Coherent Minimisation: Towards efficient tamper-proof compilation

Dan R. Ghica
University of Birmingham
d.r.ghica@cs.bham.ac.uk

Zaid Al-Zobaidi
University of Birmingham
z.k.ibrahim@cs.bham.ac.uk

Automata representing game-semantic models of programs are meant to operate in environments whose input-output behaviour is constrained by the *rules of a game*. This can lead to a notion of equivalence between states which is weaker than the conventional notion of bisimulation, since not all actions are available to the environment. An environment which attempts to break the rules of the game is, effectively, mounting a *low-level attack* against a system. In this paper we show how (and why) to enforce game rules in games-based hardware synthesis and how to use this weaker notion of equivalence, called *coherent equivalence*, to aggressively minimise automata.

1 Introduction

Computer security 'exploits' take advantages of mistakes in programs, called 'vulnerabilities', to cause unintended behaviour to occur on a computing device. Exploits are most commonly low-level attacks that violate the abstractions of the programming language to create behaviour inexpressible in the language itself. Such attacks are possible because lower-level languages ('machine code') are less constrained behaviourally than higher-level languages, so a run-time system, when confronted with executable code, cannot tell whether that code is the result of a legitimately compiled program or whether it contains behaviours deemed 'illegal'. Restricting the behaviour of machine code is the essence of 'tamper-resistant' compilation, and it can be achieved in various way: sand-boxing the code to prevent unauthorised access to memory, randomising the memory layout so that code cannot 'guess' where certain data is stored even if it has physical access to it [Shacham et al., 2004] or monitoring the control flow in a program to ensure that no arbitrary jumping occurs [Abadi et al., 2009].

A compiler and runtime system that can detect and enforce machine code behaviour so that it satisfies all the abstraction of the higher-level programming language would be, effectively, a 'fully abstract' compilation and execution environment offering the maximum level of tamper resistance: 'tamper-proof' compilation [Abadi, 1998]. In a general-purpose system this is perhaps impossible to achieve in a practical way. However, we will show how it is achievable in 'higher-level synthesis' (also known as 'hardware compilation'), the automatic synthesis of special-purpose digital circuits from programs written in conventional programming languages, using game-semantic models. We will then further show how, once guarantees on the behaviour of the environment can be effectively enforced, the automata representation of the game semantic models can be aggressively optimised. Essentially, an environment which cannot perform arbitrary actions cannot distinguish between as many states in a transition system as the unrestricted system. This is a new notion of equivalence, which we call 'coherent equivalence'.

M. Carbone, I. Lanese, A. Silva, A. Sokolova (Eds.):
5th Interaction and Concurrency Experience (ICE'12).
EPTCS 104, 2012, pp. 83–98, doi:10.4204/EPTCS.104.8

2 The GoS hardware compiler

The *Geometry of Synthesis* (GoS)[1] compiler [Ghica, 2007, Ghica and Smith, 2010, 2011, Ghica et al., 2011] produces (VHDL) descriptions of digital circuits from a conventional functional-imperative programming language. The circuits produced by the compiler are a concrete representation of the game-semantic model of the language [Ghica et al., 2006].

2.1 The language Verity

The source language of GoS is called Verity, and it is an Algol-like language in Reynolds [1981] sense. It represents a combination of the simply-typed (call-by-name) lambda calculus with the simple imperative language of while loops. Additionally, Verity has primitives for parallel execution of commands.

The combination of call-by-name and local store, although made popular in Algol 60, fell out of favour as languages with global store (and more generally, global effects) and call-by-reference (C), call-by-value (ML) and call-by-need (Haskell) became prevalent for reasons of convenience and efficiency.

However, in the case of hardware compilation the perceived disadvantages of Algol yield unexpected benefits:

Local store. The notion of global store does not fit the way memory is used in a circuit. In a circuit, stateful elements are scattered throughout the design, wherever needed. There is no need to bring them all together in a single global memory because this would be inefficient in multiple ways. Managing access to this global memory would require complex control elements which would be costly in energy, footprint and latency. It would also constitute a bottleneck for concurrency. Note that the lack of language support for global store does not mean that Verity cannot deal with programs which access off-chip RAM. It only means that such access needs to be programmed explicitly and used via library calls. This is an advantage because RAM controllers can exploit the precise memory hierarchy of the device in a way that generic language support cannot.

Call by name. Verity is a functional programming language, and it is well known that managing closures is one of the great potential sources of inefficiency in compilers. Dealing with memory management for closures in functional hardware synthesis raises additional difficulties because all usage of memory in a circuit must be bounded at synthesis time. This makes it impossible to support higher order functions [Mycroft and Sharp, 2000]. However, call-by-name closures require less storage, because of constant re-evaluation of the thunks. This provides an elegant, albeit somewhat fortuitous, solution to the problem of memory management for closures.

The syntax of the language is standard for an Algol-like language. Here we only provide two examples, to give a flavour of the language. First, a naive and highly inefficient implementation of a Fibonacci number calculator:

```
let fbn = (fix \f.\x. if x<1 then 0 else if x<2 then 1 else f(x-1)+f(x-2)) in fbn(5)
```

Second, an efficient implementation using memoisation:

```
  new mem(128) in
  new i := 0 in
  while !i < 128 do {mem(!i) := 0; i := !i + 1};
  let fbv = \l.(fix \fib.\a.\n.
    new n1 in new n2 in new n3 in new n4 in
```

[1] http://veritygos.org

```
n1 := n;
if !n1 < 2 then 1
else if !a(!n1) > 0 then !a(!n1)
else (n2 := fib(a)(!n1 - 2);
      n3 := fib(a)(!n1 - 1);
      n4 := !n2 + !n3;
      a(!n1) := !n4;
      !n4))(mem)(1) in fbv(5)
```

The examples above should serve to convince that Verity is a conventional programming language with no hardware-specific primitives or constructs, although the type system has several subtle restrictions to ensure that the game-semantic models are finite-state, as discussed below.

2.2 Theoretical and methodological background

Compared to other higher-level academic or industrial synthesis tools the emphasis of GoS is on correct and efficient support for the functional infrastructure of the language. Some restrictions are unavoidable because of the finite-state nature of the digital circuits, and the aim of GoS is to impose no additional restrictions. It is a key methodological principle of the GoS project that mature support for functions is essential in the pursuit of a useful and useable compiler. The theory behind Verity-GoS and the methodological considerations are explained at some length by Ghica [2011].

At the most abstract level, digital circuits can be seen as topological diagrams of boxes and wires. The diagrams are topological (rather than geometrical) because in design we often wish to abstract from the size and length of the connectors, and from the precise placement of the components; such low-level matters are usually sorted out algorithmically by electronic design tools. One economical, elegant and mathematically canonical representation of diagrams is using combinators, which form a mathematical structure called a *compact closed category* [Kelly and Laplaza, 1980]. What is particularly useful about such a category in our context is that it can also describe a canonical model for a higher-order programming language with affine typing. This means that that the higher-order structure of the language is reflected directly in the diagrammatic structure of the circuit, which further means that abstraction and application can be represented with zero overhead.

