Eighth International Workshop on Designing Correct Circuits

Paphos, Cyprus, 20–21 March 2010

A Satellite Event of the ETAPS 2010 group of conferences

Participants' Proceedings

Edited by Joe Stoy

Preface

This volume contains material provided by the speakers to accompany their presentations at the Eighth International Workshop on Designing Correct Circuits, held on the 20th and 21st March 2010 in Paphos, Cyprus. The workshop is a satellite event of the ETAPS group of conferences. Previous workshops in the informal DCC series were held in Oxford (1990), Lyngby (1992), Båstad (1996), Grenoble (2002), Barcelona (2004), Vienna (2006) and Budapest (2008). Each of these meetings provided a stimulating occasion for academic and industrial researchers to get together for discussions and technical presentations, and the series as a whole has made a significant contribution to supporting our research community.

The 2010 DCC workshop again brings together researchers in formal methods for hardware design and verification. It will allow participants to learn about the current state of the art in all these areas, and it is intended to further the debate about how more effective design and verification methods can be developed.

For some time now, research in hardware verification is being done in industrial laboratories, as well as in universities. Industry is commonly focussed on relatively immediate verification goals, but also keeps our work grounded in practical engineering problems. There is also a general feeling that hardware design in RTL is reaching the limits of its scalability, and alternative approaches are emerging, some based on existing software languages such as C, and others on high-level languages devised specifically for hardware design. For these approaches to continue to scale, they will need increasingly to be used with rigour based on formal foundations. To make progress on these longer-term problems in our field, academic and industrial researchers must continue to work together on the problems facing microprocessor and ASIC designers now and in the future. A major aim of the DCC series of workshops has been to provide a congenial and relaxed venue for communication among researchers in our community. We look forward to two great days of presentations and discussion in DCC2010.

I would like to express my particular gratitude to the members of the Programme Committee for their work in arranging the Workshop, and to all the speakers and participants for their contributions to Designing Correct Circuits.

Joe Stoy
February, 2010

Programme Committee

Arvind (MIT, USA)
Per Bjesse (Synopsys, USA)
Wolfgang Kunz (University of Kaiserslautern, Germany)
Bob Kurshan (Cadence Design Systems, USA)
Pete Manolios (Northeastern University, USA)
Andy Martin (IBM, USA)
Tom Melham (University of Oxford, UK)
Gordon Pace (University of Malta, Malta)
Marc Pouzet (University of Paris-Sud, France)
Carl Seger (Intel, USA)
Satnam Singh (Microsoft Research, UK)
Joe Stoy (Bluespec, USA)

Contents

Sunday 21st March 2010

Languages

Design and Synthesis

Formal Validation and Verification of Networks-on-Chips: Status and Perspective
(Draft Paper)

Julien Schmaltz[2,1], Freek Verbeek[1,2], and Tom van den Broek[1] ⋆

[1] Radboud University Nijmegen
Institute for Computing and Information Sciences
PO Box 9010 6500GL Nijmegen, The Netherlands
[2] Open University of the Netherlands
School of Computer Science
PO Box 6401DL Heerlen, The Netherlands
Email: Julien.Schmaltz@ou.nl

Abstract. Increasing the performance of computing system today means more parallelism. Systems are becoming multi-cores. The on-chip interconnect is a complex infrastructure having a crucial impact on the system global performance and functionality. In this draft paper we present recent results and work-in-progress towards a general compositional approach for the validation and verification of networks-on-chips. In particular, we discuss temporal abstractions, a refinement theorem between two architectures described at the same temporal abstraction, and two properties – namely deadlock and evacuation – proven on a model defined at a more abstract temporal layer.

1 Introduction

Increasing the performance of modern electronic systems means increasing parallelism [4]. Several processing and memory cores are integrated on a single die forming so-called Multi-Processors Systems-on-Chips (MPSoC). *Platform based design* [12] is a popular approach that provides designers with a generic architecture allowing the construction of new MPSoCs by assembling pre-designed components. The latter are often highly parametric descriptions at a high-level of abstraction. Abstractions and the interconnect become central issues in modern MPSoCs design. With the increase of the number of interconnected cores, complex on-chip networks are replacing buses [5, 1]. As for processing elements, the formal guarantee of their correct behavior will become mandatory. Our ultimate goal is to provide a formal methodology supporting the abstract specification of NoCs and the proof that an implementation – eventually described at the Register Transfer Level (RTL) – conforms to it. To this end we are developing

⋆ This research is supported by NWO/EW project Formal Validation of Deadlock Avoidance Mechanisms (FVDAM) under grant no. 612.064.811.

models and proof methods that constitute the Generic NoC (GeNoC) design and verification approach [8, 3].

GeNoC provides (1) a network model to specify the main characteristics of a NoC (e.g., topology, routing); (2) architectural models defining interaction schemes of the constituents; (3) correctness theorems for these architectural models; and (4) sufficient constraints – or proof obligations – on the constituents from which the proof of the correctness theorems follows. GeNoC is generic in the sense that the constituents are not given any specific definition but only characterized by their proof obligations. The validation of a particular NoC reduces to (1) giving a concrete definition to the constituents and (2) discharging the corresponding instantiated constraints. GeNoC has been implemented in the logic of the ACL2 theorem proving system [7] and applied to several case-studies, e.g., the HERMES [2] and Spidergon [3] designs.

In this draft paper, we present a compositional approach for the design and validation of NoCs. The focus is on the extension of the GeNoC approach with refinement theorems between architectures and an explicit notion of time and corresponding temporal abstractions. Note that this paper presents more the direction we are following and not a final result.

Section 2 presents this global approach and introduces three temporal abstractions. The lowest one is detailed in Section 3 that discusses two architectures and a refinement theorem between them. We only give an overview of this work. More details can be found in two recent publications [10, 11]. Section 4 considers the next more abstract temporal layer. It focuses on two global properties that guarantee the absence of deadlock and *evacuation*, i.e., all injected messages leave the network. We present the essential aspects related to these two properties. More details can be found in a recent publication [14]. Finally, Section 5 relates Sections 3 and 4 to the approach described in Section 2. It also concludes this paper with our current perspectives and future work.

2 A Compositional Approach

2.1 The Compositional Model

The compositional model is pictured in Fig.1. It is composed of four parts: (1) the NoC constituents; (2) the NoC Architectures; (3) the NoC Theorems; and (4) the User Input.

The *NoC Constituents* are the essentials parts common to any network. The *routing algorithm* computes for each message its next hop according to its current position an its destination. The *scheduling policy* handles the data flow. It decides if a message can progress to its next hop. The *data link layer* is responsible for the exchange of data between the current and the next position. The *injection method* controls access to the network. The last elemens of the constituents are the *characteristic parameters*, e.g., topology, buffer size, message length. Note that these parameters are most of the times left symbolic for a parametric analysis. All the constituents are *generic*. They are not given any concrete definition. Each one of them is constrained by a set of *proof obligations*.

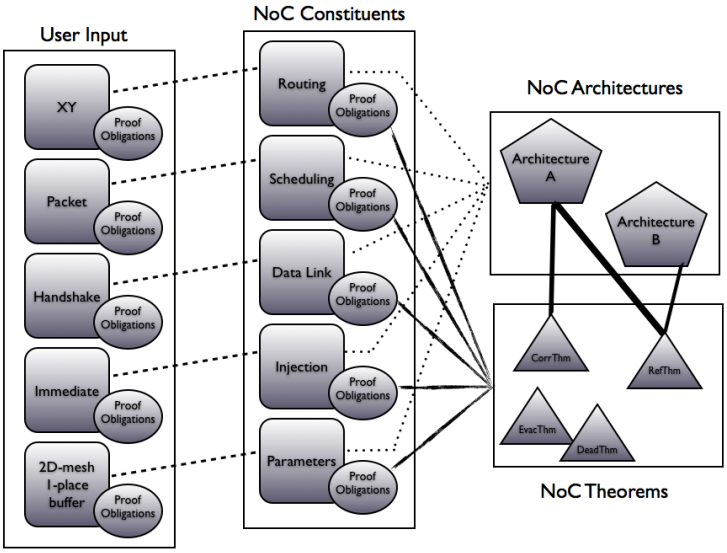


Fig. 1. Compositional Model

The *NoC Architectures* represent different ways of combining the constituents. One architecture could compute complete routes from source to destination before sending messages (*source* routing) while in another one routes are computed hop-by-hop (*distributed* routing). We will give more details about these two examples later (Section 3).

The *NoC Theorems* are global properties of one architecture or a refinement theorem between two distinct ones. Our model has currently three global properties and one refinement theorem. Global properties express (a) *functional correctness*, i.e., all messages that reach a destination reach the expected destination without modification of their content; (b) *deadlock-freedom*; and (c) *evacuation*, i.e., all messages eventually leave the network. The refinement theorem relates a source routing architecture to a distributed routing one. All these theorems are direct consequences of the proof obligations and do not depend on the particular definition of the constituents. Therefore they hold for all particular definitions satisfying the proof obligations.

This suggests the following methodology and defines the user input. First one should give a concrete definition to the constituents. For instance, an XY-routing algorithm in a 2D-mesh with packet switching as scheduling policy and where all messages are injected at time 0. Then the corresponding instances of the proof obligations are automatically generated. After discharging all of them it automatically follows that the corresponding instances of all architectures

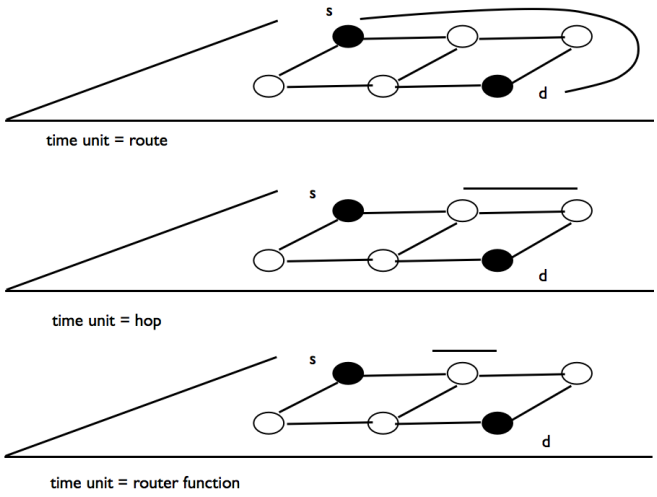


Fig. 2. Temporal Abstractions

satisfy the corresponding instances of all theorems. At the end the *User Input* consists in (1) giving a definition to the constiutents and (2) discharging the corresponding proof obligations. The rest is fully automatic.

2.2 Temporal Abstractions

We consider the temporal abstractions shown in Figure 2. Each layer depends on the definition of an atomic action. In the most abstract one, in one time unit, a message will traverse the complete network from its source to its destination. A first refinement of this abstraction restricts the time step to at most one hop. Finally, the lowest layer makes the internal structure of a node visible, in particular, the router component. The application of the function representing the router defines the time unit. In this paper, we discuss the lowest layers. The next Section details the lowest layer in Figure 2. In Section 4 we detail its first abstraction. More information about the most abstract layer can be found in the original GeNoC paper [8].

3 Refinements at the Lowest Temporal Layer

This section details the compositional model viewed at the level of a router. It defines the NoC constituents of that layer and two architectures. It then shows a refinement theorem between them.

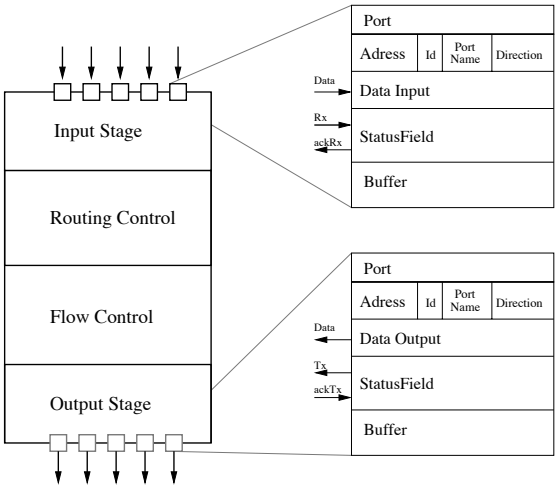


Fig. 3. A router and its ports

3.1 NoC Constituents

We assume a generic architecture composed of an arbitrary – but finite – number of nodes and a finite number of connections between any two nodes. Each node is uniquely identified by its position. A node includes a local memory and a router. A router is defined by a set of ports and four functions: input and output units, routing control, and flow control (see Fig. 3). All nodes are identical.

Ports, topology and state The main elements of a port are the data and control signals, and internal buffers (Fig. 3). A port is a tuple $\langle addr, stat, data, buff \rangle$, where *addr* is a unique address, *stat* stores the values of the control signals and other state components of a port, *data* denotes the values of the data signals, and *buff* represents the value of the buffers associated with the port. An address is a tuple $\langle coor, pid, dir \rangle$, where *coor* is the unique identifier of the node the port belongs to, *pid* is the name of the port (e.g., *west*, *south*), and *dir* is the direction, i.e., 'i' for an input port or 'o' for an output port. The topology is a list where each element is a pair of port addresses (p_i, p_j), which means that port p_i is connected to port p_j. A node is defined as the set of ports, where the address of each port p is the same. These ports define the state of the node. The set of all ports of a network defines the state of the network.

Input and output units These two functions define the low level protocols (e.g., handshake) which use the control signals to transfer the content of the data signals to the internal buffers in case of an input port, or to transfer the content of the buffers to the data signals in case of an output port.

Routing control This function applies the routing logic (e.g., dimension order routing [6]) to one or more ports of a node. It returns a list of *routed ports*,

i.e., ports together with routing information. The only function that needs to be instantiated is function `routing-logic` which implements the routing algorithm.

Flow control This function implements the switching technique, e.g., packet, circuit, or wormhole. In case of conflict, this function also resolves priorities. Function *flowcontrol* extracts from the routed ports the messages that are ready to be transmitted. The core function that needs to be instantiated is function *switch-ports* which effectively schedules messages. Those scheduled messages are moved to the output ports computed by the routing control function.

Global definition All these functions form function *router* (Figure 4), which updates a node state. Note that a node is equipped with a memory which is available to each port and each function. Argument `nstmem` represents that memory. To simplify the presentation, we assume that such a memory element is given as input argument of any function that accesses it. This argument is not explicitly mentioned any further.

```
router (nst,nstmem)  :
  let (nst nstmem) be
    RouteControl ((ProcessInputs nst), nstmem))
    in
    let  (nst nstmem) be
          Flowcontrol(nst,nstmem) in
          return (ProcessOutputs nst), nstmem
```

Fig. 4. Function *router*

3.2 Architecture Template

Function $GeNoC_t$ (Figure 5) is the core of the architecture template for our lowest temporal abstraction. It works as a simulator which applies function `router` to each node. Each recursive call defines a simulation step. Input argument `simL` defines the length of the simulation. Function $GeNoC_t$ takes as additional arguments the set of messages to be sent (`m`), the current state of the network (`ntkst`), an accumulator of messages that have reached their destination (`arr`, initially empty), the current simulation step (`z`, initially 0), and the topology (`topo`). It returns the list of arrived messages, the list of delayed messages, and the state of the network at the end of the simulation.

Function `depart` controls message injection. According to an architecture-dependent criterion, it determines which messages can be in the network. These messages have either already left their source or `depart` inserts them in the local input port of their source node. Function `depart` returns a list of updated nodes (`dep`) and a list of delayed messages (`del`). Function `step-ntk` (see below)

```
GeNoC_t(m, ntkst, arr, z, topo, simL) :
if simL = 0 return arr,m,ntkst
else
 let (dep, del) be
     depart(ntkst, m, z)
     in
     let newntkst be
         step-ntk(dep, topo)
         in
         GeNoC_t(del,newntkst, add(z,arr), z + 1,topo, simL-1)
```

Fig. 5. Architecture Template

applies function `router` to each node. This produces a list of updated nodes. Those messages that are at their destination are extracted from this new state and appended to accumulator `arr`. The next recursive call processes the delayed messages, the updated nodes, and time is incremented by 1.

Function `step-ntk` (Figure 7) is defined by two architecture dependent functions. Function `step-ntk-arch` (Figure 6) encapsulates the representation of the `router` on which the architecture is based. Function `updateNeighbours` simulates the transfer of data from output data signals to input data signals, e.g., simulate the wires between the nodes. It accomodates for particular details of one architecture.

Function `step-ntk-arch` (Figure 6) takes as arguments a list of nodes to be processed (`ntslist`) and the current network state (`ntkst`). It updates the network state. For each node, it applies function `router`. Function `ports-update` effectively updates the state of the nodes.

```
step-ntk-arch (ntslist, ntkst):
if ntslist = null return ntkst else
let newnst be router(ntslist[0]) in
    let newntkst be
        step-ntk-arch(ntslist[1..], ntkst) in
        return ports-update(newntkst,newnst)
```

Fig. 6. Function `step-ntk-arch`

Function `step-ntk` extracts the node structures from the list of ports (function `ports-nodelist`). It then calls `step-ntk-arch` to actually simulate each router. Finally, wires are updated by calling function `updateNeighbours`.

In summary, the architecture template has three architecture dependent functions:

```
step-ntk(ntkst, topo):
let newntkst be
    step-ntk-arch (ports-nodelist(ntkst), ntkst) in
    updateNeighbours(newntkst,topo)
```

Fig. 7. Function `step-ntk`

– function `depart` specifying the criteria to access the network
– function `router` specifying the router of a given architecture
– function `updateNeighbours` specifying how wires must be updated

The architecture template suggests a distributed routing network. Packets only contain information about their destination. At each intermediate step, the Routing Control part of the router decides the next hop. In the following subsection, we define a variation where routes are computed before injecting packets into the network. A packet then contains its route as well.

3.3 Source Routing Architecture

The source routing architecture is defined by the following three functions: `depart-sr`, `router-sr`, and `updateNeighbours-sr`. They define a refinement of function GeNoC named `GeNoC-sr` and follow the templates given in Section 3.2. Note that most of these functions are still generic. They are still not given an explicit definition.

Function `depart-sr` According to an architecture-dependent criterion, it determines which packets can be in the network. As the architecture is based on *source routing*, function `depart-sr` computes for each packet a route from source to destination and appends this route to the packet. Whenever the user criterion is satisfied, it inserts this extended packet in the local input port of its source node.

Function `router-sr` As shown in Figure 3, a router is composed of four parts. The routing control part is the only part that is modified. The rest is kept generic. The source routing control is defined by simply reading the next hop as the first element of the route of each packet. No computation is necessary.

Function `updateNeighbours-sr` This function updates the wires of the nodes, i.e., it simulates the transfer of data along wires. In addition it removes in each packet the first element of its route.

3.4 Refinement Theorem

The theorem connecting the two models is shown in Figure 8. Function `transform` simply removes all routes from extended packets. This theorem states that after

the application of function `transform` the lists of arrived packets, the lists of packets still en route in the network, and the final network state produced by `GeNoC` equals those produced by `GeNoC-sr`.

```
Theorem:
let (arr, m, ntkst) be
    GeNoC(m, ntkst, arr, z, topo, simL) in
    let (arr-sr, m-sr, ntkst-sr) be
        GeNoC-sr(m, ntkst, arr, z, topo, simL) in
        transform(arr-sr) = arr and
        m-sr = m and
        transform(ntkst-sr) = ntkst
```

Fig. 8. Equivalence theorem

The main proof obligation required to prove this theorem states that at each intermediate step reading the pre-computed route must be equal to the route computed in the distributed architecture. Details on these architectures and the refinement theorem can be found in previous publications [9, 11].

4 Deadlock and Evacuation

This section details the compositional model viewed at a level where a time unit is defined by one hop. We then introduce two global properties and their proof obligations. The properties deal with deadlock freedom and evacuation.

4.1 Abstract Model and Architecture

A *travel* t is a data structure which stores the progress of sending a message across a network. It is a triple $< id, c, d >$ where id is a unique identifier for the travel, c denotes the current location of the message, and d is the destination port. T denotes the list of travels that is sent across the network.

To keep the list of travels well-formed, destinations have to be reachable from their source. To this end, function $s \succ_R d$ returns true if s is reachable from d. This function is application dependent and must be instantiated. It is quite technical and not essential. We will therefore not detail it any further.

A *state* ST is a data structure which stores the current network state. The state is defined as the list of all the ports of the network. Each port is associated to the list of its buffers.

A *configuration* σ is a tuple $<T, ST, A>$, where T is a list of travels that are sent across the network, ST is a network state and A is a list of arrived travels. The travels T of configuration σ are denoted by $\sigma.T$. The set of all configurations is denoted by Σ.

Function $\mathbf{I} : \Sigma \mapsto \Sigma$ represents the *injection method*. Given a configuration, it decides which travels from T are ready for departure and injects these into the network.

Function $\mathbf{R} : P \times P \mapsto P$ represents the *routing function* of a switch. From the current position and the destination it computes the next hop. We generalize this function to apply to a configuration and to compute all hops from source to destination. We then write $\mathbf{R} : \Sigma \mapsto \Sigma$. It computes for each travel in $\sigma.T$ the route from its current location to the destination.

Function $\mathbf{S} : \Sigma \mapsto \Sigma$ represents the *switching policy*. It takes as parameter the current configuration and computes the configuration after one *switching step*, i.e., after each message that can make progression has advanced by at most one hop. If a message arrives at its destination, the corresponding travel is removed from T and added to A.

A *deadlock-configuration* is a configuration σ in which there exists no message that can make progression. This is denoted by $\Omega(\sigma)$. An interconnection network is *deadlock-free* if and only if there exists no deadlock-configuration.

4.2 Abstract Architecture Template

GeNoC takes as input argument an initial *configuration*, noted σ. The latter contains the list of messages to be sent on the network (T), the current value of the network state (ST), and the list of messages that have reached their destination (A). Function GeNoC recursively applies the composition of the three constituents to the initial configuration. The computation stops either when all messages have reached their destination or when the current configuration is in a deadlock. If the current configuration is not in deadlock, the switching policy must decrease the *termination measure*. This proves GeNoC always terminates.

Function *GeNoC* is defined as follows:

$$GeNoC(\sigma) \stackrel{\text{def}}{=} \begin{cases} \sigma & \text{iff } \sigma.T = \emptyset \\ \sigma & \text{iff } \Omega(\mathbf{R}(\mathbf{I}(\sigma))) \\ GeNoC(\mathbf{S}(\mathbf{R}(\mathbf{I}(\sigma)))) & \text{iff otherwise} \end{cases}$$

4.3 Constraints for deadlock-freedom

Dally and Seitz [6] proposed a necessary and sufficient condition for deadlock-free routing. They prove that a deterministic routing function is deadlock-free if and only if it has no cycle in its *channel* dependency graph. We have formalized a slightly different condition in GeNoC (See [13] for details). Dally and Seitz define their function at the level of processing nodes. We define our routing function at the level of ports. Let $R : P \times P \mapsto P$ be a routing function. The port dependency graph is a graph with as vertices the ports of the interconnection network and as edges the pairs of ports connected by the routing function.

Theorem 1. *R is deadlock-free if and only there is no cycle in its* port *dependency graph.*

The proof of this necessary and sufficient condition is structured in such a way that it only depends on a fixed set of constraints over the dependency graph, the routing function, and the definition of reachable destination. Let $E^R_{\mathtt{dep}}$ represent the edges of the port dependency graph. These constraints are the following:

$$\forall s, d \forall p \in \mathbf{R}(s,d) \cdot s \succ_R d \implies (s,p) \in E^R_{\mathtt{dep}} \tag{1}$$

$$\forall (p_o, p_1) \in E^R_{\mathtt{dep}} \exists d \cdot p_0 \succ_R d \wedge p_1 \in \mathbf{R}(p_0, d) \tag{2}$$

$$\forall P' \subseteq P \cdot \neg \operatorname{cycle}_{\mathtt{dep}}(P') \tag{3}$$

Constraint (1) states that each pair of ports connected by $\mathbf{R}$ must be an edge. We consider pairs resulting from reachable destinations only. Constraint (2) states that for each edge (p_0, p_1) a reachable destination port must exist such that $\mathbf{R}$ routes from p_0 to p_1. Constraint (3) states that there is no cycle in the port dependency graph.

4.4 Constraints for evacuation

All messages evacuate the network if, when GeNoC terminates, the list of arrived messages equals the list of messages that were sent. This defines the Evacuation Theorem as follows:

Theorem 2. *All messages eventually leave the network. Formally, we have the following:* $GeNoC(\sigma).A = \sigma.T$.

We assume all messages have been injected in the initial configuration. The injection method does not inject any more messages:

$$\mathbf{I}(\sigma) = \sigma \tag{4}$$

Assuming Constraints (1) through (3) are satisfied, GeNoC terminates if and only if all messages have evacuated the network. Hence, proving evacuation reduces to proving termination of function $GeNoC$. To prove termination, we define a *termination measure*, i.e., a value which decreases after each recursive call. Proving evacuation reduces to instantiating a function $\mu(\sigma)$, which computes a termination measure such that the following constraint holds:

$$\sigma.T \neq \emptyset \wedge \neg\, \Omega(\sigma) \implies \mu(\mathbf{S}(\mathbf{R}(\sigma))) < \mu(\sigma) \tag{5}$$

As long as there are messages in the network and there is no deadlock, the measure provided by μ must decrease with each switching step.

5 Conclusion and Perspectives

We presented the compositional approach that we are currently developing. It is based on a compositional model and temporal abstractions providing different views of this model. We illustrated two of these abstraction layers. At the lowest

one, we presented two architectures and exposed a refinement theorem between them. We discussed deadlock freedom and evacuation at a more abstract layer.

Our current work focuses on connecting these two layers such that properties proven for one layer are preserved in the other one. We are also considering additional layers corresponding to further refinements of the time unit. For instance, the time unit could be defined as the application of one of the four stages of the router described in Figure 3. Further refinements would aim at a cycle accurate model where a time unit coincides with a clock cycle. Our ultimate goal is to extract sufficient constraints on the constituents from which it would automatically follow that properties proven on the most abstract temporal layer are preserved in the most concrete one.

References

1. L. Benini and G. De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, 2002.
2. D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. Executable formal specification and validation of NoC communication infrastructures. In *Proceedings of the 21st annual symposium on Integrated circuits and system design (SBCCI'08)*, pages 176–181, Gramado, Brazil, September 1–4 2008. ACM.
3. D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. A formal approach to the verification of networks on chip. *EURASIP Journal on Embedded Systems*, 2009(Article ID 548324):14 pages, 2009. doi:10.1155/2009/548324.
4. W. Dally. The end of denial architecture. Keynote at Design Automation Conference (DAC'09), 2009.
5. W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the Design Automation Conference*, pages 684–689, Las Vegas, NV, 2001.
6. W.J. Dally and C.L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.
7. J. Schmaltz and D. Borrione. Towards a Formal Theory of On Chip Communications in the ACL2 Logic. In *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications, part of FloC'06*, Seattle, Washington, USA, August 14-15 2006. ACM.
8. J. Schmaltz and D. Borrione. A functional formalization of on chip communications. *Formal Aspects of Computing*, 20(3):239–348, 2008.
9. T. van den Broek. Towards the cross-layer verification of networks-on-chips. Master's thesis, Radboud University Nijmegen, The Netherlands, July 2009.
10. T. van den Broek and J. Schmaltz. A generic implementation model for the verification of networks-on-chips. In S. Ray and D. Russinoff, editors, *8th Intl. Workshop on the ACL2 Theorem Prover and Its Application*, pages 130–134, Northeastern Univ., Boston MA, USA, 2009. ACM.
11. T. van den Broek and J. Schmaltz. Towards formally verified networks-on-chips. In A. Biere and C. Pixley, editors, *9th International Conference on Formal Methods in Computer Aided Design (FMCAD'09)*, pages 184–187, Austing, Texas, USA, November 15–18 2009. IEEE Computer Society.

12. J. van Meerbergen. Networks on chip: A communication-centric approach to platform-based design. In *PROGRESS White Papers 2006*. STW, The Netherlands.
13. F. Verbeek and J. Schmaltz. Proof pearl: A formal proof of dally and seitz' necessary and sufficient condition for deadlock-free routing in interconnection networks. *J. Autom. Reasoning*, 2009. Submitted to publication.Available at: http://www.cs.ru.nl/~julien/Julien_at_Nijmegen/JAR09.html.
14. F. Verbeek and J. Schmaltz. Formal validation of networks-on-chips: Deadlock and evacuation. In *Design, Automation and Test Europe (DATE'10)*. ACM/IEEE, March 2010. To Appear.

