Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 250 (2009) 39–54
www.elsevier.com/locate/entcs

A Semantic Condition for Data Independence and Applications in Hardware Verification

Lyes Benalycherif[1,2]

STMicroelectronics SA
12 Rue Jules Horowitz, F-38019 Grenoble, France

Anthony McIsaac[1,3]

ST Microelectronics (R&D) Ltd
1000 Aztec West, Bristol BS32 4SQ, UK

Abstract

Data independence is a useful technique in reasoning about systems. Commonly, if one knows that the qualitative behaviour of a system does not depend on the specific values of data inputs, the proof of facts about its behaviour can be simplified. Such knowledge typically comes from examination of the syntax of the program for the system. Industrial hardware verification flows lead to a requirement for automated proof of data independence without intrusion into the program, where the specification on which the proof is based makes no reference to details of the program language. This paper presents and proves a sufficient condition for data independence, expressed in terms of the behaviour of inputs and outputs of a system, that can be checked in practice by a model checker; and it demonstrates how this condition is used in two design applications.

Keywords: Data independence, formal verification, model checking, PSL.

1 Introduction

Automating the verification of critical systems involves not only the mechanization of correctness proofs of complex algorithms and implementations, but also the automation of flows in which the algorithms and implementations themselves are not particularly intricate, but there is a big separation in the development process between the specification and the implementation. Automation allows different parts of the process to be carried out by very different groups of engineers: as long as

[1] Supported by the European Comission through the project Prosyd, FP6-IST-507219.
[2] Email: Lyes.Benalycherif@st.com
[3] Email: anthony.mcisaac@ntlworld.com

doi:10.1016/j.entcs.2009.08.004

the specifications are precise, they can be made without knowledge of how the implementation or verification will be done; and much of the verification can be done without a detailed knowledge of the design. This paper is concerned chiefly with results that support the automation of flows. It presents and proves a result that can be used to establish that models have the feature of data independence, either as a device for simplifying verification or as a design requirement in itself. The result is formulated in terms of inputs and outputs of models, and is suited to automated hardware verification flows.

Data independence [18] is a common technique for reducing the task of checking properties of a design model to that of checking it for a small number of data values. There are numerous results of the form: if the only operations on objects of some type in a program are assignments (or, in another variant, if the only operations are assignments and equality checks), then certain properties P(d) in which d ranges over all values of that type hold if and only if they hold when d ranges over values in some small finite type [10], [12], [18]. These results have been applied to verifying designs such as a buffer controller [16].

Checking that a program or design model is indeed data independent is usually a matter of satisfying oneself that the the only operations on data in the program are those allowed for the particular variant of data independence. For many purposes, this is satisfactory, even if the syntactic checks of the program are not automated. Many applications of model checking [7] have been concerned with the verification of complex critical components in a design, where the model checking effort is large [3], [1], and the effort required to check data independence is small by comparison. In other applications, properties are used as an additional weapon for finding or investigating complex bugs [6], rather than as a basis for exhaustive verification, and one may be satisfied with having good cause to believe the program is data independent, without checking it completely.

Recently, there has been interest in automated flows for applying reusable properties to various implementations of a design or interface. In these flows, one is not only looking to discover subtle bugs, but to provide assurance that many, perhaps quite mundane, properties hold. This is particularly the case for protocol compliance. In [17], a platform for applying reusable properties of the AHB protocol is described. This platform has elaborate tool support, incorporating the SMV model checker and the HOL theorem prover. A package for automatically checking AHB and APB compliance is described in [15]. Users provide information about the interfaces of the design, from which the properties required for compliance are deduced. These are expressed in a proprietary language, and they are checked within the package. The development of standard property specification languages such as PSL [19] has led to the possiblilty of specifying designs or protocols in temporal logic independently of any implementation, and without any particular tool as the target for carrying out the verification. For a proprietary system bus in STMicroelectronics, a flow has been developed in which the appropriate PSL properties for any particular bus module are automatically inferred from examination of its interface, and these properties can be checked, either formally or in simulation, by

any tool supporting this property language.

In automated flows where it is required to minimize the human effort in adapting a fixed set of properties to different implementations, and proving the properties with the tools available, it is not satisfactory to rely on syntactic analysis of the program in order to justify the assumption of data independence. There are three reasons for this:

- The verification task may be carried out by engineers who have no knowledge of the design, and are not in a position to analyse its internal details.
- The language of the implementation model may not be known, so it is difficult to supply with the properties a tool or script that will analyse all implementation models for data independence.
- In hardware designs especially, the models may in fact not be data independent according to a syntactic definition, although the conclusions from data independence still apply. For example, some resource in the design may be used for temporary storage of control information at some time and data at others, although whenever any value is read from that resource that finds its way to a data output, the value in the resource must have come from a data input. Data may also be transformed internally - shifted, reversed or split up.

Automatic datapath abstraction tools are described in [9] and [13]. These map a VHDL or Verilog RTL implementation model to an abstract model in some other language, which is taken as input to a verification tool. The datapath can be abstracted to a small number of elements, or to a representation in terms of uninterpreted functions. Although such tools address the point about the need for users to analyse the design, they are targeted at specific languages, and they will not detect data independence in cases where resources are used for control information at some times and data at others.