From a practical point of view, the key consequence of the GoS approach is that compiling a Verity program produces a circuit with an interface determined by the type signature of the program. It is conventional to write the type of a program as a *judgement* $x_1 : T_1, \ldots, x_n : T_n \vdash P : T$, which says that program P is well-typed of type T and has free identifiers x_i of type T_i.

Each type corresponds to a circuit interface, defined as a list of ports, each defined by data bit-width and a polarity. Every port has a default one-bit control component. For example we write an interface with n-bit input and m-bit output as $I = (+n, -m)$. More complex interfaces can be defined from simpler ones using concatenation $I_1 \otimes I_2$ and polarity reversal $I^\bullet = \mathsf{map}\,(\lambda x.{-x})I$. If a port has only control and no data we write it as $+0$ or -0, depending of polarity. Note that obviously $+0 \neq -0$ in this notation!

An interface for type T is written as $[\![T]\!]$, defined as follows:

$$[\![\mathsf{com}]\!] = (+0, -0) \qquad [\![\mathsf{exp}]\!] = (+0, -n) \qquad [\![\mathsf{var}]\!] = (+n, -0, +0, -n)$$
$$[\![T \times T]\!] = [\![T]\!] \otimes [\![T']\!] \qquad [\![T \to T']\!] = [\![T]\!]^\bullet \otimes [\![T']\!].$$

The interface for commands com has two control ports, an input for starting execution and an output for reporting termination. The interface for integer expressions exp has an input control for starting evaluation and data output for reporting the value. Assignables var have data input for a write request and control output for acknowledgment, and control input for a read request along with data output for the value. The tensor is a disjoint sum of the ports on the two interfaces while the arrow is like the tensor, but with a polarity-reversal of the ports occurring in the contra-variant position, as illustrated in the example below.

Diagrammatically, a list will correspond to ports from left-to-right and from top-to-bottom. We indicate ports of zero width (only the control bit) by a thin line and ports of width n by an additional thicker line (the data part). For example a circuit of interface $[\![\mathsf{com} \to \mathsf{com}]\!] = (-0,+0,+0,-0)$ can be written in any of these two ways in Fig. 1.

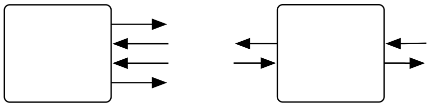

Figure 1: Equivalent representation of ports

The unit-width ports are used to transmit *events*, represented as the value of the port being held high for one clock cycle. The n-width ports correspond to data lines. We will work under the assumption that the event on the unit port is a control signal indicating the data on the data line is valid.

The significant restriction that makes support for functions so simple is that the type system is *affine*, which means that in function application the function and the argument cannot share free identifiers. This is an important restriction which has a major impact on the expressiveness of the language. For once, it is incompatible with imperative programming, in which variables naming memory locations must be reused in order to be read and written.

In order to overcome this restriction we carefully add variable sharing to the programming language, using a type system called *Syntactic control of concurrency* (SCC) [Ghica et al., 2006], which is based on Reynolds's *Syntactic control of interference* [Reynolds, 1978, O'Hearn et al., 1999]. The idea is to allow sharing of variables in product formation, but not in function application. Imperative sequential operations are then given uncurried type, so they can reuse variables. For example, the term `x:=!x+1` can be written, using a functionalised pre-fix notation as `assign (x, add(deref(x), 1))`. Assignment has type `assign : var * int -> com` and thus can share variables between its two arguments. An extra benefit of this type system is that by giving parallel command composition a curried type `par : com -> com -> com` it makes it impossible to have race conditions in the programming language, since the two arguments can never share identifiers. The program `c || c` (also written as `par c c`) does not type-check.

Unlike functions, variable sharing does not arise automatically out of the algebraic structure of the diagrammatic model. It needs to be implemented. Categorical considerations are nevertheless helpful in providing a family of equational specifications, corresponding to the notion of *Cartesian product*, which establish that variable sharing is correctly implemented (because *contraction* in the syntax corresponds to Cartesian product in the semantics).

The conditions required for the correct implementation of product in GoS amount to an input-output *protocol* which all synthesised circuits must satisfy in order to compose properly. In the implementation, this protocol amounts to a simple *bus protocol* needed for the correct time-multiplexed sharing of sequentially used circuits. They are formally described by Ghica [2007].

Diagrammatically, sharing is implemented by specialised circuits which correspond to the *diagonal* in the Cartesian category of circuits. For example, the term $\mathtt{x}:\mathsf{var} \vdash \mathtt{x} := !\mathtt{x}+\mathtt{1} : \mathsf{com}$ corresponds to the diagram sketched in Fig. 2, with the diagonal labelled Δ used to share access to variable `x`.

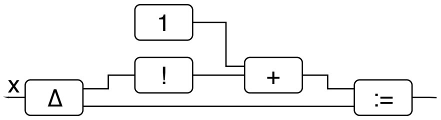

Figure 2: The diagonal circuit Δ

To give an interactive, event based semantics to the imperative constants of Verity we use the *game semantics* of the language, which is formulated in this style. The interpretation of constants is standard in game semantics and will be not detailed here. To give a flavour of the implementation we show the iterator only (for convenience we have marked the top level ports, the ports of the loop guard and the ports of the body of the loop) in Fig. 3, where OR joins two signals, T is a multiplexer and D is a unitary delay. This circuit can be realised either asynchronously [Ghica and Smith, 2010] or synchronously [Ghica and Menaa, 2011]. Its input-output behaviour is:

- receive an input signal from the top level;
- propagate the input signal to the guard;
- receive an input signal from the guard when it is ready;
- use the data line from the guard in multiplexer T to
  - propagate the signal back out to the top level if the guard is false;
  - propagate the signal out to the loop body if the guard is true;
- use the termination acknowledgement from the body to trigger an evaluation of the guard again, which will cause the process to iterate.

To have an expressive and convenient programming language the restrictions of the SCI system are still undesirable. They can be however avoided, to a great extent, using program analysis and transformation as described by Ghica and Smith [2011]. Finally, recursion can be implemented in the same framework, subject to several minor restrictions [Ghica et al., 2011]. We do not describe these features in detail here as the complications they introduce are not directly relevant to tamper-proof compilation or coherent minimisation. The techniques described apply to these features as well.

