Towards Verified Systems

$\boxed{\text{OVERVIEW}}$

Jonathan Bowen (Ed.)

October 7, 1993

$\boxed{\text{DRAFT CHAPTER BY MJCG}}$

Contents

Chapter 1

State transition assertions: a case study

M.J.C. Gordon

Overview

The temporal behaviour of programs that interact with their environment depends on the compiler used and the timing characteristics of the host processor. Working out the details can be messy. As part of the safemos project, an approach to managing this complexity based on special-purpose theorem proving tools, has been developed. Specifications are written in a state transition notation annotated with real-time constraints. Implementations are programs coded in a simple imperative language with assignments, sequencing, conditionals, asynchronous inputs, wait-statements, while-commands and forever-loops. The meaning of programs is defined by a translation to sequences of machine instructions, but automated tools can derive behavioural abstractions, called *state transition assertions* (or STAs) that enable reasoning to be conducted near the source program level.

At the core of the approach are a number of 'laws' for combining STAs. These are derived from the definition of state transition assertions and are thus theorems rather than axioms. These laws combine aspects of Hoare logic [4] and interval temporal logic [3]. A specialized theorem prover automatically generates STAs that describe the behaviour of a program considered as a sequence of assignments and jumps (see Section 1.3). Semi-automatic tools then combine these 'atomic STAs' to derive STAs for straight line code segments and certain looping structures, including wait loops. Finally, a user interactively combines these derived STAs to establish properties of the candidate implementation.

The machine used in the semantics is idealized: it has an unbounded stack and unspecified word-size. Some mechanized tools have been implemented to enable the actual resources required by a given program to be analysed (e.g. to determine the maximum stack depth). After such an analysis has been carried out, a finite machine customized to the program and suitable for physical realization can be computed. It is intended that such machines will implemented directly in hardware (e.g. with field programmable gate arrays) or via a software emulation; in both cases it is necessary to prove that the abstract machine used as a semantic base is correctly implemented. The physical realization of abstract machines is not considered here.

1.1. An example: `Mult`

The example program studied here implements a reactive system which is required to meet hard real-time constraints. It is also required to be resettable from an arbitrary processor state within a hard real-time bound; this is intended to support fault recovery protocols. The approach adopted is to count the number of machine cycles taken by the compiled code. This very fine-grained kind of analysis may be quite inappropriate in some situations; intended applications include low-level communication software, critical systems requiring very rapid responses and the software emulation of hardware functions. The example in this chapter illustrates aspects of all of these.

Programs use a very simple low-level input/output model. Inputs are 'asynchronously' supplied by the environment, outputs are 'memory mapped'. When an input is read, the program gets whatever value the environment happens to be offering at the time of the read. For example, inputs might be provided directly by a sensor. It is assumed inputs are digitized, but not that they are latched. Outputs, on the other hand, are identified with particular program variables whose values are assumed to remain stable until changed. Thus outputs are latched. This particular treatment of input/output was chosen for two reasons: it corresponds to a simple physical implementation and it can be used to model more complex regimes. For example, a regime in which inputs are latched would be modeled by requiring the environment to hold inputs stable between input events. The example in this chapter may be viewed as a study of the fine detail underlying a particular kind of synchronized communication. Juanito Camilleri has studied this topic more generally, with the eventual aim of producing a verified implementation of **occam**-style synchronized communication in terms of the simple input/output model described here [1].

1.1.1. Overview

`Mult` is a multiplier that reacts with the environment via a four-phase handshake. The first phase is a request by the environment that two numbers be read from input lines; the second phase is the reading of these by the program; the third phase is the initiation, by the environment, of the multiplication of the two numbers read in during the second phase; the fourth phase is the computation of the product. At the end of such a cycle the program outputs the computed product whilst awaiting the first phase of a new cycle. The first three phases all take place within a fixed time bound; the duration of the fourth phase depends on the size of the numbers being multiplied.

1.1.2. Informal specification of `Mult`

`Mult` has two data inputs `in1` and `in2` that carry numbers.[1] It has two control inputs `req` and `reset` that carry truthvalues (i.e. single bits). It has one numerical data output `out` and one boolean[2] control output `avail`.

In the normal operation of `Mult` there are just four possibilities, corresponding to the four phases of the handshake: (i) it is waiting to engage in a handshake; (ii) it is reading inputs; (iii) it is waiting to start a multiplication; (iv) it is performing a multiplication. Both (i) and (iii) are 'wait states': the waiting will continue indefinitely until the environment sends the appropriate signal via the `req` input (see below). The other possibilities, (ii) and (iv), are transitions between waiting states.

It is required that no matter what state the host processor is in, if the environment holds `reset` at the value 1 continuously for δ_1 cycles, then the system `Mult` will be reset to the state (i) of waiting to engage in a handshake. In this state the value output on `avail` is 1.

The environment signals the start of a handhake by changing the input `req` from 0 to 1. When this happens, it is required that the system will input the values, m and n say, on inputs `in1` and `in2` and move to a state in which it is waiting to start the multiplication of m and n. This inputting transition is required to complete within δ_2 processor cycles. The system will then wait, outputting 0 on `avail`, until the environment sets the input `req` to 0. When that happens the system will compute $m \times n$, output this value on `out`, and then return to the initial state. This multiplication transition is required to take place within $\delta_3(m)$ cycles. The timing parameters δ_1 and δ_2 are given numbers and δ_3 is a given function.

This specification requires that if `Mult` is waiting to engage in a handshake, then the product $m \times n$ can be computed in $\delta_2 + \delta + \delta_3(m)$ machine cycles, where δ is the number of machine cycles taken by the environment to set `req` to 0 after m and n have been input. But how can the environment know when `Mult` is in the state of waiting to engage in a handshake? It is specified that in this state `avail` has value 1, but the converse is not necessarily the case. For example, the value of `avail` will continue to be 1 for a short time after the environment sets `req` to 1, i.e. during the first part of phase (ii) of the handshake. It is tempting to require in the specification that `avail` be 1 if and only if the system is waiting to engage in a handshake, but such a specification would be unimplementable. This is because there will always be time delays in sensing environmental changes and then communicating the results to outputs, hence there will always be times at which internal state changes have occurred, but not yet been signalled on outputs. In general, when timing is considered, it is not possible to characterize internal states by instantaneous values on outputs. However, if inputs and outputs are observed over a sequence of cycles, then conclusions about internal states can be drawn. For example, if the input `reset` has been 1 for δ_1 cycles, then the specification requires the system to be waiting to engage in a handshake.

---

[1] For simplicity, arbitrary-precision (i.e. 'mathematical') numbers are used here but, at the expense of some arithmetical messiness, finite-precision numbers could have been used instead.

[2] The truthvalues T and F will be represented by the numbers 1 and 0, respectively, because the programming language used only supports one data type: the natural numbers $\mathbb{N}$.

1.1.3. `MultProg`: an implementation of `Mult`

Here is a program that implements `Mult`.

```
0:     MultProg =
1:      FOREVER
2:       avail := 1;
3:       IF INPUT req
4:        THEN avail := 0;
5:             x := INPUT in1;
6:             y := INPUT in2;
7:             AWAIT[NOT(INPUT req); INPUT reset];
8:             out := 0;
9:             IF (x=0) OR (y=0)
10:             THEN  SKIP
11:             ELSE  WHILE (x > 0) AND NOT(INPUT reset)
12:                    DO out := out + y;
13:                       x   := x - 1
14:                    OD
15:           avail := 1
16:       ELSE  SKIP
```

The command `FOREVER` C is an abbreviation for `WHILE TRUE DO` C. The command `AWAIT`$[E_1;\ \ldots;\ E_n]$ loops until one of E_1, …, E_n becomes true (i.e. has the value 1) and then control moves to the next command. Evaluating an expression `INPUT i` reads the current value offered by the environment at input `i` and returns the result. Outputs are identified with program variables, so are set by assignment. The other constructs in `MultProg` should be self-explanatory.

Note that there is some arbitrariness in the placement of assignments. The occurrence of `avail:=1` on line `2` could be moved to before the `FOREVER`-loop. The assignment `avail:=0` is performed before the inputs `in1` and `in2` are read, however the specification only requires `avail` to be 0 when `Mult` is waiting to start the multiplication. The verification given here also works if `avail:=0` is performed after the two reads, or even between them (perhaps this indicates that the specification is inadequate: maybe it should be required that the inputs remain stable only as long as `avail` is 1).

