SMALL: A PROGRAMMING LANGUAGE FOR STATE MACHINE DESIGN

Theodore S. Norvell

Faculty of Engineering
Memorial University of Newfoundland
St. John's, NF, A1B 3X5, Canada
theo@engr.mun.ca http://www.mun.engr.ca/~theo/

ABSTRACT

A small and simple language for sequential design is introduced.

0. INTRODUCTION

Synchronous state-machine design is often expressed using graphical notations such as Algorithmic State Machine (ASM) charts [7] and state transition diagrams. An alternative is to use common hardware description languages such as VHDL [2] or Verilog [6].

For both teaching and design purposes, neither approach is satisfactory. ASM charts and similar notations invite unstructured designs and allow parallelism only at the outermost level. Educational implementations appear not to exist. VHDL and Verilog are geared toward asynchronous behaviour — in expressing synchronous designs the clock signals must be made explicit. Also they are large, complex languages with complex semantic foundations.

SMALL (State Machine Algol-Like Language) is a small and simple imperative programming language, designed specifically for hardware implementation. It bears roughly the same relationship to ASM charts as Algol-like software languages bear to flowcharts.

1. THE SMALL LANGUAGE

1.0. Signals, registers & primitive commands

Data, in SMALL, is carried either by named signals or named registers.

Signals are used to carry data between commands executing concurrently. Data is placed on a signal s by means of an *assert command* $s!E$. The signal s may be referred to in other expressions and will have the value of expression E whenever the command $s!E$ is being executed in the same clock period.

Signals are declared using a command of the form $\mathbf{sig}\, id : T\ C$, where T is a type and C is a command. C is the scope of the declaration.

Each type T is associated with a default value $def.T$. During those clock periods in which there is no assert of a signal, the signal has its default value. For the *bool* type the default value is *false*.

Registers are used to carry data forward in time. Data is placed in a register r by means of an *assignment command* $r \leftarrow E$. The register r may be referred to in any expression. The value of r is the same as the latest value assigned to it in a previous clock period.

Registers are declared using a command of the form $\mathbf{reg}\, id : T\ C$, where T is a type and C is the scope of the register.

1.1. Composition operators

Any execution of a command takes a certain number of consecutive clock periods. The *duration* of (an execution of) a command is the difference between the initial and the final clock period. For example, in the assert and assignment commands we have seen, the initial and the final clock periods are always the same, hence the duration is 0.

The *sequential composition* $C_0\ C_1\ \cdots\ C_n$ of two or more commands is executed by executing the commands consecutively from left to right. The final clock period of C_i is also the initial clock period of C_{i+1}. I call this policy *dovetailing*. The duration of a sequential composition is the sum of the durations of its constituents

For example, a sequence of asserts and assignments has a duration of 0. For example

$$a!\mathit{false} \quad b!a \quad c \leftarrow b$$

assigns *false* to c, but so does

$$c \leftarrow b \quad b!a \quad a!\mathit{false} \qquad .$$

In either case, all three commands start at the same time.

In order to keep everything from happening at once we introduce a command *tick* that has duration 1, but no other effect. For example,

$$c \leftarrow \mathit{true} \quad \mathit{tick} \quad a!c$$

asserts a to have the value *true* during its second and final clock period.

The command *skip* is similar to *tick* in having no effect, but has a duration of 0. It is the identity of sequential composition.

The *parallel composition* **par** $C_0 \parallel C_1 \parallel \cdots \parallel C_n$ **rap** starts all its subcommands at once. The duration is the maximum of the durations of the subcommands.

The *alternative composition* **if** E **then** C_0 **else** C_1 **fi** chooses either C_0 or C_1 based on the value of E during the initial clock period. The initial clock period of the chosen command is the same as the initial clock period of the whole command. If either C_0 or C_1 is *skip*, it may be omitted, together with the preceding keyword.

The looping construct is **loop** *id* C **pool**. Within the loop body C the command **restart** *id* acts as a recursive call to the loop. There are two restrictions on the use of **restart**: it must be a tail call, and there must be at least one *tick* on every path from the start of the loop to the **restart**.

Two abbreviations capture common patterns of looping and ensure that both restrictions are fulfilled: **while** E **do** C **od** abbreviates

```
loop w    if E
          then C  tick  restart w
          fi
pool
```

and **repeat** C **until** E abbreviates

```
loop u    C
          if E
          else tick  restart u
          fi
pool
```

.

In both cases the implicit delay is inserted immediately before the return to the top of the loop.