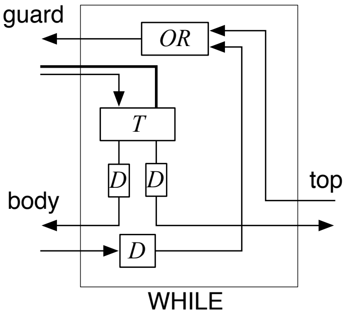

Figure 3: Iterator

The compilation process is compositional and it allows the synthesis of circuits corresponding to *open terms*. Compositionality in the compiler means that we have immediate support for *separate compilation*. This is essential for having compiler support for (pre-compiled) libraries but, most importantly, for supporting *foreign function interfaces* (FFI). Through the FFI we can interact with system-specific functionality which can be implemented outside of the programming language, using a conventional HDL. This is important as useful as low-level drivers for peripherals are written in HDL, but from the language we prefer to interact with them via function calls.

Separate compilation and foreign function interface play a great role in making a compiler useful. However, interfacing with circuits produced outside the compiler exposes the synthesised code to low-level attacks, because such circuits cannot be assumed to satisfy the input-output protocol which synthesised circuits both satisfy and assume in order to operate properly.

3 Protocols and low-level attacks

Tamper-proof compilation is relative to whatever notion of tampering we consider possible on pragmatic considerations, so a circuit is tamper resistant to the same extent as its physical substrate is. In other words, the high-level constraints needed for the proper operation of synthesised circuits cannot be violated without violating the underlying *physical* constraints of the circuit. Note that some FPGA devices, such as Altera's Cyclone III LS, have physical anti-tamper layers which include special protection for the programming ports and redundancy checks. [2]

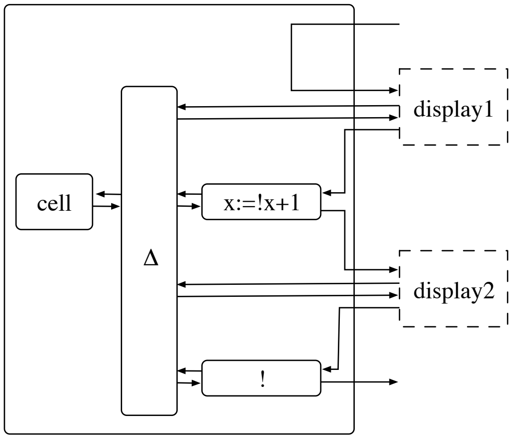

Figure 4: Example of synthesised program using the FFI

Example of physical attacks on circuits involve over-heating or over-clocking the circuit so that it behaves erroneously on an electronic level. Also of a physical nature are observations against the temperature or energy consumption of the circuit as well as timing its responses. We provide no means of resistance against such attacks, but only against attackers which provide inputs and observe outputs at the ports only, within the normally accepted parameters of operation of the device.

Let us illustrate the problem of low-level attacks with a very simple example. Consider a program which interacts with the external environment using functions `display1, display2:exp->com` to drive, for example two segmented LED displays:

```
new x := 0 in display1(!x);
x:=!x+1; display2(!x); !x
```

According to the semantics of Verity this program should first display the value 0 on device 1, then display value 1 on device 2, then return the value 1 to the top level. The high-level diagram of the concrete synthesised circuit in Fig. 4.

The circuit labelled `x:=!x+1` is the incrementer in Fig. 2. The circuit `cell` is a register storing the value of the variable and `!` is a de-referencer. The diagonal Δ shares access to `x` for the display functions, the incrementer and the final dereferencing. The dotted boxes are the implementations for the display functions, realised in HDL and visible from the programming language through the FFI. In order to simplify the drawing of the circuit the data lines are implicit where required; we only show the control lines.

Let us label the ports of the synthesised circuit top-to-bottom starting with 0 for the top-level port requesting execution and ending with 9 for the top-level port reporting the result. The correct interaction in which such a circuit is involved proceeds as follows:

1. receive a top-level input request on port 0 to start execution;
2. request external function `display1` to execute using port 1;
3. using port 2 function `display1` may inquire what the value of its argument is, zero or more times;
4. using port 3 the circuit will always provide value 0 as response, the state of `cell`;
5. eventually `display1` will terminate, reporting termination on port 4;
6. upon incrementing the register the circuit will use port 5 to request `display2` to execute;
7. using port 6 function `display2` may inquire what the value of its argument is, zero or more times;
8. using port 7 the circuit will always provide value 1 as response, the new state of `cell`;
9. eventually `display2` will terminate, reporting termination on port 8;
10. the circuit will report final value 1 on port 9.

However, the environment, consisting of the top level and the two display functions can violate the input-output behavioural assumptions of synthesised code. Consider the environment in Fig. 5. The transaction in which the circuit is now involved is:

1. receive an top-level input request on port 0 to start execution;

[2] `http://www.altera.com/corporate/news_room/releases/2009/products/nr-ciii_ls.html`

2. request external function `display1` to execute using port 1;
3. `display2` (illegally) reports termination on port 8;
4. the circuit will report final value 0, the initial state of the register, on port 9.

The unused ports are marked as black squares for emphasis.

The low-level attack violates the input-output behaviour of synthesised circuits, the essential features characterising correctly compiled Verity programs, and it causes the program to produce the wrong value 0 instead of the expected value 1. It is easy to see that the environment can manipulate the inputs and output to the two display functions so that the register `cell` and the final result can have whatever value is desired by the attacker. Obviously, from a security point of view such tampering unacceptable as it can lead to a wide range of attacks against data integrity.

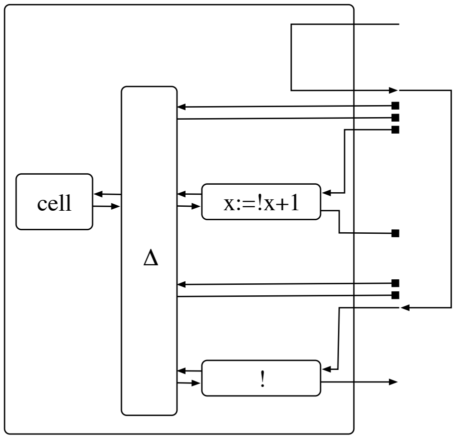

Figure 5: Environment which breaks language abstraction

4 Enforcing programming language abstractions

Low level attacks are possible when the system can perform actions that break the programming language abstractions. But can we prevent the system from performing such actions? In this particular case the answer is positive. Programming language abstractions are reflected into the structure of the synthesised circuits in two ways: statically, as the input and output ports of the circuit, or dynamically, as the input-output behaviour of the environment in which the circuit operates.

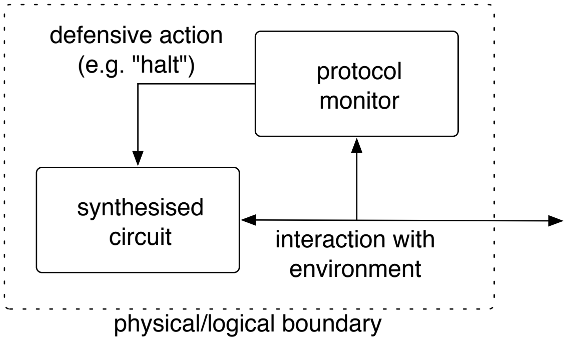

Figure 6: Architecture of a tamper-proof circuit

The static port structure cannot be violated, but the dynamic behaviour of the environment can violate the protocol-like semantics of the language. We can restrict the behaviour of the environment to legal traces by taking advantage of several facts [Ghica et al., 2006]. First, we know what the *fully abstract* model of Verity is. A fully abstract model is a correct and complete characterisation of all the traces that can be generated by a synthesised Verity program. Second, the fully abstract model of Verity has a finite-state automaton representation for any type signature. Finally, the low-latency representation of the model, which is used for hardware synthesis, also has a finite-state representation [Ghica and Menaa, 2011].

