper User Group Meeting*, November 2010.

[7] K. Shimizu, M. A. Horowitz, and D. R. Engler, “Writing, verifying, and exploiting formal specifications for hardware designs,” 2002.

[8] *AMBA® AXI™ and ACE™ Protocol Specification*, ARM.

[9] Y. Mahajan and S. Malik, “Utility of transaction-level hardware models in refinement checking,” in *High Level Design Validation and Test Workshop (HLDVT), 2010 IEEE International*, June 2010, pp. 121–128.