A Formalised Framework for Incremental Modelling of On-Chip Communication

Peter Böhm

University of Oxford
Computing Laboratory

Designing Correct Circuits, March 2010

Motivation

UNIVERSITY OF OXFORD

Goal

- Design of verified high-performance, on-chip communication protocols

Problem

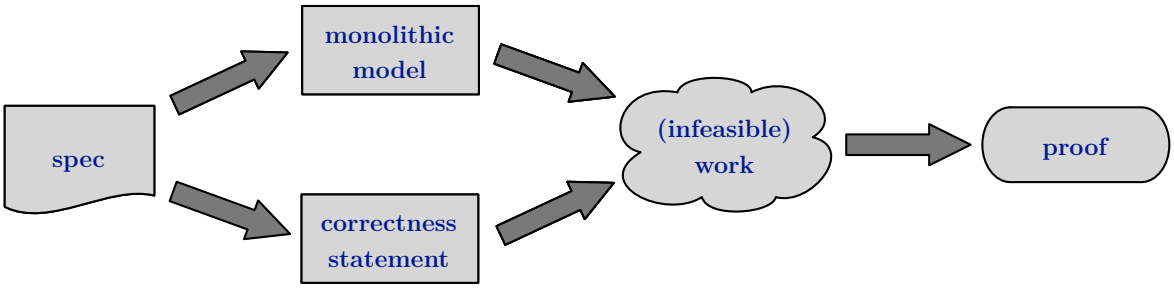
- **Communication protocols** traditionally hard to verify
- **On-chip:** increasing complexity (many-core architectures, System-on-Chips)
- **High-performance:** hard, advanced features to meet performance demands
- **Fundamental:** correct execution relies on correct data exchange

Need for functional verification

Traditional Approach

Traditional verification approach usually infeasible

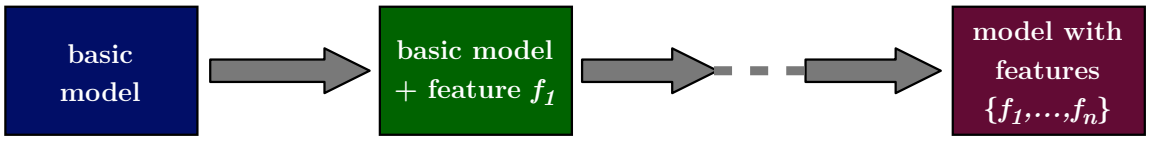
- Complex, **monolithic model**
  - High-performance features
  - Distributed, concurrent communication system
- Hard **post-hoc verification** process
  - large state space
  - complex correctness property (features)



Incremental Modelling and Verification

Idea: use **sequence of incremental modelling steps** to replace monolithic model

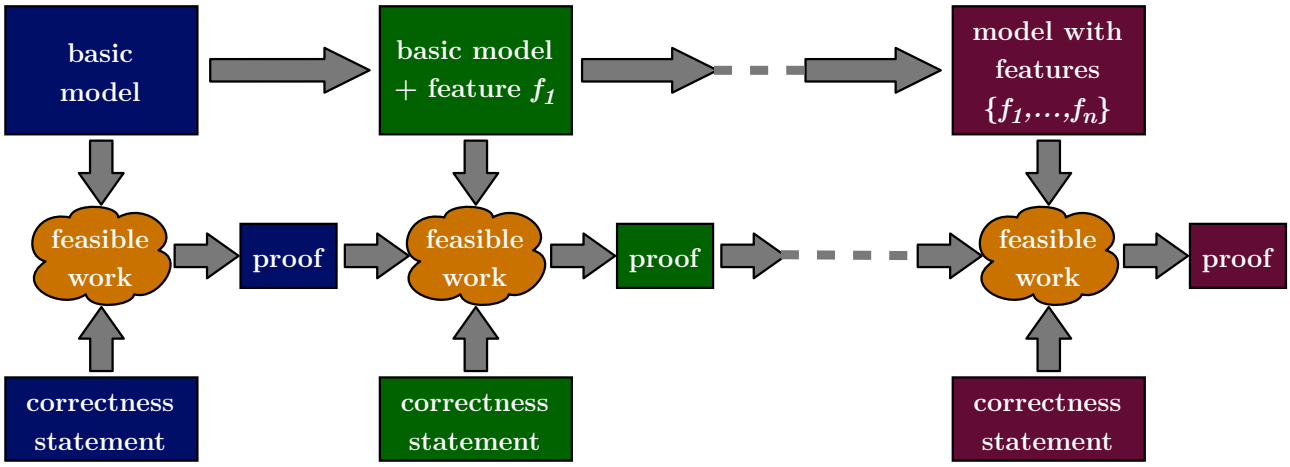
- **Basic model** with core functionality
- Incrementally add **features** in a structured, well defined way
- Features modelled independently using **transformations**
- **Complexity encapsulated**



Incremental Modelling and Verification

Idea: spread verification over modelling process

- Basic model verified using traditional approach (feasible due to model size)
- Show correctness of every modelling step
- **Leverage previous correctness properties**
- **Reuse** previously proven properties (lemmas)



This Talk

How to create a sequence of incremental models?

- Mathematical framework for incremental modelling
  - Modelling approach
  - Generic composition operators
  - Specific transformations
- Formalisation in Isabelle/HOL

How to apply the methodology?

- Overview of case study: PCI Express Transaction Layer
  - Basic model
  - Specific transformations

General Idea

Model communications system components as **state machines**

- **Mealy machines**
- Define a **generic structure** for state space, input and output sets

Extend state machines with **model of communication and composition**

- Introduce an **interface standard** for the inputs and outputs
- Provides basis for the model of composition

Define **generic transformations** using composition operators

Mealy Machines

Definition (Mealy Machine)

A state machine is given by a 6-tuple $(S, I, O, s0, \delta, \omega)$ where the components are given by

- S, I, O are the sets for state space, the inputs, and the outputs, respectively.
- $s0 \in S$ is the initial state.
- $\delta : S \times I \rightarrow S$ is the step function of the automaton, thus $\delta(s, i)$ is the next configuration of the automaton with the configuration s and the input assignment i.
- $\omega : S \times I \rightarrow O$ is the output function of the automaton, thus $\omega(s, i)$ is the assignment of the output values if the state machine is in configuration s and the input assignment is i.

Records

Sets of labelled tuples: structure the sets of a state machine

- Sets are **collections of tuples**
- Provide **names for tuple components** to access specific components

Example (Record)

Assume $R = (\!| a \in \mathbb{B}, b \in \mathbb{B} |\!)$ with $\mathbb{B} = \{\mathrm{T}, \mathrm{F}\}$.
Then,

- $\mathcal{R} = \mathbb{B}^2$
- $\mathsf{a} : \mathbb{B}^2 \to \mathbb{B}$ with $\mathsf{a}((x, y)) = x$
- $\mathsf{b} : \mathbb{B}^2 \to \mathbb{B}$ with $\mathsf{b}((x, y)) = y$
- Given $r = (\!| a = \mathrm{F}, b = \mathrm{T} |\!) \in R$, then $r.a = \mathsf{a}((\mathrm{F}, \mathrm{T})) = \mathrm{F}$

Records

UNIVERSITY OF OXFORD

Definition (Record)

A record set $R = (\!| l_0 \in \mathcal{S}_0, \ \ldots, l_i \in \mathcal{S}_i, \ \ldots, l_n \in \mathcal{S}_n |\!)$ of $(n+1)$-tuples is a set $\mathcal{R}$ with

$$\mathcal{R} = \{(s_0, \ \ldots, s_i, \ \ldots, s_n) \mid \forall j \in [0, n]. \, s_j \in S_j\} = \mathcal{S}_0 \times \ \ldots \times \mathcal{S}_i \times \ \ldots \times \mathcal{S}_n$$

together with labelling functions $\mathsf{l}_i : \mathcal{R} \to \mathcal{S}_i$ for each tuple component:

$$\mathsf{l}_i((s_0, \ \ldots, s_i, \ \ldots, s_n)) = s_i$$

Notation:

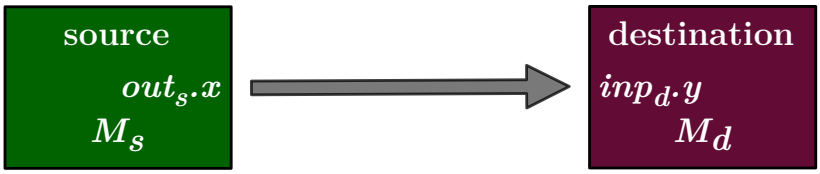
- A record instance $r \in R$ is given by $(\!| l_0 = s_0, \ \ldots, l_i = s_i, \ \ldots, l_n = s_n |\!)$ with $s_j \in S_j$ for $j \in [0, n]$.
- Given a record instance $r \in R$, we write $r.l_i \in \mathcal{S}_i$ for $\mathsf{l}_i(r)$.

Communicating State Machines

Goal

- Model **communication between network components** via channels.
- Specify **operators for composing state machines**.

Uni-directional communication



$$inp_d.y = out_s.x = (\omega_s(s_s, inp_s)).x$$

- Define communication as a **global function** over a set of state machines
- **Component aggregates** of input and output records

Component Aggregates of Records

Example

Assume $\mathcal{RS} = \{R_0, \ \ldots, R_n\}$ with $R_i = (\!| \, a \in \mathbb{B}, b \in \mathbb{B} \, |\!)$ and $n = 2$, then

$$\text{Agg}(\mathcal{RS}) = \{r_0.a, r_0.b, r_1.a, r_1.b, r_2.a, r_2.b\}$$

Definition (Component Aggregate of Records)

Given a set of records $\mathcal{RS} = \{R_0, \ \ldots, R_n\}$, we define the component aggregate of $\mathcal{RS}$ as $\text{Agg}(\mathcal{RS})$ with

$$\text{Agg}(\mathcal{RS}) = \{r_i.x \mid r_i \in \mathcal{R}_i \wedge (\exists j.\ x = l_j)\}$$

Global Communication Function

Communication among a set of state machines

- **Global function** mapping inputs to outputs.
- **Semantics:** every *data element* produced by the output is communicated to the input given by the function.
- An external input of a state machine gets defined by the output function of another state machine.

Definition

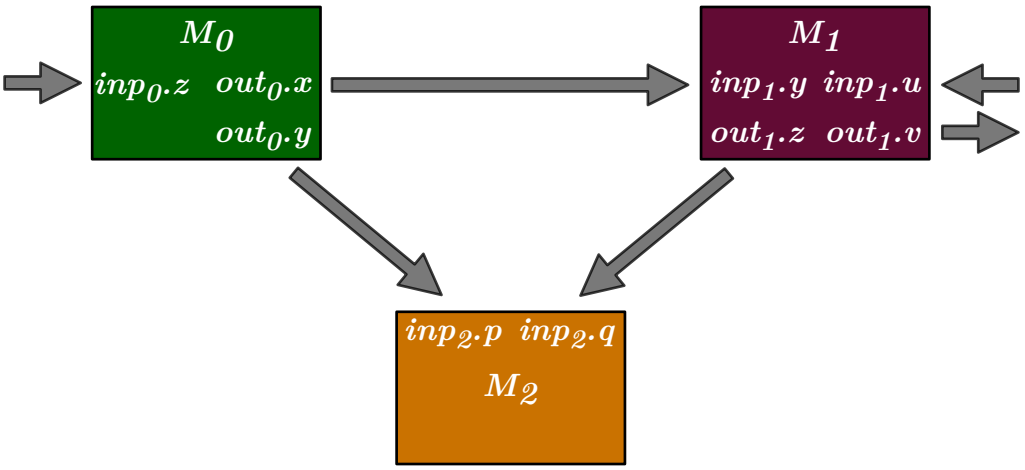
Given a set of state machines $\mathcal{M} = \{M_0, \ldots, M_n\}$ with input records I_i and output records O_i. We define the communication as a partial function $\text{com}_\mathcal{M} : \text{Agg}(\{I_i \mid i \in [0,n]\}) \rightarrow \text{Agg}(\{O_i \mid i \in [0,n]\})$ such that

$$\text{com}_\mathcal{M}(inp_i.y) = \begin{cases} out_j.x & : \text{output } x \text{ of } M_j \text{ is send to } M_i \text{ using input } y \\ \text{undefined} & : otherwise \end{cases}$$

Global Communication Function: Example

Example

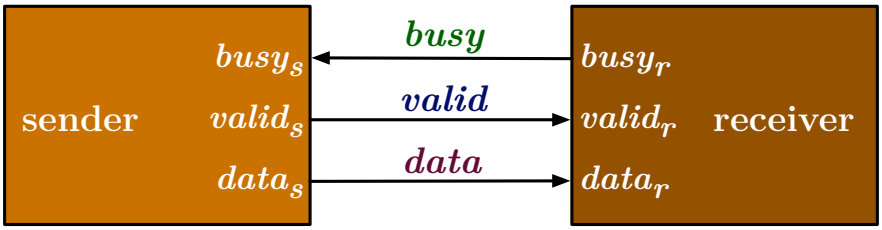
- $\mathcal{M} = \{M_0, M_1, M_2\},\ M_i = (S_i, I_i, O_i, s0_i, \delta_i, \omega_i)$
- $\text{com}_\mathcal{M} = \{(inp_1.y, out_0.x), (inp_2.p, out_0.y), (inp_2.q, out_1.z)\}$



Interface Convention

Simple handshake

- Introduce **standard interface** specification between components as basis for **composition operators**
- $busy \in \mathbb{B}$, $valid \in \mathbb{B}$, $data \in \mathcal{D}$ where $\mathcal{D}$ is the set of data elements to be communicated.



Semantics

- If sender wants to send data element x: $valid_s = \mathrm{T}$ and $data_s = x$
- If $busy_r = \mathrm{F}$: receiver samples data in the same time step.
- If $busy_r = \mathrm{T}$: receiver is busy and cannot sample data.
  Sender has to provide data until $busy_r = \mathrm{F}$, or data is not communicated.

A Generic Buffer

- Use **polymorphism** to define generic constructs
- Use the option **data type** for the data signal to formalise valid and data signals. Then the valid signal corresponds to $data = \mathrm{Some}\ x$

Definition ($(\alpha)buffer$ of finite size)

A generic buffer of finite size $l \in \mathbb{N}$ is given by the state machine $(S, I, O, s0, \delta, \omega)$ with

$$
\begin{aligned}
S &= (\!|\, data \in (\alpha)\mathsf{list}, length \in \mathbb{N} \,|\!) \\
I &= (\!|\, busy \in \mathbb{B}, data \in \alpha\ \mathsf{option} \,|\!) \\
O &= (\!|\, busy \in \mathbb{B}, data \in \alpha\ \mathsf{option} \,|\!) \\
s0 &= (\!|\, data = \mathrm{Nil}, length = l \,|\!) \\
\delta &= \lambda s \in S.\ \lambda i \in I.\ \mathrm{let} \\
&\qquad s' = \mathrm{if}\ \neg(i.busy \vee s.data = \mathrm{Nil})\ \mathrm{then}\ s' = (\mathsf{tail}\ s.data)\ \mathrm{else}\ s' = s.data \\
&\qquad s'' = \mathrm{if}\ (i.data = \mathrm{Some}\ x)\ \mathrm{then}\ s'' = s'@[x]\ \mathrm{else}\ s'' = s' \\
&\quad \mathrm{in}\ (\!|\, data = s'', length = s.length \,|\!) \\
\omega &= \lambda s \in S.\ \lambda i \in I.\ \mathrm{let} \\
&\qquad out = \mathrm{if}\ \neg(s.data = \mathrm{Nil})\ \mathrm{then}\ \mathrm{Some}\ (\mathsf{head}\ s.data)\ \mathrm{else}\ \mathrm{None} \\
&\quad \mathrm{in}\ (\!|\, busy = (\mathsf{length}\ s.data = l), data = out \,|\!)
\end{aligned}
$$

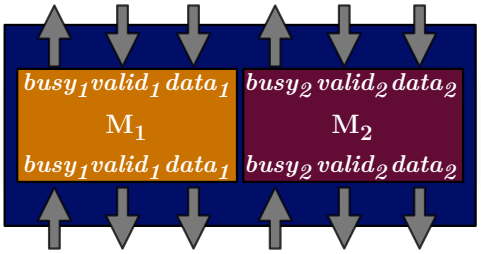
Parallel and Sequential Composition

UNIVERSITY OF OXFORD

- Standard (straightforward) composition operators
- Mainly used to **compose stack layers**

Parallel Composition

- **Goal:** Execute two state machines M_1, M_2 **in parallel**
- All inputs and outputs are inputs and outputs of the composed state machine.



Definition (Parallel Composition Operator)

The parallel composition $M_1 \text{par} M_2$ of state machines M_1 and M_2 with $M_i = (S_i, I_i, O_i, s0_i, \delta_i, \omega_i)$ is defined as $(S, I, O, s0, \delta, \omega)$ with

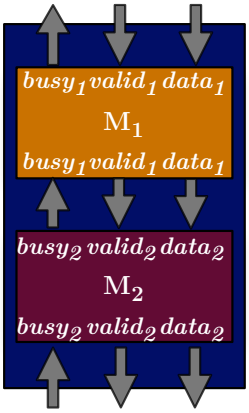
$$
\begin{aligned}
(S, I, O) &= (\{\!|\, m_1 \in S_1, m_2 \in S_2 \,|\!\}, \{\!|\, m_1 \in I_1, m_2 \in I_2 \,|\!\}, \{\!|\, m_1 \in O_1, m_2 \in O_2 \,|\!\}) \\
s0 &= \{\!|\, m_1 = s0_1, m_2 = s0_2 \,|\!\} \\
\delta &= \lambda s \in S.\ \lambda i \in I.\ \{\!|\, m_1 = \delta_1\ s.m_1\ i.m_1, m_2 = \delta_2\ s.m_2\ i.m_2 \,|\!\} \\
\omega &= \lambda s \in S.\ \lambda i \in I.\ \{\!|\, m_1 = \omega_1\ s.m_1\ i.m_1, m_2 = \omega_2\ s.m_2\ i.m_2 \,|\!\}
\end{aligned}
$$

Parallel and Sequential Composition

UNIVERSITY OF OXFORD

Sequential Composition

- **Goal:** Execute two state machines M_1, M_2 **sequentially**
- Data outputs of M_1 are connected to the inputs of M_2
- Remaining inputs and outputs are inputs and outputs of the composed state machine



Parallel and Sequential Composition

Definition (Sequential Composition Operator)

The sequential composition $M_1 \text{seq} M_2$ of state machines M_1 and M_2 with $M_i = (S_i, I_i, O_i, s0_i, \delta_i, \omega_i)$ is defined as $(S, I, O, s0, \delta, \omega)$ with

$$
\begin{aligned}
(S, I, O) &= ((\!| m_1 \in S_1, m_2 \in S_2 |\!), I_1, O_2) \\
s0 &= (\!| m_1 = s0_1, m_2 = s0_2 |\!) \\
\delta &= \lambda s \in S.\ \lambda i \in I.\ (\!| m_1 = \delta_1\ s.m_1\ int_1, m_2 = \delta_2\ s.m_2\ int_2 |\!) \\
\omega &= \lambda s \in S.\ \lambda i \in I.\ (\!| m_1 = \omega_1\ s.m_1\ int_1, m_2 = \omega_2\ s.m_2\ int_2 |\!)
\end{aligned}
$$

where

$$
\begin{aligned}
int_1 = (\!| busy = (\omega_2\ s.m_2\ (\!| busy = i.busy, valid = \mathrm{F}, data = x |\!)).busy, \\
valid = i.valid, data = i.data |\!) \text{ for some } x \\
int_2 = (\!| busy = i.busy, valid = (\omega_1\ m_1\ int_1).valid, data = (\omega_1\ m_1\ int_1).data |\!)
\end{aligned}
$$

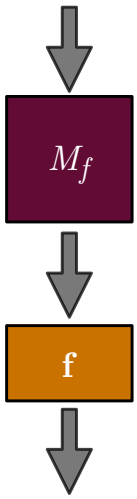
Note:

- Definition relies on the assumption that the busy output signal is independent from the valid and data input signals.
- Assumption needs to be discharged when sequential composition is used.

Combinatorial Function Composition

Goal: control and/or modify data output of a state machine.

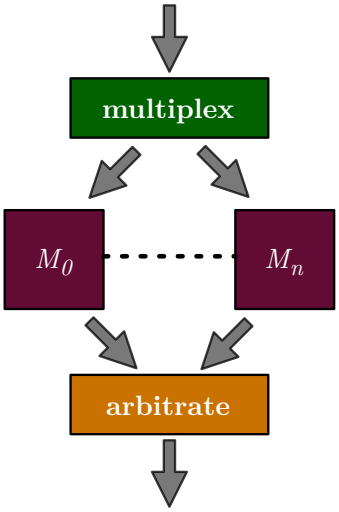
- **State space**
  $S = (\!| m \in M_f, e \in \mathcal{E} |\!)$ where $\mathcal{E}$ is a state space extension specific to the function f.
- **Input/Output domain**
  $I = I_f,\ O = (\!| busy \in \mathbb{B}, valid \in \mathbb{B}, data \in \mathcal{F} |\!)$ where $\mathcal{F}$ is the range of the function f.
- **Combinatorial function** $f : \mathcal{D} \to \mathcal{F}$ where $\mathcal{D}$ is the data output range of M_f.
  - Combinatorial in the sense that data elements are not *stored*.
  - Step function for f to update state space element e.
  - Output function for f that depends on e and the input signal, i. e. the output signals of M_f.



Generic Multiplex/Arbitrate Composition

Goal: controlled, parallel execution of $n+1$ state machines M_i while maintaining the input and output interface.

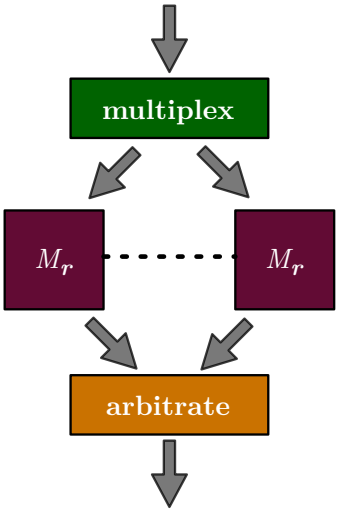
- **State space**
  $S = (\!| m_0 \in S_0, \ldots, m_n \in S_n, e \in \mathcal{E} |\!)$ where $\mathcal{E}$ is a state space extension specific to a concrete instance of the operator.
- **Input domain**
  $I = (\!| busy \in \mathbb{B}, valid \in \mathbb{B}, data \in \bigcup_i \mathcal{D}_i |\!)$ where $\mathcal{D}_i$ is the data domain of M_i. **Output domain** is defined analogously.
- **Multiplex relation** $mux \subseteq (S \times I) \times [0, n]$ to select the internal component(s) given input signal values.
- **Arbitrate function** $arb : (S \times I) \to [0, n]$ to select the component that outputs data.



Replicate Composition

Goal: controlled, parallel execution of $n+1$ **copies** of a state machine M_r.

- Similar to the generic multiplex/arbitrate, but more restrictive
- Advantage: more correctness results
- **State space**
  $S = (\!| m_0 \in S_r, \ldots, m_n \in S_r, e \in \mathcal{E} |\!)$
- **Input/Output domain**
  $I = I_r,\ O = O_r$
- **Multiplex function** $mux : (S \times I) \to [0, n]$ (instead of relation)
- **Arbitrate function** $arb : (S \times I) \to [0, n]$ analogous to multiplex/arbitrate composition



Signals and Execution Semantics

- Argue about behaviour over time
- Intuitive, standard definition
- Abstract, discrete time domain: $\mathbb{N}$

Definition (Signal)

A signal sig is a function from time $\mathbb{N}$ to a data domain $\mathcal{D}$. We write sig^t for $sig(t)$.

Definition (Execution and Output Trace)

Given a state machine $M = (S, I, O, s0, \delta, \omega)$ and input values $i^t \in I$ for $t \in \mathbb{N}$, we define the execution trace $trc_M : \mathbb{N} \to S$ and the output trace $out_M : \mathbb{N} \to O$ as

$$trc_M^t = \begin{cases} s0 & : t = 0 \\ \delta\ trc_M^{t-1}\ i^{t-1} & : otherwise \end{cases}$$
$$out_M^t = \omega\ trc_M^t\ i^t$$

Buffer Correctness

Correctness:

- Functional correctness (no data loss or modification)
- No reordering
- Liveness

Environment assumption:

- $busy$ input not constantly active

Lemma (Correctness of the Buffer FSM)

Given input signals $i^t \in I$, a generic buffer $(\alpha)buffer$ satisfies that

$$\forall x \in \alpha.\ \neg i.busy^t \wedge (i.data^t = \text{Some } x) \implies \exists k.\ (out_M^{t+k} = \text{Some } x)$$

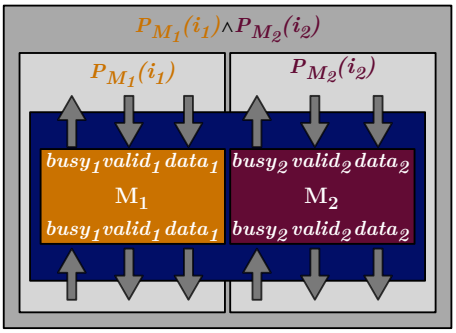
Note

- Analogous lemma with $x_1, x_2 \in \alpha$ shows in-order property.
- Easy lemma to show that data outputs independent of busy input.

Basic Compositions

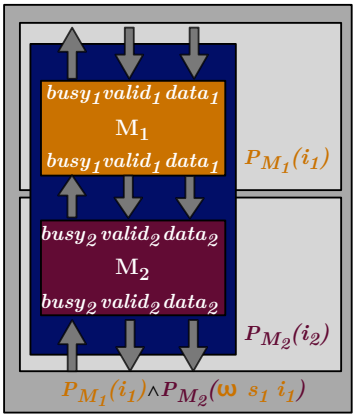
Parallel Composition

- Correctness properties of the components are maintained.
- Satisfies conjunction of the individual correctness properties.
- Environment assumptions of both state machines have to be satisfied.



Sequential Composition

- Satisfies conjunction of the correctness with the respective substitutions in P_{M_2} using ω_1.
- Analogously for the $busy$ input of M_1 (definition of sequential composition)
- Data output of M_1 has to satisfy the environment assumptions of M_2 and vice versa for the $busy$ input.

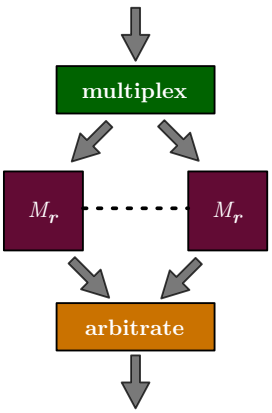


Replication Operator

Idea: Push correctness from inner components to system

Assumptions:

- M_r is **correct and ensures liveness**
- The multiplex function is **correct for valid inputs**
- The arbitration function is **fair** with respect to an active valid signal from some M_r



Theorem (Functional Correctness and Liveness)

The replication operator preserves the functional correctness and the liveness of M_r given the above assumptions.

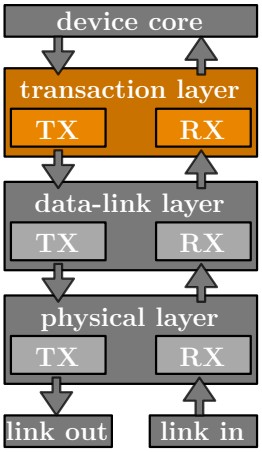
PCI Express

Protocol characteristics

- Point-to-point, packet-based communication
- Protocol stack layers: Transaction, data-Link, physical Layer
- Each layer: transmit (TX) and receive part (RX)

Memocode'09: Derivation of transaction layer

- Focus on hard transaction layer parts
  flow control, packet reordering, virtual channels
- Transformation-based modelling approach
- Formalization in Isabelle/HOL



Here: Summary of

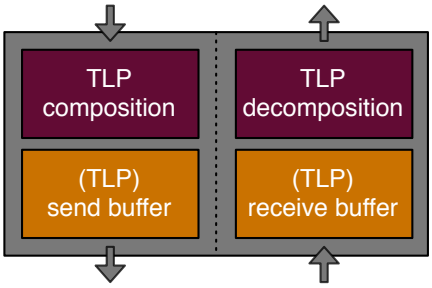
- Basic model
- Flow control

Basic Model

Data units: transaction layer packets (TLPs)

Model

- TLP composition/decomposition
- Send/receive buffers



Correctness

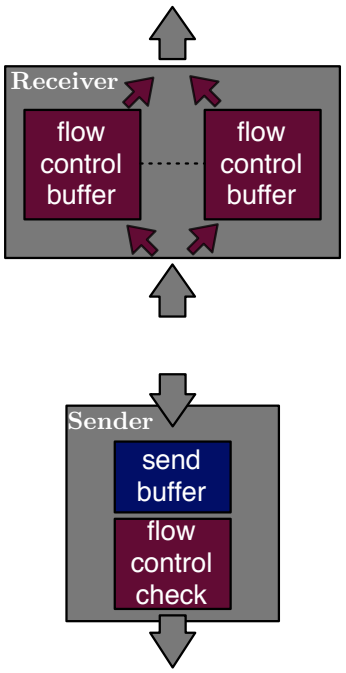
- TLP composition/decomposition (only combinatorial, easy)
- Apply correctness of generic buffer
  - Liveness
  - Ordering (no overtaking or packet loss)
  - Correct busy signal
- Sequential composition of TX, channel, and RX

Flow Control - A Specific Transformation

Goal: Sender checks locally if receiver has enough buffer space.