The three observations above mean that all the legal interactions between a circuit and its environment can be described by a finite state machine, therefore by a digital circuit. In order to achieve tamper-proof-ness a synthesised circuit must not interact with its environment directly, but the interaction must be mediated by a monitor which will detect any illegal interactions and take appropriate actions if such illegal interactions occur. As illegal interactions indicate tampering attempts, the appropriate actions may be reset, halt, intentionally erratic behaviour or even destroying the circuit, depending on the level of protection and sensitivity desired. Schematically, the tamper proof circuit will look like in Fig. 6.

The precise specification of the legal interaction protocol and its low-latency asynchronous specification used for hardware synthesis are given by Ghica and Menaa [2011] and will not be repeated here. For illustration we will detail the example of the previous section. The program has *signature* $\texttt{display1:exp}\to\texttt{com}, \texttt{display2:exp}\to\texttt{com} \vdash \texttt{M:exp}$, where *M* is the program. To wit, the program uses two non-locally defined functions, `display1, display2` which are procedures taking integer expressions as arguments, and it has type expression. We write this signature as

$$(\texttt{exp}_1 \to \texttt{com}_2) \to (\texttt{exp}_3 \to \texttt{com}_4) \to \texttt{exp}_5.$$

The game-semantic model for Verity stipulates that all legal traces in which the program can be involved have to have a form described by the Mealy machine in Fig. 7, which is dependent on the signature only:

Intuitively, the reading of the protocol is this:

1. The environment may start executing the program (q5)
2. The program may terminate immediately (d5) or may ask for either of display functions to be evaluate (r2 or r4)
3. If a function was called by the program, it is allowed to either return immediately (d2 or d4) or it can evaluate its argument (q1 or q3) any number of times.
4. The program must respond to a request to provide the argument of the function (n1 or n3).

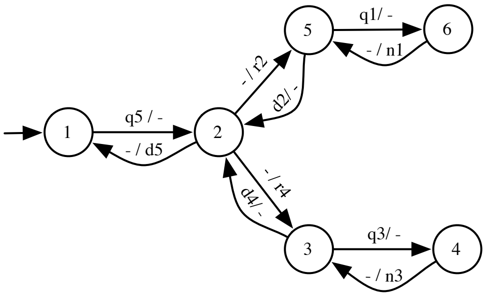

Figure 7: A game-semantic protocol, automaton representation

The protocol above is *asynchronous*, and for the purpose of hardware synthesis we use a low-latency representation called *round-abstraction* [Ghica and Menaa, 2011], allowing multiple inputs and outputs on the same transition while avoiding deadlocks and race conditions, as in Fig. 8.

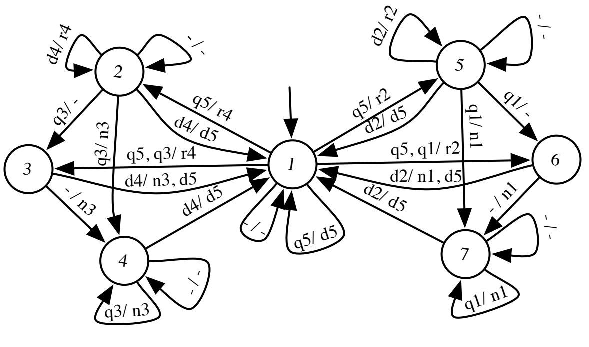

Figure 8: Synchronous representation of a protocol

Producing a circuit representation of these finite-state machines is standard. Let us call this circuit M (monitor). The tamper-proof version of the circuit from the previous section is given in Fig. 9. The input lines to M, drawn in a lighter colour, are the interactions with the environment, and the output lines from M trigger the reset lines of the component sub-circuits. Note that the monitor M can also be placed in series (rather than in parallel) with the monitored circuit so that tampering signals never reach it. This is only marginally safer but comes at a cost of increased latency.

With the tamper-proof version of the circuit the attacks of the previous section trigger resets (or whatever other defensive anti-tampering behaviour is desired in the concrete implementation) and are rendered ineffective.

5 Coherent equivalence and minimisation

By constraining the interaction between the circuit and the environment, the tamper-proof compiler makes possible more aggressive optimisations. In conventional automata optimisation two states are considered equivalent if they are not distinguishable by any environment; this concept is formalised by *bisimulation*. Bisimilar states can be identified, leading to optimised automata with fewer number of states.

In the case of the tamper-proof compiler, the interaction between the circuit (an automaton) and the environment is monitored so that only certain interactions are permitted. This makes the environment less discriminating, leading to more states being equivalent. In the limit case of an empty protocol (no interactions are allowed) for example, all states of an automaton are equivalent and can be identified. Conversely, in the other limiting example of the protocol that permits all interactions, this notion of equivalence reduces to conventional bisimulation.

In this section we formalise the concept of *coherent equivalence* for finite-state transducers and we formulate (the obvious) minimisation algorithm based on this notion of equivalence, proving its correctness. In the following section we discuss the way it is incorporated in the Verity/GoS compiler. A similar notion of coherent bisimulation could be formulated but it is more complex; since we are only interested in finite traces we will not pursue this theoretical development here.

Let a *transducer signature* A be a pair of finite sets of labels (I_A, O_A), the input and the output ports of a transducer, respectively. By $A^\bullet$ we mean signature A with input-output port polarities reversed. We call a (possibly empty) subset of A a *round*, and an occurrence of a label in a round an *event*. Let a *synchronous trace* over signature A be a sequence of rounds. Let ε be the *empty trace (empty sequence).*

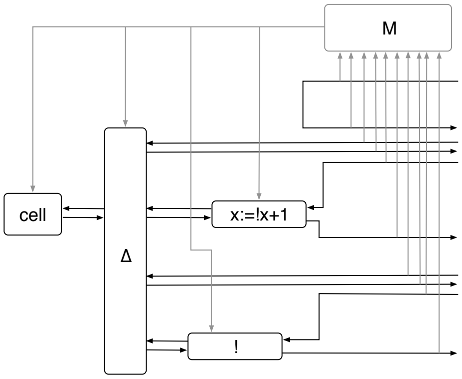

Figure 9: Tamper-proof compiled circuit with monitor

Definition 1 (Transducer). *A* transducer *with signature A, written $T : A$ is a triple $\langle S, s, \delta\rangle$ where S is a set of states, $s^0 \in S$ is the initial state and $\delta \subseteq S \times \mathscr{P}(A) \times S$ is a transition relation.*

We often write S_T to mean "the set of states S of transducer T", etc. We extend the transition relation δ to traces $\hat{\delta} \subseteq S \times \mathscr{P}(A)^* \times S$ in the usual way:

- for any state $q \in S$, $(q, \varepsilon, q) \subseteq \hat{\delta}$
- For any round $V \subseteq A$, states $s, s' \in S$ and trace $t \in \mathscr{P}(A)^*$, $(s, t \cdot V, s') \in \hat{\delta}$ if and only if there is a state $s'' \in S$ such that $(s, t, s'') \in \hat{\delta}$ and $(s'', V, s') \in \delta$.