`MultProg` works as follows: initially it is waiting to engage in a handshake by looping between lines `2`, `3` and `16` in the outer `FOREVER`-loop. If the input `req` is 1 then eventually the test at line `3` will be reached and then lines `4`, `5` and `6` will be executed in sequence resulting in `avail` being set to 0 and the values at inputs `in1` and `in2` being read into the variables `x` and `y`, respectively. The program will then loop at the `AWAIT`-command at line `7`. If either `req` is 0 or `reset` is 1 this command will terminate, control will move to line `8` and the multiplication will begin with `out` being initialized to 0. If either of the two numbers to be multiplied are 0, then nothing needs to be done (line `10`), `avail` is set back to 1 (line `15`) and the system is ready again to engage in a handshake. However, if either of `x` or `y` is non-zero, then the product is computed in `out` by repeatedly adding `y` to `out` (lines `11`—`14`). Note that each time around this `WHILE`-loop, `reset` is tested and if found to be 1 the loop is terminated. Once the loop is terminated, which (in the absence of a 1 at `reset`) will be in a time proportional to the value read into `x`, `avail` is set to 1 (line `15`) and control returns to the outer loop (lines `2`, `3`, and `16`) and the system is ready to engage in another handshake, i.e. is back in phase (i).

It is clear by inspection, that if `reset` is held at 1 for sufficiently long then control will eventually move to the outer `FOREVER`-loop. This is because all loops test `reset`.

1.2. More detailed specification of `Mult`

The view of real-time systems taken here is that they are sequential machines. A specification places requirements on the behaviour of a machine and an implementation is an actual machine that meets these requirements. The reason for this rather concrete approach is to try to minimize the gap between abstract models of behaviour and real machines. The device that ultimately runs programs implements a sequential machine, so it helps tie the hardware and software verifications together if they both use the same kind of behavioural model.

A machine is a function $\mathcal{M} : \mathit{inputs} \rightarrow (\mathit{state} \rightarrow \mathit{state})$. $\mathcal{M}$ should be thought of as an instruction processor: if the environment offers an array of inputs ι and the current state is σ, then executing the next instruction results in the state $\mathcal{M}\ \iota\ \sigma$. The state will include a program counter and a memory that associates state variables with data values. This memory is a function from names to values (i.e. numbers). Inputs too are modelled by functions from names to values. It is assumed that at each moment the environment specifies a value ι `i` for each input `i`, i.e. it determines a mapping $\iota : \mathit{name} \rightarrow \mathbb{N}$, that varies with time. For any particular system there will only be a small finite number of inputs (`in1`, `in2`, `req` and `reset` for `Mult`), but this fact does not need to be built-in to the general theory. It will be assumed that some state variables are readable by the environment (`avail` and `out` for `Mult`). During each cycle of execution of $\mathcal{M}$, an instruction is selected and executed, resulting in a new state. If the instruction is an input, then this new state will depend on the inputs supplied by the environment.

In what follows, it is first shown how the program `MultProg` in Section 1.1.3 determines a machine and then how the informal specification `Mult` in Section 1.1.2 can be expressed as a predicate on machines. Finally, a method is outlined for proving that programs meet specifications and it is illustrated using `MultProg` and `Mult`.

1.3. Determining a machine from a program

It is straightforward to define a function that recursively translates a program to a sequence of assignments and conditional jumps. For example, one particular algorithm translates `MultProg` to:

```
 0:    avail := 1
 1:    IF INPUT req THEN SKIP ELSE GOTO 18
 2:    avail := 0
 3:    x := INPUT in1
 4:    y := INPUT in2
 5:    IF NOT(INPUT req) THEN SKIP ELSE GOTO 7
 6:    GOTO 10
 7:    IF INPUT reset THEN SKIP ELSE GOTO 9
 8:    GOTO 10
 9:    GOTO 5
10:    out := 0
11:    IF NOT x OR NOT y THEN SKIP ELSE GOTO 13
12:    GOTO 17
13:    IF x > 0 AND NOT(INPUT reset) THEN SKIP ELSE GOTO 17
14:    out := out + y
15:    x := x - 1
16:    GOTO 13
17:    avail := 1
18:    GOTO 0
```

Assignments and conditional jumps require the evaluation of an expression, which may be arbitrarily large, and thus take an arbitrary amount of time to evaluate. It is thus not immediately clear how a sequence of assignments or jumps can be directly represented by a machine. Another tricky issue concerns input. Consider the conditional jump on line 13: the exact time at which `reset` is read depends on how the expression `x > 0 AND NOT(INPUT reset)` is evaluated.

To specify the detailed semantics of expression evaluation, programs will be further translated to sequences of instructions for a simple stack machine with the following instruction set.

`JMP` n	unconditional jump to instruction n
`JMZ` n	pop stack then jump to instruction n if the result is zero
`JMN` n	pop stack then jump to instruction n if the result is non-zero
`POP`	pop the top of the stack
`OP0` v	push v onto the stack
`OP1` op_1	pop one value from stack, perform unary operation op_1, push result
`OP2` op_2	pop two values from stack, perform binary operation op_2, push result
`GET` x	push the contents of memory location x onto the stack
`INP` i	push the input from i onto the stack
`PUT` x	pop the top of the stack and store the result in memory location x

It is straightforward to define a translation of assignments and jumps to sequences of machine instructions. For example, the conditional jump at line 13 is translated to the following sequence of stack machine instructions:

```
GET x
OP0 0
OP2 >
INP reset
OP1 NOT
OP2 AND
JMZ ...
```

It will be assumed that the operations `>`, `NOT` and `AND` can be performed in one machine cycle, thus the stack machine code determines that `x > 0 AND NOT(INPUT reset)` takes 7 cycles, and the input occurs on the 4th cycle.

The state of the stack machine is a triple (pc, stk, mem) consisting of a program counter $pc : \mathbb{N}$, a stack $stk : \mathrm{seq}\,\mathbb{N}$ and a memory $mem : name \to \mathbb{N}$. The model given here does not specify an upper bound on the length of the stack or size of data. Of course, real machines are finite so any actual implementation will have a bounded stack and a particular word size. The intension is to provide tools that 'fit' a given program into a finite refinement of the machine; this will not be discussed in detail here (though see the example state transition assertion in Section 1.4).

The semantics of the instructions of the stack machine is defined by a function Step. Before defining Step, some auxiliary notation is required. A conditional '*if* b *then* e_1 *else* e_2' will be written as $(b \to\ e_1 \mid e_2)$. The empty sequence is denoted by $\langle\rangle$, $\langle x\rangle$ denotes the sequence with one member, x and $\langle x_1, x_2, \ldots, x_n\rangle$ denotes the sequence (of length n) containing x_1, x_2, …, x_n. The length of a sequence s is denoted by $\#s$. The nth element of a sequence s will be denoted by $s\ n$ and the tail of s will be denoted by $\dagger s$. The tail of $\langle\rangle$ is defined to be $\langle\rangle$ (sequences will be used to represent stacks and the stack manipulating instructions are specified so that popping an empty stack leaves an empty stack). The concatenation of sequences s_1 and s_2 will be denoted by $s_1 \frown s_2$. Note that the result of 'consing' x onto a sequence s is $\langle x\rangle \frown s$. If mem is a function representing a memory (i.e. a function from names to values), then $\mathsf{Store}\ v\ x\ mem$ denotes the memory identical to mem except on argument x, which it maps to v, i.e. the memory updated with value v at x.

