Chapter 1

A Mechanized Hoare Logic of State Transitions

Mike Gordon

The field of programming logics stands on the foundation laid by C.A.R. Hoare in his seminal paper 'An axiomatic basis for computer programming' [8]. In that paper Hoare presented a calculus of formulae $P\{C\}Q$ meaning "If assertion P is true before initiation of a program C, then the assertion Q will be true on its completion." [1] Since 1969, research into Hoare logics has been a major topic in the theory of programming. Numerous variations and extensions of Hoare's original ideas have been developed. These have both been studied theoretically [1] and put into practice [3, 9]. This tradition is continued here. An interpretation of Hoare logic is described that is intended for the analysis of programs implementing real-time reactive systems. Versions of Hoare's original rules have been derived and form the basis for a prototype computer assisted program verifier. The aim has been to produce an automated system that uses 'Hoare-style' reasoning to establish both total correctness and 'fine grain' timing properties of programs. There already exist extensions of Hoare logic that enable the running time of programs to be analysed [11]. The work here extends and automates these. Formulae of the form $\{P\}\ C\ \{Q\}\ [I]\ \langle t\rangle$ are introduced where P, Q and I are assertions, C is a command (i.e. a program) and t is a number. These meaning of such formulae is "If P is true and the instructions compiled from C are executed, then in at most t machine cycles the execution of C will terminate in a state satisfying Q and all intermediate states will satisfy I."

1.1 An introductory example

The simple example given in this section aims to convey the 'look and feel' of the verification method presented in subsequent sections. Some notations and concepts appear before they are properly introduced and as a result some of the details may be obscure.

[1]From now on $\{P\}\ C\ \{Q\}$ will written instead of Hoare's original $P\{C\}Q$.

Consider the specification of the form $\{P\}\ \mathcal{C}\ \{Q\}\ [I]$ shown in box 1 below. $\mathcal{C}$ is an annotated command and P, Q, I are conditions on the values of program variables. The condition I is intended to hold throughout the computation (i.e. in the final state and all intermediate states), whereas Q is only intended to hold in the final state. The annotations in the command are an assertion after the first assignment and a variant `[x]` and invariant for the **while**-loop. Each time around the loop the invariant holds and the value of the variant decreases (values are assumed to be positive integers). Variables like `x` in teletype font are program variables; variables like x in italics are logical variables (also called 'ghost' or 'auxiliary' variables).

```
{x = x ∧ y = y}                                                        1
 out := 0;  {out = 0  ∧  x = x  ∧  y = y}
 if ¬(x = 0 ∨ y = 0)
  then while  x > 0
         do [x] {out + (x × y) = x × y  ∧  y = y}
          out := out + y;
          x := x − 1
{out = x × y }
[y  =  y]
```

The verifier initially generates eleven verification conditions. Ten of these are solved by the currently implemented simplifier leaving the following one for the user:

```
(out + (x × y) = x × y) ⇒                                              2
 ¬(x = 0) ⇒
 ((out + y) + ((x − 1) × y) = x × y)
```

This is proved manually and then the verifier generates the theorem shown in box 3 below, which has the form $\vdash \{P\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle t\rangle$, where $\mathcal{C}$ is the unannotated multiplication program, $\mathsf{MultProg}$ say. Such augmented Hoare specifications mean that $\{P\}\ \mathcal{C}\ \{Q\}$ holds (interpreted as a total correctness specification), the execution of $\mathcal{C}$ requires at most t cycles and all intermediate states satisfy I.

```
⊢ {x = x ∧ y = y}                                                      3
    out := 0;
    if ¬(x = 0 ∨ y = 0)
     then while  x > 0
           do
            out := out + y;
            x := x − 1
   {out = x × y }
   [y  =  y]
   ⟨15 + (13×x)⟩
```

Thus the multiplication takes at most $15 + (13 \times x)$ machine cycles, where x is the initial value of `x` and the value of program variable `y` remains stable throughout the computation. This bound on the number of cycles is computed (and verified) automatically. The semantics of $\{P\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle t\rangle$ is formulated directly in terms of state transitions made by machine instructions compiled from $\mathcal{C}$. The machine MultMachine in box 4 is defined by the sequence of instruction obtained by compiling MultProg (see 1.4). The instruction numbers %n% are comments.