Let $[\![T : A]\!]$ be the set of traces of a transducer, $[\![T : A]\!] \stackrel{\text{def}}{=} \{t \in \mathscr{P}(A)^* \mid \exists s \in S.(s^0, t, s) \in \hat{\delta}\}$. Given a state $s \in S$ for a transducer, it is useful to know the set of witness traces which can reach this state in the transducer, $\omega_T(s) \stackrel{\text{def}}{=} \{t \in [\![T]\!] \mid (s^0, t, s) \in \hat{\delta}\}$.

We say that two transducers with the same signature $T, T' : A$ are *equivalent* if and only if they have the same set of traces, $T \equiv_A T'$ iff $[\![T]\!] = [\![T']\!]$. This is the conventional notion of equivalence, and it is preserved by all common operations on transducers, such as intersection, product, etc.

Definition 2 (Intersection). *Given transducers $T, T' : A$, we define $T \cap T' \stackrel{def}{=} \langle S_T \times S_{T'}, (s_T^0, s_{T'}^0), \delta\rangle$, with $((s_1, s_1'), V, (s_2, s_2')) \in \delta$ iff $(s_1, V, s_2) \in \delta_T$ and $(s_1', V, s_2') \in \delta_{T'}$.*

This definition is sound in the sense that for any transducers $T, T' : A$ we have that $[\![T \cap T']\!] = [\![T]\!] \cap [\![T']\!]$. The proof is immediate. We define a laxer notion of equivalence motivated by a restricted set of interactions between the transducer and its environment. Let us define this restricted set of interactions by $P : A$, also a transducer, which we call a *protocol*.

Definition 3 (Coherent equivalence)**.** *We say that transducers $T, T' : A$ are* coherently equivalent *under protocol $P : A$, written $T \equiv_A^P T'$ if and only if $T \cap P \equiv_A T' \cap P$.*

The main definition in this section identifies when two states are coherent, i.e. equivalent under a restricted set of observations.

Definition 4 (Coherent state simulation)**.** *Given a transducer $T : A$, a protocol $P : A$ and a relation $R \subseteq S_T \times S_T$, we say that R is a* coherent simulation*, iff, for any $(s', s'') \in R$ the following two conditions hold:*

1. *For any transition $(s'', V, r'') \in \delta_T$, there exists a transition $(s', V, r') \in \delta_T$ such that $(r', r'') \in R$;*
2. *For any round $V \subseteq A$, such that there is no transition from s'' labelled with V,*
    - (a) *either there is no transition from s' labelled with V,*
    - (b) *or $\omega_T(s'') \cdot V \cap [\![P]\!] = \emptyset$.*

If $(s', s'') \in R$ for some protocol P we write $s' \frown^P s''$.

Definition 5 (Coherent state equivalence)**.** *Given transducer $T : A$, protocol $P : A$ and states $s', s'' \in S_T$ we say they are* coherently equivalent*, written $s' \asymp_A^P s''$ if and only if $s' \frown_A^P s''$ and $s'' \frown_A^P s'$.*

Given a function $f : A \to B$ we denote by $(f \mid x \mapsto y) : A \cup \{x\} \to B \cup \{y\}$ the function which maps x to y and otherwise behaves like f. We denote by $id_A : A \to A$ the identity function on A, omitting the subscript if clear from context.

We call the transducer obtained by identifying two states a *quotiented* transducer, defined as follows.

Definition 6 (Quotienting)**.** *Given transducer $T = \langle S \uplus \{s_1, s_2\}, s^0, \delta \rangle$ we define its* quotient $T/(s_1, s_2)$ *as follows:*

- $S_{T/(s_1,s_2)} = S \uplus \{s\}$
- $s^0_{T/(s_1,s_2)} = (id_S \mid s_1 \mapsto s \mid s_2 \mapsto s)(s^0)$
- $(r_1, V, r_2) \in \delta_{T/(s_1,s_2)}$ *iff there are $r_i' \in (id_S \mid s_1 \mapsto s \mid s_2 \mapsto s)^{-1}(r_i)$, $i = 1, 2$ such that $(r_1', V, r_2') \in \delta$.*

Lemma 1. *For any transducers $T, T/(s', s'') : A$ we have $[\![T]\!] \subseteq [\![T/(s', s'')]\!]$.*

The proof is immediate, as the quotiented transducer always introduces new transitions while preserving the old ones.

If two coherently equivalent states, under some protocol, are quotiented in a transducer then the resulting transducers are coherently equivalent under the same protocol.

Theorem 1 (Soundness)**.** *For any transducer $T : A$, protocol $P : A$, and states $s', s'' \in S_T$, if $s' \asymp_A^T s''$ then $T \equiv_A^P T/(s', s'')$.*

Proof. By expanding Def. 3, we need to show that $T \cap P \equiv_A T/(s', s'') \cap P$. Lem. 1 proves one direction. The other direction, by soundness of $\cap$, is $([\![T/(s', s'')]\!] \setminus [\![T]\!]) \cap [\![P]\!] = \emptyset$. We prove it by induction on the length of trace $t \in \mathscr{P}^*(A)$. Let $t \in [\![T/(s', s'')]\!]$. We need to show that either $t \in [\![T]\!]$, or $t \notin [\![T]\!]$ and $t \notin [\![P]\!]$. From the definition of $[\![T/(s', s'')]\!]$, $\exists q \in S_{T/(s',s'')}$ such that $(s^0_{T/(s',s'')}, t, q) \in \hat{\delta}_{T/(s',s'')}$. According to Def. 6 if $s^0_T = s^0_{T/(s',s'')}$ then this implies that $t \in [\![T]\!]$, otherwise $s^0_T = s'$ and $s^0_{T/(s',s'')} = s$ or $s^0_T = s''$ and $s^0_{T/(s',s'')} = s$. Clearly in the latter two cases we have $t \notin [\![T]\!]$, but since we assumed $s \asymp_A^T s'$ then by

Def. 4 and Def. 5 and by considering that ε is a witness trace for the start state s^0_T, for both cases, we get $t \notin \llbracket T \rrbracket$ and $t \notin \llbracket P \rrbracket$. So the property holds for every trace of length one.