Principle

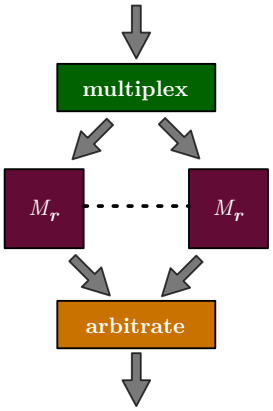
- Credit-based (header 1 credit, dw 1 credit)
- **Receiver:** Flow control buffers
  - For each message type (posted, non-posted, completion)
  - Header and payload (not every packet as payload)
  - Frequent updates to link neighbour
- **Sender:** Checks if space is available
  - Maintains available space counters
  - Checks before message transmission



Flow Control - A Specific Transformation

Receiver: Instantiate replication operator

- $n = 3$ with $(TLP, timestamp)$ flow control buffer
- Multiplex function is TLP to $[0:2]$ plus add time stamp
- Arbitrate function is n such that $timestamp(n) < timestamp(m)$ for all $m \neq n$



Flow control buffer: Instantiate multiplex/arbitrate operator

- $n = 2$ with $(TLPHeader, timestamp)$ and $(TLPData)$ data buffer
- Multiplex relation is $\{0\}$ if TLP has no data and $\{0, 1\}$ if TLP has data
- Arbitrate relation analogous to multiplex relation with respect to busy input

Sender: Instantiate combinatorial function operator

- Combinatorial function is counter test, raise $busy$ if there is not enough space

PCI Express Summary

- **Industrial-sized high-performance** communication protocol
- **Incremental modelling** of large parts of the transaction layer and data-link layer
- **Independent specification** of complex features
- **Transaction layer**
  - Flow control
  - TLP reordering
  - Packet priorities using virtual channels
- **Data-link layer**
  - Data-link layer packet arbitration
  - ACK/NAK protocol
  - CRC check
- Case study results published in MEMOCODE'09 and HFL'09

Conclusion

New methodology for an incremental modelling and verification process

- Control the model complexity by adding features incrementally
- Formalised framework with correctness results for the generic constructs
- Generalised design principle for transformations using composition operators
- HOL as design/modelling language

Long-term aim

- Increase efficiency of the model building process
- Model with significant merits against ad-hoc models
  - Functional verified
  - Independent from implementation or design architecture
  - Long-term reference model

Theorem prover

- Reduce or eliminate manual theorem proving
- Ideally modelling tool with knowledge management features

Future Work

PCI Express

- Support for power management and interrupts
- Derivation of switches (support for complex topologies)

Design and verification methodology

- Support for (automatic) refinement steps (data refinement)
- Integration of automated verification tools (model checking, SMT Solver)
- Link to HDL? (SystemVerilog)

Gap-Free verification of weakly programmable IPs against their operational ISA model

Markus Wedler, Sacha Loitz, Wolfgang Kunz
Department of Electrical and Computer Engineering,
University of Kaiserslautern, Germany
email: wedler@eit.uni-kl.de.

ABSTRACT

This paper suggests an *operational instruction set architecture (OISA)* model for specifying *weakly programmable IPs (WPIPs)*. WPIPs are application-specific programmable System-on-Chip (SoC) modules such as *application-specific instruction set processors (ASIPs)* whose individual instructions often implement large segments of an application algorithm corresponding to hundreds of usual RISC instructions. The pipeline structure of a WPIP design is usually determined by basic operations of the application algorithm. For this reason, the pipeline is often designed in a bottom-up manner where the components for the individual operations are developed first. Our OISA model reflects this design style by specifying the instruction semantics in terms of predefined operations that are associated with specific pipeline stages. The presented approach allows for a fully automatic generation of a property set that uniquely specifies the entire design. Moreover, the verification process used to design the OISA model explicitly reveals software constraints that are exploited for the optimization of the design.

1. INTRODUCTION

Formal property checking has become established in many design flows and enhances the verification methodology for System-on-Chip modules. Although formal techniques are usually considered to be no more than useful supplements to simulation-based verification there are also successes in making property checking the pre-dominant verification approach. This, however, raises the coverage issue in property checking. A methodology is required that creates a set of properties which does not only locally focus on specific design issues but completely and uniquely describe the behavior of the entire SoC module. Moreover, in order to make such a "gap-free" formal approach acceptable in practice it must be robust with respect to a large range of architectures and implementation styles being practiced in state-of-the-art SoC design.

In the design of SoC modules we observe a shift from dedicated hardware modules towards more flexible programmable devices. The goal is to provide just enough flexibility while optimizing performance and power consumption. Since general purpose processors (GPPs) do not provide the required hardware efficiency while dedicated hardware solutions are too expensive and lack flexibility there is a trend towards so called *weakly programmable IP (WPIP)* blocks with very specific programming models.

The microarchitectures of such WPIPs fundamentally differ from the usual GPP pipelines. Frequently, the instructions of a WPIP correspond to hundreds of classical RISC instructions and perform a certain well-defined part of the algorithms targeted to the WPIP. These segments of algorithms are also referred to as *nuclei*. The most prominent difference with respect to standard microprocessors is the absence of a classical ISA model specifying the effect of each instruction on the state holding elements of the processor.

Instead of refining such an ISA model top-down towards an optimized pipelined architecture, WPIP designers often follow a bottom-up approach specifying the basic operations of the pipeline first. In this bottom-up approach the semantics of WPIP instructions is implicitly given through these basic operations used to execute the instruction.

The above mentioned bottom-up design approach has a strong impact on the applicability of formal verification techniques for proving correctness of a WPIP design. The absence of a classical ISA model makes approaches based on classical refinement checking or bisimulation [10] difficult to apply. In this paper, we suggest an attractive alternative based on an *operational ISA (OISA)* model for WPIPs. This model specifies the semantics of a WPIP instruction by a collection of pipeline operations that need to be performed whenever this particular instruction is executed. We exploit that these operations usually have a well-defined impact on the WPIP design state that can also be specified in our OISA model. Additionally, our OISA model captures, by means of explicit software constraints, all dependencies between instructions that may be exploited by designers for optimization of the WPIP implementation.

Based on the OISA model we will show how to automatically generate a set properties which proves equivalence between the RTL code and the OISA model of the WPIP under verification. Due to the weak programmability of the SoC module software constraints turn out to be necessary for a successful proof of the properties against the (potentially manually) optimized RTL implementation. These software constraints may be re-used during software verification for checking compliance of software targeted to the WPIP with

these constraints.

2. INTERVAL PROPERTY CHECKING

In this section we briefly recall the basic concepts of interval property checking as it has been successfully used for the verification of industrial SOC modules specified at the RTL.

2.1 Property Language

There is a variety of temporal logics that may be used for specifying the temporal behavior of a circuit such as `CTL*,CTL,LTL, and ACTL`.

In order to make temporal logics more intuitive to circuit designers considerable efforts have been made to develop specification languages combining the expressiveness of temporal logics with syntactic features that ease property writing for designers. As a result, standardized languages such as PSL or System Verilog Assertions (SVA) are now available.

In this work, we consider a class of properties called *interval properties* or *operation properties* enabling a SAT-based verification approach capable of handling large SoC modules that are often beyond the capacity of classical (symbolic) model checking techniques.

Interval properties are safety properties that relate the signals of a design within a bounded time interval to each other. We introduce interval properties as a specific subclass of LTL. In an industrial setting, interval properties are commonly specified using the above standard specification languages.

In the sequel, we consider the signals of the device under verification (DUV) to be the atomic formulas of our property language.

DEFINITION 1. *A* timed Boolean predicate (TBP) *is a LTL formula that only uses the Boolean operators* $\wedge, \vee, \neg$ *and the next step operator* X. *An interval property is a safety property* $\mathsf{G}(p)$ *where* p *is a timed Boolean predicate.*

The generalized next state operator X^n denotes finite nestings of the next state operator, i.e., $\mathsf{X}^t(p)=\mathsf{X}(\mathsf{X}^{t-1}(p))$ for $t > 0$ and $\mathsf{X}^0(p) = p$. We will use the following notations:

$$\mathsf{During}\,[t_1,t_2](p) = \bigwedge_{t=t_1}^{t_2} \mathsf{X}^t\,(p)$$
$$\mathsf{Within}\,[t_1,t_2](p) = \bigvee_{t=t_1}^{t_2} \mathsf{X}^t\,(p)$$

DEFINITION 2. *A TBP is supposed to be in* timed normal form *if the (generalized) next step operator* X^t *is only applied to atomic formulas.*

It is straightforward that every TBP can be transformed into an equivalent formula in timed normal form. Without loss of generality we assume all TBPs to be in timed normal form, in the sequel.

DEFINITION 3. *Let* t_1 *denote the smallest and* t_2 *the largest exponent of the generalized next state operator in a TBP* p. *The interval* $[t_1, t_2]$ *is called the* inspection interval *of the TBP* p *and the corresponding interval property* $G(p)$.

Obviously, it is $\mathsf{X}^{t_1}\mathsf{X}^{t_2}(p)=\mathsf{X}^{t_1+t_2}(p)$. To further generalize the next step operator we also allow negative exponents X^{-t} such that $\mathsf{X}^t\mathsf{X}^{-t}(p)=p$. This can be applied to the notions timed normal form and inspection interval in a straightforward way. For a TBP p with inspection window $[t_1, t_2]$, $t_1 < 0$ the semantics of the formula $\mathsf{G}(p)$ is not obvious, however, and needs to be specified. Since we cannot reason about the history before the initial state of a Kripke structure, we define the semantics of this property as follows:

DEFINITION 4. *Let* p *be a TBP with inspection interval* $[t_1, t_2]$, $t_1 < 0$ *and let* q *be the timed normal form of the TBP* $\mathsf{X}^{-t_1}(p)$. *The semantics of the interval property* $\mathsf{G}(p)$ *is defined by the following equivalence*

$$\mathsf{G}(p) \equiv \mathsf{G}(q).$$

Obviously, in the above definition q has the inspection interval $[0, t_2 - t_1]$ and therefore the semantics of $\mathsf{G}(p)$ is well defined. Throughout this paper it becomes necessary to distinguish between TBPs that reason about the past and those that reason about the future only. This leads to the following definition:

DEFINITION 5. *TBPs with inspection interval* $[0, t], t > 0$ *are denoted as* future TBPs. *TBPs with inspection interval* $[-t, 0], t > 0$ *are denoted as* past TBPs.

2.2 Basic Decision Procedure for IPC

The computational model used for proving interval properties is a so called *iterative circuit model*. In this model, a finite number of copies $\delta_i, i = 0, \ldots, n$ of the DUV's transition function δ are concatenated via their present and next state variables. Each instance δ_i of the transition function is called a time frame of the iterative circuit model.

The size of this iterative circuit model corresponds to the size $n = t_2 - t_1 + 1$ of the inspection interval $[t_1, t_2]$ of the interval property under consideration.

In other words, the size of the iterative circuit model only depends on the "length" of the property. This clearly differentiates IPC from bounded model checking where the iterative circuit model may grow up to the diameter of the circuit in the worst case.

If we consider the transition function to be represented by a (bit vector) netlist we can refer to a signal s of the DUV in time frame δ_i by s_i.

A timed Boolean predicate p can easily be converted into a Boolean function in terms of the signals s_i of the iterative circuit model. Every instance $\mathsf{X}^t(s)$ of a signal s must simply be replaced by s_{t-t_1}. Concatenation of the resulting function $\hat{p}$ with the iterative circuit model Δ yields a function $\tilde{p} := \Delta \circ \hat{p}$ in terms of the inputs i_t^k of all time frames and the state variables x_0^l in time frame 0.

Obviously, the interval property $G(p)$ is valid if $\tilde{p}$ is a tautology. In practice, this is checked using a SAT/SMT solver. However, spurious counterexamples also called *false negatives* may be generated if the solver selects an assignment for the state variables x_0^l that encodes an unreachable state of the DUV.

In case of such a false negative the property can be strengthened with reachability information, e.g., provided in form of an invariant ϕ. Fortunately, in practice it turns out that most relevant properties of circuits can be proven using fairly simple invariants that are easy to be determined automatically, or manually by the verification engineer.

2.3 Operational methodology for IPC

In this section we describe a methodology for setting up a set of IPC properties that completely verifies the behavior

of a DUV. The methodology is guided by the overall goal to relate the DUV to an abstract specification FSM also called "conceptual" FSM. It pursues the following steps that will be elaborated in the sequel.

1. Identification of important abstract control states of the design.
2. Identification of possible (series of) transitions between the important states.
3. Description of the output behavior of the DUV for every transition.

Each phase results in a set of timed Boolean predicates that will finally be composed together to form the complete set of properties.

2.3.1 Important control states

In this phase, the verification engineer sets up a set of timed Boolean predicates (TBPs) that each characterize an important control state. It is up to the verification engineer how he groups the states of the DUV into abstract control states based on the existing specification documents.

This is a creative process conceptualizing about the design and thus cannot be automated. We illustrate the concept of important control states by a few examples.

- In order to check whether a device complies with a specific bus protocol, its states may be mapped to protocol states indicating whether the bus should be idle or performing a specific transaction. In the AMBA-AHB protocol states like *idle()*, *burst()* and *split()* could be specified as important control states.
- A processor could be described by a conceptual FSM consisting of important states indicating whether the processor is ready to execute the next instruction (*nextInstruction()*), has to accept an interrupt (*interruptPending()*), has to stall any of its pipelines ($stalled_i()$), needs to wait for a bus transaction to complete (*waitingForBus()*) etc.
- An arbiter could transition between important states indicating which client is granted ($grant_i()$) or whether the corresponding resource is idle (*idle()*).

2.3.2 Transitions between important states

After the conceptual states of the DUV have been identified the verification engineer will specify transactions that drive the DUV from one abstract state to another.

This shall be illustrated by means of an example. Since the main focus of this paper is on application specific processors (ASIPs) we consider again the second of the above examples.

In the usual mode of operation the processor will execute one instruction after the other. In other words, it will always loop in the state *nextInstruction()* executing one instruction after the other. Each of the instruction specified by the instruction set architecture (ISA) of the processor may be chosen as a transaction of the processor. A TBP that specifies the behavior of an addition instruction could be stated as follows:

$$\begin{aligned} &\mathit{addInstrExecuted()}:= \\ &\quad \mathit{nextInstruction()} \wedge \mathit{addFetched()} \\ &\quad \rightarrow \mathit{addPerformed()} \wedge \mathsf{X}(\mathit{nextInstruction()}). \end{aligned}$$

In this timed Boolean predicate the sub-expressions *addFetched()* and *addPerformed()* are used to specify that the addition instruction is fetched and how this instruction behaves.

In ISA models the behavior of general purpose processor instructions is typically described by the effect of the instruction onto the general purpose and status registers of the processor. However, there is not necessarily a one-to-one correspondence between the registers of the DUV and the ISA registers. Nonetheless, it should be possible for the verification engineer to devise TBPs for each of these registers. In the sequel, we consider the TBP *pc()* for the program counter and the TBPs $reg_i()$ for the general purpose registers.

Based on these TBPs the effect of an instruction on the state of the ISA model can be specified as follows:

$$\begin{aligned} &\mathit{addPerformed()}:= \\ &\quad \mathsf{X}(\mathit{pc()}) = \mathit{pc()}+2 \\ &\quad \wedge\, \mathsf{X}(\mathit{reg}_1()) = \mathit{reg}_1() + \mathit{reg}_2() \\ &\quad \wedge \textstyle\bigwedge_{i=2}^{n} \mathit{stable}(\mathit{reg}_i()), \end{aligned}$$

with $\mathit{stable}(s) := \mathsf{X}\, s = s$.

Similar interval properties like $\mathsf{G}(\mathit{addInstrExecuted()})$ can be formulated for every instruction of the ISA. For multi-cycle instructions the inspection interval may be extended to the number of cycles required for the instruction to be executed. Finally, the non-standard behavior is also specified in this manner, e.g., the transitions into the abstract states for interrupt handling and stalls.

As a result, we obtain a property set that checks whether the DUV refines its ISA model.

2.3.3 Description of output behavior

In this phase of the operational specification of a DUV the properties of the previous section are augmented with additional TBPs guaranteeing that the outputs of the block are computed during some transition of the abstract specification model.

3. IPC FOR WEAKLY PROGRAMMABLE IP

As already sketched in the introduction weakly programmable IPs impose new challenges for the classical IPC based property checking flow. The lack of a clean ISA model combined with non-standard schemes for memory access hampers the instruction-wise specification of the WPIP's behavior. Another aspect to be taken into consideration is the pipeline depth of WPIPs that is typically much deeper than for general purpose processors. This results in larger inspection intervals for the properties leading to complex decision problems for the underlying SAT engines. Furthermore we need to consider a so called *nucleus based design style* where the individual WPIP instructions subsume up to a few hundred classical RISC instructions. Often, the specification of such nuclei is only valid for a specific context in the program and a specific configuration of the processor pipeline. This imposes the need for the specification engineer to also model context and configuration while specifying a WPIP block.

In this section, we outline an operation based specification style for these blocks. The specification of every instruction is decomposed into a number of atomic operations that are assigned to a specific stage in the processor pipeline. In the specifications for all these operations we consider all contexts and configurations that are allowed for the individual operation. The following example illustrates this decomposition for an *SMBW* instruction of the WPIP presented as case study in [9]. This WPIP instruction implements an algorithmic nucleus containing a zero overhead loop that calculates a series of so called state metrics and is specified as follows:

SMBW_performed():=
decrOffsetSM() ∧ *decrSrcAdr()* ∧ *loadCV()*
∧ *calculateSMBW()* ∧ *doSMIO)()* ∧ *decideZeroOverHeadLoop()*

The operations used in this specification are likewise specified by timed boolean predicates and can be described as follows:

- decrOffsetSM(): decrement the address of the state metric register file.
- decrSrcAdr(): decrement the source address for the next state metric calculation.
- loadCV(): load the so called *channel values* from the address specified by decrSrcAdr() with respect to the selected code rate. The latter is defined by a dedicated configuration register.
- calculateSMBW(): calculate the 8 state metrics for the loaded channel values. This includes the calculation of so called branch metrics in a first step. For the calculation of the 8 state metrics 21 additions and 8 minimum selections on 12 bit words are performed in this operation in each iteration of the instruction.
- doSMIO(): store the calculated state metrics at the memory address calculated in decrOffsetSM().
- decideZeroOverHeadLoop(): The SMBW instruction is usually performed several times. For a more efficient implementation a zero overhead loop is executing the instruction for a specified number of iterations. Except for the last iteration of this zero overhead loop the PC is not incremented and the number of iterations that have to be performed is decremented instead.

In this example, the semantics of the WPIP instruction is implicitly specified by the semantics of the involved operations. We formalize this by introducing the notion of an operational ISA model.

DEFINITION 6. *Let I denote the set of instructions and O denote the set of pipeline operations of a WPIP design. An operational ISA (OISA) model of the WPIP consists of the following items:*

1. *A relation $OISA \subset I \times O$.*
2. *A set of timed Boolean predicates $instr_i Fetched()$ that specify when a specific instructions $i \in I$ has to be executed by the pipeline of the WPIP.*
3. *A set of future timed Boolean predicates $op_o()$ to specify the semantics of the individual operations $o \in O$.*

Based on the OISA model the timed Boolean predicates specifying the instruction semantics for every instruction $i \in I$ can be automatically generated as follows:

$$instr_i Performed() := \bigwedge_{(i,o)\in OISA} op_o().$$

This results in an *instruction-based property* for the standard behavior of the WPIP as given by the following TBP:
$instr_i Executed()$:=
$nextInstruction() \wedge instr_i Fetched()$
→
$instr_i Performed() \wedge \mathsf{X}^{t_i}(nextInstruction())$.
In this TBP t_i indicates the latency of the instruction i.

However, in practice a monolithic verification of complete instructions is hampered by the capacity limits of the property checkers. In order to overcome this limitation we suggest an alternative formulation by *operation-based properties*. The OISA model allows to generate triggering conditions for every operation $o \in O$ as follows:

$$op_o Triggered() := \bigvee_{(i,o)\in OISA} instr_i Fetched().$$

Using these triggers we obtain the following properties:
$op_o Executed()$:= $nextInstruction() \wedge op_o Triggered() \rightarrow op_o()$.

In the instruction-based properties for every $t \in \{t_i | i \in I\}$ the commitments

$$\mathsf{X}^t(nextInstruction())$$

have been used to specify that the DUV returns to an abstract state that allows for execution of the next instruction. Note that we can treat this predicate in the same manner as the other operations of the instruction. In this way, it is guaranteed that the set of operation-based properties is equivalent to the set of instruction-based properties that can be generated from the OISA model.

In practice, we can reduce the specification effort of the verification engineer drastically if we only require that the timed Boolean predicates *op()* for the pipeline operations specify the register/memory modifications that a specific operation carries out in the DUV. In this case, it needs to be checked that the operation does not have a side effect on any of the unspecified registers. This will be discussed in section 5.

4. CONFIGURATION AND CONTEXT DEPENDENT CONSTRAINTS

In the previous section, we introduced the operational ISA model as a basis for IPC based property checking of weakly programmable IP designs. So far, the pipeline operations of the design have been assumed to be independent of each other. This is not the case, however, in the presence of shared resources such as global registers or memories used by multiple operations that may even be located in different pipeline stages. The resulting structural hazards need to be resolved either in hardware or in software.

At this point, WPIP designers follow a strategy slightly different from common GPP design practices. As long as a structural hazard does not occur in the anticipated contexts and configurations its resolution is typically within the responsibility of the software.

Fortunately, the properties specified in the previous section can be used to detect such hazards. More precisely, at least one of the properties will fail in case of such a hazard.

Suppose two operations $op_1()$, $op_2()$ in different pipeline stages write to the same global register r. In the implementation this results in two concurrent assignments to the corresponding hardware signals. Modern property checkers provide predefined checks to detect such race conditions in advance. Additionally, the IPC checker will provide counterexamples for the interval properties $\mathsf{G}\ op_1\ Executed()$ and $\mathsf{G}\ op_2\ Executed()$. In this counterexample, both operations will be triggered by two instructions with a time offset that corresponds to the number of pipeline stages between the operations. The property checker exposes the race conditions as required to produce the respective counterexample.

If this counterexample is realistic for real application software a bug in the hardware is detected and needs to be fixed. However, in many cases such counterexamples only indicate that the designer assumed certain, often undocumented, restrictions for the software and used them for hardware optimization. In the case study of [9] such software constraints were added to the properties in a time consuming manual process.

In this work, we partially automate the detection of such software constraints. In particular, we relieve the verification engineer from the explicit specification of dependencies between instructions. Instead, the verification engineer can focus on those cases where more complex constraints regarding the context of an instruction are required. This process finally leads to a complete documentation and formal description of all software restrictions. They are added to the property set for the hardware so that false alarms related to the corresponding race conditions are avoided.

4.1 Hazard detection

The OISA model presented in the previous section can be used for detection of potential hazards with the algorithm presented in Table 1.

algorithm *HazardDetection*
  inputs
    OISA model: $(O, I, OISA)$
    Set of registers of the DUV : R
begin
  forall $(o_1, o_2 \in O)$
    if ($\exists r \in R$, *timepoints* t_1, t_2:
        $X^{t_i} r$ *is subexpression of* $o_k()$ *(k=1,2))*
      forall $(i_1, i_2 \in I : (i_1, o_1) \in OISA \wedge (i_2, o_2) \in OISA)$
        if ($t_1 > t_2$ **or** ($t_1 = t_2$ **and** $i_1 = i_2$ **and** $o_1 \neq o_2$))
          addToConflictList$(i_1, i_2, o_1, o_2, t_1 - t_2)$
end algorithm

Table 1: Hazard detection based on OISA model

This algorithm parses the TBPs $op()$ used in the OISA model for the specification of the pipeline operations. Without loss of generality, we assume all these TBPs to be in timed normal form. In this case it is sufficient to check whether a pair of operation specifications $op_1()$, $op_2()$ depends on a common register r of the DUV. This is critical if the operations are located in different stages of the pipeline, or if they are triggered within the same instruction. In both cases the corresponding instructions, operations and timing offsets of the operations are stored in a list of conflicts.

For each entry (i_1, i_2, o_1, o_2, t) of the conflict list the verification engineer has to define how the corresponding hazard should be resolved. Two options are considered:

1. Add an automatically generated software constraint $(X^{-t}\ instr_{i_2}\ Fetched()) \rightarrow (\neg\ instr_{i_2}\ Fetched())$ to the verification environment.

2. Manually find a less restrictive constraint for the context of the conflicting instructions that resolves the hazard.

While the automatically generated constraints are approriate in most situations some cases remain that require the attention of the verification engineer. For example, in the case $i := i_1 = i_2$ and $t = 0$ the automatic constraint is too restrictive as it forbids the instruction i to be used in programs. Similarly, other combinations of instructions that may be forbidden by the automatic constraints may be important for the application. In this case, it is up to the verification engineer to find a weaker constraint modeling the context of the two involved instructions that resolves the hazard.

Often these constraint can easily be specified in terms of some status registers used as flags f for the datapath of the pipeline. In general we allow constraints to be specified as a TBP of the form:

$$sw\_constraint_{i_k}() := instr_{i_k}\ fetched() \rightarrow flags_{i_k}()\ (k = 1, 2).$$

In this constraint the TBPs $flags_{i_k}()$ have to be specified by the verification engineer and should describe the status of the DUV that is required for proper execution of the instructions i_k. In order to avoid subsequent instructions to eventually influence the validity of this constraint, we restrict $sw\_constraint_{i_k}()$ to be a past TBP.

By this means the verification engineer can specify a global reachability constraint for the software. Under this constraint it should be possible to weaken the description of at least one of the operations o_1, o_2 that caused the problem. This results in TBPs $o_1'(), o_2'()$ that should not both depend on the common resource any more and have to be added to the OISA model. Moreover, the pairs $(i_k, o_k) \in OISA$ should be replaced by the pairs $(i_k, o_k') \in OISA$.

In the remainder of the paper we consider $swConstraints()$ to denote the conjunction of all the above constraints, regardless of whether they are automatically generated or manually specified by the verification engineer. When completing the verification of the hardware the operation-based properties for this modified OISA model are augmented with the constraints as stated below and re-checked against the DUV.

$$op_o\ Executed() :=$$
$$swConstraints() \wedge nextInstruction() \wedge op_o\ Triggered() \rightarrow op_o().$$

Finally, the conflict analysis of Table 1 is rerun. In this run only conflicts resolved by the automatically generated constraints are expected to pop up. If contrary to expectations new potential hazards are reported this indicates a mistake of the verification engineer when updating the corresponding operation descriptions in the OISA model. In this case the manual update process for these operations is repeated.

5. COMPLETENESS

In the previous sections we have introduced an interactive process for hardware verification of weakly programmable IPs under generated software constraints. One critical question of this approach is how to ensure that the complete functional behavior of the weakly programmable IP has been specified.

This question points to an active field of research with many recent contributions [8, 7, 5, 12, 1, 6, 3, 2, 4]. Most of the studied approaches relate the property set under consideration towards the verified design and compute some sort of mutation coverage. In this way completeness of a property set becomes a design dependent metric. The approaches of Claessen and Bormann [3, 2, 4] use a design independent notion of completeness that will be used also in this work. We briefly recall it using our notations of TBPs in the sequel.

DEFINITION 7. *A set of properties is complete if and only if any two circuit implementations satisfying the property set are sequentially equivalent modulo explicitly specified constraints C and determination requirements D.*

A constraint $c \in C$ is an arbitrary TBP while a determination requirement is a pair $(s, d_s) \in D$ consisting of a signal s and a past TBP d_s.

Note, that Claessen introduces the temporal operator *free* to characterize undetermined behavior of a signal inside a property, while Bormann uses a separate completeness specification which extends the proprietary Onespin property language ITL and includes the specification of determination requirements.

The computational model for checking completeness of a property set P of both above mentioned approaches consists of two sets of independent variables S, S' for the design signals being referred to by at least one of the properties $p \in P$.

However, the completeness check itself fundamentally differs. Claessen creates a monolithic model checking instance that checks whether the validity of the property set implies the global equality of every output $o, o' \in S$ of the design. This results in the LTL formula

$$compl(P, o) := P(s, s') \rightarrow \mathsf{G}\,(o = o')$$

for every output o of the design. In this instance $P(s, s')$ refers to the conjunction of all properties in P instantiated for both variable sets S and S'. During construction of $compl(P, o)$, undetermined signals $s \in S$ indicated by the *free* operator are immediately replaced by the corresponding signal $s' \in S'$.

By contrast, Bormann conducts a (temporal) inductive proof of completeness for a set of operation properties. The proof consists of local checks over bounded time intervals that require investigation of one or two properties each. This dramatically reduces the proof complexity and renders the approach very attractive in an IPC environment. In particular the following checks are performed:

1. *Case split test*: For every property $p \in P$ with conceptual end state s_p and every input trace $(i_t, t = 0, ..., t_{max})$ a property $p' \in P$ starting in the conceptual state s_p exists such that assumption of p' is fulfilled by s_p and the input sequence $(i_t, t = 0, ..., t_p)$.
2. *Successor test*: Checks that the successor property p' as mentioned in the case split check is uniquely determined by the input sequence $(i_t, t = 0, ..., t_{max})$.
3. *Determination test*: Checks that for every property $p \in P$ the determination requirements are fulfilled within the determination interval of p.

Note, that in the above characterization of the completeness checks t_{max} corresponds to the size of the largest inspection interval $[0, \ldots, t_{max}]$ used in P.

Moreover, it should be noted that by default, the determination interval of a property p corresponds to the inspection interval of p and may be modified by the verification engineer by an explicit specification. For simplicity of the presentation we do not detail on this technicality.

Case split and *Successor test* together guarantee that every unbounded input trace can be covered by a unique sequence of properties $(p_k \in P, k \geq 0)$ such that the assumptions of each of the properties p_k are fulfilled provided that all previous properties $p'_k, k' < k$ are valid. This allows for a local evaluation of the determination requirements.

In the case of WPIP verification the case split and successor tests for the instruction-based formulation of the properties become trivial. Recall that forbidden instructions are excluded by an explicit (software) constraint in C and that we generate a single property per instruction. Moreover, the instruction word is read from memory via inputs of the WPIP. For proper decoding the instruction opcodes are supposed to be pair-wise disjoint which is double-checked by the case split check.

Setting up and fulfilling the determination tests is, however, incomparably harder for WPIPs. In comparison with GPPs we face a lot more global registers that need to be determined. Even worse, we may face so called out-of-order register access to one and the same global register in several pipeline stages. This distributed register access further complicates the determination of those registers at every point in time. Recall that in our OISA model the verification engineer describes the behavior of the individual pipeline operations separately. In the pipeline of the WPIP multiple such operations are always active at the same time. For designers and verification engineers it is almost impossible to estimate which other operations may be active in the pipeline in a specific situation. Focusing on a specific operation it is only possible to specify which registers are written.

This of course may introduce gaps for the determination test, as the completeness checker will pin-point to a situation, where a subset of operations is active such that some of the registers will be undetermined.

