The Formal Definition of a Synchronous Hardware-Description Language in Higher Order Logic

Andrew D. Gordon
University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street,
Cambridge CB2 3QG, UK.
Email: Andrew.Gordon@cl.cam.ac.uk

Abstract

If formal methods of hardware verification are to have any impact on the practices of working designers, connections must be made between the languages used in practice to design circuits, and those used for research into hardware verification. SILAGE is a simple dataflow language used for specifying digital signal processing circuits. Higher Order Logic (HOL) is extensively used for research into hardware verification. We have used a novel combination of operational and predicative semantics to define formally a substantial subset of SILAGE by mapping SILAGE definitions into HOL predicates. Here we sketch the method used, discuss what is gained by a formal definition, and explain an immediate practical application—secure transformational design of SILAGE circuits as theorem proving in HOL.

1 Introduction

SILAGE [10] is a hardware description language (HDL) designed for specifying digital signal processing (DSP) algorithms, and is in use at several sites for research into DSP technology. SILAGE is a dataflow language intended for high-level synthesis [16] in which a specification expresses the desired behaviour of a circuit, but is free of any architectural commitment. SILAGE is the source language for synthesis environments including IMEC’s Cathedral-II [6], Philips’ PYRAMID and Phideo [13] and Berkeley’s Hyper [19].

The classic problem with high-level synthesis is the poor performance of automatically synthesised designs. In SILAGE environments, two kinds of high-level transformations are used to improve the quality of synthesis: (1) SILAGE-to-SILAGE transformations applied manually by the human designer—transformational design as described by Samsom [21] for instance—or (2) flowgraph-to-flowgraph transformations applied algorithmically by automatic tools, such as Hyper or Phideo.

In either case, the problem arises of whether the initial and final designs have the same behaviour. Current SILAGE environments offer only lengthy repeated simulation as a remedy. The goal of this project—a collaboration between Cambridge and IMEC—is to build a system to support human-directed SILAGE-to-SILAGE transformations as interactive theorem-proving. In such a system, the transformation of a specific design is proved correct by instantiating HOL theorems that represent general SILAGE transformations. We obtain a correctness proof for part of the design—and in principle reduce dependence on simulation—but as ever correctness of the whole design depends on factors—such as compiler correctness—that are not formally proved [16].

A formal definition is essential if transformations are to be proved using the HOL system, but no formal definition of the SILAGE language exists. We report here the first formal definition of a substantial subset of SILAGE, called MINI-SILAGE [8]. Apart from the immediate application, this work could be of use as the basis of a language standard, and also for verifying algorithmically applied transformations.

2 Elements of SILAGE

As a toy example, here is the Fibonacci series:

Fib@1 = 1;
Fib@2 = 1;
Fib = Fib@1 + Fib@2

Expressions in general stand for signals, which are infinite streams of samples, indexed by time. Time is modelled by the integers—negative times are used for initialisation. The first two lines say that the signal Fib is to have the sample 1 at times -1 and -2 respectively. The third line says that for all non-negative times, the stream Fib is equal to the sum of itself delayed by 1 and 2 samples, Fib@1 and Fib@2 respectively. In other words, for all i ≥ 0, the i-th sample

---

*Work funded as part of the CHEOPS ESPRIT Basic Research Action 3215.

1 Personal communication with Paul Hitfinger, January 1992.
equals the sum of the \((i-1)\)th and \((i-2)\)th samples respectively.

The samples of a signal can be vectors as well as scalars. Here is how to compute the scalar product [14] of two vectors \(A\) and \(B\):

\[
S[0] = 0; \\
(i:1..64) :: \\
begin \\
S[i] = S[i-1] + T[i-1]; \\
T[i-1] = A[i-1] \times B[i-1] \\
end; \\
C = S[64]
\]

Each sample in the scalar signal \(C\) is the scalar product of the corresponding vector samples of the signals of vectors \(A\) and \(B\). The compound definition begins with \((i:1..64)\), and \(S\) whose meaning is equivalent to the macro-expansion of its body, with \(i\) ranging from 1 up to 64.

There are two special categories of expression: a manifest expression is a combination of numeric constants, arithmetic operators and iterator variables, and is intended to be computable at compile-time to a numeric value; a selector, such as \(T[i-1]\), consists of a signal variable, with a possibly empty series of indexing expressions. Expressions used as iterator bounds and as vector indices are required to be manifest. Expressions on the left-hand side of equations are required to be selectors. These restrictions contribute to making Silage specifications executable.

3 Silage definitions as HOL predicates

HOL is a polymorphically typed classical logic, supporting a conventional collection of operators, including \(=, \neg, \wedge, \vee, \exists\) and \(\forall\), with their usual meaning, and with mechanisms to define new types and constants.