For the inductive step we want to show that for any trace $t \cdot V$ if $t \cdot V \in \llbracket T/(s', s'') \rrbracket$ then $t \cdot V \in \llbracket T \rrbracket$ or $t \cdot V \notin \llbracket T \rrbracket$, and $t \cdot V \notin \llbracket P \rrbracket$. First note that if $t \cdot V \in \llbracket T/(s', s'') \rrbracket$ then $t \in \llbracket T/(s', s'') \rrbracket$, since transducer languages are prefix-closed. Applying the induction hypothesis, either $t \in \llbracket T \rrbracket$ or $t \notin \llbracket T \rrbracket$, and $t \notin \llbracket P \rrbracket$. For the first case, as in the base case, unfold Def. 6 and take t as a witness trace; we deduce that if the target state of the trace t is s', respectively s'', and the source state of of round V is s'', respectively s', then this implies that $t \cdot V \notin \llbracket T \rrbracket$, but since we we assumed earlier in this proof that $s \asymp^T_A s'$ and by unfolding Def. 4 and Def. 5 then this means $t \cdot V \notin \llbracket P \rrbracket$. For the second case the proof is immediate from prefix closure, since either $t \cdot V \in \llbracket T \rrbracket$ or $t \cdot V \notin \llbracket T \rrbracket$, and $t \cdot V \notin \llbracket P \rrbracket$. □

Note that if the protocol $P : A$ is the trivial protocol which accepts all interactions then coherent equivalence and quotienting become the conventional notions of equivalence and minimisation.

The soundness theorem states that an environment constrained by the protocol cannot distinguish between the original and the quotiented transducer, which has a smaller number of states. Iteratively quotienting all pairs of coherently equivalent states produces a *coherently minimised* transducer.

Because we are working in a compiler, the issue of compositionality is also important. The interaction protocols between the program and the environment are dictated by the type signature of the program, therefore different programs will observe different protocols. The following result shows that coherent minimisation is not only sound, but also *compositional*, i.e. it can be applied to any sub-component of a larger system without affecting its overall properties, including coherence equivalence itself.

We denote by $t \upharpoonright A$ a trace generated by deleting all port labels that do not belong to signature A. The definition of projection ($\upharpoonright$) can be lifted to sets by pointwise application. We denote by $\theta \subseteq \mathscr{P}(A)^*$ sets of traces over signature A. We define the *interaction* of two set of traces $\theta \subseteq \mathscr{P}(A+B)^*$ and $\theta' \subseteq \mathscr{P}(B+C)^*$ as $\theta \parallel \theta' = \{t \in \mathscr{P}(A+B+C)^* \mid t \upharpoonright A+B \in \theta \wedge t \upharpoonright B+C \in \theta'\}$. *Composition* of sets of traces is defined as interaction followed by hiding (projection), which is the standard definition in trace-based models of processes $\theta;\theta' = \{t \in \mathscr{P}(A+C)^* \mid t \in \theta \parallel \theta'\}$.

Definition 7 (Transducer interaction)**.** *Given transducers $T : A+B$ and $T : B+C$, we define $T \parallel T' : A+B+C = \langle S_T \times S_{T'}, (s^0_T, s^0_{T'}), \delta \rangle$, where $((s_1, s'_1), V, (s_2, s'_2)) \in \delta$ iff $(s_1, V \upharpoonright A+B, s_2) \in \delta_T$ and $(s'_1, V \upharpoonright B+C, s'_2) \in \delta_{T'}$.*

Definition 8 (Transducer projection)**.** *Given $T : A+B$ we define $T \upharpoonright A = \langle S_T, s^0_T, \delta \rangle$, where $(s_1, V \upharpoonright A, s_2) \in \delta$ iff $(s_1, V, s_2) \in \delta_T$.*

Definition 9 (Transducer composition)**.** *Given transducers $T : A+B$ and $T : B+C$, we define $T;T' = (T \parallel T') \upharpoonright A+C$.*

Theorem 2 (Soundness of composition)**.** *For any two transducers $T : A+B$ and $T' : B+C$ we have $\llbracket T || T' \rrbracket = \llbracket T \rrbracket || \llbracket T' \rrbracket$ and $\llbracket T;T' \rrbracket = \llbracket T \rrbracket ; \llbracket T' \rrbracket$.*

Theorem 3 (Compositionality)**.** *For any transducers $T, T' : A+B, T'' : B+C$ and protocols $P : A+B, P' : B+C$ if $T \equiv^P_{A+B} T'$ then $T;T'' \equiv^{P;P'}_{A+C} T';T''$.*

Proof. Let $T \equiv^P_{A+B} T'$ which by Def. 3 is equivalent to $T \cap P = T' \cap P$. By conventional equivalence and the soundness of transducer intersection, $\llbracket T \rrbracket \cap \llbracket P \rrbracket = \llbracket T' \rrbracket \cap \llbracket P \rrbracket$. We want to show that $T;T'' \equiv^{P;P'}_{A+C} T';T''$, which by Def. 3 is $\llbracket (T;T'') \cap (P;P') \rrbracket = \llbracket (T';T'') \cap (P;P') \rrbracket$, which we do by double inclusion. $\llbracket (T';T'') \cap (P;P') \rrbracket \subseteq \llbracket (T;T'') \cap (P;P') \rrbracket$. Let $t \in \llbracket (T';T'') \cap (P;P') \rrbracket$, which by the soundness of transducer intersection and composition is equivalent to $t \in \llbracket T' \rrbracket ; \llbracket T'' \rrbracket$ and $t \in \llbracket P \rrbracket ; \llbracket P' \rrbracket$, so there exists a trace $t' : A+B+C$ such that $t' \upharpoonright A+C = t$ and $t' \in \llbracket T' \rrbracket \parallel \llbracket T'' \rrbracket$ and $t' \in \llbracket P \rrbracket \parallel \llbracket P' \rrbracket$. By the definition of trace-set interaction it follows that $t' \upharpoonright A+B \in [\![T]\!]$ and $t' \upharpoonright A+B \in [\![P]\!]$ and $t' \upharpoonright B+C \in [\![T'']\!]$ and $t' \upharpoonright B+C \in [\![P']\!]$. Since earlier we have shown $[\![T]\!] \cap [\![P]\!] = [\![T']\!] \cap [\![T']\!]$ it immediately implies that $t' \upharpoonright A+B \in [\![T]\!]$, which means that $t' \in [\![T]\!] \parallel [\![T'']\!]$. By following the definition of trace-set composition we deduce that $t' \upharpoonright A+C \in [\![T]\!];[\![T'']\!]$. Since $t = t' \upharpoonright A+C$, $t \in [\![T]\!];[\![T'']\!]$, and hence we have already $t \in [\![P]\!];[\![P']\!]$ we conclude that $t \in ([\![T]\!];[\![T'']\!]) \cap ([\![P]\!];[\![P']\!])$ which by the soundness of transducers intersection and composition is equivalent to $t \in [\![(T;T'') \cap (P;P')]\!]$. The other direction is similar. □