In order to fulfill the determination requirement for registers that may not be affected by the pipeline operations in a certain scenario the verification engineer would explicitly have to specify which registers keep their value or take a default value. Note that this has to be done for an overwhelming number of possible pipeline contexts. Due to the manifold combinations of active operations this is a very tedious task. Furthermore, the evaluation of these conditions may require additional pipeline registers for intermediate results or flags to be determined and cause a chain reaction of additional specification effort. This results in an exploding number of additional determination problems that may unnecessarily slow down the completeness checker.

Together with the inherent complexity of typical operations this quickly drives the completeness checker towards its capacity limits.

In order to overcome the above mentioned limitations and to reduce the manual specification effort we propose to automatically complete the property set. The only information that we require the verification engineer to provide is a classification of the design registers in two disjoint categories $Reg = Reg_s \cup Reg_d$. The registers $r \in Reg_s$ of the first category are supposed to keep their value unless they are explicitly modified by any of the operations. The registers $r \in Reg_d$ of the second category are supposed to take a constant default value $def(r)$ unless explicitly modified. We generate additional proof obligations in our property set to check whether the non-written registers show this default behavior.

The generation of these proof obligations will be detailed in the sequel. For every register r we compute the subset of operations $D_r \subset O$ such that the TBPs $op_o()$ specify a behavior for the register r, i.e., for every $o \in D_r$ the TBP $op_o()$ includes a subexpression $\mathsf{X}^{t_r^o} r = spec(o, r)$ where $spec(o, r)$ is a TBP with inspection interval $[t_r^o - \hat{t}, t_r^o - 1]$.

Using the corresponding trigger TBPs we can determine a condition in terms of the instruction history under which the register r remains undetermined. This condition is formulated as a past TBP as follows:

$$\mathit{default\_trigger(r)} := \bigwedge_{o \in D_r} \neg\, \mathsf{X}^{-t_r^o}\, \mathit{op_o Triggered()}$$

In order to complete our property set we generate the TBPs

$$\mathit{default\_behavior(r)} := \mathit{default\_trigger(r)} \rightarrow \mathsf{X}\,(r) = r$$
for every $r \in Reg_s$ and

$$\mathit{default\_behavior(r)} := \mathit{default\_trigger(r)} \rightarrow \mathsf{X}\,(r) = def(r)$$
for every $r \in Reg_d$.

Note that this process for completion of the property set is fully automated. Obviously, neither the above TBPs for specifying the default behavior of a register r nor the TBPs $op_o Executed()$ for specifying the operations fulfill Bormann's property-wise determination criterion where each property has to determine every relevant signal of the design.

In the remainder of this section, we prove that the properties in total nonetheless completely determine every design register. In order to simplify the presentation we omit the software constraints in the sequel. Note, that a generalization is straightforward. We show, that the TBP

$$P := \bigwedge_{r \in Reg} \mathit{default\_behavior(r)} \wedge \bigwedge_{o \in O} \mathit{op_o Executed()}$$

completely determines all registers $r \in Reg$. Note, that the above TBP is implied by our properties.

We conduct the step case of a temporal induction. The determination of the registers in the base case is ensured by a dedicated property for checking the reset behavior of the design that particularly checks the initial values of the registers. It should be noted that in practice the completeness check may be weakened by allowing some of the registers to be uninitialized after reset.

For the step case of the proof we may assume that the property $\mathsf{G}\,(P)$ determines every register $r \in Reg$ in the time interval $[0, \ldots, t-1]$ from reset.

We pick a register $r \in Reg$ and evaluate the past TBP $\mathit{default\_trigger(r)}$ at time t-1. Due to the induction hypothesis the value of this TBP is determined. In case it is true, also the register r is determined either by the default value or by its previous value which again is determined by the induction hypothesis.

In the remainder of the proof we can now focus on the non-default behavior for r, i.e., we may assume that $\mathit{default\_trigger(r)}$ evaluates to false. This implies that for at least one operation $o \in D_r \subset O$ the TBP $\mathsf{X}^{-t_r^o} \mathit{op_o Triggered()}$ evaluates to true for an instruction sequence starting at timepoint $t_0 = t - t_r^o$. We claim that for every instruction sequence this operation o is uniquely determined. This is implied by the software constraints introduced for the removal of the hazards in Section 4.1.

For determination of r we need to investigate the TBP $op_o()$. By construction of the default properties this TBP contains a subexpression $\mathsf{X}^{t_r^o} r = spec(o, r)$ that determines the value of the register r based on the values for other registers $r' \in Reg$ and the inputs. In order to check whether $spec(o, r)$ is determined we have to shift its inspection interval $[t_r^o - \hat{t}, t_r^o - 1]$ to the time point t_0. This indicates that we need to consider the value for the other registers in the time interval $[t_0 + t_r^o - \hat{t}, t_0 + t_r^o - 1] = [t - \hat{t}, t - 1]$. By induction hypothesis the registers are determined in this time interval.

5.1 Weaker determination requirements for pipeline registers

The completeness approach as presented so far does not distinguish between global design registers and temporary pipeline registers and requires global determination for every register. Some operations on pipeline registers, however, may not be relevant in every context.

In our OISA model such an operation o would only be related to an instruction i if at least one of the modified pipeline registers is relevant for the behavior of the instruction. In WPIP implementations it may happen that such operations are also started by instructions that never use the corresponding results. Note, that such a behavior may considerably simplify the decoding of the instructions.

In the optimized implementation the pipeline registers modified by o will not show the default behavior generated from the OISA model. In order to avoid unnecessary specification effort for specifying irrelevant behavior of pipeline registers in a specific implementation we introduce a special treatment for these registers in the sequel. Note, that this also increases re-usability of the specification.

Instead of globally proving determination of the pipeline registers r_p we conduct a local determination analysis whenever a pipeline register is used inside a specification $spec(o, r)$ of another register $r \in Reg$. We check that every instruction i that invokes o, i.e., $(i, o) \in OISA$, invokes exactly one other operation o' such that r_p is determined at the appropriate point in time. Of course $spec(o', r_p)$ may require other pipeline registers to be determined as well. We avoid cyclic dependencies in the specifications of pipeline registers by a restrictive use of the generalized next state operator $\mathsf{X}^{t_{r_p}}$ for these registers. Our tool checks that the following restrictions are always valid. We ensure that all specification TBPs $spec(o', r)$ use a unique value t_{r_p} for every instance of $\mathsf{X}^{t_{r_p}}$. Moreover, every operation o_p writing r_p is supposed to use a smaller exponent $t \leq t_{r_p}$ for every other pipeline register $\mathsf{X}^{t} r'_p$ used in the subexpression $\mathsf{X}^{t_{r_p}} r_p = spec(o', r_p)$ that specifies the value of r_p. The exponent t_{r_p} corresponds to the pipeline stage where the pipeline register is written. The

restriction on the usage of pipeline registers in specifications of other pipeline registers reflects the fact that pipeline registers should be written by an instruction prior to a read access of the same instruction. If this restriction is not valid for a specific register we treat it as global register.

6. APPLICATIONS

The presented techniques for gap-free formal WPIP verification have been implemented in a tool chain on top of the commercial property checking tool Onespin 360 MV [11] that supports Bormann's completeness checking approach [2].

The methodology presented in this paper has been applied to designs implementing various channel decoding algorithms partly for industrial use. More precisely we verified an academic MAP decoder design and several versions of a flexible trellis processor called FlexiTreP. The FlexiTreP designs support multiple channel decoding algorithms. This results in a large pipeline with 15 pipeline stages and a sophisticated distributed memory architecture.

The fact that both designs stem from the same application domain has made it possible to reveal a further advantage of our OISA. Although the designs have completely different instruction sets and pipeline structures they share a couple of similar operations. The verification IP for these operations turned out to be reusable with marginal adoptions. For example, the timing of an operation that has been spread over a number of subsequent pipeline stages differs from a single cycle implementation which has to be reflected in the verification IP.

Before starting formal verification all designs had been intensively simulated by the design team applying the debugging features of the industrial ASIP design tool used in the projects. Although the designers considered the respective IPs ready for sign-off we were able to identify serious bugs that lead to a code revision.

For example, we identified two bugs of the MAP design that were caused by inconsistent bit widths of operands and source registers that resulted in a wrong code for sign extensions. For the FlexiTrep design we were able to show that a saturation operation located in pipeline stage 14 showed an unintended behavior under certain conditions on the operands. For operands in a certain range two out of three saturation units evaluated a wrong saturation condition. Due to the weak controllability of the late pipeline stages this corner-case behavior was masked during the intensive sign-off simulations

Another bug identified by property checking turned out to be caused by a late code change where the designer forgot to remove a specific value assignment to a specific control signal. This resulted in a race condition for two parallel RTL assignments for the same signal. Furthermore, we discovered a bug in the RTL code generation concerning the translation of a sequence of if-statements. The corresponding bug was not found by simulation because the simulation was mainly performed within the ASIP design environment where the behavior was correctly simulated.

Besides the above mentioned bugs, the designers also considered our feedback valuable for further optimizations of the design. Finally, we also identified bugs in software programs targeted to the WPIPs where the software constraints identified by our formal techniques had been ignored by the assembler programmer. This has been checked by a fully automatic and extremely efficient compliance check that reuses the explicitly available constraints that result from the verification process.

7. CONCLUSION

This paper presents a formal verification methodology which adapts an industrial gap free property checking flow to the specific requirements of *weakly programmable IP* (WPIP) designs. We design a verification IP that we call an operational ISA (OISA) model. Based on the OISA model we generate a set of interval property that provably fulfills the completeness criterion used by the gap free verification methodology introduced by Bormann [2].

As a by-product of the formal verification efforts taken for the hardware we obtain a formal specification for the restrictions that the software must comply with when running on the verified WPIP. In an earlier case study it has already been demonstrated that an automatic compliance check for the software is feasible by making only small extensions to a state-of-the-art formal property checker.

Together with the gap free verification result for the hardware we believe that this allows for verification of the software targeted towards the WPIP with respect to an abstract model derived from the OISA model. This will be subject of our future research.

8. REFERENCES

[1] P. Basu, S. Das, A.Banerjee, P. Dasgupta, P. Chakrabarti, C. Mohan, L. Fix, and R. Armoni. Design-intent coverage-a new paradigm for formal property verification. *IEEE Transactions on Computer-Aided Design*, 25(10):1922–1934, October 2006.

[2] J. Bormann. *Vollständige Verifikation*. Dissertation, Technische Universität Kaiserslautern, 2009.

[3] J. Bormann and H. Busch. Verfahren zur Bestimmung der Güte einer Menge von Eigenschaften (Method for determining the quality of a set of properties). European Patent Application, Publication Number EP1764715, 09 2005.

[4] K. Claessen. A coverage analysis for safety property lists. In *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 0, pages 139–145, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[5] D. Grosse, U. Kühne, and R. Drechsler. Estimating functional coverage in bounded model checking. In *Proc. International Conference on Design, Automation and Test in Europe (DATE)*, pages 1176–1181, San Jose, CA, USA, 2007. EDA Consortium.

[6] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proc. International Design Automation Conference (DAC)*, pages 300–305, New York, NY, USA, 1999. ACM.

[7] S. Katz, O. Grumberg, and D. Geist. "Have I written enough Properties?" - A Method of Comparison between Specification and Implementation. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 280–297, London, UK, 1999. Springer-Verlag.

[8] T.-C. Lee and P.-A. Hsiung. Mutation coverage estimation for model checking. In *Automated*

Technology for Verification and Analysis (ATVA), 2004.

[9] S. Loitz, M. Wedler, C. Brehm, T. Vogt, N. Wehn, and W. Kunz. Proving functional correctness of weakly programmable IPs - a case study with formal property checking. In *Proc. 6th IEEE Symposium on Application Specific Processors (SASP)*, Anaheim, CA, USA, June 2008.

[10] P. Manolios and S. K. Srinivasan. A refinement-based compositional reasoning framework for pipelined machine verification. *IEEE Transactions on VLSI Systems*, 16:353–364, 2008.

[11] Onespin Solutions GmbH, Germany. OneSpin 360MV. `www.onespin-solutions.com`.

[12] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design and Test of Computers*, 2001.

A Prototype Embedding of Bluespec SystemVerilog in the SAL Model Checker

Dominic Richards and David Lester[1,2]

School of Computer Science
University of Manchester
The United Kingdom

Abstract

Bluespec SystemVerilog (BSV) is a Hardware Description Language based on the guarded action model of concurrency. It has an elegant semantics, which makes it well suited for formal reasoning. However, little work has been done on applying mechanized verification to BSV designs. We present a prototype embedding of BSV in the SAL model checker, for which translation from BSV to SAL is verified with the PVS theorem prover. We demonstrate the practical applicability of our approach by establishing deadlock freedom, mutual exclusion and liveness for a BSV implementation of Peterson's protocol.

Keywords: Bluespec, Embedding, Model Checker, Theorem Prover, Hardware Description Language.

1 Introduction

Bluespec SystemVerilog (BSV) [Nik04] is a language for high-level hardware design, and produces hardware that's competitive with hand-written RTL in terms of time and area for most designs [WNRD04,Nik04]. It developed from research using Term Rewriting Systems (TRS) to produce hardware specifications that could be synthesized and formally verified [AS99]; in common with TRS, BSV is semantically elegant, which makes it well suited for formal reasoning [ADK08]. To date, however, only one investigation has been made into the mechanized verification of BSV designs [SS08] which presented a translation schema from a subset of BSV into Promela, the input language of the SPIN model checker [Hol03].

In this paper we present a strategy for embedding a subset of BSV in the SAL model checker [dMOR+04], where the BSV-to-SAL translation can be verified with a simple proof in the PVS theorem prover [ORS92]. The subset of BSV that we embed includes a number of language features not covered in [SS08]; namely the instantiation of non-trivial nested modules, methods with arbitrary side-effects and

[1] Email:dar@cs.man.ac.uk
[2] Email:dlester@cs.man.ac.uk

return values, and rule composition from methods with arbitrary side-effects and return values. In this work we offer a translation schema: we have yet to build a BSV-to-SAL compiler, which will be developed at a later stage in tandem with our related research into automated abstraction of BSV designs.

As an example of our approach, we embed a BSV implementation of Peterson's protocol in the SAL language; we then verify the BSV-to-SAL translation with PVS and establish deadlock freedom, mutual exclusion and liveness with the SAL model checker.

1.1 The Challenges of Embedding BSV in SAL

BSV uses guarded actions to express concurrency, and so is similar to several languages that were developed for the formal study of concurrency, including UNITY [CM88], TLA [Lam02], Promela and the SAL language. There is a rich body of literature on the use of model-checkers for verifying systems expressed in these languages. However, BVS is a more complex language in some respects, being intended for hardware design rather than abstract specification:

(i) **Complex encapsulation of state.** BSV has a 'module' construct, which allows elements of state to be associated with 'rules' (guarded actions) that spontaneously transform the state, and 'methods' that can be called by other modules to return a value from the state, and possibly transform it in the process. Rules can be composed from the methods provided by other modules; in this way, the execution of a rule in one module can alter the state in another.

(ii) **Widespread presence of data paths.** Model-checking is a very useful tool for efficiently verifying concurrent systems, but can be confounded by the presence of data paths. Data paths can have very large state spaces, which can cause state space explosions in model-checkers; for this reason, model-checking has been more widely used to verify control-based systems. When abstract specification languages such as UNITY are used for hardware verification, a model can be constructed that only specifies the control-based components of a design (when the data path is irrelevant to the properties being verified), or specifies some abstract interpretation of the data path. With a direct embedding of BSV, however, the hardware design *is* the specification; we can't chose to omit data paths from our formal model. Because of this, we must find a tractable way to abstract away from data path complexity.

Therefore, in order to produce a usable general purpose embedding of BSV in a formal guarded action language, we must first bridge the *semantic gap* by expressing the constructs of BSV with the more limited constructs of our target specification language, preferably in such a way that equivalence between the two can be easily established. When we have achieved this, we must also bridge the *abstraction gap*, to obtain abstract specifications that can be efficiently verified [CDH+00].

In this paper, we concentrate on the first of these two steps. We provide an embedding strategy that maps BSV expressions to SAL expressions that can be efficiently model checked and verify the translation with PVS. In further work, we plan to investigate the use of automatic abstraction in order to bridge the abstraction gap.

After introducing BSV in section 2, we introduce the SAL language in section 3, along with our strategy for embedding BSV in SAL. Our embedding produces SAL specifications that can be efficiently model-checked but bear little resemblance to the BSV code they represent, and we discuss the problems that this can create.

In section 4 we introduce our technique for proving equivalence between BSV designs and their SAL specifications. We develop two embeddings of BSV in PVS; a 'monadic' embedding that closely resembles BSV and a 'primitive' embedding that shares the same semantics but has a syntax similar to our SAL embedding. We then show that semantic equivalence between instances of the monadic and primitive embeddings can easily be established with the powerful proof strategies of PVS.

In section 5 we report on an unsuccessful attempt to implement our 'monadic' embedding strategy in SAL, in order to produce high-level specifications that could be directly model-checked.

We demonstrate the practical applicability of our approach in section 6 by embedding a BSV implementation of Peterson's protocol, for which we verify BSV-to-SAL translation with PVS and establish deadlock freedom, mutual exclusion and liveness with the SAL model-checker. We provide code listings for this example in appendix A.

In sections 7 and 8 we discuss related work, conclude the paper and look forward to further work.

2 Bluespec SystemVerilog

BSV is a language for high-level hardware design, and produces hardware that's competitive with hand-written RTL in terms of time and area for most designs [WNRD04,Nik04]. A comprehensive tutorial can be found online at [Blu].

In BSV, hardware is specified in terms of 'modules' that associate elements of state with 'rules' (guarded actions) that transform the state and 'methods' that can be called by other modules to return values from the state, and possibly transform it in the process.

A simple example of a module is `Reg`, which specifies a register with one element of state, no internal rules and two methods, `_read` and `_write`. Other modules can create instances of `Reg`, and use the methods `_read` and `_write` in their own rules and methods. For example, consider the following rule that uses two registers, `request` and `acknowledge`, both of which hold elements of type `Bool` [3]:

```
rule request_rl (!(request._read || acknowledge._read));
  request._write(True);
endrule
```

The rule has a guard, which is a predicate on the state of the registers, and an 'action', which transforms the state of the `request` register.

In general, a rule has the form:

[3] BSV users will notice that we've de-sugared the method calls for Regs; we do this throughout the paper to simplify the semantics, and also to emphasize the use of methods inside rules.

```
rule my_rule (rl_guard);
  statement_1;
  statement_2;
  ...
endrule
```

The statements in the rule body are individual actions that transform the state in some way. The set of statements are applied *in parallel* to the state; each statement is applied to the state as it was immediately before the rule was activated, so that the changes made by `statement_1` aren't seen by `statement_2`, or any other statement. The BSV compiler ensures that the statements in a rule don't conflict with each other by simultaneously attempting to write to the same elements of state.

2.1 Term Rewriting System (TRS) Semantics

The behavior of a module can be understood in terms of a simple semantics called Term Rewriting System (TRS) semantics, also called 'one-rule-at-a-time' semantics. In TRS, a module evolves from a given state by choosing *one* rule for which the guard is true and applying the associated action to transform the state; if more than one guard is true, a nondeterministic choice is made. When actual hardware is generated from BSV designs, a number of optimizations are applied (for example, a clock is introduced and multiple rules are executed per clock cycle) but the behavior is guaranteed to comply with the TRS semantics.

3 An Embedding of BSV in the SAL Language

The TRS semantics (§ 2.1) is conveniently similar to the semantics of the SAL language. In SAL, a non-deterministic state machine is defined with an initial state and a transition relation composed of a number of guarded actions. The transition relation has the form:

```
TRANSITION
[
   guarded_action_1 : guard_1 --> action_1
[]
   guarded_action_2 : guard_2 --> action_2
[]
   ...
]
```

The guards are predicates on the state of the system and the actions are updates that change the state in some way. The system evolves from a given state by choosing one guarded action for which the guard is true and executing the action; as with BSV, when more than one guard is true for a given state, a nondeterministic choice is made.

How can we translate BSV rules to SAL guarded actions? Consider the following BSV rule, which comes from our example in section 6:

```
rule p_critical (pcp._read == Critical && fifo.notFull);
```

```
  fifo.enq (True);
  pcp._write (Sleeping);
  turn._write (False);
endrule
```

The guard of `p_critical` is a predicate over the state a register called `pcp` and a single-element FIFO called `fifo`. The action of `p_critical` changes the state of `fifo` and registers `pcp` and `turn`. Before we embed this rule in SAL, we need to define the states of `pcp`, `turn` and `fifo`. We can specify the states of generic registers and one-element FIFOs with SAL contexts:

```
Reg {T : type} : CONTEXT = BEGIN

  State : type = [# data : T #];

END

FIFOF1 {T : type} : CONTEXT = BEGIN

  State : type = [# notFull : bool, notEmpty : bool, data : T #];

END
```

We can then instantiate these contexts in our Peterson specification:

```
PC: TYPE = {Sleeping, Trying, Critical};

pcp  : Reg{PC}!State,
pcq  : Reg{PC}!State,
turn : Reg{bool}!State,
fifo : FIFOF1{bool}!State
```

We've added another register here, called `pcq`; we don't refer to it in the rule `p_critical` but it gets used in the Peterson example so we include it here for completeness. Registers `pcp` and `pcq` hold elements of type `PC` ('program counter').

Now that we have a state composed of records, we can express our rule as a guarded action in the `TRANSITION` relation by expanding all method calls to record updates and functions over record fields:

```
TRANSITION
[
   p_critical : pcp.data = Critical and fifo.notFull
                --> fifo' = (# data     := true,
                               notFull  := false,
                               notEmpty := true     #);
                    pcp'  = (# data     := Sleeping #);
                    turn' = (# data     := false    #)
[]
   ...
]
```

This is a straightforward way to express rules, and it produces SAL specifications that can be efficiently model-checked. When rules call methods that are entirely side-effecting (like the register method `_write` or the FIFO method `enq`) we can expand the method in-place to a series of record updates. Side-effect free methods (like the register method `_read`) can be expanded to pure functions on the state. Methods that produce a side-effect and also return a value can be decomposed into a pair of methods; one entirely side-effecting and one side-effect free. In this way, we can embed rules composed of methods with arbitrary side-effects and return values.

This approach seems quite simple, but can create problems. If we express a rule by fully expanding all of its method calls we expose its full complexity; BSV provides the module and method constructs to avoid just this. If we specify more complex modules in this way (for example, modules where rules and methods call methods that themselves call methods, all returning values and producing side-effects) we end up with long-winded specifications that bear little resemblance to the BSV code they represent. If we assume that the process of translating from BSV to SAL is not formally verified, it becomes difficult to provide assurance that the translation is accurate; this makes it difficult to rule out false positives when a property is proved, or conversely false negatives when a property is disproved.

Readers who are familiar with functional programming might ask: "Why don't you produce more abstraction specifications by implementing monads in the SAL language?". We tried this, and were unsuccessful; we discuss our failed attempt in section 5.

4 Verifying BSV-to-SAL Translation with PVS

We saw in the previous section that our strategy for embedding BSV in SAL produces specifications that can be efficiently model-checked, but bear little resemblance to the BSV code they represent. This can be problematic, because errors in the translation can go unnoticed.

Our solution is to verify individual translations with the PVS theorem prover. We develop two embeddings of BSV in PVS; a 'monadic' embedding that closely resembles BSV and a 'primitive' embedding that shares the same semantics but has a syntax similar to our SAL embedding. We then prove semantic equivalence between instances of the monadic and primitive embeddings quite easily with the powerful proof strategies of PVS.

Our verification strategy would fit into an automatic BSV-to-SAL translation process as follows. A program for translating BSV to SAL would parse BSV to produce an abstract syntax tree (call it the BSV AST) and transform this to an AST that represents the SAL embedding (the SAL AST) which could then be used to generate the SAL specification. In this system, the BSV AST could be converted to an instance of our monadic PVS embedding, and the SAL AST to an instance of our primitive PVS embedding; these two specifications could then be proven semantically equivalent, in order to verify the process of generating the SAL AST from the BSV AST.

To give you an idea of how the two PVS embeddings align with BSV and the SAL embedding, figure 1 shows the BSV rule `p_critical` along with its embedding

Fig. 1 A BSV rule, together with its embeddings in PVS and SAL

The BSV rule:

```
rule p_critical (pcp._read == Critical && fifo.notFull);
  fifo.enq (True);
  pcp._write (Sleeping);
  turn._write (False);
endrule
```

A monadic embedding in PVS:

```
p_critical = rule (pcp‘read = Critical ∧ fifo‘notFull)
                  (fifo‘enq (TRUE)       ≫
                   pcp‘write (Sleeping)  ≫
                   turn‘write (FALSE))
```

A primitive embedding in PVS:

```
p_critical_primitive (pre, post : Peterson) : bool
  = pre‘pcp‘data = Critical ∧ pre‘fifo‘notFull
    ∧ post = pre WITH [(fifo)  := (# data     := TRUE,
                                     notFull  := FALSE,
                                     notEmpty := TRUE  #),
                       (pcp)   := (# data     := Sleeping #),
                       (turn)  := (# data     := FALSE  #)]
```

A primitive embedding in SAL:

```
p_critical : pcp.data = Critical and fifo.notFull
             --> fifo' = (# data     := true,
                            notFull  := false,
                            notEmpty := true     #);
                 pcp'  = (# data     := Sleeping #);
                 turn' = (# data     := false    #)
```

in PVS and SAL. Notice the strong syntactic resemblance between the BSV rule and its monadic PVS embedding, and likewise for the primitive PVS embedding and the SAL embedding.

4.1 TRS Semantics in PVS

In PVS, we describe a rule as a predicate over pairs of states:

```
my_rule (pre, post: Module_State): bool = rl_guard (pre) ∧ post = rl_action (pre)
```

Here, rl_guard and rl_action are the PVS representations of the rule's guard and action, and Module_State is the PVS representation of the module's state. We can express the TRS semantics of a module by composing its rules together:

```
my_TRS_module (pre, post: Module_State): bool  = rule_1 (pre, post)
                                               ∨ rule_2 (pre, post)
                                               ∨ ...
```

my_TRS_module relates pre to post if any rule relates them when applied in isolation: we have a nondeterministic, one-rule-at-a-time semantics.

As with SAL, we can express the states of nested modules with records. We can then express the state of the top-level module as a record of records. In the case of the Peterson example, we have:

```
Peterson : TYPE = [# pcp  : Reg[PC],
                    pcq  : Reg[PC],
                    turn : Reg[bool],
                    fifo : FIFOF1[bool]#]
```

4.2 A Primitive Embedding of BSV in PVS

Now that we have a state composed of records, we can express rules in the same way that we did in our SAL embedding. For the rule `p_critical`, we have:

```
p_critical_primitive (pre, post : Peterson) : bool
  = pre`pcp`data = Critical ∧ pre`fifo`notFull
    ∧ post = pre WITH [(fifo)  := (# data     := TRUE,
                                     notFull  := FALSE,
                                     notEmpty := TRUE  #),
                       (pcp)   := (# data     := Sleeping #),
                       (turn)  := (# data     := FALSE  #)]
```

We call this embedding strategy our 'primitive' embedding.

4.3 A Monadic Embedding of BSV in PVS

Our second PVS embedding uses monads to produce specifications that are syntactically similar to the BSV code they represent. Monads are constructs for conveniently representing state in pure functional languages; a comprehensive introduction to the use of monads in functional programming can be found in [Bir98]. Before getting into the details of our embedding, let's take a look at the result. This is our monadic embedding of the rule `p_critical`:

```
p_critical = rule (pcp`read = Critical ∧ fifo`notFull)
                  (fifo`enq (TRUE)       ≫
                   pcp`write (Sleeping)  ≫
                   turn`write (FALSE))
```

In contrast to the primitive embedding, the complexity of methods and actions is factored out into monads. This yields rule specifications that are syntactically similar to the BSV rules that they represent.

Monadic rules can be expanded to expressions involving only record updates and functions over record fields. p_critical has type [[Peterson, Peterson] → bool] and is, in fact, extensionally equivalent to p_critical_primitive; this can be proven with the proof strategies `(apply-extensionality)` and `(grind)`.

4.3.1 A State Monad in PVS

So, how can we use monads to express rules and methods without expanding their full complexity in-place? Consider the action of a rule, which is composed of a sequence of statements:

```
rule my_rule (rl_guard);
    statement_1;
    statement_2;
    ...
endrule
```

Statements can either be methods (e.g. "`pcp._write(Critical);`") or let bindings (e.g. "`let x = pcp._read;`"). We'll defer our treatment of let expressions for the moment and assume that all statements are methods.

The meaning we want to capture for the entire statement block is that an initial state is transformed independently by the individual statements, and the changes made by each are combined to give a new state. We can actually achieve the same effect by applying the statements sequentially; we can apply the first statement to get a partially updated state, then apply the second statement to update this new state, and so on. This is possible because the statements are conflict free; no two statements will update the same element of state, so we don't need to worry about later statements over-writing the updates made by earlier statements. However, each statement needs access to the initial state, as earlier statements might update elements of state that later statements need to read. This suggests that we specify statements as instances of the type:

$$\text{Monad: TYPE} = [[S, S] \to [A, S]]$$

'S' is the type of the module's state; for rules in the Peterson module it's the 'Peterson' type from § 4.1. 'A' is the type of some return value (when methods are used a statements – for example "`pcp._write(Critical);`" – the return type must be Null). Instances of Monad take two copies of the module state (representing the initial state and a partially updated state) and return a value and a new instance of the state, with any additional updates added to those of the partially updated state.

4.3.2 Monad Transformers

We've seen that methods can be expressed as instances of the type Monad, which is essentially a function to transform state. When a method is used as a statement in a rule, it must operate on the state of the module containing the rule. For a rule in the Peterson module, all rule statements must operate on the Peterson state; they will have the type signature:

$$\text{Monad}\,[\text{Peterson, Null}] = [[\text{Peterson, Peterson}] \to [\text{Null, Peterson}]]$$

Remember that when methods are used as statements, their return type must be Null.