The function Step can now be defined. Its type is:

$$\mathsf{Step} : instruction \rightarrow (inputs \rightarrow state \rightarrow state)$$

where:

$$\begin{array}{lcl}
inputs & = & name \rightarrow \mathbb{N} \\
state & = & program{-}counter \times stack \times memory \\
program{-}counter & = & \mathbb{N} \\
stack & = & \mathrm{seq}\ \mathbb{N} \\
memory & = & name \rightarrow \mathbb{N}
\end{array}$$

Step is defined by:

$$\begin{array}{lcl}
\mathsf{Step}\ (\mathtt{JMP}\ n)\ \iota\ (pc, stk, mem) & = & (n, stk, mem) \\
\mathsf{Step}\ (\mathtt{JMZ}\ n)\ \iota\ (pc, stk, mem) & = & ((stk\ 1\ =\ 0\ \rightarrow\ n \mid pc{+}1), \dagger stk, mem) \\
\mathsf{Step}\ (\mathtt{JMN}\ n)\ \iota\ (pc, stk, mem) & = & ((stk\ 1\ =\ 1\ \rightarrow\ n \mid pc{+}1), \dagger stk, mem)) \\
\mathsf{Step}\ (\mathtt{POP})\ \iota\ (pc, stk, mem) & = & (pc{+}1, \dagger stk, mem) \\
\mathsf{Step}\ (\mathtt{OP0}\ v)\ \iota\ (pc, stk, mem) & = & (pc{+}1, \langle v\rangle \frown stk, mem) \\
\mathsf{Step}\ (\mathtt{OP1}\ op_1)\ \iota\ (pc, stk, mem) & = & (pc{+}1, \langle op_1(stk\ 1)\rangle \frown \dagger stk, mem) \\
\mathsf{Step}\ (\mathtt{OP2}\ op_2)\ \iota\ (pc, stk, mem) & = & (pc{+}1, \langle op_2(stk\ 2, stk\ 1)\rangle \frown \dagger\dagger stk, mem) \\
\mathsf{Step}\ (\mathtt{GET}\ x)\ \iota\ (pc, stk, mem) & = & (pc{+}1, \langle mem\ x\rangle \frown stk, mem) \\
\mathsf{Step}\ (\mathtt{INP}\ i)\ \iota\ (pc, stk, mem) & = & (pc{+}1, \langle \iota\ i\rangle \frown stk, mem) \\
\mathsf{Step}\ (\mathtt{PUT}\ x)\ \iota\ (pc, stk, mem) & = & (pc{+}1, \dagger stk, \mathsf{Store}\ (stk\ 1)\ x\ mem)
\end{array}$$

The machine Machine $instrs$ determined by a sequence of instructions $instrs$ is defined by:

$$\begin{array}{l}
\mathsf{Machine}\ instrs\ \iota\ (pc, stk, mem)\ \ = \\
\quad (pc < \#instrs\ \rightarrow\ \mathsf{Step}\ (instrs(pc{+}1))\ \iota\ (pc, stk, mem)\ \mid\ (0, stk, mem))
\end{array}$$

The reason for $pc{+}1$ is because the program counter starts at 0 not 1. Note that if the program counter pc points outside the program (i.e. $pc \geq \#instrs$) then the machine jumps to 0.

If $\mathcal{P}$ is a program, let Compile $\mathcal{P}$ denote the translation of $\mathcal{P}$ to stack machine instructions; this is the composition of the translation to assignments and jumps with the translation of these to machine instructions. The machine determined by $\mathcal{P}$ is thus Machine(Compile $\mathcal{P}$). For example, the machine corresponding to `MultProg` is denoted by Machine(Compile `MultProg`); this will be called `MultMachine`.

1.4. State transition assertions

Specifications are formalized as predicates on machines. The informal specification of `Mult` given in Section 1.1 involved a number of transitions between wait states. These can be represented using a kind of assertion, called a *state transition assertion* (or STA), that combines aspects of the 'leads-to' and 'until' operators of temporal logic and also resembles a state delta [5]. The general form of an STA is:

$$\mathcal{M} \models A \xrightarrow[P]{Q} B$$

where:

- $\mathcal{M} : inputs \to state \to state$ is a machine;
- $A : state \to \mathbb{B}$ is called the *state precondition*;
- $B : state \to \mathbb{B}$ is called the *state postcondition*;
- $P : \mathrm{seq}\, inputs \to \mathbb{B}$ is called the *input precondition*;
- $Q : \mathrm{seq}\, state \to \mathbb{B}$ is called the *output postcondition*.

The intuition behind state transition assertions is straightforward: if $\mathcal{M}$ is in a state satisfying A and a sequence of inputs arrives that satisfies P, then a state satisfying B will be reached and the sequence of intermediate states will satisfy Q. The formal definition is slightly delicate as it has to cover the possibility that inputs start to arrive satisfying P, but then stop satisfying it before a state satisfying B is reached. A *trace* of machine $\mathcal{M}$ is an infinite sequence $\langle(\iota_0, \sigma_0), (\iota_1, \sigma_1), \ldots, (\iota_n, \sigma_n), \ldots\rangle$ such that $\sigma_{m+1} = \mathcal{M}\ \iota_m\ \sigma_m$ for all m. It is an A-trace iff $A\ \sigma_0$. Observe that the state entered by $\mathcal{M}$ after a sequence $\iota_0, \ldots, \iota_n$ of inputs have arrived is σ_{n+1}. Thus the machine generates the sequence $\langle\sigma_1, \ldots, \sigma_n, \sigma_{n+1}\rangle$ of states from the inputs $\iota_0, \ldots, \iota_n$. With this observation in mind, the following auxiliary concepts are defined.

- *B succeeds at n in trace* $\langle(\iota_0, \sigma_0), (\iota_1, \sigma_1), \ldots, (\iota_n, \sigma_n), \ldots\rangle$ iff $B\ \sigma_{n+1}$.
- *P fails at n in trace* $\langle(\iota_0, \sigma_0), (\iota_1, \sigma_1), \ldots, (\iota_n, \sigma_n), \ldots\rangle$ iff $\neg P\langle\iota_0, \ldots, \iota_n\rangle$.
- *Q holds until n in trace* $\langle(\iota_0, \sigma_0), (\iota_1, \sigma_1), \ldots, (\iota_n, \sigma_n), \ldots\rangle$ iff $Q\langle\sigma_1, \ldots, \sigma_m\rangle$ for all m such that $1 \le m \le n{+}1$.

The state transition assertion:

$$\mathcal{M} \models A \xrightarrow[P]{Q} B$$

holds iff for every A-trace τ of $\mathcal{M}$ there exists an n such that (i) either B succeeds at n in τ or P fails at n in τ and (ii) Q holds until the first such n in τ.

An example of a state transition assertion is shown below using a number of notational conventions that are explained immediately afterwards. It is true of a machine $\mathcal{M}$ if whenever the predicate $\mathsf{Available}$ is true and the variables $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$ have the values x, y, z respectively, then as long as the $\mathtt{reset}$ line keeps at 0, a sequence of steps of length less than $\delta\ x$ will be traversed in which $\mathtt{y}$ remains stable with value y, $\mathsf{Available}$ is false and the length of the stack is less than d. Furthermore, the sequence ends in a state in which $\mathsf{Available}$ is false, $\mathtt{x}$ is 0, $\mathtt{y}$ is y and $\mathtt{z}$ is $x \times y$.

$$\mathcal{M} \quad \models \quad \begin{array}{l} \mathsf{Available} \\ \mathtt{x} \equiv x \\ \mathtt{y} \equiv y \\ \mathtt{z} \equiv z \end{array} \quad \xrightarrow[\textstyle [\mathtt{reset} \equiv 0]]{\begin{array}{c} \mathsf{By}(\delta\ x) \\ \left[\begin{array}{l} \mathtt{y} \equiv y \\ \neg\mathsf{Available} \\ \mathsf{StackMax}\ \mathsf{d} \end{array}\right] \end{array}} \quad \begin{array}{l} \neg\mathsf{Available} \\ \mathtt{x} \equiv 0 \\ \mathtt{y} \equiv y \\ \mathtt{z} \equiv x \times y \end{array}$$

Vertical stacking means conjunction. The notation $\mathtt{v} \equiv x$ (where $\mathtt{v}$ is a name and x a value) is overloaded; it is used both for predicates on states and for predicates on inputs:

$$\begin{array}{lll} (\mathtt{v} \equiv x)(pc, stk, mem) & =_{def} & mem(\mathtt{v}) = x \\ (\mathtt{v} \equiv x)\iota & =_{def} & \iota(\mathtt{v}) = x \end{array}$$

$\mathsf{StackMax}\ m$ is true of a state if the length of the stack is less than m.

$$\mathsf{StackMax}\ m\ (pc, stk, mem) \quad =_{def} \quad \#stk \leq m$$

If μ is either a state or an input, M is either a predicate on states or a predicate on inputs, then:

$$\begin{array}{lll} (\neg M)\,\mu & =_{def} & \neg(M\ \ \mu) \\ [M]\langle\mu_1, \ldots, \mu_n\rangle & =_{def} & M\ \ \mu_1 \wedge \ldots \wedge M\ \ \mu_n \\ (\mathsf{By}\ m)\langle\mu_1, \ldots, \mu_n\rangle & =_{def} & n \leq m \end{array}$$

The notation $[M]$ asserts that M holds at all points in a sequence, so it is analogous to the modal formula $\Box M$. As a mnemonic, think of sawing the box operator $\Box$ in two and writing the first half before M and the other half after M.