The idea of specifying circuits by predicates is well-known in the hardware verification field [9]. Earlier CBEQ [6,12] proposed to model Silage definitions as predicates, but concentrated on how to model bitstring datatypes, and suggested designing an extended language Silage+. In defining MINI-SILAGE [8] we have ignored the issues of low-level datatypes or multi-rate signals, wishing to concentrate on support for transformations, and have defined precisely how to map from a true subset of Silage into HOL predicates.

We take time to be the integers and model signals with samples of type \(\tau\) by the HOL function type:

\[
(\tau) \text{ signal} \stackrel{\text{def}}{=} \text{int} \rightarrow \tau
\]

We model each Silage operation by defining a HOL constant. With these constants one can model each Silage definition as a predicate that must hold of the signals in a correct implementation. Given the constants,

\[
A \sim B \stackrel{\text{def}}{=} \forall i \geq 0. A(i) = B(i) \\
(A \bowtie n) t \stackrel{\text{def}}{=} A(t - n) \\
(A \text{ SIGPLUS } B) t \stackrel{\text{def}}{=} A(t) + B(t)
\]

we can model the Fibonacci specification:

\[
\begin{align*}
\text{Fib}(\neg 1) &= 1 \\
\text{Fib}(\neg 2) &= 1 \\
\text{Fib} \sim (\text{Fib} \bowtie 1) \text{ SIGPLUS} (\text{Fib} \bowtie 2)
\end{align*}
\]

We model vectors in HOL as finite lists of samples, so the numeric vector \(S\), say, in the scalar product example is modelled by a term of HOL type \((\text{num}\ \text{list})\) signal. Given some more constant definitions,

\[
\begin{align*}
\forall i :: m \text{ TO } n, P & \quad \text{def} \quad \forall i. (m \leq i \leq n) \supset P \\
(A ! n) t & \quad \text{def} \quad \text{EL} n (A(t)) \\
(A \text{ SIGMULT } B) t & \quad \text{def} \quad A(t) \times B(t)
\end{align*}
\]

where \(\text{EL} i [a_1, \ldots, a_n] = a_i\), here is the HOL meaning of the iterator from the scalar product example:

\[
\begin{align*}
(\text{S ! i}) & \sim \\
(S ! i) & \sim \quad \forall i. (i \geq 0) \supset P \\
(T ! (i - 1)) & \sim \\
(A ! (i - 1)) & \sim \quad \forall i. (i \geq 0) \supset P
\end{align*}
\]

4 A sketch of the definition

There are two parts of the definition of Mini-Silage in HOL: a written definition [8] and its implementation running in the HOL system. The machine-independent definition on paper is important as a precise human-readable account of Mini-Silage, and as a specification of the implementation. The machine implementation was written after the paper definition was considered complete, and under the discipline that each set or rule in the paper definition corresponded exactly to a type or function in the implementation. Of course, this discipline exposed several defects in the paper definition, which evolved during its implementation, but on completion there is an exact correspondence between the two.

The written definition of MINI-SILAGE follows the style of Standard ML [17] and consists of three parts: BNF rules determine the abstract syntax; static semantics rules determine the well-typed and well-formed expressions and definitions; predicative semantics rules map definitions into HOL terms. Existing informal Silage texts [11, 18] define the concrete syntax precisely, so it has not been redefined. The point of having a formal definition is to make precise certain aspects of well-formedness or program behaviour left vague in the informal texts.

The implementation consists of a tool running in the HOL system that allows the user to have Silage function definitions translated mechanically into HOL constant definitions.

To illustrate the definition we discuss the semantic treatment of initialisations—such as the first two lines of the Fibonacci example—whose abstract syntax takes the form \(e_1 \bowtie e_2 = e_3\), where each \(e_i\) is an expression. Here is the static semantics rule:

\[
\vdash e_1 \bowtie e_2 \Rightarrow \text{Nat}, \text{man} \quad \vdash e_3 \Rightarrow \sigma, \text{man}
\]

\[
\vdash e_1 @ e_2 = e_3 \text{ ok}
\]
The predicate on the bottom says that the initialisation is well-formed, and the rule says that this is so just when the three predicates on the top line hold, each of the form \( \Gamma \vdash e \sigma \sigma_a \), pronounced "expression \( e \) has type \( \sigma \) and attribute \( \sigma_a \)." An attribute \( \sigma \in \{ \text{man}, \text{sel}, \text{other} \} \) indicates whether an expression is manifest, or is a selector, or neither. The rule above requires that expression \( e_1 \) and \( e_2 \) have the same type, \( \sigma_1 \), that \( e_1 \) is a selector, and that \( e_2 \) and \( e_3 \) are manifest numeric expressions. The SILAGE texts attempt to use BNF rules to assert type and well-formedness conditions; the rules used here are more lucid and precise.

Here is the predicative semantics:

\[
[e_1 @ e_2 = e_3] = (e_3) - (e_2)^M = (e_2)^M
\]

If \( \phi \) is a phrase of abstract syntax, \([\phi]\) is its translation into the HOL logic. Any well-formed expression, such as \( e_1 \), can be translated to a signal, written \([e_1]\). A manifest expression, such as \( e_2 \), can also be interpreted as a scalar in HOL, written \([e_2]^M\). The predicate requires that the signal denoted by \( e_1 \) have the sample denoted by \( e_3 \) at negative time \( e_2 \).