In some applications, data independence is not so much a useful way of reducing the cost of proof, but an essential part of the functional specification. The work presented here arose from the verification of a component in a security-critical random number generator [5]. The component reads input streams that can be assumed to be random, and is required to deliver random output streams. One requirement in the specification is that each output data value is equal to some associated input data value, where the association between the inputs and outputs is independent of the data values at the inputs. The approach to specifiying the component as a whole is described in [5], but no justification is given there for the way this requirement is proved. The results in this paper provide this justification (Section 3.2).

For all these reasons, we want a way of establishing data independence from consideration of the interface behaviour of a model. There is a semantic definition in [18], although it is not proposed as a basis for checking in practice, being expressed in terms of all possible mappings from one set of data values to any other set. A very general framework for semantic definitions of data independence is provided in [12]. These definitions are formulated in terms of the operational semantics of the program. In an application to any particular instance, the definitions say things

not only about inputs and outputs, but also about the transitions between states involving internal program variables, and in order to make use of the definitions, one would need knowledge of the specific program language and its operational semantics.

In Section 2, we prove the sufficiency of certain semantic conditions for data independence, expressed in terms of inputs and outputs of a hardware model. These conditions are practical for checking in industrial applications. Informally, they state that any data value seen at the outputs must have been seen at the inputs; and that if two copies of the design have the same control inputs and their data outputs differ at any point, there must have been a point at which the data inputs differ in the same way as the data outputs.

In Section 3, we describe two applications of these results, to a random number generator and a fifo.

2 A sufficient condition for data independence

We consider a hardware design with a number of input and output ports. The conditions for data to be captured at an input port or released at an output port are given in terms of the values, and possibly the histories, of control signals (including clocks, for synchronous designs). There are data input and output signals at the ports, and the control outputs have no dependencies on the data inputs.

For the purposes of this section, we consider just one input and one output port. The data signals at these ports are *data_in* and *data_out* respectively. A strong condition for data independence is that the output data streams are samplings of the input data streams, in the following sense:

Consider a behaviour B of the design, in which there is a sequence of points $t\_in_1, t\_in_2, \ldots$ at which data is captured, and a sequence of points $t\_out_1, t\_out_2, \ldots$ at which data is released. Then there is a mapping associating each release point t_out_i with a capture point t_in_j, such that, in any possible behaviour of the design in which the control signals have the same values as in B (and therefore the sequences of capture and release points are the same as in B), the value of *data_out* at t_out_i is equal to the value of *data_in* at t_in_j.

Note that, in this definition, one capture point may be associated with more than one release point.

If the output streams are samplings of the input streams in this sense, then the intuitive characterization of data independence in [18] is satisfied: "if we change the input data of our program, the behaviour of the program will not change, except for the corresponding values of the output data". Further, it is possible to write a program determining the values of the data outputs, in which the only operations on data are assignments: namely by adding to the code for the control signals (which has no dependency on data values) statements keeping track of the data inputs and assigning the value of *data_out* at t_out_i to the value of *data_in* at t_in_j. So the design will be data independent according to the many other definitions in, for example, [18] and [12].

The condition that the output data streams are samplings of the input data streams cannot be checked directly in a practicable way, and cannot be expressed in a temporal property language such as PSL. However, sufficient conditions for this are provided by the following two properties, which can be expressed in a way suitable for direct checking.

(1) If a data value v is released at any time, then v must have been captured at the same or an earlier time.

(2) Suppose that there are two copies of the design, and the control inputs to both copies are the same. Then, if there is some point where the data value v is released in one copy and the data value w in the other, where $v \neq w$, there must be some point where the data values v and w respectively are captured in the two copies.

We will in fact prove that the conditions (1) and (2′) are sufficient, where (2′) is weaker than (2):

(2′) Suppose that there are two copies of the design; the control inputs to both copies are the same; and the input data streams to each copy are identical, except at precisely one point where data is captured, when the input data values are different. Then, if there is a difference between the output values at any point, it must be the same difference as that between the input data values at the point where the input data streams differ.

The deduction of the condition that the output data streams are samplings of the input data streams from (1) and (2′) is not straightforward; indeed these conditions are not sufficient if there are only two data values. For suppose that there are two data values, x and y. The first output value is x if either of the first two input values is x; otherwise the first output value is y. The second output value is the third input value; the third output value is the fourth input value, etc. Then property (1) is clearly satisfied. To see that property (2′) is satisfied, we need to know that if the input value changes at precisely one point, and there is a point at which the output value changes, then the ouput value changes in the same way as the input value. Now if just the nth input value changes, for any $n > 2$, then the only change in the outputs is that the $(n-1)$th output value changes in the same way. If either of the first two input values changes from x to y, then either the first output value remains the same (if the other of the first two input values is x), or it changes from x to y (if the other input value is y). And if either of the first two input values changes from y to x, then either the first output value remains the same (if the other input value is x), or it changes from y to x (if the other input value is y).

However, the output stream is not a sampling of the input stream at points independent of the data values, because the identification of the input point associated with the first output point depends on the first two data input values.