4

```
MultMachine = Machine [OP0 0;     %0 %
                       PUT out;   %1 %
                       GET x;     %2 %
                       OP1 ¬;     %3 %
                       GET y;     %4 %
                       OP1 ¬;     %5 %
                       OP2 ∨;     %6 %
                       OP1 ¬;     %7 %
                       JMZ 22;    %8 %
                       GET x;     %9 %
                       OP0 0;     %10%
                       OP2 >;     %11%
                       JMZ 22;    %12%
                       GET out;   %13%
                       GET y;     %14%
                       OP2 +;     %15%
                       PUT out;   %16%
                       GET x;     %17%
                       OP0 1;     %18%
                       OP2 −;     %19%
                       PUT x;     %20%
                       JMP 9]     %21%
```

Expanding the theorem of the form $\{P\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle t\rangle$ into its semantics yields the state transition assertion (or STA) [6] in box 5. State transition assertions are defined in 1.7. They are formulae of the form:

$$\mathcal{M}\ \models\ A \xrightarrow{\mathcal{I}} B$$

which means "If machine $\mathcal{M}$ is in a state satisfying A then a state satisfying B will be reached and the sequence of intermediate states will satisfy $\mathcal{I}$."

In the STA in box 5, vertical stacking means conjunction, At m is true when the program counter is m, $[I]$ is true of a sequence if I is true of all elements in the sequence and By m is true of any sequence of length less than m.

5

$$\mathsf{MultMachine}\ \models\ \begin{array}{l}\mathsf{At}\ 0\\ \mathtt{x}=x\\ \mathtt{y}=y\end{array} \xrightarrow{\begin{array}{l}\mathsf{By}(15+(13\times x))\\ [\mathtt{y}=y]\end{array}} \begin{array}{l}\mathsf{At}\ 22\\ \mathtt{out}=x\times y\end{array}$$

This says that if control is at instruction number 0 and the values in locations x and y are x and y, respectively, then within $15+(13\times x)$ cycles control will reach instruction 22 and the value of out will then be $x\times y$ and the value of y will have remained stable throughout the computation. A similar state transition assertion going from instruction n to instruction $n+22$ could be deduced if the compiled instructions from MultProg were loaded at position n in memory.

This result is a worst case analysis. If further information is known about the starting state, then a tighter time bound may be provable. For example, in the following annotated specification the precondition has the extra assumption $x = 0$ and the conditional has been annotated with `[F]` (which is necessary for the verification condition generator – see 1.6).

$\boxed{6}$

```
{x = x ∧ y = y ∧ x = 0}
 out := 0;  {out = 0  ∧  x = x  ∧  y = y ∧ x = 0}
 if ¬(x = 0 ∨ y = 0) [F]
  then while  x > 0
         do [x] {out + (x × y) = x × y  ∧  y = y}
          out := out + y;
          x := x − 1
{out = x × y }
[y = y]
```

The five verification conditions (see 1.6.5) from this are all solved automatically and the resulting STA follows:

$\boxed{7}$

$$\mathsf{MultMachine}\ \models\ \begin{array}{l}\mathsf{At}\ 0\\ \mathtt{x} = x\\ \mathtt{y} = y\\ x = 0\end{array} \xrightarrow{\begin{array}{l}\mathsf{By}\ 11\\ [\mathtt{y} = y]\end{array}} \begin{array}{l}\mathsf{At}\ 22\\ \mathtt{out} = x \times y\end{array}$$

This shows that if x is initially 0 then the computation takes at most 11 cycles. The STAs in boxes 5 and 7 can be combined into a single STA covering both cases:

$\boxed{8}$

$$\mathsf{MultMachine}\ \models\ \begin{array}{l}\mathsf{At}\ 0\\ \mathtt{x} = x\\ \mathtt{y} = y\end{array} \xrightarrow{\begin{array}{l}\mathsf{By}(x = 0\ \rightarrow\ 11 \mid 15 + (13 \times x))\\ [\mathtt{y} = y]\end{array}} \begin{array}{l}\mathsf{At}\ 22\\ \mathtt{out} = x \times y\end{array}$$

where $b\ \rightarrow\ p \mid q$ is the conditional *if b then p else q*.