Consider again the monadic specification of rule `p_critical` form the Peterson module:

```
p_critical = rule (pcp‘read = Critical ∧ fifo‘notFull)
                  (fifo‘enq (TRUE)        ≫
                   pcp‘write (Sleeping)  ≫
                   turn‘write (FALSE))
```

The three statements in the action (fifo‘enq (TRUE), pcp‘write (Sleeping) and turn‘write (FALSE))) have type Monad[Peterson, Null]; that is to say, they must operate on instances of the Peterson state, despite the fact that they only influence individual registers or FIFOs within that state. We can achieve this by specifying the generic register and FIFO methods as monads that act on types Reg[T] and FIFOF1[T] respectively, and lifting them to monads that act on type Peterson with monad transformers. For the register methods, we have:

```
read : Monad[Reg[T], T]
  = λ (init, updates: Reg[T]): (init‘data, updates)

write (d: T) : Monad [Reg[T], Null]
   = λ (init, updates : Reg[T]) : (null, (# data := d #))

Transformer : TYPE = [Monad [R, A] → Monad [S, A]]

transform (get_R: [S → R], update_R: [[S, R] → S]): Transformer
  = λ (m: Monad [R, A]) (init, updates: S) :
       LET (data, new_updates) = m (get_R (init), get_R (updates))
       IN  (data, update_R (updates, new_updates))
```

A function of type Transformer takes a monad over state R and lifts it to become a monad over state S. We can use the ‘transform’ function to produce a Transformer that lifts the generic register functions (read and write) to operate on our Peterson state. For example, we can do this for the pcp register:

```
pcpT : Transformer [Reg[PC], Peterson, T] = transform (get_pcp, update_pcp)
```

where get_pcp and update_pcp access and update the pcp field of a Peterson record. We can then define our lifted monads pcp‘read and pcp‘write:

```
pcp : [# read  : Monad[Peterson, PC],
         write : [PC → Monad[Peterson, Null]] #]
   = (# read  := pcpT[PC] (read),
         write := λ (x: PC): pcpT[Null](write (x)) #)
```

4.3.3 Monad Connectors

We can compose monadic statements to form statement blocks with the standard monad connectors [Bir98], with bind (≫=) adapted to accept a pair of input states:

```
≫= (m: Monad[S, A], k: [A → Monad [S, B]]): Monad [S, B]
  = λ (init, updates: S) : LET (data, new_updates) = m (init, updates)
                           IN  k (data) (init, new_updates)

≫ (m: Monad [S, A], n : Monad [S, B]) : Monad[S, B]
  = m ≫= λ (data : A) :  n
```

We can use the bind function to express 'let' statements. For example, the statement block:

```
let x = pcp._read; pcq._write (x)
```

becomes:

pcp‘read ≫= λ x : pcq‘write (x)

or simply:

pcp‘read ≫= pcq‘write

We can use monads directly in the guard if we overload the standard boolean and equality operators with functions over monads. For example:

$$\wedge\ (\text{m, n: Monad}[S, \text{bool}]):\ \text{Monad}[S, \text{bool}]$$
$$= \lambda\ (\text{init, updates : S}):\ \text{LET}\ b_1 = (\text{m (init, updates)})\text{‘1,}$$
$$b_2 = (\text{n (init, updates)})\text{‘1}$$
$$\text{IN}\quad (b_1 \wedge b_2, \text{ updates})$$

When we have monadic specifications of a rule's guard and body, we can form a 'rule' that is a predicate over pairs of states with the function:

rule (guard: Monad [S, bool]) (action: Monad [S, Null]) (pre, post: S): bool =
(guard (pre, pre))‘1 ∧ post = (action (pre, pre))‘2

5 An Unsuccessful Monadic Embedding in the SAL Language

We saw in section 4 that monads can be used to produce specifications that bear a strong syntactic resemblance to the BSV code they represent, so that compilation from BSV to the target language is reduced to translation of the concrete syntax. If we could implement monads in the SAL language and model check monadic specifications directly, there would be no need for us to use PVS. We tried to do this, and were unfortunately unsuccessful.

SAL has a very expressive language that allowed us to implement our monadic embedding strategy. To give you an idea of the concrete syntax, this is our embedding of the `p_critical` rule:

```
p_critical = (pcp_read = Critical and fifo_notFull,
              bool_bool!seq (fifo.enq (true),
              PC_bool!seq   (pcp.write  (Sleeping),
                             turn.write (false))))
```

The rule is expressed as a pair of monads representing the rule's guard and action. SAL doesn't support type inference or the definition of infix operators, so our monadic specifications were a little less concise in SAL compared to PVS.

With rules expressed as monads, the transition relation reduced to 'boilerplate', where the rules' monads are applied to the current state:

```
TRANSITION
[
```

```
    ...
[]
    p_critical : p_critical.1 --> s' = bool_!exec (p_critical.2, s)
[]
    q_critical : q_critical.1 --> s' = bool_!exec (q_critical.2, s)
[]
    ...
]
```

The various SAL back ends (deadlock checker, symbolic model-checker etc.) would happily accept our monadic specifications, but struggled to reduce them to BDD form.

6 Experimental Results: A BSV implementation of Peterson's Protocol

To demonstrate the practical applicability of our approach, we verified a BSV design for a 2-process version of Peterson's protocol [dM04]. We hand-embedded the BSV design in SAL, verified the BSV-to-SAL translation with PVS and used the SAL model-checker to establish deadlock freedom, mutual exclusion and liveness. The BSV design amounts to just over 50 lines of code; extracts of the design and its embeddings in SAL and PVS can be found in appendix A.

In our implementation of Peterson's protocol, two processes simultaneously attempt to enter a 'critical mode' that can only be entered 'safely' by one process at a time. Each process has a program counter that can be in one of three modes: Sleeping, Trying and Critical. A process starts in the Sleeping mode, then wakes up, entering the Trying mode and attempts to enter the Critical mode. Each process can read the program counter of the other, and will not enter Critical mode if the other process is Critical. To ensure that the protocol is fair (a Trying process will always eventually gain access to the Critical mode) there is a boolean 'turn' flag. The state of the turn flag gives priority to one of the two processes; when a process enters the Critical mode, it sets the turn flag to give the other process priority.

In our BSV design, the two program counters and the turn flag are all held in registers. To add some complexity to the example, we also have a one element FIFO that the processes write to when they are in the Critical mode.

We established deadlock freedom for our SAL embedding with the SAL deadlock checker. We stated mutual exclusion and fairness with LTL assertions in SAL. The mutual exclusion theorem states that the two processes will never be in critical mode at the same time:

```
mutex: THEOREM
  System |- G(NOT(pcp.data = Critical AND pcq.data = Critical))
```

The liveness theorem states that a Trying process will always (eventually) gain access to the Critical mode:

```
liveness: THEOREM
  System |- (G(F(pcp.data = Trying)) => G(F(pcp.data = Critical)))
```

```
and (G(F(pcq.data = Trying)) => G(F(pcq.data = Critical)))
```

These two theorems were proved by the SAL symbolic model checker in 0.1 and 0.2 seconds respectively.

To verify our BSV-to-SAL translation, we produced a primitive embedding and a monadic embedding in PVS (extracts are given in appendix A) and proved them equivalent. The proof required only two strategies; `(apply-extensionality)` followed by `(grind)`. For readers who aren't familiar with proof in PVS, the entire user input for this proof is `M-x prove`, `TAB E`, `TAB G`. The proof re-runs in 0.45 seconds.

7 Related Work

There has been one previous investigation into the mechanized verification of BSV designs. In [SS08] Singh and Shulka present a translation schema from a subset of BSV into Promela, the input language of the SPIN model-checker. For this subset, they use SPIN to verify that rule scheduling is a valid refinement and demonstrate the use of SPIN for verifying LTL assertions. The subset of BSV that they consider is similar to ours, but does not address the instantiation of non-trivial nested modules, methods with arbitrary side-effects and return values, or rule composition from methods with arbitrary side-effects and return values. They translate directly to Promela in a similar way to our 'primitive' SAL translation.

In the wider literature, our work sits somewhere between theorem prover embeddings of formal guarded action languages and model-checking of main-stream, informal hardware and software design languages. There are a number of theorem-prover embeddings of the well-known guarded action languages. In [Pau00], Paulson embeds UNITY in the Isabelle [Pau94] higher order theorem prover. He uses a set-based formalism in his embedding that allows efficient leverage of Isabelle's proof tactics., and demonstrates efficient verification of a UNITY specification of Peterson's protocol. In [CDLM08], Chaudhuri *et. al.* present a proof environment for TLA$^+$ [Lam02]. As with Paulson, they use automated deduction to lessen the proof burden. They first attempt to prove theorems with the Zenon first-order tableau prover, and if this is unsuccessful, call the Isabelle higher order theorem prover. In [SSA01] Stoy *et. al.* provide an embedding of TLA in PVS, along with a method of proving temporal logic assertions about TLA state machines by deductive reasoning about properties that are invariant for reachable states under the transition relation. They use this embedding to verify a TLA specification of Cachet, an adaptive cache-coherence protocol.

Because languages like UNITY and TLA+ were developed for specification rather than design, there is less emphasis on the use of abstraction for state-space reduction, which will be necessary in a general-purpose verification tool for the BSV language. There have been a number of investigations into the use of abstraction and model-checking to verify programs expressed in hardware and software design languages. Some of the larger projects include the Java model-checking environments Bandera [CDH$^+$00] and Java PathFinder [VHBP00], and the C model-checkers FeaVer [HS00] and SLAM [BR02]. All of these tools employ some combination of static analysis, slicing, abstract interpretation and specialization to reduce the

state space. The issue of semantic mismatch between object language and specification language is tackled in different ways: Bandera simplifies Java bytecode and translates the result to the input languages of SPIN and SMV; PathFinder provides a custom model-checker for Java bytecode; FeaVer uses user-defined lookup tables to translate C to Promela on a line-by-line basis; SLAM translates C programs to equivalent boolean programs.

Monads have been used several times before to express state in theorem prover specification languages. In [Fil03], Filliâtre uses monads to specify non-functional programs in Coq. Monads are used in [KM02] for the specification of BDD algorithms in Isabelle, and in [BKH+08] to express imperative programs in Isabelle.

8 Conclusions and Further Work

We have presented a strategy for embedding a subset of Bluespec SystemVerilog in the SAL language, yielding SAL specifications that can be efficiently model-checked. We also presented a technique for verifying the BSV-to-SAL translation with the PVS theorem prover, by producing two embeddings in PVS, with one equivalent to the BSV design and one equivalent to its SAL specification, modulo differences in concrete syntax.

In further work, we plan to extend our approach with automated abstraction. We also plan to extend the subset of BSV that can be embedded to include static elaboration and nested modules with internal rules. We hope to embed the latter by promoting nested rules to rules in the top-level module with monad transformers.

In related work that's currently under consideration for publication elsewhere, we have develped a technique for directly verifying instances of our monadic PVS specifications. We hope to combine our work in PVS and SAL by producing a translator that compiles BSV designs to both languages, giving users the benefits of both tools. We intend to road-test our combined system by verifying a transaction level BSV model of the communications network of the SpiNNaker super-computer [FTB06], which we have previously specified and verified using Haskell [RL09]; verification of this system was the initial motivation for our work.

References

[ADK08] Arvind, Nirav Dave, and Michael Katelman. Getting formal verification into design flow. In *Proc. FM '08*, 2008.

[AS99] Arvind and Xiaowei Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3):36–46, 1999.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2 edition, 1998.

[BKH+08] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with Isabelle/HOL. In *Proc. TPHOLs '08*, 2008.

[Blu] Bluespec Inc. Bluespec SystemVerilog online training. http://www.demosondemand.com/dod/proddemos/vendors/pd_bluespec.aspx.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. POPL '02*, 2002.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE '00*, 2000.

[CDLM08] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. A TLA+ Proof System. In *Proc. KEAPPA*, 2008.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison Wesley Publishing Company, Inc., Reading, Massachusetts, 1988.

[dM04] Leonardo de Moura. SAL: Tutorial. Technical report, SRI International, November 2004.

[dMOR+04] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In *Proc. CAV'04*, 2004.

[Fil03] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *J. Funct. Program.*, 13(4), 2003.

[FTB06] Steve Furber, Steve Temple, and Andrew Brown. High performance computing for systems of spiking neurons. In *Proc. AISB '06*, 2006.

[Hol03] Gerard Holzmann. *SPIN model checker, the: primer and reference manual*. Addison-Wesley, 2003.

[HS00] Gerard J. Holzmann and Margaret H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.

[KM02] Sava Krstic and John Matthews. Verifying BDD algorithms through monadic interpretation. In *Proc. VMCAI '02*, 2002.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Nik04] R. Nikhil. Bluespec SystemVerilog: efficient, correct RTL from high level specifications. In *Proc. MEMOCODE '04*, 2004.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Proc. CADE-11*, 1992.

[Pau94] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*. LNCS. 1994.

[Pau00] Lawrence C. Paulson. Mechanizing UNITY in Isabelle. *ACM Trans. Comput. Logic*, 2000.

[RL09] Dominic Richards and David Lester. Concurrent functions: A system for the verification of networks-on-chip. *Proc. HFL'09*, 2009.

[SS08] Gaurav Singh and Sandeep K. Shukla. Verifying compiler based refinement of Bluespec specifications using the SPIN model checker. In *Proc. SPIN '08*, Berlin, Heidelberg, 2008.

[SSA01] Joseph E. Stoy, Xiaowei Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In *Proc. FME '01*, 2001.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proc. ASE '00*, 2000.

[WNRD04] Weng-Fai Wong, R.S. Nikhil, D.L. Rosenband, and N. Dave. High-level synthesis: an essential ingredient for designing complex ASICs. In *Proc. CAD04*, 2004.

A Extracts from a BSV Specification of Peterson's Protocol and its Specifications in PVS and SAL

Fig. A.1 Extracts from a BSV Specification of Peterson's Protocol

```
typedef enum {Sleeping, Trying, Critical} PC deriving (Bits, Eq);

Reg#(PC) pcp <- mkReg(Sleeping); Reg#(PC) pcq <- mkReg(Sleeping);
Reg#(Bool) turn <- mkReg(True);  FIFOF#(Bool) fifo <- mkFIFOF1;

rule wake_p (pcp._read == Sleeping);
  pcp._write (Trying);
  turn._write (False);
endrule

rule wake_q (pcq._read == Sleeping);
  pcq._write (Trying);
  turn._write (True);
endrule

rule grant_p (pcp._read == Trying
              && (turn._read || pcq._read == Sleeping));
  pcp._write (Critical);
endrule

rule grant_q (pcq._read == Trying
              && (!turn._read || pcp._read == Sleeping));
  pcq._write (Critical);
endrule

rule p_critical (pcp._read == Critical && fifo.notFull);
  fifo.enq (True);
  pcp._write (Sleeping);
  turn._write (False);
endrule

rule q_critical (pcq._read == Critical && fifo.notFull);
  fifo.enq (False);
  pcq._write (Sleeping);
  turn._write (True);
endrule

rule read_fifo (fifo.notEmpty);
  fifo.deq;
endrule
```

Fig. A.2 Extracts from the PVS Embeddings

```
wake_p    = rule (pcp‘read = Sleeping)
                 (pcp‘write (Trying) ≫
                  turn‘write (FALSE))

grant_p   = rule (pcp‘read = Trying ∧ (turn‘read ∨ pcq‘read = Sleeping))
                 (pcp‘write (Critical))

p_critical = rule (pcp‘read = Critical ∧ fifo‘notFull)
                  (fifo‘enq (TRUE)       ≫
                   pcp‘write (Sleeping)  ≫
                   turn‘write (FALSE))

...

wake_p_ primitive (pre, post) : bool
  = pre‘pcp‘data = Sleeping
    ∧ post = pre WITH [(pcp)  := (# data := Trying #),
                       (turn) := (# data := FALSE #)]

grant_p_ primitive(pre, post): bool
  = pre‘pcp‘data = Trying ∧ (pre‘turn‘data ∨ pre‘pcq‘data = Sleeping)
    ∧ post = pre WITH [(pcp) := (# data := Critical #)]

p_critical_primitive (pre, post) : bool
  = pre‘pcp‘data = Critical ∧ pre‘fifo‘notFull
     ∧ post = pre WITH [(fifo)  := (# data     := TRUE,
                                      notFull  := FALSE,
                                      notEmpty := TRUE  #),
                        (pcp)   := (# data     := Sleeping #),
                        (turn)  := (# data     := FALSE  #)]

...

transitions (pre, post) : bool = wake_p    (pre, post) ∨
                                 grant_p   (pre, post) ∨
                                 p_critical (pre, post) ∨ ...

primitive_transitions (pre,post) : bool = wake_p_primitive    (pre, post) ∨
                                          grant_p_primitive   (pre, post) ∨
                                          p_critical_primitive (pre, post) ∨ ...

transitions_lem : LEMMA transitions = primitive_transitions
  % Proven with (apply-extensionality) and (grind)
```

Fig. A.3 The Transition Relation of the SAL Embedding

```
TRANSITION
[  wake_p     : pcp.data = Sleeping
                --> pcp'  = (# data := Trying #);
                    turn' = (# data := false #)
[]
   wake_q     : pcq.data = Sleeping
                --> pcq'  = (# data := Trying #);
                    turn' = (# data := true #)
[]
   grant_p    : pcp.data = Trying
                and (turn.data or (pcq.data = Sleeping))
                --> pcp' = (# data := Critical #)
[]
   grant_q    : pcq.data = Trying
                and (not turn.data or (pcp.data = Sleeping))
                --> pcq' = (# data := Critical #)
[]
   p_critical : pcp.data = Critical and fifo.notFull
                --> fifo' = (# data     := true,
                               notFull  := false,
                               notEmpty := true     #);
                    pcp'  = (# data     := Sleeping #);
                    turn' = (# data     := false    #)
[]
   q_critical : pcq.data = Critical and fifo.notFull
                --> fifo' = (# data := true,
                               notFull := false,
                               notEmpty := true #);
                    pcq'  = (# data := Sleeping #);
                    turn' = (# data := true #)
[]
   read_fifo  : fifo.notEmpty
                --> fifo' = (# data := fifo.data,
                               notFull := true,
                               notEmpty := false #)
] END;
```

Introducing Kind #: The Numeric Type System of Bluespec SystemVerilog

Ravi Nanavati

December 8, 2009

Hardware designers routinely develop and use circuits that are polymorphic over key hardware parameters and sizes. For example, a FIFO might be polymorphic over the width of its elements or a register file might permit different numbers of read and write ports. As aa consequence, any practical hardware design language must have support for numeric polymorphism. Current, mainstream hardware design languages have weak numeric type systems that often lead to size-mismatch bugs.

Bluespec SystemVerilog (BSV) is a high-level, statically typed hardware design language based on functional programming technology. BSV's type system is based on Haskell's and includes Haskell-style typeclasses with several familiar extensions (most notably multi-parameter typeclasses with functional dependencies). To support numeric types, BSV adds a new primitive kind (#), whose members are the natural numbers, and implements numeric type relations as built-in numeric typeclasses (e.g. `Add`, `Mul`) with functional dependencies. BSV also supports numeric type functions (e.g. `TAdd`, `TMul`). These type functions were introduced as a way of preventing the propagation of some vacuous numeric constraints, but they turn out to have several other uses (e.g. capturing numeric constraints in syntactic contexts where typeclass constraints are not permitted).

The numeric type extensions described so far have had several important successes. Most notably, they are the infrastructure underlying BSV's `Bits` typeclass—a typeclass that converts abstract types to and from canonical bit-level representations. However, additional features are required to create a robust and usable numeric type system. One issue is that structural unification is not an appropriate way to check the equality of numeric types. There are many numeric equalities that cannot be proved structurally (e.g. `(TMul 2 a)=(TAdd a a)`). Even worse, there are situations where structural unification makes incorrect deductions. To resolve these problems, BSV's numeric type system replaces structural unification with an approach based on a numeric equality typeclass, `NumEq`. Numeric equality constraints are gathered during ordinary unification and converted into typeclass constraints. Either these typeclass constraints are solved by BSV's type-checker (and a numeric equality witness is created for future stages of the compiler) or type checking fails with a suitable error message about the problem.

Unfortunately, improved numeric equality is not enough to resolve some of the usability problems of the system described so far. One problem is handling local non-syntactic numeric equalities when type-checking definitions with non-trivial numeric constraints. Theoretically, it is possible to recursively deduce these equalities using the given numeric constraints. In practice, however, this results in exponential type-checking time in some important cases. Instead, it is more practical to eliminate any determined numeric variables in a definition's numeric constraints. When this can be done, any local equalities become purely syntactic and, therefore, easier to handle. In some cases, eliminating a determined numeric type variable might require introducing a subtraction. Since BSV's numeric types are required to be non-negative, this is only permissible if the result of that subtraction will be non-negative. To ensure this, BSV provides a second numeric constraint typeclass, `AtMost`, that can be introduced to guard a subtraction.

The `AtMost` typeclass has several uses beyond enabling more aggressive elimination of determined numeric variables. It can help improve numeric type-checking error messages because reasoning involving `AtMost` more closely corresponds to the reasoning designers do when thinking about the numeric constraints related to their circuits. It is the natural place to capture greater-than and less-than numeric reasoning involving numeric type functions and constants. The `AtMost` typeclass is also a logical place to insert special-case numeric type-checking extensions that handle transitive greater-than and less-than reasoning (which is required to support some kinds of obvious numeric reasoning). Finally, the `AtMosta` typeclass captures easily checked witnesses for all of these more advanced kinds of numeric reasoning.

BSV has a complex numeric type system with several different moving parts. This complexity is driven by the different kinds of numeric reasoning required for correctness and usability. That is why we believe that most of the complexity in BSV's numeric type system is necessary. We also believe that this system is a compelling demonstration of the power, flexibility and usability achievable with language-level numeric types.

Modular Refinement for Bluespec Hardware Designs

Michael Katelman[1] Nirav Dave[2]

[1] University of Illinois at Urbana-Champaign
[2] Massachusetts Institute of Technology

DCC 2010

Introduction

Bluespec:

> a high-level hardware design language based on *guarded atomic actions*; also called *rules*.

Refinement:

> coarse-grained actions to fine-grained sets of actions.

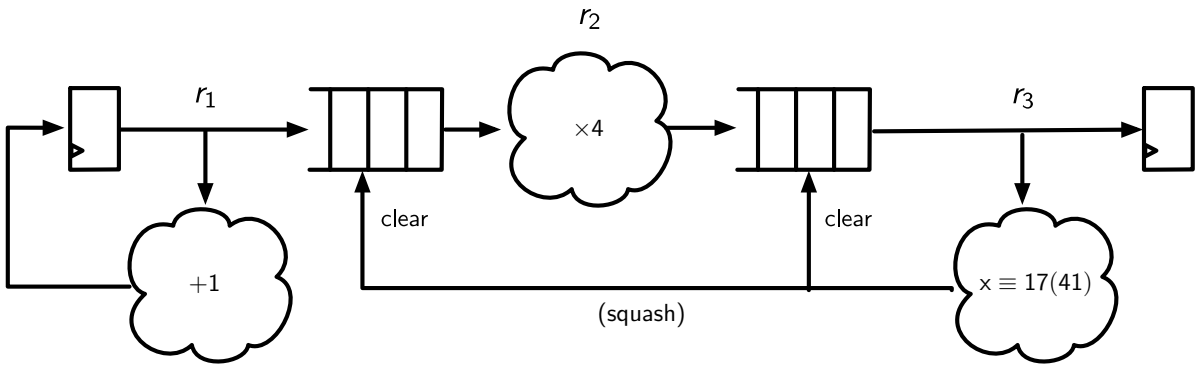
we will show a general mechanism to establish, formally, the correctness of a refinement.

Outline

- *demonstrate a common refinement paradigm;*
- *formalize the correctness criterion;*
- *describe the automated correctness checking procedure we've implemented.*

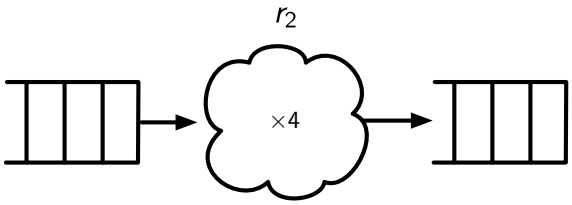
Part I: Example Refinement

Example Refinement

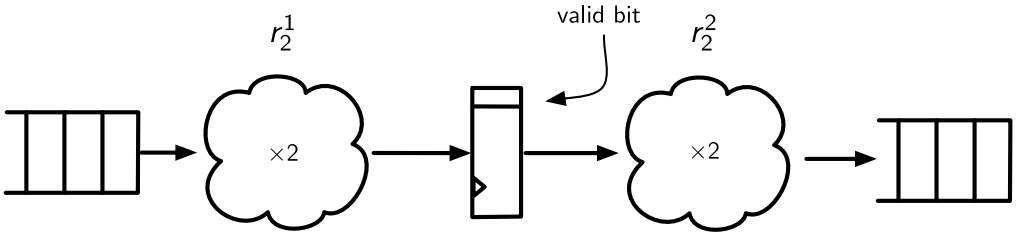


assume "×4" is an expensive operation that we want to refine, e.g., to allow for a faster clock in the synthesized device.

Example Refinement



. . . gets refined to . . .



substituting $\{r_2^1, r_2^2\}$ for r_2 as above, is the design correct? what about r_3 and the squash signal?

Refinement Pattern

- *monolithic rule acting as a bottleneck in the synthesized design;*
- *split into multiple rules and add new state;*
- *possibly update other rules, accounting for the split.*

Part II: Definition of Correctness

Bluespec (abstractly)

clearly, we need a formal definition of correctness that exposes errors during a refinement. it should also be intuitively satisfying and susceptible to automated proof.

to start, let us take an abstract view of what a Bluespec design is.

Definition

A *Bluespec design* is a pair

$$\mathcal{D} = (2^i, \{r_1, \ldots, r_n\})$$

where $i \in \omega$ and for all $1 \leq j \leq n$, $r_j : 2^i \rightharpoonup 2^i$.

$\mathcal{T}(\mathcal{D})$

another way to view a Bluespec design is as a transition system defined by its rules.

Definition

Let $\mathcal{D} = (2^i, \{r_1, \ldots, r_n\})$ be a Bluespec design, we let

$$\mathcal{T}(\mathcal{D}) = (2^i, \longrightarrow_{\mathcal{D}} \subseteq 2^i \times 2^i)$$

denote the transition system where $(s, s') \in \longrightarrow_{\mathcal{D}} \Leftrightarrow r_j(s) = s'$ for some $r_j \in \mathcal{D}$

the Bluespec compiler essentially implements hardware which is cycle-consistent with $\longrightarrow_{\mathcal{D}}^{+}$.

Definition of Correctness

of course . . . this is ultimately about refinement.

Definition

Let $\mathcal{D} = (2^i, \{r_1, \ldots, r_n\})$ be a Bluespec design, a *refinement* of $\mathcal{D}$ is any design

$$\mathcal{D}' = (2^{i+k}, \{\bar{r}_1, \ldots, r_l^1, \ldots, r_l^m, \ldots, \bar{r}_n\})$$

where $\bar{r}_j$ extends $r_j \in \mathcal{D}$ so that

$$\bar{r}_j(s)/_{2^i} = r_j(s/_{2^i}) \quad \text{whenever } r_j(s) \text{ defined;}$$

and r_l is replaced by $\{r_l^1, \ldots, r_l^m\}$.

Definition of Correctness

Definition

Let $\mathcal{D}$ be a Bluespec design and $\mathcal{D}'$ a refinement of $\mathcal{D}$. We call $\mathcal{D}'$ a *correct refinement* of $\mathcal{D}$ iff

$$\longrightarrow_{\mathcal{D}'} /_{2^i} \subseteq \longrightarrow_{\mathcal{D}}^{=} \qquad \text{with respect to } \mathcal{T}(\mathcal{D}), \mathcal{T}(\mathcal{D}');$$

where $\longrightarrow_{\mathcal{D}}^{=}$ denotes the reflexive closure of $\longrightarrow_{\mathcal{D}}$.

there is no single "right answer" when we consider what it means for a refinement to be correct, but the above definition is a very strong and natural correspondence.

Part III: Checking Correctness Automatically

Checking Correctness Automatically

we now have a definition of what it means for a refinement to be correct, but how can we actually check it?

- first, we reduce to establishing a certain *simulation*

$$\mathcal{H} : \mathcal{T}(\mathcal{D}') \longrightarrow \mathcal{T}(\mathcal{D});$$

- second, we convert the simulation conditions to a *symbolic form* suitable for SMT tools;
- finally, an efficient SMT solver *discharges* the obligations establishing the simulation.

Simulations

Definition

Let $\mathcal{A} = (A, \longrightarrow_A \subseteq A \times A)$ and $\mathcal{B} = (B, \longrightarrow_B \subseteq B \times B)$ be transition systems. A *simulation*

$$\mathcal{H} : \mathcal{B} \longrightarrow \mathcal{A}$$

consists entirely of a relation $H \subseteq B \times A$ such that for all $a, a' \in A$ and $b \in B$ where $(a, b) \in H$ and $a \longrightarrow_A a'$, there exists a $b' \in B$ such that $b \longrightarrow_B b'$ and $(a', b') \in H$.

Simulations

simulations are typically used in abstractions relating Kripke structures.

- (strict) simulations reflect satisfaction of ACTL* formulae

our goal is slightly different, so that we want to establish a particular mapping as a simulation, namely

$$\mathrm{rem}_{2^i} : 2^{i+k} \longrightarrow 2^i$$

where i, k are as above for designs $\mathcal{D}, \mathcal{D}'$.