We prove a theorem from which we can deduce the sufficiency of conditions (1) and (2′) when there are at least three distinct data values.

Theorem 2.1 *Let N be the set of positive integers $1, 2, \ldots$ and let V be any set*

with at least three elements. Let $p : N \to N$ be a function. Let Φ be a set of pairs (f, g), where $f : N \to V$ and $g : N \to V$ are functions.

Suppose that Φ has the following properties.

(i) *For every function $f : N \to V$, there is a function $g : N \to V$ such that (f, g) is in Φ.*

(ii) *For every positive integer n, if (f, g) is in Φ and $g(n) = v$, then there is a positive integer $m \leq p(n)$ such that $f(m) = v$.*

(iii) *For every positive integer n, if (f_1, g_1) is in Φ, and $f_1(m) = f_2(m)$ for all $m \leq p(n)$, then there is some $g_2 : N \to V$ such that (f_2, g_2) is in Φ and $g_1(n) = g_2(n)$.*

(iv) *Suppose that (f_1, g_1) and (f_2, g_2) are elements of Φ, where there is some positive integer m_0 such that $f_1(m) = f_2(m)$ for all positive integers $m \neq m_0$. Then for every positive integer n, either $g_1(n) = g_2(n)$, or $g_1(n) = f_1(m_0)$ and $g_2(n) = f_2(m_0)$.*

Then there is a function $s : N \to N$ such that for all (f, g) in Φ and all positive integers n, $g(n) = f(s(n))$.

Theorem 2.1 allows us to conclude that the conditions (1) and (2′) are sufficient for the stream of released data values to be a sampling of the stream of captured data values, where the sampling points are independent of the input data values. For if V is the set of data values, Φ can be viewed as the set of pairs of input and output data streams that are consistent with the behaviour of the design, with N being used for a sequence of points in time, either the sequence of points where data is captured or the sequence of points where data is released. The function $p(n)$ describes the relationship in time between points of the output stream and points of the input stream: for any positive integer n, $p(n)$ is the number of data input capture times up to and including the time of the n-th data output release.

Then conditions (ii) and (iv) of Theorem 2.1 are immediate consequences of (1) and (2′). Condition (i) of Theorem 2.1 holds because there are no constraints on the data inputs: for every possible data input stream there is some behaviour of the design consistent with it, and that behaviour yields some stream of data outputs. Condition (iii) of Theorem 2.1 simply states that the possible values of the data output at any time are not influenced by input data values at later times.

The conclusion from Theorem 2.1 is then that every data release time *t_out* can be associated with some data input time *t_in* (namely, t_out_n is associated with $t\_in_{s(n)}$), such that the value of the data output at *t_out* is equal to the value of the data input at *t_in*. Since $s(n)$ does not depend on the specific pair of input and output data streams, this association is independent of the actual data values.

We start by proving a sequence of three lemmas. In these lemmas, we focus on the output data values at just a single point in time: we fix some positive integer n, and consider the values of $g(n)$ in pairs (f, g) as in the statement of the Theorem.

Lemma 2.2 *Under the conditions of Theorem 2.1, suppose that (f_1, g_1), (f_2, g_2) and (f_3, g_3) are in Φ, and there is a positive integer m_0 such that:*

- *for all* $m \neq m_0$, $f_1(m) = f_2(m) = f_3(m)$.
- $f_1(m_0) = x$, $f_2(m_0) = y$ *and* $f_3(m_0) = z$, *where* x, y *and* z *are distinct elements of* V.
- $g_1(n) = x$ *and* $g_2(n) = y$.

Then $g_3(n) = z$.

Proof. On the one hand, since f_1 and f_3 differ only at m_0, and (f_1, g_1) and (f_3, g_3) are in Φ, it follows from condition (iv) of Theorem 2.1 that either $g_3(n) = g_1(n)$, in which case $g_3(n) = x$, or g_1 and g_3 differ at n in the same way as f_1 and f_3 differ at m_0, in which case $g_3(n) = z$.

On the other hand, f_2 and f_3 also differ only at m_0, and (f_2, g_2) and (f_3, g_3) are in Φ. A similar argument shows that $g_3(n)$ is either y or z. The only possibility consistent with the previous conclusion that $g_3(n)$ is either x or z is for $g_3(n)$ to be equal to z. This proves the lemma. □

Lemma 2.3 *Under the conditions of Theorem 2.1, suppose that* (f_1, g_1) *and* (f_2, g_2) *are in* Φ, *and there is a positive integer* m_0 *such that:*

- *for all* $m \neq m_0$, $f_1(m) = f_2(m)$.
- $f_1(m_0) \neq f_2(m_0)$.
- $g_1(n) = f_1(m_0)$ *and* $g_2(n) = f_2(m_0)$.

Then for all (f, g) *in* Φ *such that* $f(m) = f_1(m)$ *for all* $m \neq m_0$, $g(n) = f(m_0)$.

Proof. If $f(m_0)$ is distinct from both $f_1(m_0)$ and $f_2(m_0)$, the conclusion follows immediately from Lemma 2.2.

Suppose that $f(m_0) = f_1(m_0)$.