1.2 Overview

The verifier illustrated in the previous section is built on top of a version of Hoare logic for judgements of the form $\{P\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle t\rangle$. This, in turn, is built on top of the theory of state transition assertions and a compiler for a simple programming language. Finally, these are defined directly in higher order logic. Here's a diagram:

Verification conditions	
Hoare logic of STAs	
Theory of STAs	Compiler
Higher order logic	

The approach is purely definitional in that each layer is defined in terms of the concepts of a lower one. The theory of STAs and their use in reasoning about machine instructions is described elsewhere [6]. To make this paper self-contained, a simplified version of the theory is outlined in 1.7. The general idea of mechanising Hoare logics by generating verification conditions and then feeding them to a theorem prover is standard [3, 5, 13]. The particular approach used here was originally developed for non-timed Hoare logics [4]. Verification conditions are described in 1.6. The main contribution of this paper is to make the use of STAs for reasoning about data-processing algorithms much easier by defining a Hoare logic on top of them.

1.3 Timed Hoare specifications

The syntax of expressions $\mathcal{E}$ and commands $\mathcal{C}$ is given by the following BNF, where $\mathcal{N}$ ranges over the natural numbers, $\mathcal{V}$ ranges over the set *Var* of program variables, $\mathcal{U}$ ranges over unary operators and $\mathcal{B}$ ranges over binary operators.

$$\mathcal{E} \;::=\; \mathcal{N} \mid \mathcal{V} \mid \mathcal{U}\,\mathcal{E} \mid \mathcal{E}_1\,\mathcal{B}\,\mathcal{E}_2$$

$$\begin{array}{lll} \mathcal{C} & ::= & \mathcal{V} := \mathcal{E} \\ & \mid & \mathcal{C}_1 \;;\; \mathcal{C}_2 \\ & \mid & \textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathcal{C} \\ & \mid & \textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2 \\ & \mid & \textbf{while}\ \mathcal{E}\ \textbf{do}\ \mathcal{C} \end{array}$$

This BNF syntax is ambiguous; if necessary, brackets will be used to disambiguate particular examples.

Expressions and commands are executed by translating them to sequences of instructions for a simple stack machine and then running the resulting machine code programs. The state of the target 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 $mem : Var \rightarrow \mathbb{N}$.

The meaning of timed Hoare specifications $\{P\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle t\rangle$ is defined in terms of the execution of instructions compiled from $\mathcal{C}$.

The specification $\{P\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle t\rangle$ is true iff [2] whenever the program counter points to the beginning of the instructions compiled from C and P is true then the target machine goes through a sequence of states satisfying I and reaches within t steps a state in which Q is true and the program counter points to the end of instructions compiled from C.

This informal definition is refined and formalized in 1.7 using state transition assertions.

[2] "iff" abbreviates "if and only if".

1.4 A simple machine and compiler

Programs are compiled to sequences of instructions from 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` $\mathcal{U}$	pop one value from stack, perform unary operation $\mathcal{U}$, push result
`OP2` $\mathcal{B}$	pop two values from stack, perform binary operation $\mathcal{B}$, 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