1.4.1. Holding states

Part of the specification of `Mult` is that it remain waiting to engage in a handshake as long as the request line `req` is 0. If 'waiting to engage in a handshake' is represented by the predicate $\mathsf{Available}$, then this part of the specification can be represented by:

$$\mathtt{Mult} \quad \models \quad \mathsf{Available} \xrightarrow[\textstyle [\mathtt{req} \equiv 0]]{\mathsf{By}\ 1} \mathsf{Available}$$

which will be abbreviated to:

$$\mathtt{Mult} \quad \models \quad \mathtt{req} \equiv 0\ \textbf{Holds}\ \mathsf{Available}$$

notice that "By 1" means "1 host machine cycle" not "1 program step" (whatever that might mean).

Whilst `Mult` is waiting to engage in a handshake it should be outputting the result of the previous handshake on `out` and outputting 1 on `avail`. Therefore in the actual specification given in Section 1.5, instead of the predicate $\mathsf{Available}$, a parameterized predicate $\mathit{Ready}\ z$ is used, with the interpretation "ready to engage in a handshake whilst outputting z on `out` and 1 on `avail`". In general, waiting states are characterized by a parameterized predicate (e.g. $\mathit{Ready}\ z$, see Section 1.8.4), an invariant (e.g. $\mathtt{avail} \equiv 1$ and $\mathtt{out} \equiv z$) and a holding condition (e.g. $\mathtt{req} \equiv 0$). In the diagram representing the formal specification of `Mult`, the existence of such a waiting state is represented graphically by:

$\mathit{Ready}\ z$
$\mathtt{avail} \equiv 1$ $\mathtt{out} \equiv z$
$\mathtt{req} \equiv 0$

This abbreviates the conjunction of an STA representing the holding condition and a formula expressing the invariant (which uses the logical operators $\bigwedge$ and $\Longrightarrow$ which are explained below).

$$\mathtt{Mult}\ \models\ \mathtt{req} \equiv 0\ \mathbf{Holds}\ (\mathit{Ready}\ z) \quad \wedge \quad \mathit{Ready}\ z \Longrightarrow \mathtt{avail} \equiv 1 \bigwedge \mathtt{out} \equiv z$$

where, letting μ range over states or inputs as before:

$$(M_1 \bigwedge M_2)\,\mu \quad =_{def} \quad (M_1\ \mu) \wedge (M_2\ \mu)$$

$$(M_1 \Longrightarrow M_2) \quad =_{def} \quad \forall \mu\,.\,(M_1\ \mu) \Rightarrow (M_2\ \mu)$$

Other similar notation used later includes:

$$\begin{aligned}
\mathsf{True}\ \mu \quad &=_{def} \quad \mathsf{T} \\
(M_1 \Rrightarrow M_2)\,\mu \quad &=_{def} \quad M_1\ \mu \Rightarrow M_2\ \mu \\
(M_1 \bigvee M_2)\,\mu \quad &=_{def} \quad M_1\ \mu \vee M_2\ \mu \\
(\exists x.\ M\ x)\,\mu \quad &=_{def} \quad \exists\, x.\ (M\ x)\,\mu \\
(P_1 \bigwedge P_2)\langle\mu_1,\ldots,\mu_n\rangle \quad &=_{def} \quad P_1\langle\mu_1,\ldots,\mu_n\rangle \wedge P_2\langle\mu_1,\ldots,\mu_n\rangle \\
(P_1 \bigvee P_2)\langle\mu_1,\ldots,\mu_n\rangle \quad &=_{def} \quad P_1\langle\mu_1,\ldots,\mu_n\rangle \vee P_2\langle\mu_1,\ldots,\mu_n\rangle
\end{aligned}$$

Notice that there are three different kinds of implication: ordinary logical implication $\Rightarrow$ and two relations between predicates $\Rrightarrow$ and $\Longrightarrow$. The relations between predicates are connected by $M_1 \Longrightarrow M_2 \;=\; \forall \mu\,.\,(M_1 \Rrightarrow M_2)\,\mu$.

1.5. Formal specification of `Mult`

A program implementing `Mult` must be able to cycle within two sets of states representing waiting to engage in a handshake and waiting to start a multiplication. Thus the formal specification asserts the existence of two predicates representing these sets of states. In addition, an implementation must support various invariants and transitions, which can be expressed using STAs. The complete specification can be represented by the following diagram, which represents a conjunction of STAs (details below).

$$\mathsf{True} \xrightarrow[{[\texttt{reset} \equiv 1]}]{\mathsf{By}\ \delta_1} \begin{array}{|c|}\hline \mathit{Ready}\ z \\ \hline \texttt{avail} \equiv 1 \\ \texttt{out} \equiv z \\ \hline \texttt{req} \equiv 0 \\ \hline \end{array} \xrightarrow[\left[\begin{array}{l}\texttt{req} \equiv 1 \\ \texttt{reset} \equiv 0 \\ \texttt{in1} \equiv x \\ \texttt{in2} \equiv y\end{array}\right]]{\begin{array}{l}\mathsf{By}\ \delta_2 \\ {[\texttt{out} \equiv z]}\end{array}} \begin{array}{|l|}\hline \mathit{Ack}\ x\ y\ z \\ \hline \texttt{avail} \equiv 0 \\ \texttt{x} \equiv x \\ \texttt{y} \equiv y \\ \texttt{out} \equiv z \\ \hline \texttt{req} \equiv 1 \\ \hline \end{array} \xrightarrow[\left[\begin{array}{l}\texttt{req} \equiv 0 \\ \texttt{reset} \equiv 0\end{array}\right]]{\mathsf{By}(\delta_3\ x)} \begin{array}{|l|}\hline \mathit{Ready}(x \times y) \\ \hline \texttt{avail} \equiv 1 \\ \texttt{out} \equiv x \times y \\ \hline \texttt{req} \equiv 0 \\ \hline \end{array}$$

The predicate *Ready* z is true of states that are passed through whilst waiting to engage in a handshake. During this waiting, `out` has value z. The predicate *Ack* $x\ y\ z$ is true of states that are passed through whilst waiting to start a multiplication. During this waiting `x`, `y`, `z` have the values x, y, z, respectively. This diagram defines a predicate $\mathsf{MultSpec}(\delta_1, \delta_2, \delta_3)$ on machines $\mathcal{M}$ by the following formula (the abbreviations Reset, ReadyInv, ReadyHold, ReadyToAck, AckInv, AckHold, AckToReady are explained later).

$$\begin{array}{l}
\mathsf{MultSpec}\ (\delta_1, \delta_2, \delta_3)\ \mathcal{M} = \\
\quad \exists\, \mathit{Ready}\ \mathit{Ack}. \\
\quad\quad (\mathcal{M}\ \models\ \mathsf{Reset}(\delta_1, \mathit{Ready})) \qquad \wedge \\
\quad\quad (\forall\, z.\ \mathsf{ReadyInv}(\mathit{Ready}, z)) \qquad \wedge \\
\quad\quad (\forall\, z.\ \mathcal{M}\ \models\ \mathsf{ReadyHold}(\mathit{Ready}, z)) \qquad \wedge \\
\quad\quad (\forall\, x\ y\ z.\ \mathcal{M}\ \models\ \mathsf{ReadyToAck}(\delta_2, \mathit{Ready}, \mathit{Ack}, x, y, z)) \qquad \wedge \\
\quad\quad (\forall\, x\ y\ z.\ \mathsf{AckInv}(\mathit{Ack}, x, y, z)) \qquad \wedge \\
\quad\quad (\forall\, x\ y\ z.\ \mathcal{M}\ \models\ \mathsf{AckHold}(\mathit{Ack}, x, y, z)) \qquad \wedge \\
\quad\quad (\forall\, x\ y\ z.\ \mathcal{M}\ \models\ \mathsf{AckToReady}(\delta_3, \mathit{Ready}, \mathit{Ack}, x, y, z))
\end{array}$$

The reset condition $\mathsf{Reset}(\delta_1, \mathit{Ready})$ asserts that if `reset` is held equal to 1 for at least δ_1 then the system will be in a state satisfying *Ready* z, for some z.