Choose $f_3 : N \to V$ so that $f_3(m) = f_1(m)$ for $m \neq m_0$, and $f_3(m_0)$ is distinct from both $f_1(m_0)$ and $f_2(m_0)$. This is possible, since V has at least three elements. Choose g_3 such that (f_3, g_3) is in Φ (using condition (i) of Theorem 2.1). By Lemma 2.2, $g_3(n) = f_3(m_0)$.

Now apply Lemma 2.2 again, with (f_3, g_3) playing the role of (f_1, g_1), (f_2, g_2) playing the role of (f_2, g_2), and (f, g) playing the role of (f_3, g_3). It follows that $g(n) = f(m_0)$.

The argument for the case $f(m_0) = f_2(m_0)$ is similar.

This completes the proof of Lemma 2.3. □

Lemma 2.4 *Under the conditions of Theorem 2.1, let* n *be any positive integer. There is some function* $f_0 : N \to V$ *and some* $m_0 \leq p(n)$ *in* N *such that, whenever* (f, g) *is in* Φ *and* $f(m) = f_0(m)$ *for all* $m \neq m_0$, $g(n) = f(m_0)$.

Proof. Let (ϕ, γ) be any element of Φ, and let $\gamma(n) = x$.

Let $M = \{\mu_1, \ldots, \mu_k\}$ be the set of all positive integers $m \leq p(n)$ such that $\phi(m) = x$. By condition (ii) of Theorem 2.1, M is non-empty.

Let y be an element of V different from x. Construct functions $\phi_0, \phi_1, \phi_2, \ldots, \phi_k : N \to N$ as follows.

$\phi_0(m) = \phi(m)$ for all m.
$\phi_1(m) = \phi_0(m)$ for $m \neq \mu_1$; $\phi_1(\mu_1) = y$.
$\phi_2(m) = \phi_1(m)$ for $m \neq \mu_2$; $\phi_2(\mu_2) = y$.
...
$\phi_k(m) = \phi_{k-1}(m)$ for $m \neq \mu_k$; $\phi_k(\mu_k) = y$.

Choose functions $\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_k : N \to N$ such that (ϕ_i, γ_i) is in Φ for all $0 \leq i \leq k$, and $\gamma_0 = \gamma$. This is possible by condition (i) of Theorem 2.1.

By construction, there is no $m \leq p(n)$ such that $\phi_k(m) = x$. Then by condition (ii) of Theorem 2.1, $\gamma_k(n) \neq x$. Let j be the smallest positive integer such that $\gamma_j(n) \neq x$. Since $\gamma_0(n) = x$, $j > 0$.

Now ϕ_{j-1} and ϕ_j have the same value except at one point, namely μ_j, where ϕ_{j-1} has the value x and ϕ_j has the value y. By condition (iv) of Theorem 2.1, if $\gamma_{j-1}(n)$ and $\gamma_j(n)$ differ, then $\gamma_{j-1}(n) = x$ and $\gamma_j(n) = y$. Since $\gamma_{j-1}(n) = x$ and $\gamma_j(n) \neq x$, we must have $\gamma_j(n) = y$.

Now let $m_0 = \mu_j$, and apply Lemma 2.3 with $(\phi_{j-1}, \gamma_{j-1})$ in the role of (f_1, g_1) and (ϕ_j, γ_j) in the role of (f_2, g_2). It follows that, whenever (f, g) is in Φ and $f(m) = \phi_{j-1}(m)$ for all $m \neq m_0$, $g(n) = f(m_0)$. Taking $f_0 = \phi_{j-1}$, this proves Lemma 2.4. □

In terms of the relationships between data streams, Lemma 2.4 says that there is some particular set of input data values at times other than m_0, such that the output value at n is always equal to the input value at m_0, as long as the inputs at times other than m_0 have these particular values. We have to extend this to say that the output value at n is equal to the input value at m_0, whatever the values of the inputs at other times.

We now prove the theorem.

Proof. Let n be any positive integer.

By Lemma 2.4, there is a function $f_0 : N \to V$ and some positive integer m_0 such that, whenever (f, g) is in Φ and $f(m) = f_0(m)$ for all $m \neq m_0$, $g(n) = f(m_0)$. For these values of f_0 and m_0, we prove by induction on k that, whenever (f, g) is in Φ and $f(m)$ differs from $f_0(m)$ for precisely k values of m other than m_0, then $g(n) = f(m_0)$.

For $k = 0$, this is precisely the conclusion of Lemma 2.4.

Assume now that the claim holds for $k = i$. Suppose that (f, g) is in Φ and $f(m)$ differs from $f_0(m)$ for $i+1$ values of m other than m_0. Let m_1 be one of these values. We consider separately the cases where $f(m_0) = f_0(m_1)$ and $f(m_0) \neq f_0(m_1)$. Suppose first that $f(m_0) \neq f_0(m_1)$.

Define $f_1 : N \to V$ by setting $f_1(m) = f(m)$ for $m \neq m_1$ and $f_1(m_1) = f_0(m_1)$. Then $f_1(m)$ differs from $f_0(m)$ for precisely i values of m other than m_0. Let $g_1 : N \to V$ be such that (f_1, g_1) is in Φ. By the inductive hypothesis, $g_1(n) = f_1(m_0)$.