6 Symbolic transducers and synthesis

Finite-state transducers are unsuitable in dealing with numbers due to the very large numbers of states required. Transducers interpret the entire state space explicitly, and hence, it is too computationally expensive to construct them for arbitrary values. A standard technique is to use symbolic representations to overcome these limitations. Several transitions from one source state to different target states can be combined into a single transition governed by a symbolic condition, a *guard*. A symbolic finite-state transducer (SFST) uses two components to represent state: a finite set of *control states* and a finite set of *registers* of unbounded size to handle data. Registers have initial values and can be modified explicitly via symbolic expressions (*updates*). In our concrete implementation, symbolic guards and updates use the expressions syntax of YICES, a state of the art of tool for solving satisfiabilty modulo theory (SMT). [3] Generating synthesisable HDL descriptions of circuits from symbolic transducers is straightforward.

Definition 10 (SFST)**.** *A Symbolic Finite State Transducer T over port signature A written $T : A$ is defined by as a tuple $T \overset{def}{=} \langle S_T, Reg_T, s^0, \delta_T \rangle$, where:*

- *a finite set of* states*, S_T*
- *a finite set of* local stores *(*registers*) Reg_T*
- *a designated* start state*, $s^0 \in S_T$*
- *A* transition relation*, $\delta_T \subseteq S_T \times \mathscr{P}(A) \times \Gamma \times \Delta \times S_T$*

Note that the *guard* and the *update* are symbolic expressions specified in a fixed grammar, in our case that of YICES. Let Γ and Γ' be the set of guard and update expressions, respectively and let $\Delta \subseteq Reg \times \Gamma'$ be a language of *updates* which includes the identity. Every register $x \in Reg$ is assumed to have initial value 0.

As a simple example, a SFST which reads an integer twice from port x and outputs the sum of the two values, only if it is positive, to port r can be defined as below.

$$T = \langle \{A,B,C\}, \{y,z\}, A, \{(A,\{x\},true, y \leftarrow x, B), (B,\{x\},true, z \leftarrow x, C), (C,\{r\}, y+z>0, r \leftarrow y+z, A)\}$$

All definitions from the previous section lift in the obvious way to SFSTs since their correctness is not contingent on the set of states being finite, and since all SFSTs can be obviously represent explicitly by mapping the register values into a concrete state. The key difference between FSTs and SFSTs is in their computational properties, which for the latter can be undecidable. For this reason, the key notion of coherent equivalence and quotienting only apply to control states, rather than symbolic states encompassing both control states and register values. For computational reasons we also restrict the notion of protocol to the order in which ports are activated, ignoring the values on the ports. So, as a protocol we could specify that an operation reads the input twice then produces output, but we cannot specify that it is an adder.

[3] http://yices.csl.sri.com/

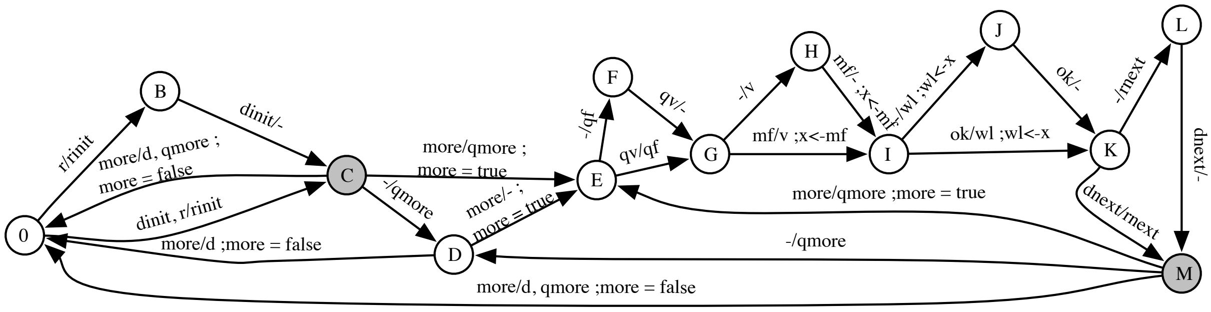

Figure 10: Original representation of in-place map

Definition 11 (Symbolic protocol)**.** *A SFST T is a* symbolic protocol *only if* $\Gamma = \{true\}$ *and* $\Delta = id$.

Definition 12 (SFST-Coherent state simulation)**.** *Given a SFST* $T : A$*, a symbolic protocol* $P : A$ *and a relation* $R \subseteq S_T \times S_T$*, we say that R is a* coherent simulation*, iff, for any* $(s', s'') \in R$ *the following two conditions hold:*

1. *For any transition* $(s'', V, g'', \Delta'', r'') \in \delta_T$*, there exists a transition* $(s', V, g', \Delta', r') \in \delta_T$ *such that* $(r', r'') \in R$ *and* $g' \equiv g''$ *and* $\Delta' \equiv \Delta''$*;*
2. *For any round* $V \subseteq A$*, such that there is no transition from* s'' *labelled with V,*
   - *(a) either there is no transition from* s' *labelled with V,*
   - *(b) or* $\omega_T(s'') \cdot V \cap [\![P]\!] = \emptyset$.

6.1 Example: coherent optimisation of in-place map

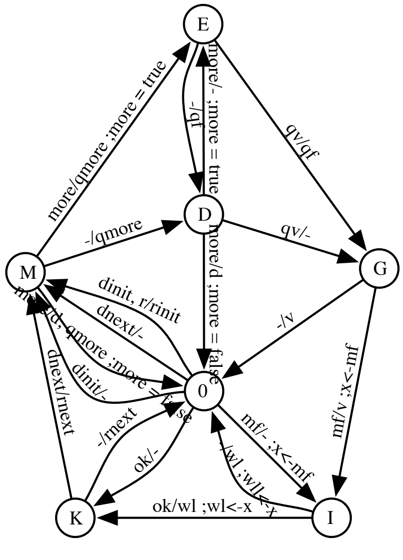

Figure 11: Coherently minimised transducer

Consider a Verity program which applies a function `f` to a data structure through an iterator: `init` initialises it, `more` tests whether there are more elements, `l` gives write-access to the current location, `v` gives the value of the current location and `next` moves to the next location. The code is:

```
import f, init, more, l, v, next;
init; while more do {l := f(v); next}
```

The (asynchronous) protocol P that is considered here for the minimisation process, given by the type signature of the program, can be described by the following regular expression: $(r(q_{more}b_{more} + q_f^1(q_f^2 m_f^2)^* m_f^1 + r_{init}d_{init} + r_{next}d_{next} + w_l^n ok_l + q_v m_v)^* d)^*$. The synchronous version is derived from this using round abstraction and is significantly more complicated.