$$\mathsf{Reset}(\delta_1, \mathit{Ready}) \ =_{def}\ \mathsf{True} \xrightarrow[{[\texttt{reset} \equiv 1]}]{\mathsf{By}\ \delta_1} \exists z.\ \mathit{Ready}\ z$$

States satisfying *Ready* z should output 1 on `avail` and z on `out`.

$$\mathsf{ReadyInv}(\mathit{Ready}, z) \ =_{def}\ \mathit{Ready}\ z \Longrightarrow \texttt{avail} \equiv 1 \ \wedge\ \texttt{out} \equiv z$$

States continue to satisfy *Ready* z as long as `req` is 0.

$$\mathsf{ReadyHold}(\mathit{Ready}, z) \ =_{def}\ \texttt{req} \equiv 0\ \textbf{Holds}\ (\mathit{Ready}\ z)$$

If the system is in a state satisfying *Ready* z and for at least δ_2 cycles `req` is held at 1, `reset` at 0, `in1` at x and `in2` at y, then the system will be in a state satisfying *Ack* $x\ y\ z$.

$$\mathsf{ReadyToAck}(\delta_2, \mathit{Ready}, \mathit{Ack}, x, y, z) \ =_{def}\ \mathit{Ready}\ z \xrightarrow[\left[\begin{array}{l}\texttt{req} \equiv 1\ \wedge\ \texttt{reset} \equiv 0 \\ \texttt{in1} \equiv x\ \wedge\ \texttt{in2} \equiv y\end{array}\right]]{\mathsf{By}\ \delta_2} \mathit{Ack}\ x\ y\ z$$

States satisfying $Ack\ x\ y\ z$ should output 0 on `avail` and x, y, z on `x`, `y`, `out`, respectively.

$$\mathsf{AckInv}(Ack, x, y, z) \ =_{def}\ Ack\ x\ y\ z \Longrightarrow \texttt{avail} \equiv 0 \wedge \texttt{x} \equiv x \wedge \texttt{y} \equiv y \wedge \texttt{out} \equiv z$$

States continue to satisfy $Ack\ x\ y\ z$ as long as `req` is 1 and `reset` is 0.

$$\mathsf{AckHold}(Ack, x, y, z) \ =_{def}\ (\texttt{req} \equiv 1 \wedge \texttt{reset} \equiv 0)\ \mathbf{Holds}\ (Ack\ x\ y\ z)$$

If the system is in a state satisfying $Ack\ x\ y\ z$ and then for at least $\delta_3\ x$, `req` is held at 0 and `reset` at 0 then the system will be in a state satisfying $Ready(x \times y)$.

$$\mathsf{AckToReady}(\delta_3, Ready, Ack, x, y, z) \ =_{def}\ Ack\ x\ y\ z \xrightarrow[{[\texttt{req} \equiv 0\ \wedge\ \texttt{reset} \equiv 0]}]{\mathsf{By}(\delta_3\ x)} Ready(x \times y)$$

1.6. Correctness of `MultProg`

The program `MultProg` is correct if the machine, `MultMachine`, that it defines satisfies the predicate $\mathsf{MultSpec}(\delta_1, \delta_2, \delta_3)$. In fact, it follows that:

$$\texttt{MultSpec}\ (43,\ 13,\ \lambda x.\ 30 + (15 \times x))\ \texttt{MultMachine}$$

which establishes correctness with the timing parameters $\delta_1 = 43$, $\delta_2 = 13$ and the function δ_3 defined by $\delta_3\ x = 30 + (15 \times x)$.

The rest of this chapter is devoted to outlining how such correctness results can be proved. Mechanized theorem proving tools are used since there is a large amount of detail in even small examples.[3]

1.7. Generating atomic STAs

The program is translated to a sequence of *intermediate commands* that are either assignments or jumps and then two STAs are generated for each intermediate command. The first of these describes transitions from the beginning of a comand to its end; the second describes transitions from anywhere inside the command to its end. Before explaining this the formalization of “beginning”, “inside” and “end” must be given.

Figure 1.1 shows both the intermediate commands and the final machine instructions for `MultProg`. Consider the intermediate command with number `13`: the beginning of this is the machine instruction numbered `27` (i.e. `GET x`) and the end of it is after the machine instruction numbered `33` (i.e. `JMP 42`). The execution of `MultProg` is said to be inside intermediate command number `13` if it is executing a machine instruction whose number is in the set $\{\texttt{27}, \texttt{28}, \texttt{29}, \texttt{30}, \texttt{31}, \texttt{32}, \texttt{33}\}$. Both intermediate commands and machine instructions will be indexed by their position. A program $\mathcal{P}$ defines, via the compiler, a mapping $\mathsf{Positions}(\mathcal{P})$ from command numbers to instruction numbers in which each command number is mapped to the number of its first instruction. For example, $\mathsf{Positions}(\texttt{MultProg})$ is the following mapping.

[3] 383,547 primitive inference steps were performed, mostly automatically, to verify `MultProg`.

```
0:  avail := 1 ......................................... 0:  OP0 1
                                                         1:  PUT avail
1:  IF INPUT req THEN SKIP ELSE GOTO 18 .................. 2:  INP req
                                                         3:  JMZ 44
2:  avail := 0 ......................................... 4:  OP0 0
                                                         5:  PUT avail
3:  x := INPUT in1 ..................................... 6:  INP in1
                                                         7:  PUT x
4:  y := INPUT in2 ..................................... 8:  INP in2
                                                         9:  PUT y
5:  IF NOT(INPUT req) THEN SKIP ELSE GOTO 7 .............. 10: INP req
                                                         11: OP1 NOT
                                                         12: JMZ 14
6:  GOTO 10 ............................................ 13: JMP 18
7:  IF INPUT reset THEN SKIP ELSE GOTO 9 ................. 14: INP reset
                                                         15: JMZ 17
8:  GOTO 10 ............................................ 16: JMP 18
9:  GOTO 5 ............................................. 17: JMP 10
10: out := 0 ........................................... 18: OP0 0
                                                         19: PUT out
11: IF NOT x OR NOT y THEN SKIP ELSE GOTO 13 ............. 20: GET x
                                                         21: OP1 NOT
                                                         22: GET y
                                                         23: OP1 NOT
                                                         24: OP2 OR
                                                         25: JMZ 27
12: GOTO 17 ............................................ 26: JMP 42
13: IF x > 0 AND NOT(INPUT reset) THEN SKIP ELSE GOTO 17 .. 27: GET x
                                                         28: OP0 0
                                                         29: OP2 >
                                                         30: INP reset
                                                         31: OP1 NOT
                                                         32: OP2 AND
                                                         33: JMZ 42
14: out := out + y ..................................... 34: GET out
                                                         35: GET y
                                                         36: OP2 +
                                                         37: PUT out
15: x := x - 1 ......................................... 38: GET x
                                                         39: OP1 pre
                                                         40: PUT x
16: GOTO 13 ............................................ 41: JMP27
17: avail := 1 ......................................... 42: OP0 1
                                                         43: PUT avail
18: GOTO 0 ............................................. 44: JMP 0
```

Figure 1.1. Intermediate commands and machine instructions for `MultProg`

$$
\begin{array}{lllllll}
\{0 \mapsto 0, & 1 \mapsto 2, & 2 \mapsto 4, & 3 \mapsto 6, & 4 \mapsto 8, & 5 \mapsto 10, & 6 \mapsto 13, \\
7 \mapsto 14, & 8 \mapsto 16, & 9 \mapsto 17, & 10 \mapsto 18, & 11 \mapsto 20, & 12 \mapsto 26, & 13 \mapsto 27, \\
14 \mapsto 34, & 15 \mapsto 38, & 16 \mapsto 41, & 17 \mapsto 42, & 18 \mapsto 44\} & &
\end{array}
$$

Recall that the state of the stack machine is a triple (pc, stk, mem) consisting of a program counter $pc : \mathbb{N}$, a stack $stk : \mathrm{seq}\,\mathbb{N}$ and a memory $mem : name \to \mathbb{N}$. Define:

$$
\begin{array}{lcl}
\mathsf{At}(\mathcal{P})\ n\ (pc, stk, mem) & = & (pc\ =\ \mathsf{Positions}(\mathcal{P})(n)) \\
\mathsf{In}(\mathcal{P})\ n\ (pc, stk, mem) & = & (\mathsf{Positions}(\mathcal{P})(n) \leq pc < \mathsf{Positions}(\mathcal{P})(n+1))
\end{array}.
$$

Then $\mathsf{At}(\mathcal{P})n$ is the predicate on states that is true of σ iff σ is at the beginning of the intermediate command numbered n in program $\mathcal{P}$ and $\mathsf{In}(\mathcal{P})n$ is the predicate on states that is true of σ iff σ is inside command n.