- a straightforward implication is that $\longrightarrow_{\mathcal{D}'} /_{2^i} \subseteq \longrightarrow_{\mathcal{D}}$

Caveat

unfortunately, we don't expect rem_{2^i} *to define a simulation, in general.*

- the issue is that the "extra" state bits, $i \leq b < k$, can take on values which are unreachable during normal operation.

. . . therefore, we need to do something more; specifically, constrain the state space

- for this we introduce the concept of *transaction consistency*.

Transaction Consistency

Definition

Let $\mathcal{D}'$ be a refinement of $\mathcal{D}$ as given earlier. A state $s \in 2^{i+k}$ is *transaction consistent* with respect to $\mathcal{D}, \mathcal{D}'$ if

$$s \longrightarrow^{!}_{\{r_l^1,\ldots,r_l^m\}} s' \quad \text{implies} \quad s/_{2^i} \longrightarrow^{!}_{\{r_l\}} s'/_{2^i}$$

where the refinement takes r_l to $\{r^1, \ldots, r_l^m\}$ as above, and $\longrightarrow^{!}$ denotes a terminating sequence of transitions.

transaction consistency is essentially just a flushing condition.

Bluespec to SMT

we have implemented a tool to convert arbitrary sequences of Bluespec rules to logical formulae; specifically

$$r \quad \text{gets converted to} \quad \pi(\vec{x}), \delta(\vec{x}, \vec{y})$$

where π characterizes the guard of r and δ its action, and $\vec{x}, \vec{y}$ are symbolic variables representing elements of 2^{i+k}.

- formulas are formatted for *STP*, a fast bit-vector solver

Transaction Consistency to SMT

let $n \in \omega$ and $\vec{x}$ etc. be symbolic variables representing states of 2^{i+k};

$$\zeta_n(\vec{x}, \vec{x}') \stackrel{def}{=} \vec{x}_0 = \vec{x} \wedge \bigwedge_{o<n} \left[\bigvee_{1<p\leq m} \delta_{r_l^p}(\vec{x}_o, \vec{x}_{o+1}) \right] \wedge \bigwedge_{1<p\leq m} \left[\neg \pi_{r_l^p}(\vec{x}_n) \right] \wedge \vec{x}' = \vec{x}_n$$

where $\vec{x}_0, \ldots, \vec{x}_n$ are fresh variables. any satisfying assignment ρ to the variables of $\zeta_n(\vec{x}, \vec{x}')$ means that

$$\rho(\vec{x}) \longrightarrow^{!}_{\{r_l^1, \ldots, r_l^m\}} \rho(\vec{x}') \quad \textit{in exactly } n \textit{ steps.}$$

Transaction Consistency to SMT

now, letting $\vec{y}$ *etc. be symbolic variables representing states of* 2^i*;*

$$\xi_n(\vec{y}, \vec{y}') \stackrel{def}{=} \vec{y}_0 = \vec{y} \wedge \bigwedge_{o<n} [\delta_{r_l}(\vec{y}_o, \vec{y}_{o+1})] \wedge \neg\pi_{r_l}(\vec{y}_n) \wedge \vec{y}' = \vec{y}_n$$

where $\vec{y}_0, \ldots, \vec{y}_n$ *are fresh variables. any satisfying assignment* ρ *to the variables of* $\xi_n(\vec{y}, \vec{y}')$ *means that*

$$\rho(\vec{y}) \longrightarrow^{!}_{\{r_l\}} \rho(\vec{y}') \quad \textit{in exactly } n \textit{ steps.}$$

Transaction Consistency to SMT

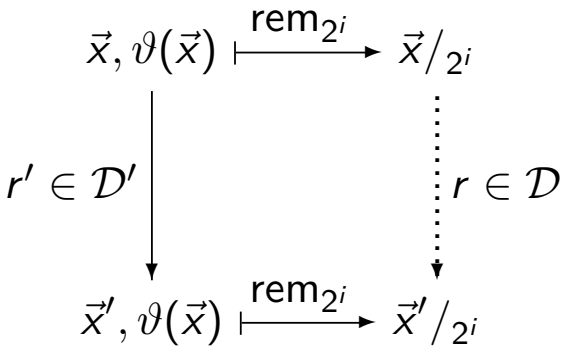
for a given state of 2^{i+k} *represented by a symbolic variable* $\vec{x}$*, let*

$$\vartheta(\vec{x}) \stackrel{def}{=} \bigvee_{n<t'} [\zeta_n(\vec{x}, \vec{x}')] \wedge \bigvee_{n<t} [\xi_n(\vec{x}/_{2^i}, \vec{x}'/_{2^i})]$$

any satisfying assignment ρ *to the variables of* $\vartheta(\vec{x})$ *means that* $\rho(\vec{x})$ *is transaction consistent.*

Refinement Correctness to SMT

the rest is just a matter of converting diagrams



to SMT formulae.

Refinement Correctness to SMT

for a given $r' \in \mathcal{D}'$, let $\psi_{r'}$ denote

$$\psi_{r'} \stackrel{def}{=} \delta_{r'}(\vec{x}, \vec{x}') \wedge \vartheta(\vec{x}) \wedge \bigwedge_{r \in \mathcal{D}} \left[\neg \pi_r(\vec{x}/_{2^i}) \vee \delta_r(\vec{x}/_{2^i}, \vec{y}'_r)\right] \Rightarrow \bigvee_{r \in \mathcal{D}} \left[\pi_r(\vec{x}/_{2^i}) \wedge \vec{x}'/_{2^i} = \vec{y}'_r\right]$$

to check a given refinement for correctness, we need to establish that $\psi_{r'}$ is a tautology, for all $r' \in \mathcal{D}'$; this can be done with STP by negating $\psi_{r'}$ and checking for non-satisfiability. note that we also have to establish preservation of transaction consistency under r', which can be done through a separate characterization of $\neg\vartheta$.

Summary

- *formalized a natural correctness condition for a common refinement pattern used in Bluespec;*
- *described an automated correctness checking procedure we've implemented;*
- *in the future we need to apply this technique to substantial case studies and determine the capacity/bottlenecks of the approach.*

Acknowledgment

we thank Arvind and José Meseguer, both of whom have helped us shape this work; mistakes are ours alone.

A Proposal for a More Generic, More Accountable, Verilog *

Cherif Salama Walid Taha
Rice University
{cherif,taha}@rice.edu

Despite its wide use in industry, Verilog provides limited support for capturing (not to mention statically checking) parametric circuit descriptions. The 2001 IEEE Verilog standard [6] introduces generative constructs that offer a concise way of describing circuit families; however, these constructs are limited to loops and conditionals. In addition, the new standard still restricts module parameters to integer values. In this work, we address these expressivity limitations, identifying the key extensions needed to overcome these limitations, and showing that our previous work on statically checking parameterized Verilog modules can be extended to these significant extensions

Verilog's support for generative constructs provides a natural way to capture the design of circuits with linear structures. While many important circuits like adders, multipliers, and counters fit this pattern, there are other equally important circuits that have so far required ad hoc program generation techniques to describe. Examples of such circuit patterns include tree shaped circuits [4] and butterfly circuits [7]. Additionally, even when they only feature linear structures, generic composition patterns like the common ripple pattern shown in Figure 1, cannot be expressed in Verilog. These expressivity restrictions do not exist in functional hardware description languages like Lava [2]. For example, the ripple pattern can be described in Lava as follows:

```
ripple circ (cin, []) = ([], cin)

ripple circ (cin, a:as) = (b:bs, cout)
  where
    (b, carry) = circ (cin, a)
    (bs, cout) = ripple circ (carry, as)
```

* This work was supported by the National Science Foundation (NSF) SoD award 0439017, and the Semiconductor Research Consortium (SRC) Task ID: 1403.001 (Intel custom project).

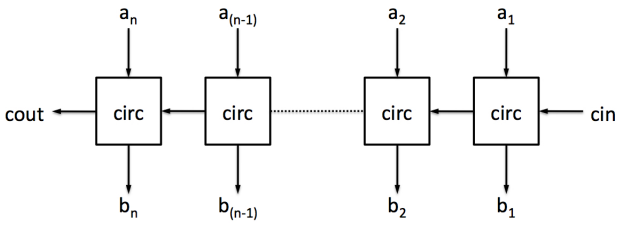


Figure 1. The ripple pattern connecting `n` instances of `circ` (adapted from [2])

Describing such patterns requires the use of recursion, polymorphism, as well as the ability to parametrize one circuit by other circuits. These points were made in early works by Sheeran et al [2] and by O'Leary et al [8], and these ideas have also been incorporated into systems such as BlueSpec [3].

Our goal is to re-introduce these techniques into Verilog as extensions that can be implemented through a pre-processor, that is, as syntactic sugar. In addition, and most importantly, we wish to demonstrate that powerful static analyses are still possible with these extensions.

Achieving these goals requires: 1) extending the syntax in a manner that is compatible with standard Verilog, 2) defining the semantics for the new extensions by a translation into Verilog, and 3) defining the static analysis at the level of this extended source language.

For syntax, we propose a syntax which would allow expressing the above Lava example as follows:

```
module ripple
 (output 't1 [N:1] b, output 't3 cout,
  input 't2 [N:1] a, input 't3 cin);

  parameter N = 4;
  modparameter circ;
  't3 carry;
```

```
  if (N=0)
    assign cout = cin;
  else begin
    circ c (b[1],carry,cin,a[1]);
    ripple #(N-1) #M(circ) r
         (b[N:2],cout,a[N:2],carry);
  end
endmodule
```

The above generic module recursively describes the ripple pattern. It is parametrized by an integer parameter `N` and by the module `circ`. It also allows the ports `a` and `b` to be arrays of arbitrary types. Although fairly concise, the Verilog description is still longer than its Lava counterpart primarily due to its explicit type declarations (as opposed to Lava's inferred types). This price cannot be avoided since we aim to provide extensions that fit naturally with the rest of Verilog, and that existing Verilog users can easily adopt. The technical novelty in our approach, therefore, is not in its syntax, but in the powerful static analyses that it supports.

For the above code to be legal, the pre-processing uses a source-to-source translation to define the following semantic extensions: 1) Recursive modules, 2) higher order modules, and 3) parametric polymorphism.

The semantics of the program are defined only when instantiated with concrete values for both the integer and module parameters at which point it is replaced with the corresponding specialized module. The resulting module is parameter and recursion free.

For the static analysis, we extend type-checking of circuit families [5], bus-width checking [9], and gate count estimation [10]. Adding recursive modules is the most challenging extension to handle because it introduces the possibility of pre-processing divergence which requires static termination analysis. Additionally, recursive modules makes static resource estimation harder because the amount of resources required by a module may potentially be a recurrence relation. We propose using PURRS [1], a recurrence relation solver, and show that it can be used to obtain a closed form estimate.

Using a variety of different examples of circuit families, we illustrate the expressivity gained by the language extensions, and the feasibility of the static analyses even in the presence of challenging circuit patterns.

References

[1] Roberto Bagnara, Andrea Pescetti, Alessandro Zaccagnini, Enea Zaffanella, and Tatiana Zolo. PURRS: the parma university's recurrence relation solver. `http://www.cs.unipr.it/purrs/`.

[2] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. *SIGPLAN Not.*, 34(1):174–184, 1999.

[3] Bluespec, Inc. *Bluespec SystemVerilog Version 3.8 Reference Guide*, 2006.

[4] Annette Bunker. A hardware combinator for tree-shaped circuits. Master's thesis, Brigham Young University, 1998.

[5] Jennifer Gillenwater, Gregory Malecha, Cherif Salama, Angela Yun Zhu, Walid Taha, Jim Grundy, and John O'Leary. Synthesizable high level hardware descriptions: using statically typed two-level languages to guarantee Verilog synthesizability. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 41–50, New York, NY, USA, 2008. ACM.

[6] IEEE Standards Board. *IEEE Standard Verilog Hardware Description Language*. Number 1364-2001 in IEEE Standards. IEEE, 2001.

[7] O.A. Mukhanov and A.F. Kirichenko. Implementation of a fft radix 2 butterfly using serial rsfq multiplier-adders. *Applied Superconductivity, IEEE Transactions on*, 5(2):2461–2464, Jun 1995.

[8] John W. O'Leary, Mark H. Linderman, Miriam Leeser, and Mark Aagaard. Hml: A hardware description language based on standard ml. In *CHDL '93: Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications*, pages 327–334, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[9] Cherif Salama, Gregory Malecha, Walid Taha, Jim Grundy, and John O'Leary. Static consistency checking for Verilog wire interconnects: Using dependent types to check the sanity of Verilog descriptions. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2009.

[10] Cherif Salama, Walid Taha, Jim Grundy, and John O'Leary. The VPP verilog preprocessor. In *HFL '09: Hardware design and Functional Languages*, 2009.

A High-level Language for Testing

Michael Katelman José Meseguer

University of Illinois at Urbana-Champaign

DCC 2010

Contemporary Verification Methodology

simulation dominates industrial verification methodology.

- constrained random testbench + compute cluster.
- $\approx$ 95% of bugs found through simulation (ITRS 2007/2008)

enormous engineering effort is required to build, adjust, and maintain testbenches.

- *"...sources report that in current development projects verification engineers outnumber designers, with this ratio reaching two to one for the most complex designs."*

(ITRS 2007/2008)

Solution Space

so, testing effort dictates verification time; how can the efficacy of this testing regime be improved?

- smart testing algorithms, *e.g.*, Magellan, DeNibulator;
- language-level improvements, *e.g.*, concepts from OOP, temporal assertions, and randoms all appear in SystemVerilog;
- management tools for assessing completeness of testing effort, *e.g.*, coverage statistics.

A High-level Language for Testing

we're proposing a testbench language with the following features:

- embedded in a high-level *declarative language*;
- simulation context of design-under-test is a *first-class object*;
- simulation context is *symbolic*;
- integration with very general, efficient bit-level *SMT solver*.

overarching idea is testing-as-meta-language, enabling verification engineers to develop tailored, smart testing algorithms.

Outline

- *explain what the language is;*
- *demonstrate the utility of its novel features;* and
- *show how it can be effectively implemented.*

Part I: Defining the Testing Language

Overarching Idea

testing-as-meta-language means considering testing as a program that manipulates simulation contexts.

- power to look into the *future, past, and parallel universes*; and
- programmatically *orchestrate* multiple simulation contexts;
- a SystemVerilog testbench, by contrast, exists at the same level as the device-under-test;

Core API (in Haskell)

the "core" API must provide a small, versatile interface; first and foremost, a mechanism for symbolic simulation:

```
eval :: Int -> VSI ()
```

- `VSI` is a "state monad" encapsulating the simulation context;
- it essentially denotes a *function* from context to context;

```
eval :: Int -> (Context -> Context)
```

Core API

the core API also includes a set of functions for creating device-level predicates and interacting with the SMT solver;

```
solve  :: VlogExp -> VSI (Maybe Subst)
applyM :: Subst -> VSI ()
apply  :: Subst -> VlogExp -> VlogExp
at     :: VlogExp -> Int -> VSI VlogExp
```


Core API

also, for getting a list of all identifiers, all inputs, and all outputs;

```
listIds   :: VSI [VlogId] -- all source ids
listIs    :: VSI [VlogId] -- all top-level inputs
listOs    :: VSI [VlogId] -- all top-level outputs
```

. . . and various other functions; at the moment, the API is in flux.

Derived Operations

some "derived" functions we will be using in the examples below:

- simulation with a partial substitution, leaving all undefined values symbolic:

```
simulate :: Int -> [[(VlogId,VlogExp)]] -> VSI ()
```

- completely random simulation on all inputs:

```
simulateRand :: Int -> VSI ()
```

- resolve symbolic variables to concrete values so that the expression is logical 1 during some cycle already simulated;

```
solveAnyCycle1 :: VlogExp -> VSI (Maybe (Subst,Int))
```


Formalization of the Language

briefly, let us mention a formalization of the language which we've done using a logic called rewriting logic.

- analogous to meta-languages used to control theorem provers.
- we formalize the *semantics of Verilog* in rewriting logic. this corresponds to the *axioms* being reasoned from in the analogy.
- via *reflection*, rewriting logic also becomes the meta-language; serving the function of, *e.g.*, ML.
- the core API corresponds to rules of inference in the analogy.

although our implementation is in Haskell, it is also possible to implement the language directly in rewriting logic, e.g., using a tool such as Maude.

Part II: Demonstrating the Language

Example

```
module maze(i,clk,pos);
   input           i,clk;
  output reg [2:0] pos  ;

  always @(posedge clk)
  case (pos)
    0 : pos <= i ? 3 : 1;
    1 : pos <= i ? 2 : 0;
    2 : pos <= i ? 7 : 3;
    3 : pos <= i ? 4 : 2;
    4 : pos <= i ? 4 : 5;
    5 : pos <= i ? 6 : 7;
    6 : pos <= 6; // ``out''
    7 : pos <= 7; // ``dead-end''
  endcase
endmodule
```

Example

first, a simple example using symbolic simulation and employing an automated solver;

```
test :: VSI ()
test = do
   simulate 10 [[]]
   Just (s,j) <- solveAnyCycle1 ("pos" `expEq` 6)
   applyM s
```

. . . result is a concrete simulation with signal pos *equal to* 110 *at cycle j; found via symbolic simulation and a query to an SMT solver.*

Example

. . . solving the same problem, but with multiple simulation contexts and complex control;

```
test = runContT (callCC $ \exit -> g [] exit) return
  where g xs exit = do
          p <- lift (valOfId "pos")
          when (p == expConst 6) $ exit []
          if p `elem` xs
            then return xs
            else do
              ctxt <- lift get
              lift (simulate 1 [[("i",exp1)]])
              xs'  <- g (p:xs) exit
              lift (put ctxt)
              lift (simulate 1 [[("i",exp0)]])
              g xs' exit
```

Example

. . . we can also combine the two, e.g., to get an effect similar to Magellan.

```
strat :: VSI a -> Ctxt -> VSI (a,Ctxt)
                       -- run with context

trial = do
  simulateRand 5
  simulate 5 [[]]
  solveAnyCycle1 ("pos" `expEq` 6)

test = do
  ctxt <- get
  xs   <- mapM (strat trial) (repeat ctxt)
  let Just (c,y) = find (isJust . snd) xs
      Just (s,j) = y
  put c ; applyM s
```


Input Class

we also provide a type-class so that a test can be built at a high-level; consider

```
module counter(ctrl,amt,clk,cnt);
   input      [1:0] ctrl,amt;
   input            clk;
  output reg [7:0]  cnt;

  always @(posedge clk)
  case (ctrl)
    0: cnt <= 0;
    1: cnt <= cnt + amt;
    2: cnt <= cnt - amt;
    3: cnt <= cnt;
  endcase
endmodule
```

Input Class

we can define a data type for the module's operations and make it an instance of the appropriate type-class.

```
data CntrIntf = RST | INC Int | DEC Int | STOP

instance Input CntrIntf where
  toDUT (RST  ) = [[("ctrl",0)]]
  toDUT (INC i) = [[("ctrl",1),("amt",i)]]
  toDUT (DEC i) = [[("ctrl",2),("amt",i)]]
  toDUT (STOP ) = [[("ctrl",3)]]

test = do
  let xs = [RST,replicate 4 (INC 1), STOP]
  simulate (length xs) xs
```

- note that `simulate`'s type needs to become more general.

Part III: Implementation

Overview

our implementation is called "vsi": for verilog symbolic interpreter.

- written entirely in Haskell and compiles with ghc;
- handles both *synthesizeable* and *behavioral* code;
- integrated with the STP, a solver for bit-vectors and arrays;
- will be released under a common open-source license.

Contexts

after parsing and a canonicalization phase, all the data for symbolic simulation is stored in a record:

```
data Ctxt = Ctxt {
  srcSt  :: Map (VlogId        ) (VlogExp),
  tmpSt  :: Map (VlogId        ) (VlogExp),
  usrSt  :: Map (VlogId        ) (VlogExp),
  hist   :: Map (VlogId,Integer) (VlogExp),  ...
```

the state is split into separate maps for (a) source ids, (b) temporary ids, and (c) symbolic variables added by the user.

Contexts

to support behavioral Verilog code, we maintain a set of process queues; as opposed to, e.g., doing a synthesis step;

```
immQ      :: [Process ()     ]          ,
activeQ   :: [Process ()     ]          ,
inactiveQ :: [Process ()     ]          ,
nonBlkQ   :: [Process ()     ]          ,
futureQ   :: [Process Integer]          ,
assigns   :: Map VlogId [VlogStmt]      ,
waitQ     :: Map VlogId [Process VlogEvt],
```

. . . a process is essentially a guarded list of Verilog statements, plus some extra data depending on context.

```
type (Process a) = (a,VlogExp,[VlogStmt]) -- (_,guard,_)
```


Simulation

as noted earlier, the top-level type denoting a simulation is a "state monad" on contexts; specifically

```
type (VSI a) = StateT Ctxt IO a
```

. . . which is essentially equivalent to a function with type . . .

```
Ctxt -> IO (a,Ctxt)
```

- `IO` is stuffed into the monad so that *external calls* to the SMT solver can be made.

Evaluation

events are processed as per the scheduling semantics defined by the Verilog standard;

```
delta :: VSI Integer
delta = do
  done <- epsilon
  if not done
    then delta
    else tickClock
```

```
epsilon :: VSI Bool
epsilon = do
  xs <- gets immQ
  if (not . null) xs
    then do
      mapM evalP xs
      return False
    else do
      xs <- gets activeQ
      ...
      ...
        else return True
```


Evaluation

symbolic execution is cased on each statement construct:

```
evalP :: Process () -> VSI ()
evalP (_,guard,stmts) = do
  modify (\ctxt -> ctxt{guard = guard})
  evalS stmts

evalS :: [VlogStmt] -> VSI ()
evalS (IF'THEN'ELSE c s1 s0 : stmts) =
  if is1 c
    then evalS (s1:stmts)
    else if is0 c
           then evalS(s0:stmts)
           else do
             p1 <- procGuarded () (       c) (s1:stmts)
             p2 <- procGuarded () (expNot c) (s0:stmts)
             toActiveQ [p1,p2]
```

Future Work

- *pin-down the language, or "core api";*
- *work on a set of substantial case studies;*
- *continue to improve the performance of vsi;*
- *functionality outside of testing, e.g., coverage statistics.*

Summary

we've presented a high-level language for hardware testing that promotes a paradigm of testing-as-meta-language and . . .

- *allows multiple, symbolic simulations to be orchestrated in unison* and
- *integrates formal algorithms in a general way*; plus, we've
- *demonstrated its utility and feasibility of implementation*.

Clock Typing of n-Synchronous Programs

Louis Mandel Florence Plateau Marc Pouzet
Université Paris-Sud 11 and INRIA Saclay

Synchronous functional languages such as Lustre or Lucid Synchrone define a restricted class of Kahn Process Networks which can be executed without buffers. This condition is ensured by a dedicated type system, the clock calculus. Every stream is associated to a *clock* defining the instants where the stream is present. Every expression must in turn verify a type constraint such as:

$$\frac{H \vdash e_1 : ck_1 \mid C_1 \quad H \vdash e_2 : ck_2 \mid C_2}{H \vdash e_1 + e_2 : ck_3 \mid \{ck_1 \ \mathbf{=} \ ck_2 \ \mathbf{=} \ ck_3\} \cup C_1 \cup C_2}$$

which states that under the typing environment H, $e_1 + e_2$ has clock ck_3 if e_1 has clock ck_1, e_2 has clock ck_2 and $ck_1 \ \mathbf{=} \ ck_2 \ \mathbf{=} \ ck_3$. Constraints C_1 and C_2 are gathered during typing. Synchronous languages only consider equality constraints. An expression is well typed if its actual clock equals its expected clock and this means that no buffer will be necessary to store or delay it. n-Synchrony [1] relaxes these constraints by allowing to compose streams whose clocks are not equal but can be synchronized through the introduction of bounded buffers. It is obtained by extending the clock calculus with a *subtyping* rule which defines points where a buffer should be inserted. If a stream x with clock ck can be consumed later on a clock ck' using a bounded buffer, we shall say that ck is a subtype of ck' and we shall write $ck \ \mathbf{<:} \ ck'$.

$$\frac{H \vdash e : ck \mid C}{H \vdash \texttt{buffer}(e) : ck' \mid \{ck \ \mathbf{<:} \ ck'\} \cup C}$$

In term of sequence of values, $\texttt{buffer}(e)$ is equivalent to e but it may delay its input using a bounded buffer. The purpose of the extended clock calculus is to compute this bound. The designer can write $\texttt{buffer}(e)$ everywhere in the program as potential places where a buffer can be inserted. Then, the clock calculus automatically computes bounds for these buffers.

n-Synchrony can be defined for any language of clocks provided we are able to test equality ($ck_1 \ \mathbf{=} \ ck_2$), subtyping ($ck_1 \ \mathbf{<:} \ ck_2$) and $\texttt{size}(ck_1, ck_2)$ to compute the buffer to synchronize ck_1 and ck_2. An interesting case is the one where clocks are restricted to be *ultimately periodic binary sequences* for which all the above properties are decidable.

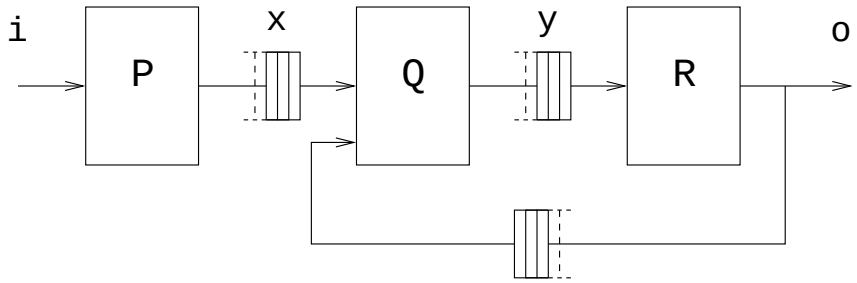
Last year, we presented how to abstract clocks in order to check the subtyping relation in an efficient manner [2]. This year, we shall present Lucy-n, the first implementation of a n-synchronous programming language. In this talk, we will describe the language as an extension of Lustre. Then, we will explain its clock calculus and the constraints resolution algorithm. Finally, we will illustrate it on a multimedia application.

References

[1] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. *N*-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems. In *ACM International Conference on Principles of Programming Languages*, January 2006.

[2] A. Cohen, L. Mandel, F. Plateau, and M. Pouzet. Relaxing synchronous composition with clock abstraction. In *Hardware Design using Functional languages*, pages 35–52, York, UK, 2009.

Kahn Process Networks [Gilles Kahn, 1974]



Network of processes

- concurrent execution
- communication through buffers of sufficient size

If processes are deterministic then the network is deterministic

Programming Kahn Process Networks

Problem: computation of sufficient buffer sizes

- risk of data loss, of deadlock
- sometimes, need of infinite buffers

Goal:

- rejection of infinite buffers
- automatic sizing of buffers

Related work:

- Synchronous Data Flow and variants [Lee *et al.*] [Buck]
- scheduling [Carlier, Chretienne] [Baccelli, Cohen, Quadrat]
- Network Calculus [Cruz], Real-time Calculus [Thiele *et al.*]

Dataflow Synchronous Model

Programming Kahn networks without buffers:

- programming languages: Lustre, Signal, Lucid Synchrone
- instantaneous consumption of produced data
- strong guaranties: bounded memory, absence of deadlocks

But: communication without buffers sometimes too restrictive
(e.g. multimedia applications)

n-Synchronous Model: Programming Kahn Networks with Bounded Memory

Automatic methods at compile time to:

- accept to store data in buffers
- reject networks needing infinite memory
- compute activation paces of computations nodes
- compute sufficient buffers sizes

More flexibility with the same guaranties

Overview

1. Lucy-n: a n-Synchronous Extension of Lustre
2. Periodic Clocks
3. Abstract Clocks
4. Conclusion and Future Work

Lucy-n: a n-Synchronous Extension of Lustre

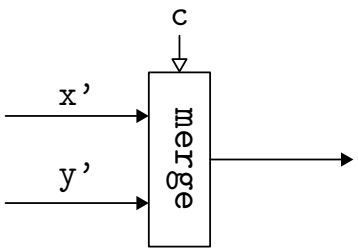
flow	values								clock
x	5	7	3	6	2	8	1	...	1111111...
c	1	0	1	0	1	0	1	...	
x when c	5		3		2		1	...	1010101...
c'	1		0		1		1	...	
(x when c) when c'	5				2		1	...	1000101...

clock(x when c) = *clock*(x) *on* c

clock((x when c) when c') = *clock*(x when c) *on* c'

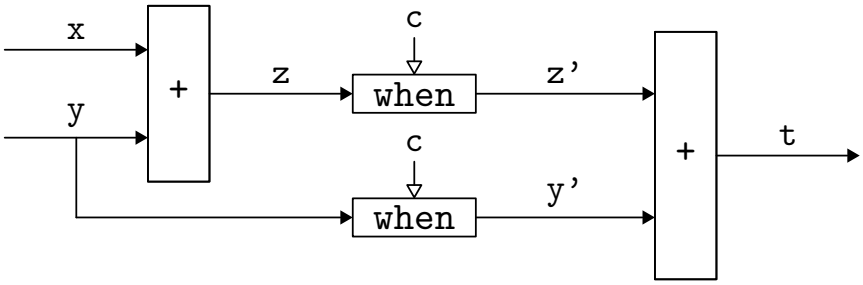
on operator :

$$0.w_1 \textit{ on } w_2 \stackrel{def}{=} 0.(w_1 \textit{ on } w_2)$$
$$1.w_1 \textit{ on } 1.w_2 \stackrel{def}{=} 1.(w_1 \textit{ on } w_2)$$
$$1.w_1 \textit{ on } 0.w_2 \stackrel{def}{=} 0.(w_1 \textit{ on } w_2)$$



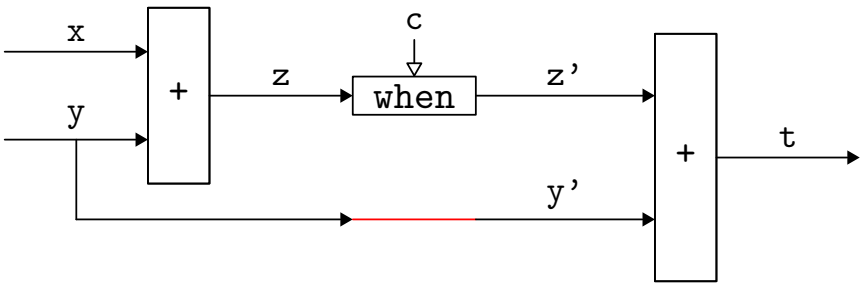
flow	value								clock
x'	5		3		2		1	...	1010101...
y'		2		5		1		...	0101010...
c	1	0	1	0	1	0	1	...	
merge c x' y'	5	2	3	5	2	1	1	...	1111111...