Let $[\![\mathcal{E}]\!]$ denote the instructions that expression $\mathcal{E}$ compiles to and let $s_1 {}^\frown s_2$ denote the concatenation of sequences s_1 and s_2, then:

$$\begin{array}{lcl}
[\![\mathcal{N}]\!] & = & \texttt{OP0}\ \mathcal{N} \\
[\![\mathcal{V}]\!] & = & \texttt{GET}\ \mathcal{V} \\
[\![\mathcal{U}\ \mathcal{E}]\!] & = & [\![\mathcal{E}]\!] {}^\frown \texttt{OP1}\ \mathcal{U} \\
[\![\mathcal{E}_1\ \mathcal{B}\ \mathcal{E}_2]\!] & = & [\![\mathcal{E}_1]\!] {}^\frown [\![\mathcal{E}_2]\!] {}^\frown \texttt{OP2}\ \mathcal{B}
\end{array}$$

It is easy to prove that if $|\mathcal{E}|$ is the number of instructions in $[\![\mathcal{E}]\!]$ then:

$$\begin{array}{lcl}
|\mathcal{N}| & = & 1 \\
|\mathcal{V}| & = & 1 \\
|\mathcal{U}\ \mathcal{E}| & = & |\mathcal{E}|\ +\ 1 \\
|\mathcal{E}_1\ \mathcal{B}\ \mathcal{E}_2| & = & |\mathcal{E}_1|\ +\ |\mathcal{E}_2|\ +\ 1
\end{array}$$

Let $[\![\mathcal{C}]\!]\ n$ be the sequence of instructions that command $\mathcal{C}$ compiles to if the first instruction is placed at position n. Let $|\mathcal{C}|$ be the number of instructions in $[\![\mathcal{C}]\!]$, then:

$$\begin{array}{lcl}
|\mathcal{V}\ \texttt{:=}\ \mathcal{E}| & = & |\mathcal{E}| + 1 \\
|\mathcal{C}_1\ \texttt{;}\ \mathcal{C}_2| & = & |\mathcal{C}_1| + |\mathcal{C}_2| \\
|\textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathcal{C}| & = & |\mathcal{E}| + |\mathcal{C}| + 1 \\
|\textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2| & = & |\mathcal{E}| + |\mathcal{C}_1| + |\mathcal{C}_2| + 2 \\
|\textbf{while}\ \mathcal{E}\ \textbf{do}\ \mathcal{C}| & = & |\mathcal{E}| + |\mathcal{C}| + 2
\end{array}$$

and

$$\begin{array}{lcl}
[\![\mathcal{V} \texttt{ := } \mathcal{E}]\!]\ n & = & [\![\mathcal{E}]\!] \frown \texttt{PUT}\ \mathcal{V} \\
[\![\mathcal{C}_1 \texttt{ ; } \mathcal{C}_2]\!]\ n & = & [\![\mathcal{C}_1]\!]\ n \frown [\![\mathcal{C}_2]\!](n{+}|\mathcal{C}_1|) \\
[\![\textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathcal{C}]\!]\ n & = & [\![\mathcal{E}]\!] \frown \texttt{JMZ}(n{+}|\mathcal{E}|{+}|\mathcal{C}|{+}1) \frown [\![\mathcal{C}]\!](n{+}|\mathcal{E}|{+}1) \\
[\![\textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2]\!]\ n & = & [\![\mathcal{E}]\!] \frown \texttt{JMZ}(n{+}|\mathcal{E}|{+}|\mathcal{C}_1|{+}2) \frown [\![\mathcal{C}_1]\!](n{+}|\mathcal{E}|{+}1) \\
 & & \quad \frown \texttt{JMP}(n{+}|\mathcal{E}|{+}|\mathcal{C}_1|{+}|\mathcal{C}_2|{+}2) \frown [\![\mathcal{C}_2]\!](n{+}|\mathcal{E}|{+}|\mathcal{C}_1|{+}2) \\
[\![\textbf{while}\ \mathcal{E}\ \textbf{do}\ \mathcal{C}]\!]\ n & = & [\![\mathcal{E}]\!] \frown \texttt{JMZ}(n{+}|\mathcal{E}|{+}|\mathcal{C}|{+}2) \frown [\![\mathcal{C}]\!](n{+}|\mathcal{E}|{+}1) \frown \texttt{JMP}\ n
\end{array}$$

1.5 A timed Hoare logic

The axioms and rules in this section can all be derived from the definition of $\{P\}\ C\ \{Q\}\ [I]\ \langle t\rangle$ given in 1.7 below.