Consider a program $\mathcal{P}$ containing an assignment at position $\mathtt{n}_2$:

```
    :
n1:  ...
n2:  x := ((INPUT in) + y)
n3:  ...
    :
```

Given a list of significant state variables (e.g. `x`, `y` and `z`), the STA generator will automatically deduce two STAs for the command at $\mathtt{n}_2$:

$$
\mathsf{Machine}(\mathsf{Compile}\ \mathcal{P})\ \models\
\begin{array}{l} \mathsf{At}(\mathcal{P})\ \mathtt{n}_2 \\ \mathtt{x} \equiv x \\ \mathtt{y} \equiv y \\ \mathtt{z} \equiv z \end{array}
\ \xrightarrow[{[\mathtt{in} \equiv in]}]{\mathsf{By}\ 4\ \left[\begin{array}{l} \mathsf{In}(\mathcal{P})\ \mathtt{n}_2 \\ \mathtt{y} \equiv y \\ \mathtt{z} \equiv z \end{array}\right]}\
\begin{array}{l} \mathsf{At}(\mathcal{P})\ \mathtt{n}_3 \\ \mathtt{x} \equiv in{+}y \\ \mathtt{y} \equiv y \\ \mathtt{z} \equiv z \end{array}
$$

$$
\mathsf{Machine}(\mathsf{Compile}\ \mathcal{P})\ \models\
\begin{array}{l} \mathsf{In}(\mathcal{P})\ \mathtt{n}_2 \\ \mathtt{x} \equiv x \\ \mathtt{y} \equiv y \\ \mathtt{z} \equiv z \end{array}
\ \xrightarrow[\mathsf{True}]{\mathsf{By}\ 4\ \left[\begin{array}{l} \mathsf{In}(\mathcal{P})\ \mathtt{n}_2 \\ \mathtt{y} \equiv y \\ \mathtt{z} \equiv z \end{array}\right]}\
\begin{array}{l} \mathsf{At}(\mathcal{P})\ \mathtt{n}_3 \\ \mathtt{y} \equiv y \\ \mathtt{z} \equiv z \end{array}
$$

The first of these asserts that if the input `in` is held stable with value in, then there is a transition taking at most four cycles from the beginning of the comand at $\mathtt{n}_2$ to the beginning of the command at $\mathtt{n}_3$. During this transition only states in $\mathtt{n}_2$ are passed through and the values of `y` and `z` are unchanged, but the value of `x` changes to $in{+}y$, where y is the value of the variable `y`.

The second of these asserts that under arbitary input conditions, there is a transition from anywhere in $\mathtt{n}_2$ to the begining of $\mathtt{n}_3$. This transition takes at most four cycles, only passes through states in $\mathtt{n}_2$ and doesn't change the values of `x` and `y`.

The implemented tool also automatically proves that the stack will grow by at most two during the transition, but because this feature is not used in the `Mult` example the details are not discussed here.

The first stage in verifying `MultProg` is to generate two STAs for each of the nineteen intermediate commands. The result of this will not be shown here due to lack of space. Note, however, that users of the verification tool are not expected to have to study these atomic STAs in detail; they are fed into other tools that derive higher level results. These tools combine the atomic STAs using various derived laws.

1.8. Laws for combining STAs

Some of the laws for combining STAs are analogous to rules of Hoare logic. In what follows P, Q etc. will range over predicates on sequences, p, q etc. will range over predicates on the elements of sequences and A, B etc. will range over predicates on states.

Each law applies to an arbitrary machine. For conciseness, "$\mathcal{M} \models$" has been omitted from STAs in the hypothesis and conclusion of the laws.

1.8.1. The consequence rule

The following is similar to the rule of consequence of Hoare logic. It allows preconditions to be strengthened and postconditions to be weakened

$$\frac{A' \Longrightarrow A \qquad P' \Longrightarrow P \qquad A \xrightarrow[P]{Q} B \qquad B \Longrightarrow B' \qquad Q \Longrightarrow Q'}{A' \xrightarrow[P']{Q'} B'}$$

1.8.2. The sequencing rule

The sequencing rule allows sequences of transitions to be combined into a single long transition.

$$\frac{A \xrightarrow[{[p_1]}]{\mathsf{By}\ \delta_1\ \wedge\ [q_1]} B \qquad B \xrightarrow[{[p_2]}]{\mathsf{By}\ \delta_2\ \wedge\ [q_2]} C}{A \xrightarrow[{[p_1\ \wedge\ p_2]}]{\mathsf{By}(\delta_1+\delta_2)\ \wedge\ [q_1\ \vee\ q_2]} C}$$

This rule requires that the input preconditions $[p_1]$ and $[p_2]$ are conjoined in the conclusion, so that both p_1 and p_2 are required to hold throughout the combined transition. This is sufficient for the `Mult` example, but a stronger rule would have $[p_1]\,;[p_2]$ as the input precondition of the conclusion, where `;` is a chop operator of interval temporal logic [3].

1.8.3. Cases rules

There are two cases rules. One for case analysis of state preconditions:

$$\frac{A_1 \xrightarrow[P]{Q} B \qquad\qquad A_2 \xrightarrow[P]{Q} B}{A_1 \vee A_2 \xrightarrow[P]{Q} B}$$

and the other for case analysis of input preconditions:

$$\frac{A \xrightarrow[P_1]{Q} B \qquad\qquad A \xrightarrow[P_2]{Q} B}{A \xrightarrow[P_1 \vee P_2]{Q} B}$$

A combination of sequencing and cases can be used to establish the resetting behaviour:

$$\texttt{MultMachine} \models \mathsf{True} \xrightarrow[[\texttt{reset} \equiv 1]]{\mathsf{By}\ 43} \mathsf{At}(\texttt{MultProg})\ \texttt{0}$$

The argument proceeds by first splitting the universally true predicate True into a 20-way disjunction asserting that control is either outside the program or is inside one of the intermediate commands. For each case, the sequencing rule applied to atomic STAs can be used to show that under the assumption of $\texttt{reset} \equiv 1$ the postcondition $\mathsf{At}(\texttt{MultProg})\ \texttt{0}$ is eventually achieved. Showing these amounts to symbolic execution from an arbitrary starting position. The results of each of these cases are combined using a cases rule. Although there is a substantial amount of detail, a reset-analysis tool has been implemented that performs the proof automatically.

1.8.4. The wait loop rule

The next STA rule enables the existence of wait states to be deduced. Unfortunately the rule is rather complicated and contains a number of hypotheses that are hard to motivate in a general way (their necessity only becomes apparent when the detailed derivation of the rule is considered, which is not done here).

To see why waiting states can be rather subtle consider `MultSpec`. To meet this specification it is necessary to have a predicate, $\mathsf{Ready}\ z$ say, that satisfies both

$$\mathsf{Ready}\ z \Longrightarrow (\texttt{avail} \equiv 1 \ \wedge\ \texttt{out} \equiv z)$$

and

$$\texttt{req} \equiv 0\ \textbf{Holds}\ (\mathsf{Ready}\ z)$$

Consider now the execution of `MultProg` when it is waiting to start a handshake. It will be cycling between intermediate commands `0`, `1` and `18` waiting for `req` to become 1 (see Figure 1.1). Abbreviate $\mathsf{In}(\texttt{MultProg})$ to In. Perhaps $\mathsf{Ready}\ z$ could be defined by:

$$\mathsf{Ready}\ z \ \ =_{def?}\ \ (\mathsf{In}\ \texttt{0} \vee \mathsf{In}\ \texttt{1} \vee\ \mathsf{In}\ \texttt{18}) \wedge (\texttt{avail} \equiv 1 \ \wedge\ \texttt{out} \equiv z)$$

Unfortunately this definition will not ensure $\texttt{req} \equiv 0$ **Holds** ($\mathsf{Ready}\ z$). To see this suppose control is at machine instruction 3 (JMZ 44), avail is 1 and out is z. The predicate In 1 will be true (since machine instruction 3 is part of intermediate command 1) and hence Ready z is true. Suppose now that the environment makes input req be 0. If $\texttt{req} \equiv 0$ **Holds** ($\mathsf{Ready}\ z$) then the next state of the machine (i.e. the one after executing JMZ 44) must satisfy Ready z, however this will only be the case if the top of the stack contains 0, which will only be the case if req were true on the previous cycle (i.e. when INP req was executed). Thus req being 0 at instruction 3 does not ensure that Ready z is held. It is necessary to assume that the top of the stack is 0 rather than that the input req is 0. Thus the definition of Ready given above will not work. The solution used by the wait loop rule is to define Ready z to be true of a state σ if σ is reachable from a state satisfying At 1 via a trace in which all the intermediate states satisfy the invariant $\texttt{avail} \equiv 1 \land \texttt{out} \equiv z$ and all the intermediate inputs satisfy $\texttt{req} \equiv 0$. If this is the case then machine instruction 3 will have been reached from a preceding state in which 0 was read onto the top of the stack.