Now f and f_1 differ at only one point, namely m_1. It follows from (iv) that if $g(n) \neq g_1(n)$, we must have $g(n) = f(m_1)$ and $g_1(n) = f_1(m_1)$. But $g_1(n) = f_1(m_0)$, and $f_1(m_0) = f(m_0)$, since f and f_1 differ only at m_1. Therefore $f(m_0) = f_1(m_1)$.

But $f_1(m_1) = f_0(m_1)$ by construction, and we are assuming that $f(m_0) \neq f_0(m_1)$. So we cannot have $g(n) \neq g_1(n)$. So in this case $g(n) = g_1(n) = f_1(m_0) = f(m_0)$, as required.

On the other hand, suppose that $f(m_0) = f_0(m_1) = x$. Choose y and z in V distinct from each other and from x. Define f_y and $f_z : N \to V$ to be identical to f except at m_0, with $f_y(m_0) = y$ and $f_z(m_0) = z$. Let g_y and $g_z : N \to V$ be such that (f_y, g_y) and (f_z, g_z) are in Φ. Now $f_y(m)$ differs from $f_0(m)$ for precisely $i+1$ values of m other than m_0, among them m_1; and $f_y(m_0) \neq f_0(m_1)$. The same argument as applied to (f, g) in the case where $f(m_0) \neq f_0(m_1)$ can now be applied to (f_y, g_y), with the conclusion that $g_y(n) = f_y(m_0) = y$.

Similarly, $g_z(n) = f_z(m_0) = z$. By Lemma 2.3, with (f_y, g_y) and (f_z, g_z) in the roles of (f_1, g_1) and (f_2, g_2), it follows that $g(n) = f(m_0)$ as required.

This completes the proof of the claim that, whenever (f, g) is in Φ and the number of values of m for which $f(m)$ differs from $f_0(m)$ is finite, then $g(n) = f(m_0)$.

To complete the proof of the theorem, define $s(n)$, for each positive integer n, to be some positive integer $m_0 \leq p(n)$ such that there is a function f_0 for which, whenever (f, g) is in Φ and the number of values of m for which $f(m)$ differs from $f_0(m)$ is finite, then $g(n) = f(m_0)$.

Let (f, g) be any element of Φ and let n be any positive integer, and let $m_0 = s(n)$. Let f_0 be as above. Define $f_1 : N \to V$ by $f_1(m) = f(m)$ for $m \leq p(n)$ and $f_1(m) = f_0(m)$ for $m > p(n)$. By condition (iii), there is some $g_1 : N \to V$ such that (f_1, g_1) is in Φ and $g_1(n) = g(n)$. Now the number of values of m for which $f_1(m)$ differs from $f_0(m)$ is finite. Therefore $g_1(n) = f_1(m_0)$. Further, m_0 was chosen to be not greater than $p(n)$. So $f_1(m_0) = f(m_0)$. Therefore $g(n) = g_1(n) = f_1(m_0) = f(m_0) = f(s(n))$ as required.

This completes the proof of Theorem 2.1. □

3 Applications

We describe two applications where the results of the previous section can be used to help automate the verification flow. As far as possible, we want to verify design implementations against specifications that are neutral with respect to both the verification tools to be used and the language of the implementation model. For maximum portability, we seek to express the requirements as temporal formulas in a standard language such as PSL.

3.1 PSL

PSL [19] is a language designed for the specification of temporal properties of industrial hardware designs. Its core language is a linear-time logic based on LTL, and it has a rich set of features, including

- Syntactic sugar for the LTL operators and simple combinations of them; for example `always` for G, `until` for the weak until operator W, and `next_event(p)(q)` to say that, on the next occasion when p holds, q also holds.

- Suffix implication constructs such as *sequence* $|->$ *sequence*. The simplest sequences have forms such as {`p;q;r`}, where p, q and r are boolean expressions that can be evaluated on states of the system; a system satisfies {`p;q;r`} `|->` {`s;t`} if, for every behaviour in which p, q and r hold in the first, second and third states respectively, s and t hold in the third and fourth states respectively. More complex sequences can be formed using a * operator for regular expressions: `p[*]` represents any number of repetitions of p, and `[*]` represents any number of repetitions of *true*. There is a range of further operators, including `[->]`, where {`p[->]`} is equivalent to {`!p[*];p`}, with `!p` being the negation of `p`.
- Built-in operators on boolean terms, such as `rose(`x`)` to say that x is 1 in the current cycle and was 0 in the previous cycle.

PSL is used for the illustrations of properties in the applications described below.

3.2 Random Number Generator

As discussed in Section 1 above, the example in which the need to prove that the output streams are a sampling of the input streams first arose was a random number generator (RNG). More accurately, the design is a random number distributor, receiving streams of random 8-bit numbers from a block containing analog hard macros, and supplying streams of random 8- or 16-bit numbers to several clients, including a host CPU and a complex memory controller. There are two data input ports at the interface with the analog side; it can be assumed that, as long as the interval between successive reads at each port is greater than a certain value, the streams of input data at the two ports are independently random. The RNG also receives monitoring information through a dedicated port, indicating if there is any reason to suppose that the analog block is not producing random numbers. At the interfaces with the clients, there is a request/grant protocol for transfer of output data values. The RNG supplies the clients with data values according to priorities and rules assigned in configuration registers. The configuration registers also support various functions: enabling/disabling each fifo; enabling the analog hard macro; loading specific values into fifos in debug mode; recording status information such as any fifo being full or warnings from the monitoring port. The RNG has two clocks, one, clk_a, coming from the analog side, and the other, clk_b, governing the digital side.