1.2. Open systems

In order to communicate with the rest of the world there must be some provision for input and output. In SMALL this is accomplished via signals and registers that are declared **global** When commands having the same **global** name are composed, that name must have the same type and kind (register or signal) in both. Any **global** signals and registers of subcommands become global signals of the larger construct.

1.3. Types and expressions

The only types currently supported by SMALL are *bool* and arrays: **array** n **of** T. There are a number of operations that act on booleans and arrays, including arithmetic operations that treat arrays of booleans as numbers using either unsigned-magnitude or two's complement encoding. The assert and assignment commands have forms to allow assertion of or assignment to array elements or segments of arrays.

```
global sig go : bool
global sig done : bool
global sig multiplier : array N of bool
global sig multiplicand : array N of bool
global sig product : array 2 × N of bool
while true
do  reg pl : array N of bool
    reg pu : array N of bool
    reg a : array N of bool
    reg b : array N of bool
    reg count : array ⌈lg.N⌉ of bool
    repeat  pu ← 4 of 0
            count ← ⌈lg.N⌉ of 0
            a ← multiplicand
            b ← multiplier
    until go
    tick
    repeat  sig sum : array N + 1 of bool
            if b[0]
            then sum ! pu uplus a
            else sum ! pu ++[0]
            fi
            pu ← sum[N@1]
            pl ← pl[N − 1@1] ++[sum[0]]
            b ← b[N − 1@1] ++[0]
            count ← count + 1
    until count = (N − 1) as ⌈lg.N⌉ bits
    tick
    done ! true
    product ! pl ++ pu
od
```

Figure 0: A multiplier

1.4. Restrictions

Because of parallelism, it is possible to assign to the same register, or assert the same signal more than once during a single clock period. This is allowed, but there is a restriction that all values given to each location in a single clock period must be the same.

Another restriction requires that assert commands executed in the same clock period must not be circularly dependent. These examples all violate this rule:

$a!a$

$a!b \quad b!a$

$a!b \quad b!\mathbf{not}\,a$

if a **then** $a!true$ **else** $a!false$ **fi**

if a **then** $a!false$ **else** $a!true$ **fi** .

2. EXAMPLE

Figure 0 shows an example program. It demonstrates some of the varieties of expressions in the language. Numbers are represented by arrays such that the least-significant bit is at index 0. Array operations include

$$a!E = (\tau' = \tau \wedge a.\tau = E.\tau \wedge \bar{a}.\tau \wedge \neg\bar{b}.\tau \wedge \neg\bar{q}.\tau \wedge \neg\bar{r}.\tau)$$

$$q \leftarrow E = (\tau' = \tau \wedge r.(\tau+1) = E.\tau \wedge \neg\bar{a}.\tau \wedge \neg\bar{b}.\tau \wedge \neg\bar{r}.\tau \wedge \bar{q}.\tau)$$

$$skip = (\tau' = \tau \wedge \neg\bar{a}.\tau \wedge \neg\bar{b}.\tau \wedge \neg\bar{r}.\tau \wedge \neg\bar{q}.\tau)$$

$$tick = \left(\begin{array}{ll} & \tau' = \tau + 1 \wedge \neg\bar{a}.\tau \wedge \neg\bar{b}.\tau \wedge \neg\bar{r}.\tau \wedge \neg\bar{q}.\tau \\ \wedge & \neg\bar{a}.\tau' \wedge \neg\bar{b}.\tau' \wedge \neg\bar{r}.\tau' \wedge \neg\bar{q}.\tau' \end{array} \right)$$

$$\mathbf{sig}\, c : T\ C = (\exists c, \bar{c} \cdot C \wedge (\forall \hat{\tau} \mid \tau \leq \hat{\tau} \leq \tau' \wedge \neg\bar{c}.\hat{\tau} \cdot c.\hat{\tau} = def.T))$$

$$\mathbf{reg}\, p : T\ C = (\exists p, \bar{p} \cdot C \wedge (\forall \hat{\tau} \mid \tau \leq \hat{\tau} < \tau' \wedge \neg\bar{p}.\hat{\tau} \cdot p.(\hat{\tau}+1) = p.\hat{\tau}))$$

$$\mathbf{global\,sig}\, a : T_a\ C = C$$

$$\mathbf{global\,reg}\, q : T_q\ C = C$$

$$\mathbf{if}\, E\, \mathbf{then}\, C\, \mathbf{else}\, D\, \mathbf{fi} = ((E.\tau \Rightarrow C) \wedge (\neg E.\tau \Rightarrow D))$$

$$C\ D = (\exists \hat{\tau} \cdot C^{\tau'}_{\hat{\tau}} \bigtriangleup D^{\tau}_{\hat{\tau}})$$

$$\mathbf{par}\, C \parallel D\, \mathbf{rap} = ((C\ wait) \bigtriangleup D) \vee (C \bigtriangleup (D\ wait))$$

$$\mathbf{loop}\, l\ w.(\mathbf{restart}\, l)\, \mathbf{pool} = (\sqcap C \mid C = w.C \wedge Prog.C \cdot C)$$