1.5.1 The assignment axiom

The notation $P\,[\mathcal{E}/\mathcal{V}]$ denotes the result of substituting $\mathcal{E}$ for $\mathcal{V}$ in P. The assignment axiom states:

$$\vdash\ \{P\,[\mathcal{E}/\mathcal{V}]\}\ \mathcal{V}\texttt{:=}\mathcal{E}\ \{P\}\ [P \vee P\,[\mathcal{E}/\mathcal{V}]]\ \langle|\mathcal{E}|+1\rangle$$

The assignment takes one more cycle than the evaluation of $\mathcal{E}$ and during its execution the value of $\mathcal{V}$ is either its initial value or the value of $\mathcal{E}$.

1.5.2 The sequencing rule

$$\frac{\vdash\ \{P\}\ \mathcal{C}_1\ \{Q\}\ [I_1]\ \langle t_1\rangle \qquad \vdash\ \{Q\}\ \mathcal{C}_2\ \{R\}\ [I_2]\ \langle t_2\rangle}{\vdash\ \{P\}\ \mathcal{C}_1\texttt{;}\ \mathcal{C}_2\ \{R\}\ [I_1 \vee I_2]\ \langle t_1 + t_2\rangle}$$

The time taken to execute $\mathcal{C}_1\texttt{;}\mathcal{C}_2$ is the sum of the times taken by $\mathcal{C}_1$ and $\mathcal{C}_2$ and throughout the combined computation either I_1 or I_2 holds of the memory. The invariant could probably be strengthened to say that first I_1 and then I_2 holds; this could be expressed with a 'chop' operator from interval temporal logic [10]. So far this strengthening has not been needed and the rule above has been sufficient.

1.5.3 The conditional rules

The one-armed conditional rule is:

$$\frac{\vdash\ \{P \wedge \mathcal{E}\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle t\rangle \qquad \vdash\ P \wedge \neg\mathcal{E}\ \Rightarrow\ Q \qquad \vdash\ P\ \Rightarrow I}{\vdash\ \{P\}\ \textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathcal{C}\ \{Q\}\ [I]\ \langle|\mathcal{E}| + t + 1\rangle}$$

The time taken by **if** $\mathcal{E}$ **then** $\mathcal{C}$ is at most the time taken to evaluate $\mathcal{E}$ plus the time taken to execute $\mathcal{C}$ plus 1. If the invariant I is initially true and its truth is maintained by $\mathcal{C}$, then it is true throughout the entire computation of the conditional.

If it is known that the test $\mathcal{E}$ is false, then the time bound can be improved and the invariant strengthened to the precondition (since expression evaluations cannot change the memory).

$$\frac{\vdash\ P\ \Rightarrow\ \neg\mathcal{E}}{\vdash\ \{P\}\ \textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathcal{C}\ \{P\}\ [P]\ \langle|\mathcal{E}| + 1\rangle}$$

The two-armed conditional rule is:

$$\frac{\begin{array}{l}\vdash\ \{P \wedge \mathcal{E}\}\ \mathcal{C}_1\ \{Q\}\ [I_1]\ \langle t_1\rangle \\ \vdash\ \{P \wedge \neg\mathcal{E}\}\ \mathcal{C}_2\ \{Q\}\ [I_2]\ \langle t_2\rangle \\ \vdash\ P \wedge \mathcal{E}\ \Rightarrow\ I_1 \\ \vdash\ P \wedge \neg\mathcal{E}\ \Rightarrow\ I_2\end{array}}{\vdash\ \{P\}\ \textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2\ \{Q\}\ [I_1 \vee I_2]\ \langle|\mathcal{E}| + \mathsf{Max}(t_1, t_2) + 2\rangle}$$

The time taken to execute **if** $\mathcal{E}$ **then** $\mathcal{C}_1$ **else** $\mathcal{C}_2$ is at most the time taken to evaluate $\mathcal{E}$ plus the maximum of the times taken to execute $\mathcal{C}_1$ and $\mathcal{C}_2$ plus 2. If when $\mathcal{C}_1$ is executed then I_1 holds throughout the computation and when $\mathcal{C}_2$ is executed I_2 holds, then $I_1 \vee I_2$ holds no matter what arm of the conditional is taken.