As described in [5], there are four types of requirements on the RNG: interface protocols; performance; register-related properties; and end-to-end relationships between input and output data. All but the last of these requirements can be straightforwardly specified with PSL properties. The end-to-end requirement is intuitively simple: the output streams to the clients must be independent random streams, as long as the input streams are independent random streams. But in this form, the requirement cannot be formalized by PSL properties. Instead, we identify properties that can be expressed in PSL and show that they imply preservation of randomness.

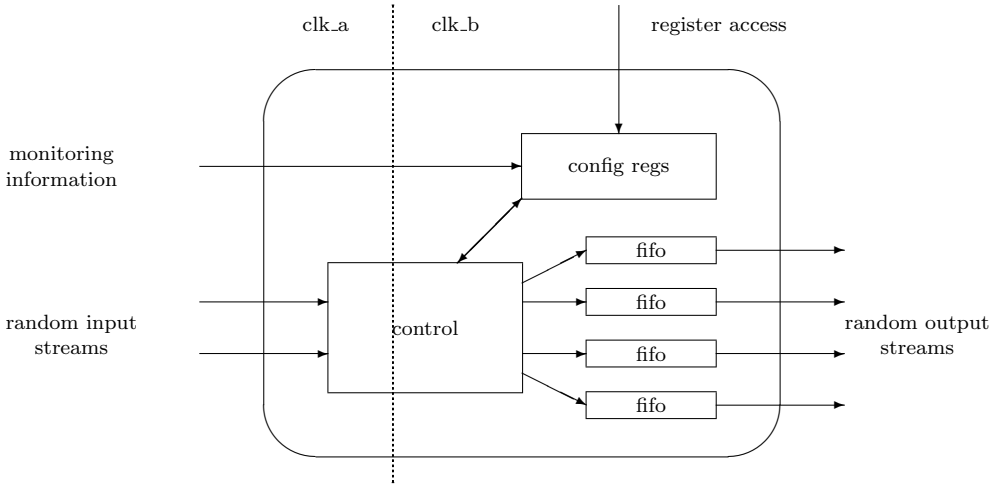

Figure 1 - Block diagram of Random Number Generator

Satisfaction of the final requirement is proved in three steps.

(i) We prove that the design complies with the set-up on which the results in Section 2 are based - namely that there is no dependency of the control outputs on the data inputs. This can be done by showing that the data inputs are not in the cone of influence of the control outputs. But, although many tools perform cone of influence reduction, this is usually as part of the model-checking algorithm, and users do not have access to a stand-alone dependency analyzer. An alternative way to specify the requirement is in terms of two copies of the design: if the values of the control inputs to each copy are identical at all times, then, whatever the values of the data inputs, the values of the control outputs are equal at all times. This requirement can be expressed in a temporal logic such as PSL; when it is checked in any particular tool, the cone of influence reductions will normally come into play, and the proof will be rapid.

The use of two copies of the design to prove complete independence of the control outputs from the data inputs has similarities with the conditions in Section 2; but we are concerned here with a different feature of the design. At this point, we are establishing that the control outputs have no dependencies whatsoever on the data inputs; we still have to prove the more complex property that the data outputs depend on the data inputs only in a certain way, which we can express precisely as in Section 2.

(ii) For each output port, we prove the two requirements (1) and (2) of Section 2 above, namely that no data value is output unless it has previously been input, and that if the data outputs at any time differ, then there is some time at which the input values at some port differ in the same way as these output values. In Section 2, we supposed there was only one input port, but the generalization

to a finite number of input ports is straightforward: if data is captured at more than one input port at any time, we can put an arbitrary ordering on the input ports, and consider the several input streams as a single stream.

These requirements can again be expressed in a temporal logic such as PSL, and model checking of them is feasible. For the implementation in the actual application, the check of requirement (2) took 1min 21secs and used 52MB on a 440MHz Sun Sparc Ultra 10 workstation 2, using the RuleBase model checker from IBM [3].

This establishes a mapping from the points where data is transmitted to the points where data is read, such that the data values read and transmitted are the same. It may be that some data values are read but never transmitted to any client; this is acceptable within the requirements. But we still have to prove that a data value read at one time is not transmitted more than once, whether to the same client or different ones.

(iii) We prove that, for each data capture point (i.e. for any input port, and any time when data is read at that port), there is at most one data release point (i.e. at most one output port, and at most one time when data is transmitted at that port) that is mapped to that capture point. In proving this, we can make use of the data independence already established. We can check a property saying that, for some specific data value d, if there is only one input port at which the data value read is ever d, and the value read is d precisely once at that port, then there is at most one output port at which the value d is ever transmitted, and it is transmitted at most once at that port.