Figure 1: Semantic equations for small

indexing ($A[i]$), segment formation ($A[n@i]$ is the segment of A of length n that starts at index i), catenation ($A +\!\!+ B$), and unsigned addition ($A\ \mathbf{uplus}\ B$).

3. FORMAL SEMANTICS

A *specification* is a predicate on behaviours. Those behaviours for which the predicate is true are considered acceptable to the specification [0].

Behaviours for SMALL are described by variables representing times and time varying functions. We take $\mathsf{xnat} = \mathsf{nat} \cup \infty$ as a set of times. We define $wire.T = \mathsf{xnat} \to T$. To describe the behaviour of a command with a global signal $a : T_a$ and global register $r : T_r$, we use the following variables:

$\tau : \mathsf{xnat}$	The starting time.
$\tau' : \mathsf{xnat}$	The final time.
$a : wire.T_a$	The value of a during each clock period.
$\bar{a} : wire.bool$	Whether a is asserted during each clock period.
$r : wire.T_r$	The value of r during each clock period.
$\bar{r} : wire.bool$	Whether r recieves a new value after each clock period.

An example specification is

$$\tau' = \tau + 1 \wedge \bar{a}.\tau \wedge a.\tau = r.\tau \wedge \bar{r}.\tau \wedge r.(\tau + 1) = 10\ .$$

A command that implemented this specification would have a duration of 1, would assert a to have the same value as r during its first clock period, and would assign 10 to r between its two clock periods.

$$C \bigtriangleup D = \left(\begin{array}{r} \exists \dot{a}, \dot{b}, \dot{q}, \dot{r}, \ddot{a}, \ddot{b}, \ddot{q}, \ddot{r} \cdot C^{\bar{a},\bar{b},\bar{q},\bar{r}}_{\dot{a},\dot{b},\dot{q},\dot{r}} \wedge D^{\bar{a},\bar{b},\bar{q},\bar{r}}_{\ddot{a},\ddot{b},\ddot{q},\ddot{r}} \\ \wedge\, \bar{a} = (\dot{a} \vee \ddot{a}) \wedge \bar{b} = (\dot{b} \vee \ddot{b}) \\ \wedge\, \bar{q} = (\dot{q} \vee \ddot{q}) \wedge \bar{r} = (\dot{r} \vee \ddot{r}) \end{array} \right)$$

$$wait = (\tau' \geq \tau \wedge (\forall \hat{\tau} \mid \tau \leq \hat{\tau} \leq \tau' \cdot \neg\bar{a}.\hat{\tau} \wedge \neg\bar{b}.\hat{\tau} \wedge \neg\bar{r}.\hat{\tau} \wedge \neg\bar{q}.\hat{\tau}))$$

$$Prog.C = (\tau' \geq \tau \sqsubseteq C)$$

$$(C \sqsubseteq D) = (\forall \tau, \tau', a, \bar{a}, b, \bar{b}, q, \bar{q}, r, \bar{r} \cdot C \Leftarrow D)$$

Figure 2: Auxilliary notations

The formal semantics is given by equating primitive commands to specifications and by equating composition operators (parallel, sequential, alternative, and looping) to operators on specifications. The equations for these definitions are given in Figures 1 and 2. For simplicity, in these equations we assume there are two global signals a and b, and two global registers q and r. Assignment to and signalling of array elements and segments is not dealt with. The operator $C \bigtriangleup D$ denotes the combining of two components in such a way that both control their common set of signals and registers.

The looping operator requires some explanation. The relation $C \sqsubseteq D$ denotes refinement; a specification C is refined by a specification D iff every behavior accepted by D is also accepted by C. This partial order gives rise to a complete lattice of specifications in which $\sqcap$ is the join. The predicate $Prog.C$ means that C ensures that the final time is no less than the initial time. If we regard a loop's body as a function, w, from specifications to specifications, the loop is the least-refined progressive fixed-point of this function [3, 4].