If the precondition P determines the value of $\mathcal{E}$, then a tighter time bound can be derived. In the case that P forces $\mathcal{E}$ to be true:

$$\frac{\vdash\ \{P\}\ \mathcal{C}_1\ \{Q\}\ [I]\ \langle t\rangle \qquad \vdash\ P\ \Rightarrow\ \mathcal{E} \qquad \vdash\ P\ \Rightarrow\ I}{\vdash\ \{P\}\ \textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2\ \{Q\}\ [I]\ \langle|\mathcal{E}| + t + 2\rangle}$$

In the case that P forces $\mathcal{E}$ to be false:

$$\frac{\vdash\ \{P\}\ \mathcal{C}_2\ \{Q\}\ [I]\ \langle t\rangle \qquad \vdash\ P\ \Rightarrow\ \neg\mathcal{E} \qquad \vdash\ P\ \Rightarrow\ I}{\vdash\ \{P\}\ \textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2\ \{Q\}\ [I]\ \langle|\mathcal{E}| + t + 1\rangle}$$

The reason for “2” in the time bound when $\mathcal{E}$ is true, but “1” when it is false is that the compiler generates a jump instruction from after the code for $\mathcal{C}_1$ to the end of the conditional. This jump is not executed if the **else**-arm is taken.

1.5.4 The while rule

In the rule that follows $\mathcal{V}$ is a variant, i.e. a variable whose value strictly decreases each time around the loop; note that all values are natural numbers, so the value of $\mathcal{V}$ cannot be negative. The function $\mathsf{WhileTime}$ is defined by:

$$\mathsf{WhileTime}(e,t,n) \quad =_{def} \quad e \ + \ 1 \ + \ n \times (e+t+2)$$

$\mathsf{WhileTime}(e,t,n)$ is an upper bound on the number of cycles taken to execute n iterations of **while** $\mathcal{E}$ **do** $\mathcal{C}$, where e is an upper bound on the number of cycles to evaluate $\mathcal{E}$ and t is an upper bound on the number of cycles to execute $\mathcal{C}$.

$$\frac{\vdash\ \forall v.\ \{P \wedge \mathcal{E} \wedge \mathcal{V} = v\}\ \mathcal{C}\ \{P \wedge \mathcal{V} < v\}\ [I]\ \langle t\rangle \qquad\quad \vdash\ P\ \Rightarrow\ I}{\vdash\ \forall v.\ \{P \wedge \mathcal{V} = v\}\ \textbf{while}\ \mathcal{E}\ \textbf{do}\ \mathcal{C}\ \{P \wedge \neg\mathcal{E}\}\ [I]\ \langle\mathsf{WhileTime}(|\mathcal{E}|, t, v)\rangle}$$

If it is known that $\mathcal{E}$ is false, then the body of the while loop is never executed. This is reflected in the following rule.

$$\frac{\vdash\ P\ \Rightarrow\ \neg\mathcal{E}}{\vdash\ \{P\}\ \textbf{while}\ \mathcal{E}\ \textbf{do}\ \mathcal{C}\ \{P\}\ [P]\ \langle|\mathcal{E}|+1\rangle}$$

1.5.5 The consequence rule

Preconditions can be strengthened and postconditions, invariants and time-bounds weakened:

$$\frac{\begin{array}{l}\vdash\ \{P\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle t\rangle\\ \vdash\ P'\ \Rightarrow\ P\\ \vdash\ Q\ \Rightarrow\ Q'\\ \vdash\ I\ \Rightarrow\ I'\\ \vdash\ t \leq t'\end{array}}{\vdash\ \{P'\}\ \mathcal{C}\ \{Q'\}\ [I']\ \langle t'\rangle}$$

1.5.6 The cases rule

$$\frac{\begin{array}{l}\vdash\ \{P_1\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle t\rangle\\ \vdash\ \{P_2\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle t\rangle\end{array}}{\vdash\ \{P_1 \vee P_2\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle t\rangle}$$

1.5.7 An example proof

The following proof establishes the specification in box 6 on page 4.