This is enough to prove that, if the data streams at the input ports are independently random, so are the data streams at the output ports. For the events of data transmission are in one-to-one correspondence with a subset of the events of data reads, and therefore the probability of any proposition about the values at data transmission events is equal to the probability of the corresponding proposition about the values at data read events.

The PSL specification of the whole random number generator took six weeks of effort, not including the time spent proving Theorem 2.1, and involved 150 properties. The majority of the properties were concerned with details of the relationship between the values in configuration registers and the behaviour of the design. The time was spent in understanding the design and the timing requiremants at the interfaces; clarifying the specification of these timing requirements with the designer (which led to a correction of the microarchitecture); finding appropriate properties of internal signals where the properties of interface signals were too complex for model checking; and enumerating the many register properties. The specification of the sufficient conditions for data independence took one day; the complications were identification of the precise conditions for data capture and release, and dealing with the merging of some pairs of 8-bit inputs into 16-bit outputs.

3.3 Fifos

The results of Section 2 can also be used in developing automatically checkable specifications of common generic designs such as fifos. In verifying the implementation of a fifo, if one only has access to interface events and data values, it is necessary to keep track of the number of fifo elements ahead of any given one, in order to compare the data values for the same element as it enters and leaves the fifo. This cannot be expressed directly in temporal logic, and it is usually necessary to supplement the temporal properties with some sort of modelling code, for example, with code defining the behaviour of variables used in a PSL property:

```
assign
 next(entries_ahead) := case
  element_has_entered & exit_event : entries_ahead - 1;
  element_has_entered & exit_event & !entry_event : entries_ahead - 1;
  !element_has_entered & entry_event & !exit_event : entries_ahead + 1;
  else : entries_ahead;
 esac;
```

This defines a variable *entries_ahead* that keeps track of the number of places in the fifo that are occupied by data that has entered the fifo before the data that enters at some distinguished time. There will be further definitions for *entry_event*, *exit_event* and *element_has_entered*. The PSL property itself is

```
assert
  forall v in data_values :
     {[*]; entry_event & data_in = v & rose(element_has_entered);
     (exit_event & (elements_ahead = 0))[->]}
     |->  {data_out = v};
```

For portability, we wish to avoid the use of such modelling code. It is possible to write the code in the language of the design, and combine it with the design model; but we are aiming for a flow that can fit any language of the design implementation.

Instead, the results of Section 2 can be used to establish a correspondence between entry and exit events for the fifo. The condition (1) in that section can be coded in the temporal layer of PSL as

```
assert
  forall v in data_values:
    !(exit_event & data_out = v) until (entry_event & data_in = v);
```

while (2), which applies to two copies of the design, can be coded as

```
assert
  forall v, w in data_values :
   !(v = w) ->
     (!(exit_event_1 & data_out_1 = v & exit_event_2 & data_out_2 = w)
      until
      (entry_event_1 & data_in_1 = v & entry_event_2 & data_in_2 = w));
```

This establishes that each event of a data value leaving the fifo can be associated with an event when the same data value enters the fifo. In order to prove that this association is a one-to-one ordered correspondence, we need to prove that

- Every data value that enters the fifo subsequently leaves it;
- For every data value v, if there is precisely one time when v enters the fifo, then there is precisely one time when v leaves it;
- For every pair of data values v and w, if v and w enter the fifo precisely once each, with v entering before w, then v leaves the fifo before w.

These properties can be expressed in the temporal layer of PSL without additional modelling code. Because of the data independence already established, these properties need to be proved only for one specific data value v, or one specific pair of data values v and w.

The motivation for including the sufficient conditions for data independence as an explicit part of the PSL specification in this example is to eliminate modelling code. The fact that the other properties then need to be proved for only one or two data values is a bonus; but it will always be necessary to prove some properties for all data values, as the sufficient conditions themselves are universally quantified over data values. However, the checks of the sufficient conditions often turn out to be rapid in practice, and there are reasons to expect that proofs of the sufficient conditions may be less complex than proofs of other properties. For example, if the model is in fact data independent according to the syntactic definition, then in checking (2), it will be possible first to use model reduction to reduce all the control state to just one copy of the design, since the control state variables in the two copies are equivalent. There will then be localization reduction algorithms [11] - allowing arbitrary behaviour of control variables - that will enable the properties to be proved in no more iterations than the number of registers a data value passes through between input and output.

4 Conclusion

The sufficient conditions proved above provide a novel and flexible way of establishing data independence that is not bound to a particular implementation language or verification tool. It has been demonstrated to work efficiently in practice, and meets concrete verification needs arising in system on chip design.

The sufficient conditions of this paper differ from commonly used syntactic criteria in three ways: in their strength, their complexity, and the applications for which they are appropriate.

The strength of these semantic conditions is not directly comparable with that of common syntatic criteria. On the one hand, the conditions presented here are strong: they exclude both non-deterministic systems and programs in which the operations on data include equality testing. However, as noted in Section 1, our semantic conditions apply to some systems that are not data independent according to syntactic definitions - when there is a register that is sometimes used for data

values, and sometimes for other information, and to know that the system is data independent involves checking that if a value is read from this register and subsequently reaches the data outputs, it must have arrived at the register by some route from the data inputs. The semantic conditions are still strong enough to imply data independence according to the intuitive notion that Definition 4.1 of [18] aims to capture, and to justify checking properties only for small finite data domains.