4. SPECIFICATIONS

By extending the language with a few constructs intended for specification purposes, we obtain an expressive language of which SMALL programs form a subset. For example, defining $\langle E \rangle$ to mean $\tau' = \tau \wedge E.\tau$ and $\Box E$ to mean $\tau' \geq \tau \wedge (\forall \hat{\tau} \mid \tau \leq \hat{\tau} < \tau' \cdot E.\hat{\tau})$, we obtain a form of interval temporal logic [1]. A multiplier with a simple handshake interface can be specified as:

$$\begin{array}{ll} S = & \mathbf{global\,sig}\ go : bool\ \cdots \\ & \mathbf{while}\ true \\ & \mathbf{do}\ (\ \exists A, B \cdot \\ & \qquad \Box \neg \overline{go} \\ & \qquad \langle go \wedge multiplier = A \wedge multplicand = B \rangle \\ & \qquad tick \\ & \qquad \Box \neg \overline{done} \\ & \qquad \langle done \wedge product = A \times B \rangle\ ) \\ & \mathbf{od} \end{array}$$

A SMALL program P implements this specification iff $S \sqsubseteq P$. Because the various program composition operators are monotone with respect to refinement, the derivation of programs from specifications can proceed in a step-wise fashion.

5. IMPLEMENTATION

The current implementation of the SMALL language translates source programs to an extended form of ASM chart. The ASM chart can then either be simulated or translated to a gate level implementation.

5.0. Translation to ASM charts

SMALL programs can be translated quite directly to an extended form of ASM charts. Each *tick* command is translated to an empty state node (a rectangle). Assignment and assert commands are translated to ordinary nodes (ovals). Alternative compositions give rise to decision nodes (diamonds). The composition operators control how nodes are connected. An extra state node is added at the start of the chart to represent the initial state.

Parallel composition introduces the possibility that a state node may have more than one successor state node. This is usually not considered proper in ASM charts, but causes no difficulty; we simply allow more than one state node to be active in a single clock period. Thus sets of state nodes, rather than the state nodes themselves, represent the states of the finite state machine. Coordinating the termination of parallel processes is done by introducing an extra signal for each process that indicates its completion; an extra state node is added at the end of each process as a wait state.

Simulation consists of interpreting the ASM charts.

5.1. Translation to gate-level

The translation to gate-level is from the ASM charts. The state-encoding currently used is a one-hot encoding —i.e. there is one flip-flop per state node—, although, as noted above, more than one state node may be active at one time. Encoding schemes that use fewer flip-flops could be devised. Optimizations such as the sharing of components among commands that can not be active in the same cycle are not currently done and will probably be implemented using source-to-source translation.

6. CONCLUSIONS

The SMALL language provides a structured approach to the design of sequential circuits. The language has a simple semantic basis and can be embedded in a larger specification language.

The most similar work reported in the literature is the Handel language designed and implemented by Page and his group [5]. The most striking difference is that, in SMALL, assignment and assertion commands have a duration of 0 rather than 1; this leads to what I have called 'dovetailing' and allows fast designs to be expressed more easily. Handel has no equivalent to assert commands, but does support communication primitives that include synchronization. Such communication primitives could be added to SMALL.

Future work is needed. First, there are language enhancements. The language lacks several useful constructs such as modules, macros, and a parallel 'for' construct. Second, the question of deriving implementations from their specifications needs to be addressed. SMALL, or a similar language, augmented with specification constructs, provides a good language for expressing specifications, implementations, and intermediate steps. The correctness of each refinement step should be checkable by a theorem prover.

7. ACKNOWLEDGMENTS

This work was supported by the National Sciences and Engineering Research Council. I would like to thank Kong Fook Lai for his initial implementation of the gate-level translation.

8. REFERENCES

[0] Eric C. R. Hehner. Predicative programming. *Communications of the ACM*, 27(2):134–151, 1984.

[1] Ben Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.

[2] Zainalabedin Navabi. *VHDL: Analysis and Modeling of Digital Systems.* McGraw-Hill, 1993.

[3] Theodore S. Norvell. Machine code programs are predicates too. In David Till, editor, *Sixth Refinement Workshop*, Workshops in Computing, pages 188–204. Springer Verlag, 1994.

[4] Theodore S. Norvell. Predicative semantics of loops. In Richard Bird and Lambert Meertens, editors, *Algorithmic Languages and Calculi.* Chapman-Hall, 1997. Forthcoming.

[5] Ian Page. Constructing hardare-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[6] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language.* Kluwer, 1991.

[7] David Winkel and Franklin Prosser. *The Art of Digital Design.* Prentice-Hall, 1980.