1. By the assignment axiom:

   $\vdash \{\mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0 \wedge 0 = 0\}$
   $\quad\mathtt{out}\ \mathtt{:=}\ 0$
   $\{\mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0 \wedge \mathtt{out} = 0\}$
   $[(\mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0 \wedge \mathtt{out} = 0) \vee (\mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0 \wedge 0 = 0)]$
   $\langle 2\rangle$

2. By the consequence rule this entails:

   $\vdash \{\mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0\}$
   $\quad\mathtt{out}\ \mathtt{:=}\ 0$
   $\{\mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0 \wedge \mathtt{out} = 0\}$
   $[\mathtt{y} = y]$
   $\langle 2\rangle$

3. Since $(\mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0 \wedge \mathtt{out} = 0) \Rightarrow \neg(\neg(\mathtt{x} = 0 \vee \mathtt{y} = 0))$ and $|\neg(\mathtt{x} = 0 \vee \mathtt{y} = 0)| = 8$, it follows by the second one-armed conditional rule (the one for when the test is false) that:

   $\vdash \{\mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0 \wedge \mathtt{out} = 0\}$
   $\quad\mathtt{if}\ \neg(\mathtt{x} = 0 \vee \mathtt{y} = 0)\ \mathtt{then}\ \mathcal{C}$
   $\{\mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0 \wedge \mathtt{out} = 0\}$
   $[\mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0 \wedge \mathtt{out} = 0]$
   $\langle 9\rangle$

4. This simplifies by the consequence rule to:

   $\vdash \{\mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0 \wedge \mathtt{out} = 0\}$
   $\quad\mathtt{if}\ \neg(\mathtt{x} = 0 \vee \mathtt{y} = 0)\ \mathtt{then}\ \mathcal{C}$
   $\{\mathtt{out} = x \times y\}$
   $[\mathtt{y} = y]$
   $\langle 9\rangle$

5. Applying the sequencing rule to 2 and 4 yields:

   $\vdash \{\mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0 \wedge \mathtt{out} = 0\}$
   $\quad\mathtt{out}\ \mathtt{:=}\ 0;$
   $\quad\mathtt{if}\ \neg(\mathtt{x} = 0 \vee \mathtt{y} = 0)$
   $\quad\ \ \mathtt{then}\ \mathcal{C}$
   $\{\mathtt{out} = x \times y\}$
   $[\mathtt{y} = y]$
   $\langle 11\rangle$

1.6 Verification conditions

The proof in the previous section was a sequence of lines each of which was an axiom or followed from earlier lines by a rule of inference. Such forward proofs are tedious to produce. An alternative is to proceed backwards, by starting from the goal to be proved and then splitting this into subgoals, subsubgoals etc. until instances of axioms are reached. A traditional way of organizing such goal-directed proofs is to use verification conditions [3, 13]. The idea is to generate from a goal $\{P\}\ \mathcal{C}\ \{Q\}$ a set of purely logical formulae – the verifications conditions – that have the property that if they are true then the Hoare specification from which they were generated is also true. To enable verification conditions to be easily generated, the command $\mathcal{C}$ needs to be annotated with hints; in particular the variant and invariants for while loops need to be supplied (though attempts have been made to generate this information automatically [14]). The verification conditions are generated by a straightforward recursion on the structure of $\mathcal{C}$.