In the complicated looking rule that follows, the predicate A characterizes the top of a wait loop. If the environment maintains the holding condition p_1, then the waiting is maintained and A is true each time the loop starts a new iteration. The invariant q holds during the wait loop. The predicate B is true of the first state not in the loop; it is reached if the environment maintains the breakout condition p_2 for at least δ_1 machine cycles. The first and last hypotheses of the rule are necessary technical conditions. The first hypothesis says that the breakout state is not passed though during the wait loop. The last hypothesis says that if the loop is started then no matter what inputs arrive, within δ_2 cycles control will either return to the top of the loop or have left the loop.

$$\frac{(q \Longrightarrow \neg B) \land \left(A \xrightarrow[{[p_1]}]{[q]} A\right) \land \left(A \xrightarrow[{[p_2]}]{\mathsf{By}\ \delta_1} B\right) \land \left(A \xrightarrow[\mathsf{True}]{\mathsf{By}\ \delta_2} A \lor B\right)}{\exists\, W.\ \ (p_1\ \textbf{Holds}\ W) \land \left(W \xrightarrow[{[p_2]}]{\mathsf{By}(\delta_1+\delta_2)} B\right) \land (A \Longrightarrow W) \land (W \Longrightarrow q)}$$

The application of this rule to MultProg is now shown. The hypotheses of the application are the following four facts. The first one follows directly from definitions; the other three can be deduced from automatically generated atomic STAs.

$$(\mathsf{In}\ 0 \lor \mathsf{In}\ 1 \lor \mathsf{In}\ 18) \land (\texttt{avail} \equiv 1 \land \texttt{out} \equiv z) \Longrightarrow \neg(\mathsf{At}\ 2 \land \texttt{avail} \equiv 1 \land \texttt{out} \equiv z)$$

$$\begin{array}{l}\mathsf{At}\ 1\\ \texttt{avail} \equiv 1\\ \texttt{out} \equiv z\end{array} \xrightarrow[{[\texttt{req} \equiv 0]}]{\left[\begin{array}{l}\mathsf{In}\ 0 \lor \mathsf{In}\ 1 \lor \mathsf{In}\ 18\\ \texttt{avail} \equiv 1 \land \texttt{out} \equiv z\end{array}\right]} \begin{array}{l}\mathsf{At}\ 1\\ \texttt{avail} \equiv 1\\ \texttt{out} \equiv z\end{array}$$

$$\begin{array}{l}\mathsf{At}\ 1\\ \texttt{avail} \equiv 1\\ \texttt{out} \equiv z\end{array} \xrightarrow[{[\texttt{req} \equiv 1]}]{\mathsf{By}\ 2} \begin{array}{l}\mathsf{At}\ 2\\ \texttt{avail} \equiv 1\\ \texttt{out} \equiv z\end{array}$$

$$\begin{array}{l}\mathsf{At\ 1}\\ \mathtt{avail}\equiv 1\\ \mathtt{out}\equiv z\end{array} \xrightarrow[\mathsf{True}]{\mathsf{By\ 5}} \left(\begin{array}{l}\mathsf{At\ 1}\\ \mathtt{avail}\equiv 1\\ \mathtt{out}\equiv z\end{array} \vee \begin{array}{l}\mathsf{At\ 2}\\ \mathtt{avail}\equiv 1\\ \mathtt{out}\equiv z\end{array}\right)$$

From these hypotheses, the wait loop rule directly yields the existence of a predicate $W\ z$ such that:

$$(\mathtt{req}\equiv 0)\ \textbf{Holds}\ (W\ z)$$

and

$$W\ z \xrightarrow[[\mathtt{req}\equiv 1]]{\mathsf{By\ 7}} \begin{array}{l}\mathsf{At\ 2}\\ \mathtt{avail}\equiv 1\\ \mathtt{out}\equiv z\end{array}$$

and

$$\mathsf{At\ 1} \wedge \mathtt{avail}\equiv 1 \wedge \mathtt{out}\equiv z \Longrightarrow W\ z$$

and

$$W\ z \Longrightarrow (\mathsf{In\ 0} \vee \mathsf{In\ 1} \vee \mathsf{In\ 18}) \wedge (\ \mathtt{avail}\equiv 1 \wedge \mathtt{out}\equiv z)$$

The desired result is obtained by defining Ready to be W. The application of the wait loop rule has been fully automated and so most of the details just shown are generated by a procedure and need not concern the user. The existence of a predicate $\mathsf{Ack}\ x\ y\ z$ can be deduced similarly.

1.8.5. The while rule

The while rule for STAs is analogous to the while rule of a Hoare logic of total correctness. It is formulated here in terms of predicates on stack machine states, though it could be expressed abstractly in terms of arbitrary predicates as was done for the wait loop rule. First some notation. A vector of names $\langle \mathtt{x}_1, \ldots, \mathtt{x}_n\rangle$ will be abbreviated to $\vec{\mathtt{x}}$ and similarly a vector $\langle x_1, \ldots, x_n\rangle$ of values will be abbreviated to $\vec{x}$. The predicate $\vec{\mathtt{x}} \equiv \vec{x}$ abbreviates the conjunction of predicates $\mathtt{x}_1 \equiv x_1 \wedge \ \ldots \ \wedge \mathtt{x}_n \equiv x_n$. The predicate $\mathtt{v} \ll x$ is true of a state (pc, stk, mem) iff $mem\ \mathtt{v} < x$. If f is a function from vectors of values to vectors of values and A, B are predicates on vectors of values, then the Hoare logic like notation $\{A\}f\{B\}$ means $\forall\, \vec{x}.\ A\ \vec{x} \Rightarrow B(f\ \vec{x})$.

In the while rule for STAs that follows, *Inv* and B are arbitrary predicates on vectors of values called the *invariant* and *test*, respectively. The function f specified the state-change each time around the loop (i.e. the 'meaning' of the body of the 'while loop'). The first hypothesis of the rule says that *Inv* is an invariant and the value of $\mathtt{x}_i$ decreases each time around the loop (i.e. $\mathtt{x}_i$ is a variant). The second hypothesis is that the loop starts with the values $\vec{x}$ of $\vec{\mathtt{x}}$ in the memory satisfying *Inv*. It is assumed that the top of a while loop is at itermediate command $\mathtt{n}$. If the test B fails, the loop exits to the command numbered $\mathtt{n}_2$, without change of memory and taking δ_1 cycles. If B succeeds then control transfers to $\mathtt{n}_1$, without changing the memory and taking δ_1 cycles, and then back to $\mathtt{n}$ with the memory changed by f and taking another δ_2 cycles.