A Dataflow Synchronous Kernel



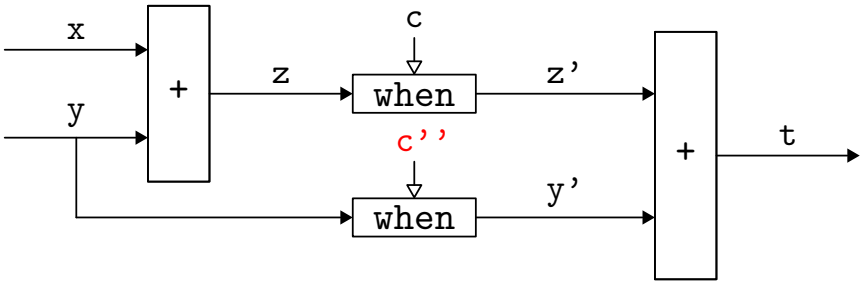
flow	values								clock
x	5	7	3	6	2	8	1	...	111111...
y	3	2	1	5	4	1	7	...	111111...
z = x + y	8	9	4	11	6	9	8	...	111111...
c	1	0	1	0	1	0	1	...	
z' = z when c	8		4		6		8	...	101010...
y' = y when c	3		1		4		7	...	101010...
t = z' + y'	11		5		10		15	...	101010...

A Dataflow Synchronous Kernel



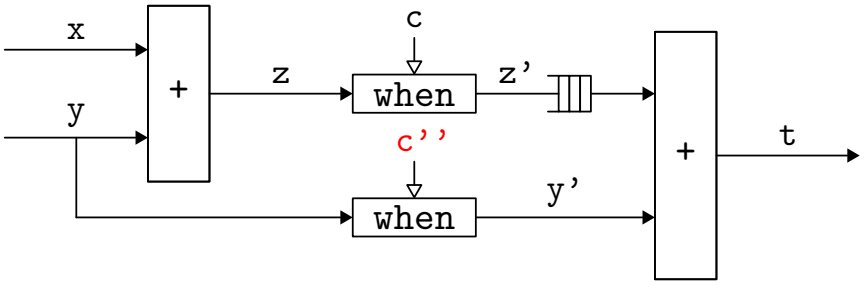
flow	values								clock
x	5	7	3	6	2	8	1	...	111111...
y	3	2	1	5	4	1	7	...	111111...
z = x + y	8	9	4	11	6	9	8	...	111111...
c	1	0	1	0	1	0	1	...	
z' = z when c	8		4		6		8	...	101010...
y' = y	3	2	1	5	4	1	7	...	111111...
t = z' + y'	rejected								

A Dataflow Synchronous Kernel



flow	values								clock
x	5	7	3	6	2	8	1	...	111111...
y	3	2	1	5	4	1	7	...	111111...
z = x + y	8	9	4	11	6	9	8	...	111111...
c	1	0	1	0	1	0	1	...	
z' = z when c	8		4		6		8	...	101010...
y' = y when c''		2		5		1		...	010101...
t = z' + y'	rejected								

n-Synchronous Extension: Bufferization Operator

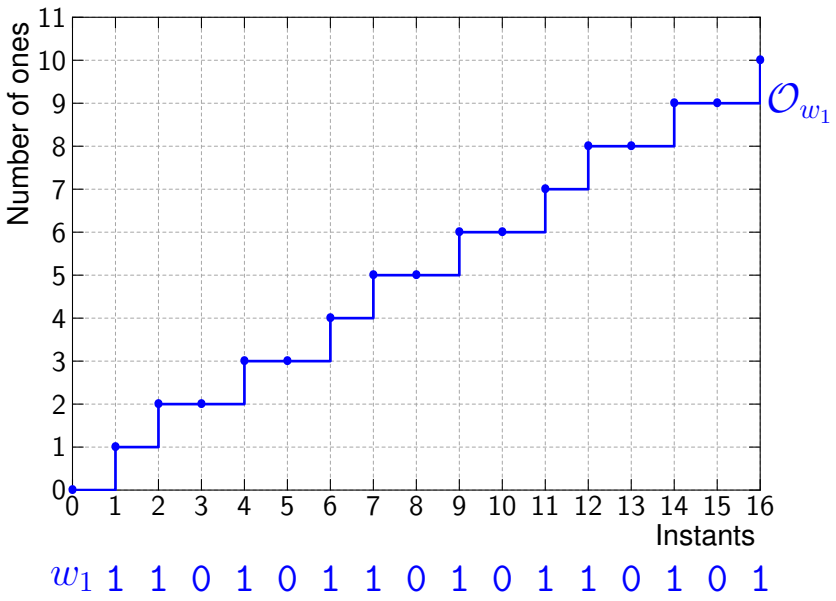


flow	values				clock
z' = z when c	7	4	6	8 ...	101010...
z'' = buffer(z')	7	4	6	...	010101...
y' = y when c''	2	5	1	...	010101...
t = z'' + y'	9	9	7	...	010101...

- adaptability relation $\Rightarrow$ communication through a bounded buffer
- example : 101010... $<:$ 010101...

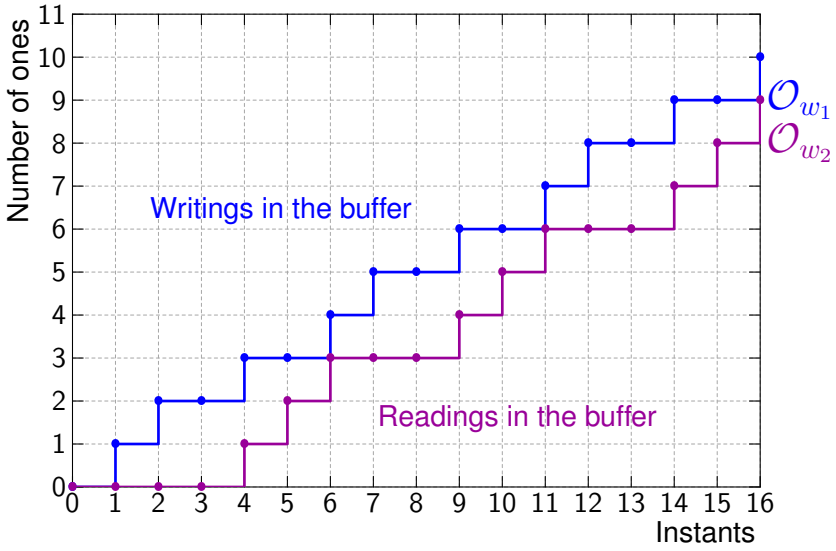
Adaptability Relation

Cumulative Function



$\mathcal{O}_{w_1}$: cumulative function of the word w_1

Adaptability Relation



buffer size	$size(w_1, w_2) = \max_{i \in \mathbb{N}}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$
adaptability	$w_1 <: w_2 \overset{def}{\Leftrightarrow} \exists n \in \mathbb{N}, \forall i,\ 0 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq n$
synchronizability	$w_1 \bowtie w_2 \overset{def}{\Leftrightarrow} \exists b_1, b_2 \in \mathbb{Z}, \forall i,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$
precedence	$w_1 \preceq w_2 \overset{def}{\Leftrightarrow} \forall i,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$

Typing

We want to guaranty at compile time that:

- clocks are equal when the communication is done without buffers
- clocks are adaptable when communication is done through buffers

It is done by typing techniques.

- The clock of a node is described by a type scheme:

$$\sigma \quad ::= \quad \forall \alpha_1, \ldots, \alpha_n . (ck \times ... \times ck) \rightarrow (ck \times ... \times ck)$$

- The clock of a flow is described by a type:

$$ck \quad ::= \quad \alpha \quad | \quad ck \texttt{ on } ce \quad | \quad ck \texttt{ on not } ce$$

- Equality of clocks is ensured by equality of types: $ck_1 \ = \ ck_2$
- Adaptability of clocks is ensured by subtyping: $ck_1 \ <: \ ck_2$

This type calculus is named clock calculus

Clock Calculus

Equality and subtyping constraints are collected during typing:

$$C \quad ::= \quad \{ck_1 \ = \ ck_2\} \cup C \quad | \quad \{ck_1 \ <: \ ck_2\} \cup C \quad | \quad \varnothing$$

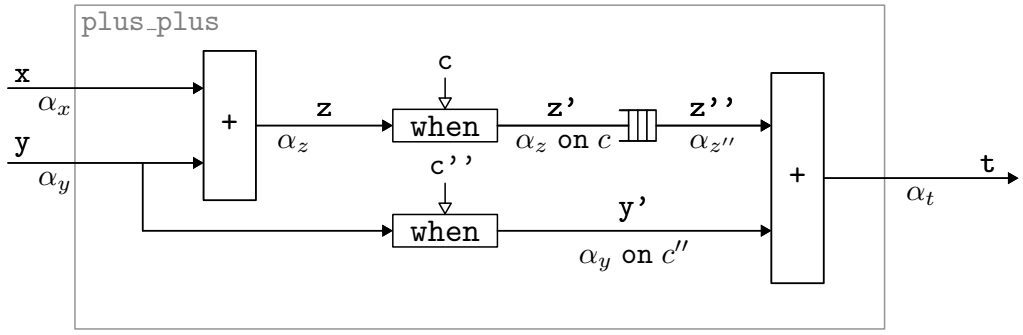
Examples of typing rules:

$$\frac{H \vdash e_1 : ck_1 \mid C_1 \qquad H \vdash e_2 : ck_2 \mid C_2}{H \vdash e_1 \ + \ e_2 : \alpha \mid \{ck_1 \ = \ ck_2 \ = \ \alpha\} \cup C_1 \cup C_2}$$

$$\frac{H \vdash e : ck \mid C}{H \vdash \texttt{buffer(}e\texttt{)} : \alpha \mid \{ck \ <: \ \alpha\} \cup C}$$

$$\frac{H \vdash e : ck \mid C}{H \vdash e \texttt{ when } ce : ck \texttt{ on } ce \mid C}$$

Clock Calculus — Example



$$(\alpha_x \times \alpha_y) \to \alpha_t \qquad \text{such that} \quad C = \left\{ \begin{array}{l} \alpha_x = \alpha_y = \alpha_z; \\ \alpha_y \text{ on } c'' = \alpha_{z''} = \alpha_t; \\ \alpha_z \text{ on } c \mathbin{<:} \alpha_{z''} \end{array} \right\}$$

$$\rightsquigarrow \quad (\alpha \times \alpha) \to \alpha \text{ on } c'' \quad \text{such that} \quad C = \left\{ \alpha \text{ on } c \mathbin{<:} \alpha \text{ on } c'' \right\}$$

Clock Calculus — Type Inference

Constraints solving

- simple case:

$$\alpha \text{ on } w_1 \mathbin{<:} \alpha \text{ on } w_2 \quad \Leftrightarrow \quad w_1 <: w_2$$

  $\Rightarrow$ verification of adaptability relation on words $w_1 <: w_2$

- more difficult case:

$$\alpha_1 \text{ on } w_1 \mathbin{<:} \alpha_2 \text{ on } w_2 \quad \Leftarrow \quad \left\{ \begin{array}{l} \alpha_1 \leftarrow \alpha \text{ on } c_1 \\ \alpha_2 \leftarrow \alpha \text{ on } c_2 \\ \alpha \text{ on } c_1 \text{ on } w_1 \mathbin{<:} \alpha \text{ on } c_2 \text{ on } w_2 \end{array} \right.$$

  $\Rightarrow$ inference of paces c_1 and c_2 such that c_1 ***on*** $w_1 <: c_2$ ***on*** w_2

Buffer size computation

$$\texttt{size}(\alpha \text{ on } w_1, \alpha \text{ on } w_2) = size(w_1, w_2)$$

Periodic Clocks

Ultimately Periodic Clocks

Example : $0(00111) = 0\,00111\,00111\ldots$

Adjustment

- increase of the prefix size: $0(00111) = 0\,001(11\,001)$
- repetition of the periodic pattern: $0(00111) = 0(00111\,00111\,00111)$

Verification of relations on clocks

- equality test: $0(00111\ 00111) = 0\ 00(111\ 00)$
- synchronizability test: $(11010) \bowtie 0(00111)$
- precedence test: $(11010) \preceq 0(00111)$
- adaptability test: conjunction of synchronizability and precedence

Clock expressions

- computation of on : $0(00111)\ on\ (101) = 0(00101)$
- computation of not : $not\ 0(00111) = 1(11000)$

Paces Inference

Example of a system without prefix:

$$C_1 = \left\{ \begin{array}{lcl} \alpha_1 \text{ on } \mathit{not}\ (100) & <: & \alpha_2 \text{ on } (10) \\ \alpha_1 \text{ on } (1) & <: & \alpha_3 \text{ on } (01) \\ \alpha_3 \text{ on } (011)\ \mathit{on}\ (10) & <: & \alpha_2 \text{ on } (01) \end{array} \right\}$$

1. Computation of clock expressions

$$C_1 = \left\{ \begin{array}{lcl} \alpha_1 \text{ on } (011) & <: & \alpha_2 \text{ on } (10) \\ \alpha_1 \text{ on } (1) & <: & \alpha_3 \text{ on } (01) \\ \alpha_3 \text{ on } (010) & <: & \alpha_2 \text{ on } (01) \end{array} \right\}$$

2. Instantiation of type variables

$$\alpha_1 \leftarrow \alpha \text{ on } c_1, \quad \alpha_2 \leftarrow \alpha \text{ on } c_2, \quad \alpha_3 \leftarrow \alpha \text{ on } c_3$$

$$C_1 = \left\{ \begin{array}{lcl} \alpha \text{ on } c_1\ \mathit{on}\ (011) & <: & \alpha \text{ on } c_2\ \mathit{on}\ (10) \\ \alpha \text{ on } c_1\ \mathit{on}\ (1) & <: & \alpha \text{ on } c_3\ \mathit{on}\ (01) \\ \alpha \text{ on } c_3\ \mathit{on}\ (010) & <: & \alpha \text{ on } c_2\ \mathit{on}\ (01) \end{array} \right\}$$

Paces Inference

3. Transformation into an adaptability constraints system

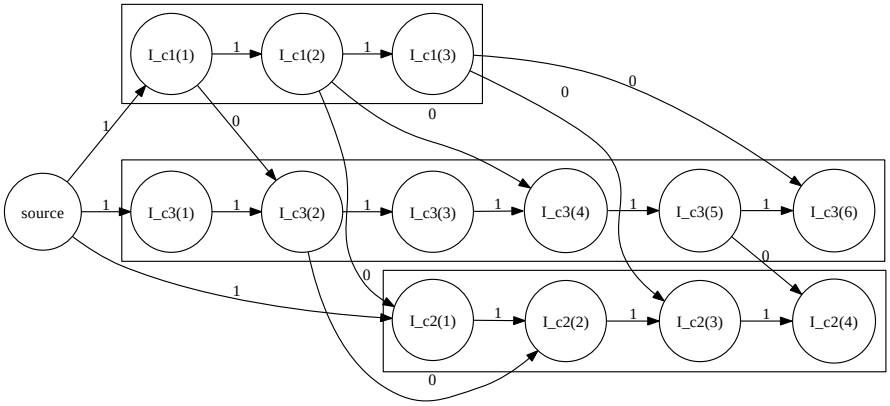
$$A_1 = \left\{ \begin{array}{lcl} c_1\ \mathit{on}\ (011) & <: & c_2\ \mathit{on}\ (10) \\ c_1\ \mathit{on}\ (1) & <: & c_3\ \mathit{on}\ (01) \\ c_3\ \mathit{on}\ (010) & <: & c_2\ \mathit{on}\ (01) \end{array} \right\}$$

4. Transformation into adjusted forms

$$A_1' = \left\{ \begin{array}{lcl} c_1\ \mathit{on}\ (011) & <: & c_2\ \mathit{on}\ (1010) \\ c_1\ \mathit{on}\ (111) & <: & c_3\ \mathit{on}\ (010101) \\ c_3\ \mathit{on}\ (010010) & <: & c_2\ \mathit{on}\ (0101) \end{array} \right\}$$

Paces Inference

5. Transformation into linear inequations on indexes of 1s in clock variables c_i



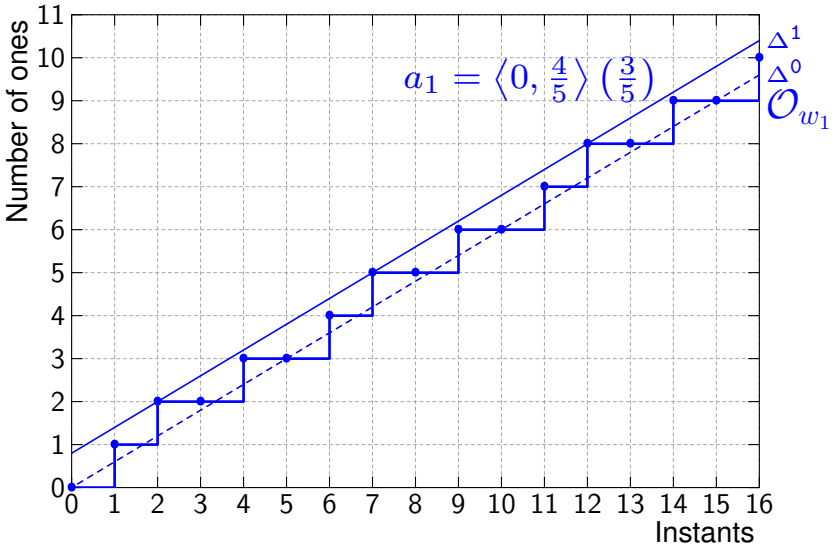
Summary:

- proved correct
- problem with prefixes
- may be expansive: the number of linear inequations is proportional to the number of 1s in the system

24

Abstract Clocks

Abstract Clocks: $abs\,(w) = \langle b^0, b^1 \rangle\,(r)$

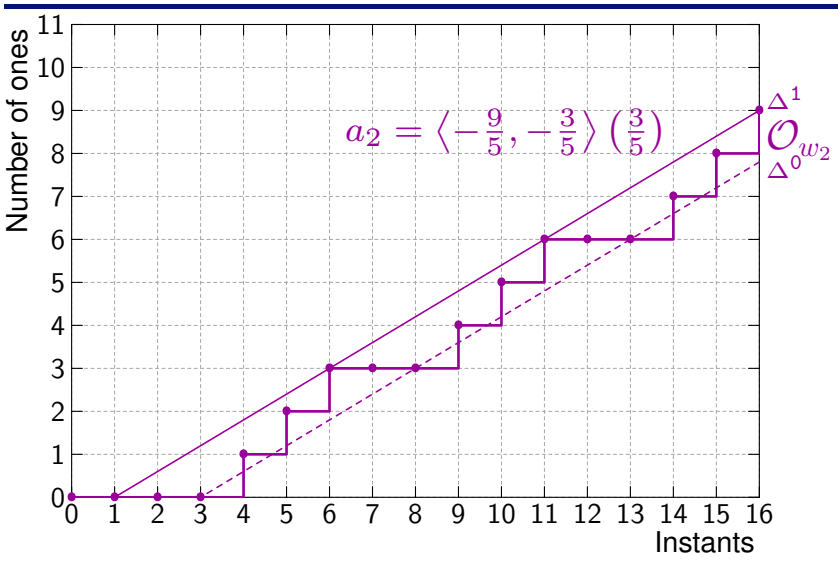


$\Delta^1 : r \times i + b^1$

$\Delta^0 : r \times i + b^0$

$$concr\,(\langle b^0, b^1 \rangle\,(r)) = \left\{ w \;\middle|\; \begin{array}{lcl} w[i] = 1 & \Rightarrow & \mathcal{O}_w(i) \leq \Delta^1(i) \\ w[i] = 0 & \Rightarrow & \mathcal{O}_w(i) \geq \Delta^0(i) \end{array} \right\}$$

Abstraction of Clocks



Abstraction of ultimately periodic words:

$$abs\,(0(00111)) = \left\langle -\frac{9}{5}, -\frac{3}{5} \right\rangle \left(\frac{3}{5}\right)$$

Abstraction of Clocks

Abstraction of clock expressions:

$$\begin{aligned} abs(\mathit{not}\ w) &= \mathit{not}^{\sim}\ abs(w) \\ abs(ce_1\ \mathit{on}\ ce_2) &= abs(ce_1)\ \mathit{on}^{\sim}\ abs(ce_2) \end{aligned}$$

Correctness property of operators:

$$\begin{aligned} \mathit{not}\ w &\in concr(\mathit{not}^{\sim}\ abs(w)) \\ ce_1\ \mathit{on}\ ce_2 &\in concr(abs(ce_1)\ \mathit{on}^{\sim}\ abs(ce_2)) \end{aligned}$$

Definition of abstract operators:

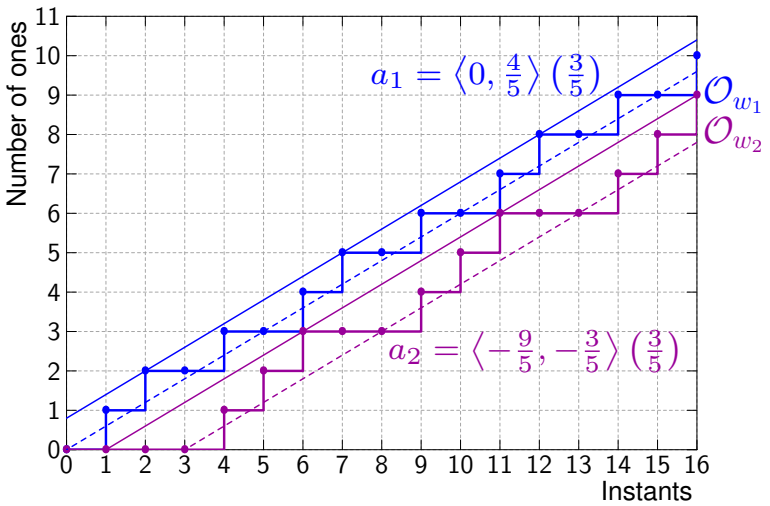
$$\mathit{not}^{\sim}\ \langle b^0, b^1 \rangle\,(r) \stackrel{def}{=} \langle -b^1, -b^0 \rangle\,(\,1 - r)$$

$$\langle b^0{}_1, b^1{}_1 \rangle\,(r_1)\ \mathit{on}^{\sim}\ \langle b^0{}_2, b^1{}_2 \rangle\,(r_2) \stackrel{def}{=}$$

$$\langle b^0{}_1 \times r_2 + b^0{}_2, b^1{}_1 \times r_2 + b^1{}_2 \rangle\,(r_1 \times r_2) \text{ with } b^0{}_1 \leq 0 \text{ and } b^0{}_2 \leq 0$$

Adaptability of Abstract Clocks

$\bowtie^{\sim}$ and $\preceq^{\sim}$



synchronizability $\langle {b^0}_1, {b^1}_1 \rangle (r_1) \bowtie^{\sim} \langle {b^0}_2, {b^1}_2 \rangle (r_2) \overset{def}{\Leftrightarrow} r_1 = r_2$

precedence $\langle {b^0}_1, {b^1}_1 \rangle (r) \preceq^{\sim} \langle {b^0}_2, {b^1}_2 \rangle (r) \overset{def}{\Leftrightarrow} {b^1}_2 - {b^0}_1 < 1$

buffer size $size^{\sim}(a_1, a_2) = \lfloor {b^1}_1 - {b^0}_2 \rfloor$

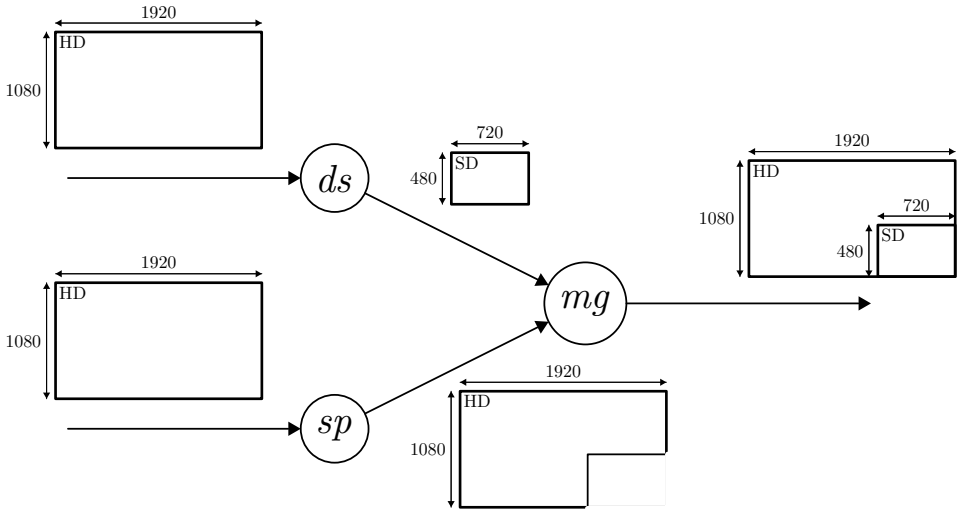
Properties: $a_1 \bowtie^{\sim} a_2 \Rightarrow \forall w_1 \in concr(a_1), \forall w_2 \in concr(a_2), w_1 \bowtie w_2$

$a_1 \preceq^{\sim} a_2 \Rightarrow \forall w_1 \in concr(a_1), \forall w_2 \in concr(a_2), w_1 \preceq w_2$

$\forall w_1 \in concr(a_1), \forall w_2 \in concr(a_2), size(w_1, w_2) \leq size^{\sim}(a_1, a_2)$

29

Video Application

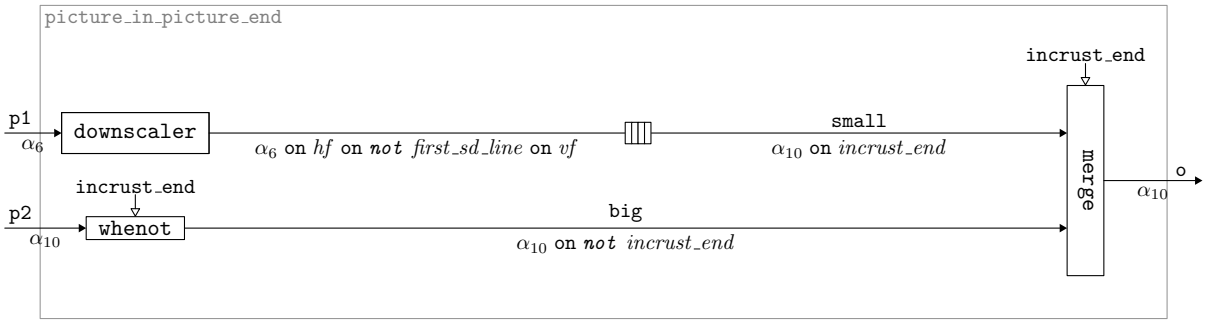


Picture in Picture:

- takes two High Definition images as input
- downscales the first one to obtain a Small Definition image
- embeds the small image in the big one

30

Video Application




```
51  (* picture in picture *)
52  let clock incrust_end =
53    (0^(1920 * (1080 - 480)) {0^1200 1^720}^480)
54
55  let node picture_in_picture_end (p1, p2) = o where
56    rec small = buffer(downscaler p1)
57    and big = (p2 whenot incrust_end)
58    and o = merge incrust_end small big
```

31

Video Application

System to solve:

$$\{\alpha_6 \text{ on } hf \text{ on not } first\_sd\_line \text{ on } vf \;<:\; \alpha_{10} \text{ on } incrust\_end\}$$

1. Adaptability constraints
   $\alpha_6 \leftarrow \alpha$ on c_6, $\quad \alpha_{10} \leftarrow \alpha$ on c_{10}
   $\{c_6 \;on\; hf \;on\; not\; first\_sd\_line \;on\; vf \;<:\; c_{10} \;on\; incrust\_end\}$

2. Abstraction of constraints
   $\{abs\,(c_6) \;on^{\sim}\; abs\,(hf) \;on^{\sim}\; not^{\sim}\; abs\,(first\_sd\_line) \;on^{\sim}\; abs\,(vf)$
   $<:^{\sim} abs\,(c_{10}) \;on^{\sim}\; abs\,(incrust\_end)\}$
   i.e. $\{abs\,(c_6) \;on^{\sim}\; \langle -720, \frac{481}{3}\rangle\,(\frac{1}{6}) <:^{\sim} abs\,(c_{10}) \;on^{\sim}\; \langle -192200, 0\rangle\,(\frac{1}{6})\}$

3. System of linear inequations

4. Solution: $c_6 = 0^{4315}(1), \quad c_{10} = (1)$

$$picture\_in\_picture\_end :: \forall\alpha.\,(\alpha \times \alpha \text{ on } 0^{4315}(1)) \rightarrow \alpha \text{ on } 0^{4315}(1)$$

32

Summary

	delay	buffer size
minimal result	$1\,920$ ($\approx$ 1 HD line)	$191\,970$ (≈ 266.6 SD lines)
abstract result	$4\,315$ ($\approx$ 2 HD lines)	$193\,079$ (≈ 268.1 SD lines)

- proved correct
- incomplete by nature
- handles systems with prefixes
- good performances: the number of linear inequations is proportional to the number of adaptability constraints in the system

33

Conclusion and Future Work

Conclusion

- n-synchronous model:
  more flexible composition of nodes without loss of guaranties
- two clocks languages studied
- algorithm with abstraction:
  efficient and working on non periodic clocks
- algorithms implemented in Lucy-n
- coq proofs

Future Work

- integer clocks
- paces inference on periodic clocks with prefixes
- buffers with limited size, strict buffers
- links with static scheduling in Latency Insensitive Design
  [de Simone, Boucaron, Millo]
- code generation

Synthesis of Data Parallel GPU Software into FPGA Hardware

Satnam Singh

Microsoft Research

This presentation describes an embedded domain specific language for expressing data parallel computations. The Accelerator library is embedded in C++ and also in C# and it can be used from other .NET languages such as F#. The library provides data-parallel arrays and data-parallel operations which are designed to be applied to whole arrays rather than individual elements e.g. transpose, stride operations and shifts which are similar to stencil style computations.