The compilation of the program gives the SFST in Fig. 10, with 13 states. Because the protocol is enforced by the monitor we can apply coherent minimisation, which results in a version with 7 states (Fig. 11). The tuples of coherently equivalent states are $D \asymp^P F$, $0 \asymp^P B \asymp^P H \asymp^P J \asymp^P L$, $C \asymp^P M$, and can be quotiented away. Using conventional bisimulation, such as implemented by the Hopcroft minimisation algorithm, we get an automaton with 12 states as only *C* and *M* are bisimilar.

7 Towards practical and efficient secure computation

Higher-level synthesis via GoS augmented with the protocol-monitoring mechanism of Sec. 3, can produce tamper-proof instances of arbitrary libraries with rich functional interfaces. This can offer a new approach to the problem of collaborative computation without data disclosure, i.e. *secure computation* [Ben-Or et al., 1988]. Much research in this area is concerned with writing programs that do not inadvertently leak information, e.g. privacy-preserving data mining [Agrawal and Srikant, 2000], but system-level security guarantees in the form of absence of low-level attacks and exploits are essential to make this practical. Because system-level security is difficult to guarantee on the desktop, secure computation is generally thought of in the context of distributed and cloud computing, where the physical separation of resources makes system-level security guarantees easier to achieve.

On the desktop, on the other hand, secure computation requires the presence of a trusted hardware module that can prevent sensitive data (keys, value registers, etc.) from being tampered with, the typical example being the *Trusted Platform Module* (TPM)[4]. A TPM can also be used to authenticate arbitrary binary code and authorise its access to sensitive data, so it is possible to set up a secure computation framework using it. The most practical way of doing this is using virtualisation to set up a secure virtual machine for the execution of trusted code, and interfacing it with the rest of the machine as if it was a distinct physical computer. This works because modern processors support virtualisation natively and protect the memory space of the virtual machine.

Hardware compilation is a way to produce fully customised secure hardware modules that can interface with the rest of the system using a convenient higher-order interface. Low-level attacks and exploits on the module are prevented first via the physical tamper-proof mechanism of the FPGA fabric, and logically through a monitoring mechanism which prevents interactions that do not respect the programming language protocol. This allows properties established by reasoning at the programming languages level to be guaranteed in the implementation. Compared with TPM-based approaches, this approach has two potential advantages. First, is its simplicity. No special TPM is required and no native virtualisation support is needed in the untrusted device. Through higher-level synthesis we can produce special-purpose devices that provide only restricted functionality. Verifying the logical security properties of such devices is significantly easier than verifying the security properties of general-purpose software and hardware mechanisms such as TPMs and virtualisation frameworks. The second advantage is that of low overhead. TPM-based secure computation on the desktop involves a significant amount of overhead, as does the communication between the secure virtual machine and the rest of the system. On the other hand a FPGA can be set up to interface with a CPU on a physical level via the system bus; a variety of FPGA-based PCI cards are commercially available and can be used to implement this system. This is further work.

We also note that once the interaction between a system and its environment can be effectively constrained, more states in the system are observably equivalent. This novel notion of equivalence, called coherent equivalence, can be used to aggressively optimise such systems by reducing the number of states. Coherent equivalence is, of course, not restricted to our particular semantic model, which is

[4] See `http://www.trustedcomputinggroup.org/`

meant to serve as motivation and illustration, but to any automata operating in restricted environments. For example, APIs themselves can be enforced by a monitor, stipulating that functions belonging to the API must be called in a particular order.

Our formulation is a first step, but more work needs to be done: the main definition (Def. 4) has a formulation which relies on the trace-semantic interpretation of the transducer; a direct definition similar to that of bisimulation would be preferable. This is also future work. Most importantly, the simulation relation of Def. 4 is in general not transitive. This raises one technical problem which is somewhat unpleasant although not critical. Our state-reduction algorithm will lead to results which are dependent of the order in which coherently equivalent states are identified and quotiented. If $s_1 \frown s_2$ and $s_2 \frown s_3$ and if we quotient s_1, s_2 (as a new state s') we don't know a priori whether $s' \frown s_3$. From experiments we know that sometimes it holds sometimes it doesn't. Practically, this makes the state-reduction algorithm slower because $\frown$ needs to be recalculated after each quotienting.

References

M. Abadi. Protection in programming-language translations. In *ICALP*, pages 868–883, 1998. doi: 10.1007/BFb0055109.

M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009. doi: 10.1145/1609956.1609960.

R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 439–450, New York, NY, USA, 2000. ACM. ISBN 1-58113-217-4. doi: 10.1145/342009.335438.

M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 1–10, New York, NY, USA, 1988. ACM. ISBN 0-89791-264-0. doi: 10.1145/62212.62213.

D. R. Ghica. Geometry of Synthesis: a structured approach to VLSI design. In *POPL*, pages 363–375, 2007.

D. R. Ghica. Function interface models for hardware compilation. In S. Singh, B. Jobstmann, M. Kishinevsky, and J. Brandt, editors, *MEMOCODE*, pages 131–142. IEEE, 2011. ISBN 978-1-4577-0117-7. doi: 10.1109/MEMCOD.2011.5970519.

D. R. Ghica and M. N. Menaa. Synchronous game semantics via round abstraction. In *FOSSACS*, pages 350–364, 2011. doi: 10.1007/978-3-642-19805-2_24.

D. R. Ghica and A. Smith. Geometry of Synthesis II: From games to delay-insensitive circuits. *Electr. Notes Theor. Comput. Sci.*, 265:301–324, 2010. doi: 10.1016/j.entcs.2010.08.018.

D. R. Ghica and A. Smith. Geometry of Synthesis III: Resource management through type inference. In *POPL*, pages 345–356, 2011. doi: 10.1145/1926385.1926425.

D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Syntactic control of concurrency. *Theor. Comput. Sci.*, 350(2-3):234–251, 2006. doi: 10.1016/j.tcs.2005.10.032.

D. R. Ghica, A. Smith, and S. Singh. Geometry of Synthesis IV: compiling affine recursion into static hardware. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *ICFP*, pages 221–233. ACM, 2011. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034805.

G. M. Kelly and M. L. Laplaza. Coherence for compact closed categories. *Journal of Pure and Applied Algebra*, 19:193–213, 1980.

A. Mycroft and R. Sharp. A statically allocated parallel functional language. In *ICALP*, pages 37–48, 2000. doi: 10.1007/3-540-45022-X_5.

P. W. O'Hearn, J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. *Theor. Comput. Sci.*, 228(1-2):211–252, 1999. doi: 10.1016/S0304-3975(98)00359-4.

J. C. Reynolds. Syntactic control of interference. In *POPL*, pages 39–46, 1978. doi: 10.1145/512760.512766.

J. C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981.

H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security*, pages 298–307, 2004. doi: 10.1145/1030083.1030124.