Thus while B remains true the program loops from $\mathtt{n}$ to $\mathtt{n}_1$ then back to $\mathtt{n}$ taking $\delta_1 + \delta_2$ cycles and transforming the values of the vector of variables $\vec{\mathtt{x}}$ by f each time. The conclusion of the rule is that the loop will terminate within $\delta_1 + (\delta_1+\delta_2) \times x_i$ cycles, where

x_i is the value of the variant $\mathtt{x}_i$ when the loop started. On termination the invariant still holds, but the test B is false. Here is the while rule for STAs.

$$\frac{\begin{array}{l}\{Inv \wedge B \wedge \mathtt{x}_i \equiv x_i\}\ f\ \{Inv \wedge \mathtt{x}_i \ll x_i\} \\ Inv\ \vec{x} \\ \forall \vec{x}.\ \left(\begin{array}{l}\mathsf{At}\ \mathtt{n} \\ \vec{\mathtt{x}} \equiv \vec{x}\end{array} \xrightarrow[{[p_1]}]{\mathsf{By}\ \delta_1} \begin{array}{l}\mathsf{At}(B\ \vec{x} \ \rightarrow\ \mathtt{n}_1 \mid \mathtt{n}_2) \\ \vec{\mathtt{x}} \equiv \vec{x}\end{array}\right) \\ \forall \vec{x}.\ \left(\begin{array}{l}\mathsf{At}\ \mathtt{n}_1 \\ \vec{\mathtt{x}} \equiv \vec{x}\end{array} \xrightarrow[{[p_2]}]{\mathsf{By}\ \delta_2} \begin{array}{l}\mathsf{At}\ \mathtt{n} \\ \vec{\mathtt{x}} \equiv\ f\ \vec{x}\end{array}\right)\end{array}}{\exists \vec{x}'.\ \left(\begin{array}{l}\mathsf{At}\ \mathtt{n} \\ \vec{\mathtt{x}} \equiv \vec{x}\end{array} \xrightarrow[{[p_1\ \wedge\ p_2]}]{\mathsf{By}(\delta_1 + (\delta_1 + \delta_2) \times x_i)} \begin{array}{l}\mathsf{At}\ \mathtt{n}_2 \\ \vec{\mathtt{x}} \equiv \vec{x}'\end{array}\right) \ \wedge\ Inv\ \vec{x}' \ \wedge\ \neg(B\ \vec{x}')}$$

As an example of the STA while rule consider the iteration:

```
WHILE (x > 0) AND NOT(INPUT reset)
 DO out := out + y;
    x   := x - 1
 OD
```

This translates to the following intermediate form (see Figure 1.1).

```
13: IF x > 0 AND NOT(INPUT reset) THEN SKIP ELSE GOTO 17 .. 27: GET x
                                                            28: OP0 0
                                                            29: OP2 >
                                                            30: INP reset
                                                            31: OP1 NOT
                                                            32: OP2 AND
                                                            33: JMZ 42
14: out := out + y ........................................ 34: GET out
                                                            35: GET y
                                                            36: OP2 +
                                                            37: PUT out
15: x := x - 1 ............................................ 38: GET x
                                                            39: OP1 pre
                                                            40: PUT x
16: GOTO 13 ............................................... 41: JMP27
```

From which the following two STAs can be generated (the free variables $reset$, x, y and out are assumed to be universally quantified).

$$\begin{array}{l}\mathsf{At}\ 13 \\ \mathtt{x} \equiv x \\ \mathtt{y} \equiv y \\ \mathtt{out} \equiv out\end{array} \xrightarrow[{[\mathtt{reset} \equiv reset]}]{\mathsf{By}\ 7} \begin{array}{l}\mathsf{At}(x > 0 \wedge \neg(reset = 1)\ \rightarrow\ 14 \mid 17) \\ \mathtt{x} \equiv x \\ \mathtt{y} \equiv y \\ \mathtt{out} \equiv out\end{array}$$

$$\begin{array}{l}\mathsf{At}\ 14 \\ \mathtt{x} \equiv x \\ \mathtt{y} \equiv y \\ \mathtt{out} \equiv out\end{array} \xrightarrow[\mathsf{True}]{\mathsf{By}\ 8} \begin{array}{l}\mathsf{At}\ 13 \\ \mathtt{x} \equiv x - 1 \\ \mathtt{y} \equiv y \\ \mathtt{out} \equiv out + y\end{array}$$

For arbitary r take:

$$
\begin{array}{lcl}
Inv_r\langle x, y, out\rangle & = & (out{+}(x{\times}y) = r) \\
B\langle x, y, out\rangle & = & x > 0 \wedge \neg(reset = 1) \\
f\langle x, y, out\rangle & = & \langle x{-}1, y, out{+}y\rangle
\end{array}
$$

It follows that $\{Inv_r \,\wedge\, B \,\wedge\, \mathtt{x} \equiv m\}\; f\; \{Inv_r \,\wedge\, \mathtt{x} < m\}$ and hence by the while rule, if $out{+}(x{\times}y) = r$ then there exists x', y' and out' such that:

$$
\left(\begin{array}{lcl}
\mathsf{At}\ 13 & & \mathsf{At}\ 17 \\
\mathtt{x} \equiv x & \xrightarrow[\left[\mathtt{reset}\,\equiv\, reset\right]]{\mathsf{By}(7\,+\,(7{+}8)\,\times\, x)} & \mathtt{x} \equiv x' \\
\mathtt{y} \equiv y & & \mathtt{y} \equiv y' \\
\mathtt{out} \equiv out & & \mathtt{out} \equiv out'
\end{array}\right) \;\wedge\; Inv_r\langle x', y', out'\rangle \;\wedge\; \neg B\langle x', y', out'\rangle
$$

Hence $out'{+}(x'{\times}y') = r$ (the invariant still holds) and $\neg(x' > 0 \wedge \neg(reset = 1))$ (the test is false). If $reset = 0$ then it follows from the test being false that $x' = 0$ and then if r is taken to be $x \times y$ it follows from the invariant still holding that $out' = x \times y$. Hence:

$$
\begin{array}{lcl}
\mathsf{At}\ 13 & & \mathsf{At}\ 17 \\
\mathtt{x} \equiv x & \xrightarrow[\left[\mathtt{reset}\,\equiv\, 0\right]]{\mathsf{By}(7{+}15{\times}x)} & \mathtt{x} \equiv 0 \\
\mathtt{y} \equiv y & & \mathtt{y} \equiv y' \\
\mathtt{out} \equiv 0 & & \mathtt{out} \equiv x{\times}y
\end{array}
$$

By choosing a slightly more complex invariant it could also be shown that the value of y is unchanged by the iteration.

1.9. Conclusions

The analysis of real-time programs is notoriously complex. The approach outlined here tightly couples the formalism used (STAs) with theorem proving tools, the aim being to automate away as much detail as possible. The current mechanization requires the user to invoke tools, such as the wait loop synthesizer and the while rule, on an intermediate representation of the high-level program consisting of sequences of assignments and jumps. In the future, it is hoped to try to hide this level completely by guiding the verification via annotations in the program. The goal, only partially achieved so far, is to require the user to manually prove 'mathematical' verification conditions, but to have all STA manipulations performed automatically. Progress towards this goal appears in the paper entitled 'A Hoare logic of state transitions' included in the Festschrift for Professor Hoare, edited by Bill Roscoe and published by Prentice-Hall in 1994. In that paper it is shown how the while rule for STAs can be automatically invoked via annotations in the high-level program. This is achieved by defining a Hoare logic of state transitions.

The approach taken here can be viewed as lying somewhere in the middle of a spectrum with conventional verification plus a verified compiler at one end, and pure machine code verification at the other. Conventional verification using a high-level semantics has many advantages: properties of programs can be proved that are independent of the compiler used. If a verified compiler is available, then analysis can be conducted within an abstract semantics and then applied, via a compiler correctness statement, to machine code [2].

At the other extreme, one can model the host machine semantics and then verify machine code programs (got, for example, by running production Ada compilers) by reasoning about processor transitions. Impressive work of this sort has been done by Yuan and Boyer [6]. Between these two extremes lies the work presented here. The techniques are in the spirit of Yuan and Boyer in that they are based on a semantics derived from the execution of machine instructions (though Yuan and Boyer use a real machine in the 68000 family, whereas an enormously simpler abstract machine is used here). However, the reasoning is conducted through an abstract view of the machine code provided by a high level programming language. Proofs are conducted using Hoare-style proof rules normally associated with high-level languages; but the interpretation of the Hoare-sentences is low-level.

Bibliography

REFERENCES

1 J. A. Camilleri. Symbolic compilation and execution of programs by proof: A case study in HOL. Technical Report 240, Computer Laboratory, University of Cambridge, UK, December 1991.

2 P. Curzon. Deriving correctness properties of compiled code. In L. Claesen and M. Gordon, editors, *Proceedings of the International Workshop on Higher Order Logic Theorem Proving and its Applications*. North-Holland, 1992.

3 J. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals. In *Proc. 10th International Colloquium on Automata, Languages and Programming, Barcelona, Spain*, 1983.

4 C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, October 1969.

5 B. Levy, I. Filippenko, L. Marcus, and T. Menas. Using the state delta verification system. In *Proc. IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design*, pages 337–360. North-Holland, June 1992.

6 Y. Yu. *Automated Proofs of Object Code for a Widely Used Microprocessor*. PhD thesis, The University of Texas at Austin, yuanyu@com.dec.src, 1992.