Syntactic checks of data independence are of course very simple, and can be done in one pass of the program text. As can be seen from Section 3.3 above, the semantic conditions can be expressed as PSL formulae that can be evaluated by checking CTL formulae of the form $A[pWq]$, so in general the complexity of their checks is as great as that of CTL model checking - in particular, it can be exponential in the number of state variables in the model. However, in practice, checks of the conditions (1) and (2) of Section 2 can often be performed quickly, as was seen for the random number generator. If the model is in fact data independent according to the syntactic definition, then, as explained in Section 3.3, it is likely that model checking will be much more efficient than the theoretical worst case.

The efficiency of syntactic checking means that it will normally be the method of choice if it is available. Applying model checking for the semantic conditions of this paper will be appropriate either if syntactic checking is not possible, as in cases where models are not data independent according to syntactic definitions, or where syntactic checking does not fit well into the flow. This is typically the case in large-scale projects, where the specification is required to be suitable for a number of different design languages, implementations and verification tools, and the verification is highly automated.

References

[1] Aagard, M. D., R. B. Jones, T. F. Melham, J. W. O'Leary and C.-J. Seger, *Practical Formal Verification in Microprocessor Design*, IEEE Design and Test of Computers **18** (2001), 16–25.

[2] Andraus, Z. S., and K. A. Sakallah, *Automatic Abstraction and Verification of Verilog Models*, Proceedings of the Design Automation Conference, San Diego, 2004. ACM (2004), 218–223.

[3] Barrett, G., and A. McIsaac, *Model Checking in a Microprocessor Design Project*, Proceedings of the 9th International Conference on Computer Aided Verification, Lecture Notes in Computer Science **1254** (1997), Springer-Verlag, London, 214–225.

[4] Beer, I., S. Ben-David, C. Eisner and A. Landver, *RuleBase: An Industry-Oriented Formal Verification Tool*, Proceedings of the Design Automation Conference (1996), 655–660.

[5] Benalycherif, Lyes, Neil Dunlop and Anthony McIsaac, *Structured Approach to Property Specification and Verification of Hardware IP*, Proceedings of the 18th IEEE International Workshop on Rapid System Prototyping (RSP'07), Porto Alegre, Brazil, May 2007. IEEE Computer Society Press.

[6] Bjesse, P., T. Leonard and A. Mokkedem, *Finding Bugs in an Alpha Microprocessor using Satisfiability Solver*, Proceedings of the 13th International Conference on Computer Aided Verification, Paris 2001. Lecture Notes in Computer Science **2102** (2001), Springer-Verlag, 454–464.

[7] Clarke, E. M., O. Grumberg and D. A. Peled, "Model Checking", MIT Press, 1999.

[8] Gott, R. M., J. R. Baumgartner, P. Roessler and S. E. Joe, *Functional Formal Verification on Designs of pSeries Microprocessors and Communication Subsystems* IBM Journal of Research and Development **49**, No. 4/5 (2005).

[9] Hojati, R., and R. Brayton, *Automatic Data Path Abstraction in Hardware Systems*, Proceedings of the 7th International Conference on Computer Aided Verification (1995).

[10] Hojati, R., D. L. Dill and R. K. Brayton, *Verifying Linear Temporal Properties of Data Insensitive Controllers using Finite Instantiations*, Proceedings of the 15th International Conference on Computer Hardware Description Languages and their Applications (1997).

[11] Kurshan, R.P., "Computer-Aided Verification of Coordinating Processes", Princeton Univeristy Press, 1994.

[12] Lazic, R. S., and D. Nowak, *A Unifying Approach to Data Independence*, Proceedings of the 11th International Conference on Concurrency, Lecture Notes in Computer Science **1877** (2000), Springer-Verlag, 581–595.

[13] Paruthi, V., N. Mansouri and R. Vemuri, *Automatic Data Path Abstraction for Verification of Large Scale Designs*, IEEE International Conference on Computer Design (1998), 192–194.

[14] Raimi, R., and J. Lear, *Analyzing a PowerPC620 Microprocessor Silicon Failure using Model Checking*, Proceedings of the International Test Conference (1997).

[15] Sayer, C., and J. Sonander, *Formal Verification of AMBA Bus Systems*, Information Quarterly **2**, No. 3 (2003), 42–44.

[16] Stangier, C., and U. Holtmann, *Applying Formal Verification with Protocol Compiler*, Euromicro Digital System Design, Warsaw (2001).

[17] Susanto, K. M., and T. Melham, *An AMBA-ARM7 Formal Verification Platform*, ICFEM 2003, Lecture Notes in Computer Science **2885** (2003), Springer, 43–67.

[18] Wolper, P., *Expressing Interesting Properties of Programs in Propositional Temporal Logic*. Proceedings of the 13th Annual Symposium on Principles of Programming Languages, ACM (1986), 184–193.

[19] IEEE Property Specification Language LRM, IEEE Standard 1850, Oct 2005.