One can prove desired properties of the definition itself, for instance:

**Theorem.** If expression \( e \) has type \( \sigma \), its translation \([e]\) is a HOL term of type \([\sigma]\) signal, where \([\sigma]\) is the translation of type \( \sigma \) as a HOL type. ■

5 Application of the definition

We illustrate how the HOL semantics of SILAGE can be used in practice with a pipelining transformation taken from Lippens' study of SILAGE transformations [14]. The initial program is the scalar product program from Section 2. Lippens supposes that the target architecture has both an adder and a multiplier. The form of the iterator in the initial program suggests that \( S[i] \) and \( T[i-1] \) be computed on each cycle of the iteration, which would mean that the two execution units cannot be used concurrently because the computation of \( S[i] \) depends on the value of \( T[i-1] \).

Lippens suggests the following tuned replacement for the iterator:

\[
T[0] = A[0] \times B[0];
(i : 1..63) ::
begin
T[i] = A[i] \times B[i];
S[i] = S[i-1] + T[i-1]
end;
S[64] = S[63] + T[63]
\]

Here the computation of \( T[i-1] \) occurs in the cycle before that of \( S[i] \), so there is no data dependency between the addition and multiplication in the loop body, allowing concurrent use of the execution units.

Lippens had no semantics to prove the transformation correct, but we have proved his example as a HOL theorem, by appealing to the following equational transformations, which are also proved within the HOL system:

\[
\begin{align*}
(i : m..n) :: \text{begin} d_1 ; d_2 \text{ end } &= \left( (i : m..n) :: d_1 ; (i : m..n) :: d_2 \right) \\
(i : m..n) :: d &= \\
\left( d^{(m\%)} ; (i : (m + 1)..n) :: d \right) & \text{ if } m < n \\\n(i : m..n) :: d &= \\
\left( (i : m..(n - 1)) :: d ; d^{(m\%)} \right) & \text{ if } m < n \\\n(i : (m + k)..(n + k)) :: d &= \left( (i : m..n) :: d \left( i + k \right) \right)
\end{align*}
\]

The notation \( d^{(m\%)} \) stands for the outcome of substituting expression \( e \) for each occurrence of variable \( i \) in the definition \( d \).

Other loop pipelining and retiming examples have been mechanically proved, demonstrating the feasibility of this approach for transformational design.

6 Related work

McFarland and Parker [15] report the seminal work on proving HDL transformations correct in terms of a behavioural semantics. Their emphasis was on proving algorithmic transformations correct manually; conversely, our emphasis is on proving manually-applied transformations correct mechanically.

We have deliberately adopted the simplest semantic embedding—programs as predicates—able to support behaviour-preserving transformation so as to focus on the practicality of using an interactive proof assistant to construct correct transformations. Our embedding follows that used for ELLA in HOL [3]; in comparison, our work focuses on transformational design and the need for a written formal definition [2]. Other embeddings of HDLs in theorem provers [20, 7] employ more expressive but more complex semantic embeddings. Busch [5] uses a theorem-prover to develop refinements, but expresses hardware using logic rather than an HDL. We have not investigated refinement of SILAGE programs; if desired, refinements could be represented in HOL as implicative theorems.

There are mechanical transformation systems not based on a general-purpose theorem-prover, such as the Indiana DDD system [4]. The advantage of using an LCF-style theorem-prover such as HOL is that theorems are an abstract type distinguished from sentences of the logic, and so programming errors are unlikely to lead to a proof of a false transformation. Any such "proof" would imply either an inconsistency in higher order logic or an undiscovered bug in the HOL system.

Vemuri [22] considers the completeness of a set of interactively-applied transformations. An interactive system for transformational design based on a theorem-prover need not be based on a fixed set of transformations: any logically provable transformation can be made available by resorting to interactive theorem-proving.

7 Conclusions

We have reported work on the first formal definition of a substantial subset of SILAGE. We combine the language semantics methodology of ML [17] with the idea of circuits as HOL predicates [9] to make a
language definition that is precise and amenable to formal proof. We have implemented the definition in HOL, mechanised the denotations of several SILAGE programs as predicates, and examined how transformational design can be rendered as theorem proving in HOL. Work is in progress in collaboration with IMEC to tackle realistic examples that have arisen in practice, and to find suitable user interfaces to the system.

It is a pleasure to thank Catia Angelo, Richard Boulton, Luc Claesen, Mike Gordon, Brian Graham, Sara Kalvala, Mats Larsson, Wim Ploegaets and Hans Samsom for fruitful discussions about this work, and for comments on drafts of this paper. Catia Angelo first suggested to me the idea of modelling SILAGE definitions as predicates.

References