The original target for this library was GPU code via a DirectX-based compilation flow; after that a target for x64 multi-core processes using SIMD instructions was developed. This presentation describes a new target for this library which supports efficient compilation to Xilinx FPGAs. We show how the discipline of the whole-array operations allows us to generate efficient address-generation hardware.

Chalk: a language and tool for architecture design and analysis

Wouter Swierstra Koen Claessen Carl Seger
Mary Sheeran Emily Shriver

January 20, 2010

Abstract

By virtue of Moore's law, today's hardware designs are extremely complex. A modern microprocessor contains multiple cores (often heterogenous), multiple levels of caches, complex high-speed interconnects and is built using over a billion transistors. Designing a functionally correct high-performance architecture of such a complex system is very challenging. However, as hardware migrates from air-conditioned computer rooms to our pockets, a range of non-functional design aspects take on the same urgency and importance as the functional correctness and performance aspects. Current hardware and architecture description languages are struggling to keep up with this burgeoning complexity. We see two possible ways forward: either bite the bullet and start trying to reason about SystemC, as recently argued for by Vardi [15], or start again and try to provide exactly the right abstractions in a much simpler setting. We have chosen the latter approach, in a joint project that has been running since Feb. 2009. This talk is very much a description of work in progress and is aimed at stimulating discussion.

1 Introduction

Computer architecture design is a complex multi-faceted engineering task. Traditionally, the primary focus was on functional correctness and performance. However, today other design parameters have emerged that are taking on the same, or even greater, importance. For example, as a result of the proliferation of mobile electronic devices, power consumption is now as important a design consideration as raw performance. Of course, these design requirements are often contradictory and require careful tradeoffs.

Architects must strike a balance between performance, power consumption, cost, and many other non-functional design constraints. The exact repercussions of early design decisions are not always clear: how will an additional cache improve performance? How much additional power will the cache require? How much will it cost? It is crucial to provide architects with tool support to make such decisions.

To partially address these problems, we are designing Chalk, a domain specific embedded language for computer architecture design. We are implementing Chalk as a combinator library in the general purpose functional programming language Haskell [14]. The design of Chalk has been driven by three key goals:

Simplicity The core combinators that Chalk provides are simple. Users do not need to be familiar with advanced functional programming techniques to grasp the core language design.

Abstraction Chalk exposes all of Haskell's language features to computer architects. Architects may capture common design patterns using polymorphism, higher-order functions, and algebraic data types.

Analysis Circuit descriptions in Chalk are executable. Furthermore, we provide several example non-standard interpretations of Chalk circuits. Crucially, however, we enable users to define their own analyses of Chalk circuits specific to any particular domain.

The pursuit of these goals has led to new insights. More specifically, this paper makes the following research contributions:

- We have given a comparison of existing domain-specific embedded languages for hardware description (Section 2). Based on our observations, we motivate the design of a *deep embedding* of a *behavioural* architecture description language, addressing some of the weaknesses of previous attempts (Section 3). We illustrate our design choice with an example architecture taken from the literature (Section 4).
- A deep embedding enables us to define *new behaviours*, that is, non-standard interpretations of our circuits. We present several example analyses that can be formulated as such non-standard interpretations, including the maximum delay/pipeline depth, circuit visualisation, and a simple activity analysis.
- Finally, this paper presents a case study in applying the recent extensions to the Glasgow Haskell Compiler (GHC), such as Generalised Algebraic Data Types, may be used in the design of domain specific embedded languages.

2 Embedded hardware description languages

Our work on Chalk draws inspiration from two existing domain specific languages for hardware design: Lava [1, 4] and Hawk [10, 5, 8]. Both Lava and Hawk are languages embedded in Haskell that were developed approximately ten years ago. Despite these superficial similarities, Lava and Hawk are poles apart. Before describing our work on Chalk we will give a brief overview of both Lava and Hawk.

2.1 Lava

Lava is a combinator library for gate-level descriptions of hardware circuits. The core idea underlying Lava is to define a data type with separate constructors for different hardware gates, for example:

```
data Gate c where
  And :: c -> c -> Gate
  Or :: c -> c -> Gate
  Not :: c -> Gate
  ...

data Lava where
   Circuit :: Ref (Gate Lava) -> Lava
```

Rather than represent gates as a straightforward inductive data type, there is an additional level of indirection here. The `Lava` data type wraps every recursive subgate in an additional `Ref` type constructor. This indirection is necessary to *observe sharing* in our embedded language. Consider the following, somewhat contrived, example:

```
silly x = or x x
```

If we generate hardware circuits from a gate-level description without this extra indirection, using the `silly` function results in *two* copies of its argument circuit. The programmer probably meant to generate a *single* copy of the function's argument, but duplicate the result of this circuit using a fan-out.

The situation is even worse in the presence of recursively defined circuits. Any function traversing a recursively defined circuit may fail to terminate. Such a function expects a finite circuit, but traversing the circuit repeatedly unfolds the recursive definition, resulting in divergence.

To address this problem, Lava wraps all recursive subcircuits in a additional `Ref` type. The interface for manipulating these references is:

```
type Ref a
ref :: a -> Ref a
deref :: Ref a -> a
unique :: Ref a -> Unique
```

Although the `Ref` type is kept abstract, it is implemented as a pair of a value of type `a`, together with some unique identifier (generated using GHC's library `Data.Unique`). The creation of references is not a pure function. It uses GHC's primitive function `unsafePerformIO` to enable the presentation of a pure interface to the user. The consequences of breaking purity have been discussed in the literature [2]. Dereferencing, on the other hand, is entirely safe: it simply discards the unique identifier. Similarly, the `unique` function projects out the unique identifier from a reference. When we compare to values of type `Ref a` for equality, we compare their unique identifiers and not the values.

This is not the only way to make sharing observable. Earlier versions of Lava used a state monad to generate unique names. Recent work by Gill [7] gives a good overview of the different solutions that have been proposed to this problem.

If we introduce smart constructors for individual gates that take care of the reference creation, we can define a multiplexer between two bits as follows:

```
mux :: Lava -> Lava -> Lava -> Lava
mux c t e = or (and t c) (and e (not c))
```

The `mux` function is a bit-level if-than-else: if the signal `c` is true, it will return the value of `t`; otherwise it will return the value of `e`. Here we have to describe the *behaviour* of the multiplexer by defining its implementation in terms of logical gates.

Lava defines a rich combinator library for composing circuits. Besides obvious candidates such as sequential and parallel composition, Lava illustrates how many complex circuit patterns such as butterfly circuits can be described succinctly using recursion [3].

The embedding of Lava into Haskell is *deep*, that is, we have a Haskell data type representing Lava's abstract syntax tree. As a result, functions that traverse the tree can compute different values of interest. For instance, we can simulate circuits, compute input for automated theorem provers, or generate VHDL descriptions.

Lava is a structural hardware description language. All circuits are implemented from a handful of simple, primitive gates. It uses Haskell's abstractions to provide combinators for assembling circuit descriptions. When designing new circuits, however, thinking at the gate-level can be too restrictive. The gate-level design of a circuit may not be available until later on in the design process. The fact that integers as well as bits can "flow" on Lava wires alleviates this problem a little for some classes of circuits, but still one is anchored to a low level of abstraction and this can be over-restrictive.

2.2 Hawk

In contrast to Lava, Hawk is a behavioural hardware description language. A circuit's behaviour is defined to be the (infinite) list of values that are produced at consecutive clock cycles:

```
type Hawk a = [a]
```

Based on this simple design principle, there are several combinators that turn out to be useful when defining Hawk circuits:

```
constant :: a -> Hawk a
constant x = x : constant x

lift :: (a -> b) -> Hawk a -> Hawk b
lift f (x : xs) = f x : lift f xs
```

```
delay :: a -> Hawk a -> Hawk
delay x xs = x : xs
```

The definition of a multiplexer in Hawk is very different from the Lava definition we saw previously. One possible definition simply pattern matches on its argument lists:

```
mux :: Hawk Bool -> Hawk a -> Hawk a -> Hawk a
mux (c:cs) (t:ts) (e:es) = x : mux cs ts es
  where
  x = if c then t else e
```

This definition uses Haskell's if-then-else construct instead of describing a series of gates that have the same behaviour. Furthermore, this definition is *polymorphic* – the two branches may be Hawk circuits of any type.

The real strength of Hawk is its ability to quickly give executable specifications of complex architecture designs [10]. Having all of Haskell at your disposal, including polymorphism and algebraic data types, can make complex architecture descriptions surprisingly simple.

Hawk does have its drawbacks compared to Lava. Hawk is a *shallow embedding* – in contrast to Lava, there is no Haskell representation of the abstract syntax tree of Hawk circuits. As a result, the only thing you can do directly with Hawk circuits is simulate them. In contrast to Lava, there is no immediate way to generate hardware descriptions or analyse the circuits. (Note, however, that initial studies on transforming and verifying Hawk descriptions have been performed nonetheless [12, 11]. We hope to enable similar reasoning more directly in our future work.)

3 Introducing Chalk

The previous section discussed how Lava is a deep embedding of a structural hardware description language and Hawk is a shallow embedding of a behavioural hardware description language. Chalk aims to combine the best of both worlds: a *deep* embedding of a *behavioural* description language.

Chalk provides several combinators and types to write architecture descriptions. This section outlines how to use these combinators by means of several small examples. The next sections will discuss more substantial examples and the underlying implementation of the library.

The simplest combinator is called `pure`. It has the following type:

```
pure :: a -> Circuit a
```

In other words, it turns any Haskell value into a (constant valued) circuit. A circuit producing a value of type `a` has the Haskell type `Circuit a`. For example, we may want to write the signal that is always `False`:

```
zero :: Circuit Bool
zero = pure False
```

Similarly, if we want to describe a component that inverts a signal, we can also use the `pure` function:

```
inverter :: Circuit (Bool -> Bool)
inverter = pure not
```

Here `not` is Haskell's Boolean negation function. Note that these signals do not need to be first-order – here we have a signal of type `Bool -> Bool`.

Now we may want to 'connect' the `inverter` and `zero` signals. There is a second combinator, written using the infix operator `<*>`, that does just this:

```
<*> :: Circuit (a -> b) -> Circuit a -> Circuit b
```

You may want to think of this combinator as making function application explicit. Using this combinator we can wire together the inverter and `zero` signal to get a signal that is always `one`

```
one :: Circuit Bool
one = inverter <*> zero
```

Using these two basic combinators, it is fairly straightforward to write more complicated signal functions. For instance, the `mux` takes three input signals. Based on its first input signal, it outputs one of its other arguments.

```
mux  ::  Circuit Bool -> Circuit a -> Circuit a -> Circuit a
mux cs ts es = pure cond <*> cs <*> ts <*> es
  where
  cond :: Bool -> a -> a -> a
  cond c t e = if c then t else e
```

Besides the two primitive combinators we have seen so far, we also need some notion of delay:

```
delay :: a -> Circuit a -> Circuit a
```

You can use the `delay` function to write recursive signal functions. For example, you may want to repeatedly apply a function to some initial value:

```
iterator :: a -> Circuit (a -> a) -> Circuit a
iterator x h = delay x (h <*> iterator h x)
```

In the first clock cycle this will return `x`, in the second cycle it will return `h(x)`, in the third cycle it will return `h (h(x))`, and so forth.

Finally, you may want to organise circuits into a hierarchy. There is one last combinator, `component`, that explicitly groups a circuit into one logical unit:

```
component :: String -> Circuit a -> Circuit a
```

The `String` argument gives some name to the component. It need not be unique. As we will see, it can be useful to name subcircuits in order to make the results of circuit analyses easier to understand.

4 A simple microprocessor

Before describing the implementation of Chalk, we will present an example architecture description. Hopefully this will illustrate that Chalk circuits have the same 'behavioural feel' as those written in Hawk.

As a running example, we will take an example from the Hawk literature [10]. The Simple HAwk Microprocessor, or SHAM, only consists of an ALU and a register file. There are four registers. The ALU can add, multiply, or increment integers. We start our specification with the following data types:

```
data Reg = R0 | R1 | R2 | R3

data Cmd = ADD | MUL | INC
```

The `Reg` data type corresponds to the four registers, `R0` through `R3`. The `Cmd` data type describes the three ALU commands. Figure 1 gives a visualisation of the SHAM architecture.

The definition of the ALU is extremely simple. Given a command and a pair of integers, it computes the result of executing that command with those integers as input:

```
alu :: Circuit Cmd -> Circuit (Int, Int) -> Circuit Int
alu cmds xys =
  component "ALU" (pure interpret <*> cmds <*> xys)
    where
    interpret :: Cmd -> (Int, Int) -> Int
    interpret ADD  (x,y)  = x + y
    interpret MUL  (x,y)  = x * y
    interpret INC  (x,_)  = x + 1
```

Note that by convention, `INC` increments the first integer, ignoring the second integer.

The register file is slightly more complicated. The type of register file is:

```
regFile  :: Circuit Reg -> Circuit Int
         -> Circuit Reg -> Circuit Reg
         -> Circuit (Int, Int)
```

The first two arguments contain write information: the register file should write the value of the second argument to the register given by the first argument. The `regFile` should return a pair of the integers stored in the registers specified by the third and fourth argument respectively.

The register file is defined as follows:

```
regFile wr val rd1 rd2 =
  component "RegisterFile" res
    where
    initRegs = (0,0,0,0)
    rf :: Circuit ((Int, Int), RegState)
```

```
    rf = pure step  <*> wr <*> val <*> rd1 <*> rd2
                    <*> delay initRegs regStates
    (res, regStates) = (fmap fst rf, fmap snd rf)

type RegState = (Int, Int, Int, Int)

step  :: Reg -> Int -> Reg -> Reg
      -> RegState -> ((Int, Int), RegState)
step wr x rd1 rd2 regs = ((res1, res2), regs')
  where
  regs' = updateReg (wr,x) regs
  res1 = lookupReg rd1 regs'
  res2 = lookupReg rd2 regs'

updateReg :: (Reg,Int) -> RegState -> RegState
lookupReg :: Reg -> RegState -> Int
```

We initialize all the registers to `0`. We define the register file using an auxiliary circuit definition, `rf`. The `rf` circuit has the same arguments as the register file, but also maintains the current state of the registers.

The bulk of the work is done by the `step` function which is given similar arguments to those of `regFile` – only now we have an additional argument of type `RegState`, representing the current state of the registers. It computes the new state of the registers, `reg'`, that results from performing the required update. The `step` function also looks up the two argument registers `rd1` and `rd2` in this newly computed register state. It returns the results of this read (`res1` and `res2`), together with the new register state `reg'`. We need two auxiliary definitions, `updateReg` and `lookupReg`, to complete the definition of the register file. These definitions are both unremarkable and have been omitted.

Now we can assemble the ALU and register file into a complete microprocessor. The microprocessor receives a sequence of register assignments, such as `R3 <- ADD R2 R1`. It proceeds by looking up the values of the argument registers. The resulting integers are passed to the ALU; the result of the computation is then written to the destination register. The output of the microprocessor is a sequence of values and destination registers, corresponding to the assignments made after every clock cycle.

With this in mind, we define the `sham` circuit as follows:

```
sham  :: Circuit Cmd -> Circuit Reg
      -> Circuit Reg -> Circuit Reg
      -> (Circuit Reg, Circuit Int)
sham cmd dest arg1 arg1 = (dest' , aluOutput')
  where
  aluOutput = alu cmd aluInputs
  aluInputs = regFile dest' aluOutput' arg1 arg2
  dest' = delay R0 dest
```

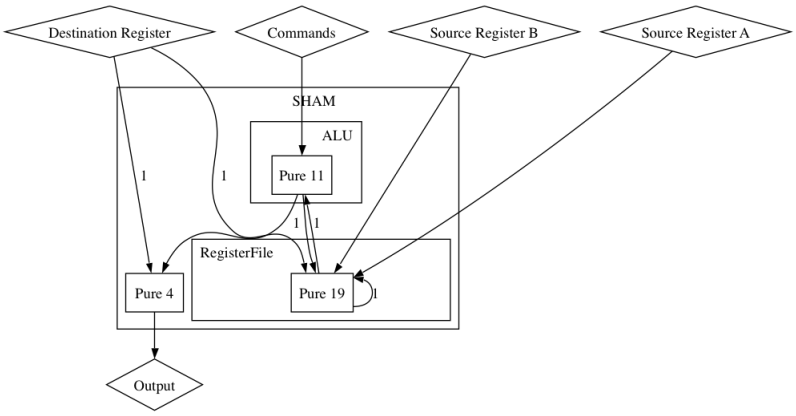


Figure 1: The SHAM microprocessor

```
aluOutput' = delay 0 aluOutput
```

The `aluInputs` are the result of consulting the register file. To prevent a zero-delay loop between the register file and ALU, we explicitly delay the initial input to the register file. The results from the register file are passed to the ALU, together with the commands specified in the `cmd` signal, yielding the ALU outputs. The assignments resulting from the ALU are passed back to the register file, that is updated accordingly.

5 Implementation and analysis

The implementation of the Chalk primitives is very simple. We implement the primitive combinators (`delay`, `pure`, `component`, and `<*>`) by defining the following data types:

```
data CircuitF c a where
  Pure :: a -> Circuit c a
  App :: c (b -> a) -> c b -> Circuit c a
  Delay :: a -> c a -> Circuit c a
  Component :: String -> c a -> Circuit a

data Circuit a where
  Circuit :: Ref (CircuitF Circuit a) -> Circuit a
```

Just as in Lava, we use references to make sharing observable.

The Chalk interface is *applicative* [13]. A *functor* is said to be applicative if we can define instances of the following two classes:

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The applicative instance for our `Circuit` data type are entirely straightforward:

```
instance Applicative Circuit where
  pure x = Circuit (ref (Pure x))
  c1 <*> c2 = Circuit (ref (App c1 c2))
```

Note that much of `Chalk`'s interface is overloaded. In later sections, we will show how to exploit this overloading to assign non-standard interpretations to circuits.

It is worth mentioning that the instance definition that we have given above does not satisfy the laws of applicative functors. For example, the following law will not hold:

```
pure f <*> pure x = pure (f x)
```

The left-hand side is (roughly) equal to `App (Pure f) (Pure x)`, while the right-hand side is `Pure (f x)`. We believe this is not real problem, provided we do not pattern match on the structure of circuits explicitly during their definition. Keeping this data type abstract when circuits are defined should achieve just this.

Simulation Like in Hawk, we can still simulate our circuits, producing an infinite list of values. The `simulate` function takes a signal function and returns an infinite list corresponding to the values produced by the circuit at every clock cycle.

```
simulate :: Circuit a -> [a]
simulate (Circuit c) = sim (deref c)
  where
  sim :: CircuitF Circuit a -> [a]
  sim (Pure x) = repeat x
  sim (App f x) = zipWith id (simulate f) (simulate x)
  sim (Delay x xs) = x : simulate xs
  sim (Component nm c) = simulate c
```

In the case of `Pure x` constructor, we construct the infinite list where every element is equal to `x`. To handle application we simulate both the 'function' circuit and the 'argument' circuit, yielding an infinite list of functions and an infinite list of arguments. We return the result of point-wise applying every function to its corresponding argument. Simulating `Delay x c` adds `x` to the head of the list obtained by simulating `c`. Finally, to simulate a component, we ignore the `String` and simulate its underlying circuit.

Readers already familiar with applicative functors may recognise the `simulate` function as a type class morphism [6] – although in this instance perhaps an applicative functor homomorphism is more accurate – mapping the `Circuit` applicative functor into streams. The `pure` and `<*>` definitions of the `Circuit` type are mapped to the corresponding `pure` and `<*>` definitions for infinite lists.

This is not the only such applicative functor homomorphism. For instance, the applicative homomorphism from Chalk circuits to the identity applicative functor computes the first value produced by a circuit:

```
first :: Circuit a -> a
first (Circuit c) = sim (deref c)
  where
  sim :: CircuitF Circuit a -> [a]
  sim (Pure x) = x
  sim (App f x) = (first f) (first x)
  sim (Delay x xs) = x
  sim (Component nm c) = first c
```

Pipeline depth In contrast to Hawk, we can now define *non-standard interpretations* of our circuits. A simple example of such an interpretation is to compute the maximum number of latches on all possible evaluation paths between the input and output of a circuit. This example is inspired by a similar analysis by Luk [9].

To count the maximum number of latches, we traverse our `Circuit` data type. As this data type may contain loops, we need to maintain a list of all the nodes we have already visited during the traversal. Every time we encounter a new node, we add it to the list of nodes we have visited. If we encounter a node we have visited previously, we simple return `0`. On the other hand, if we encounter a new node, we continue our traversal.

```
delaySim :: Circuit a -> Int
delaySim c = evalState (sim c) []
  where
  sim :: Circuit a -> State [Unique] Int
  sim (Circuit c) = do
    visited <- get
    if unique r `elem` visited
      then return 0
      else  modify (r:) >>
            step (deref r)
  step :: CircuitF Circuit a -> State [Unique] Int
  step (Pure x) = return 0
  step (App f x) = liftM2 max (sim f) (sim x)
  step (Delay x c) = fmap (1 +) (sim c)
  step (Component nm c) = sim c
  step (Input nm) = return 0
```

The most important computation happens in the `step` function, where we pattern match on the possible constructors of the `CircuitF` data type. A `Pure` constructor has no delay. In the case of the `App` constructor, we take the maximum of the delay of the two subcircuits. In the branch for the `Delay` constructor, we increment the maximum delay.

This can be easily generalised to any monoid. For example, we may want to collect the unique identifiers of all the nodes along the critical path.

This is once again a homomorphism between applicative functors. To see this, consider that every monoid is a phantom applicative functor [13]. The `delaySim` function maps `pure` to `0` and `<*>` to `max`.

Activity analysis When estimating the power usage of a high-level architecture, one of the key parameters is the *switching frequency*, the probability that a signal will change from one clock cycle to the next. Using Chalk we can define a simulation that measures just that.

Although we could simulate a circuit and inspect adjacent elements in the stream of values, there is an alternative. Using the `first` function we saw previously, we can delay an arbitrary circuit by a single clock cycle:

```
delayed :: Circuit a -> Circuit a
delayed c = delay (first c) c
```

To monitor the switching frequency, we simply compare the original circuit with its delayed counterpart:

```
activity :: (a -> a -> b) -> Circuit a -> Circuit b
activity m c = pure m <*> delayed c <*> c
```

Here we have abstracted over the exact function that compares the values that a circuit produces.

As a small example, we will show how to do a simple activity analysis on the register file of the simple microprocessor defined in Section 4. The real work is done in the definition of the `rf` circuit of type

```
Circuit ((Int, Int), RegState)
```

There are several metrics in which we could be interested. The simplest analysis only measures if the signal has changed or not:

```
hasChanged :: Circuit Bool
hasChanged = activity (==) rf
```

This is much too coarse to be useful. A much better measure would count the number of bits that has changed. To do so, we need to define the following auxiliary function:

```
bitDiff :: Int -> Int -> Int
bitDiff x y = count (complement (xor (bit x) (bit y)))
```

The `bitDiff` function converts its two integer arguments to a 32-bit word. It proceeds by using an `xor` and `complement` operation to compute a word whose only non-zero entries correspond to changed bits. Finally, it counts the number of one bits in the resulting word. By applying this function to each of the integers returned by the `rf` circuit, we can get a much more refined activity analysis. Of course, we could also choose to measure only changes in the registers, and not take changes to the other signals into account, such as the integers that result from looking up the values stored in the register file.

6 Design exploration

The implementation of the ALU in the microprocessor from Section 4 was entirely straightforward. Other alternatives are certainly possible. In this section, we show how to use Chalk to explore and evaluate different design alternatives.

Suppose we expect most numbers handled by the ALU to be reasonably small. In that case, it might be worthwhile to define two different multipliers: one that can only multiply small numbers and another that can multiply larger numbers, but consumes more power. When is such a design preferable?

To make an informed estimate, we need to keep track of how often each multiplier is used. One way to do so is by defining the following types:

```
data Ticked a = T {val :: a, cost :: Double}

type TCircuit a = Circuit (Ticked a)
```

Instead of just computing the values at every clock cycle, the `TCircuit` type also computes an estimated cost for every cycle.

The central observation we now make is that the `TCircuit` type is still an applicative functor. To see this, define the applicative instance for the `Ticked` type and observe that applicative functors are closed under composition:

```
instance Functor Ticked where
  fmap f (T x cost) = T (f x) cost

instance Applicative Ticked where
  pure x = T x 0.0
  (T f c1) <*> (T x c2) = T (f x) (c1 + c2)
```

We can now define a variation of the `pure` operator that introduces non-zero costs:

```
costed :: Double -> a -> TSignal a
costed i x = pure (T x i)
```

Using these ingredients, we could define our new, clever multiplier as follows:

```
multiplier :: TCircuit (Int, Int) -> TCircuit Int
multiplier xys =
  mux (sizeTest xys) (cheapMul xys) (dearMul xys)

sizeTest ::  TCircuit (Int, Int) -> TCircuit Bool
sizeTest xys = costed 0.1 (\(x,y) ->  x < threshold &&
                                      y < threshold) <*> xys

cheapMul, dearMul :: TCircuit (Int, Int) -> TCircuit Int
cheapmul xys = costed 0.2 (uncurry (*)) <*> xys
dearMul xys = costed 0.5 (uncurry (*)) <*> xys
```

Here we have chosen fixed constants for the cost of the multipliers and control logic. This circuit now introduces a bit of control logic to decide which multiplier to use. Depending on the argument values, the mux returns the result computed by one of the two multipliers.

There is an error in this definition. The cost per clock cycle is constant: regardless of which branch is chosen, the cost of both branches is still summed. To solve this, we need to define a variation of the `mux` we have seen previously:

```
smartMux :: Circuit (Ticked Bool) -> Circuit (Ticked a)
 -> Circuit (Ticked a) -> Circuit (Ticked a)
smartMux bs ts es = pure f <*> bs <*> ts <*> es
  where
  f :: Ticked Bool -> Ticked a -> Ticked a -> Ticked a
  f (T True c) t e = T (tval t) (c + cost t)
  f (T False c) t e = T (tval e) (c + cost e)
```

This version of the `mux` only sums the costs of the branches that are chosen. By simulating a multiplier using this new multiplexer, we can evaluate the power cost of this alternative design.

Discussion

Although we believe the design of Chalk is sound in principle, we believe we can sharpen these ideas further with specific examples. In particular, we would like to study how Chalk's applicative interface can be used during the design of more realistic architectures.

Acknowledgements

This work was funded by a grant from Intel to the Functional Programming Group at Chalmers.

References

[1] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, 1998.

[2] K. Claessen and D. Sands. Observable sharing for functional circuit description. In P.S. Thiagarajan and R. Yap, editors, *Advances in Computing Science ASIAN'99; 5th Asian Computing Science Conference*, volume 1742 of *Lecture Notes in Computer Science*, pages 62–73. Springer-Verlag, 1999. Extended Version Available.

[3] K. Claessen, M. Sheeran, and S. Singh. The Design and Verification of a Sorter Core. In *Correct Hardware Design and Verification Methods, CHARME*, volume 2144 of *Lecture Notes in Computer Science*, pages 355–369. Springer, 2001.

[4] Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Department of Computer Science, Chalmers University of Technology, 2001.

[5] B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in Hawk. *Formal Techniques for Hardware and Hardware-like Systems. Marstrand, Sweden*, 1998.

[6] Conal Elliott. Denotational design with type class morphisms. Technical Report 2009-1, LambdaPix, 2009.

[7] Andy Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, Sep 2009.

[8] J. Launchbury, J.R. Lewis, and B. Cook. On embedding a microarchitectural design language within Haskell. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 60–69. ACM New York, NY, USA, 1999.

[9] W. Luk. Analysing parametrised designs by non-standard interpretation. In *Application Specific Array Processors, 1990. Proceedings of the International Conference on*, pages 133–144, 1990.

[10] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *Computer Languages, 1998. Proceedings. 1998 International Conference on*, pages 90–101, 1998.

[11] John Matthews. *Algebraic Specification and Verification of Processor Microarchitectures*. PhD thesis, Oregon Graduate Institute, 2000.

[12] John Matthews and John Launchbury. Elementary Microarchitecture Algebra. In *Int. Conf. on Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*. Springer, 1999.

[13] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2007.

[14] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[15] Moshe Vardi. Formal Techniques for System- Level Verification, invited tutotial at Int. Conf. on Formal Methods in Computer Aided Design (FMCAD), 2009. url: http://fmv.jku.at/fmcad09/slides/vardi.pdf.