Verification conditions are related to Dijkstra's weakest preconditions [2], though they predate it. Dijkstra's idea was to replace $\{P\}\ \mathcal{C}\ \{Q\}$ by the purely logical formula $P \Rightarrow \mathsf{wp}(\mathcal{C}, Q)$, where $\mathsf{wp}(\mathcal{C}, Q)$ is the weakest precondition for $\mathcal{C}$ to establish Q. The rules for calculating $P \Rightarrow \mathsf{wp}(\mathcal{C}, Q)$ are similar to the rules for generating verification conditions from $\{P\}\ \mathcal{C}\ \{Q\}$.

The verifier described here requires the variant and invariant of all while loops to be supplied, as well as an assertion before each command in a sequence that is not an assignment. These assertions should be statements that are true when control reaches the point at which they occur. Additional optional assertions of the form `[T]` or `[F]` may also be added after the test in conditional and while commands; these indicate the truth value of the test.

A goal has the form $\{P\}\ \mathcal{C}\ \{Q\}\ [I]$, where $\mathcal{C}$ is an annotated command. It is assumed that each while command has a distinct variant $\mathtt{v}$ (and associated auxiliary variable v) and that for each such variant there is a conjunct $\mathtt{v} = v$ in P. The annotations in $\mathcal{C}$ enable an expression $\mathsf{Time}\ \mathcal{C}$ to be computed syntactically that gives a bound on the running time of $\mathcal{C}$ in terms of the initial values of the variants.

$$
\begin{array}{lcl}
\mathsf{Time}(\mathcal{V}\ \mathtt{:=}\ \mathcal{E}) & = & |\mathcal{E}| + 1 \\
\mathsf{Time}(\mathcal{C}_1\ ;\ \mathcal{C}_2) & = & \mathsf{Time}\ \mathcal{C}_1 + \mathsf{Time}\ \mathcal{C}_2 \\
\mathsf{Time}(\mathcal{C}_1\ ;\ \{R\}\ \mathcal{C}_2) & = & \mathsf{Time}\ \mathcal{C}_1 + \mathsf{Time}\ \mathcal{C}_2 \\
\mathsf{Time}(\mathbf{if}\ \mathcal{E}\ \mathbf{then}\ \mathcal{C}) & = & |\mathcal{E}| + \mathsf{Time}\ \mathcal{C} + 1 \\
\mathsf{Time}(\mathbf{if}\ \mathcal{E}\ \mathtt{[F]}\ \mathbf{then}\ \mathcal{C}) & = & |\mathcal{E}| + 1 \\
\mathsf{Time}(\mathbf{if}\ \mathcal{E}\ \mathbf{then}\ \mathcal{C}_1\ \mathbf{else}\ \mathcal{C}_2) & = & \mathsf{Time}\ \mathcal{E} + \mathsf{Max}(\mathsf{Time}\ \mathcal{C}_1, \mathsf{Time}\ \mathcal{C}_2) + 2 \\
\mathsf{Time}(\mathbf{if}\ \mathcal{E}\ \mathtt{[T]}\ \mathbf{then}\ \mathcal{C}_1\ \mathbf{else}\ \mathcal{C}_2) & = & \mathsf{Time}\ \mathcal{E} + \mathsf{Time}\ \mathcal{C}_1 + 2 \\
\mathsf{Time}(\mathbf{if}\ \mathcal{E}\ \mathtt{[F]}\ \mathbf{then}\ \mathcal{C}_1\ \mathbf{else}\ \mathcal{C}_2) & = & \mathsf{Time}\ \mathcal{E} + \mathsf{Time}\ \mathcal{C}_2 + 1 \\
\mathsf{Time}(\mathbf{while}\ \mathcal{E}\ \mathbf{do}\ \mathtt{[v]}\ \{R\}\ \mathcal{C}) & = & \mathsf{WhileTime}\ (|\mathcal{E}|, \mathsf{Time}\ \mathcal{C}, v) \\
\mathsf{Time}(\mathbf{while}\ \mathcal{E}\ \mathtt{[F]}\ \mathbf{do}\ \mathtt{[v]}\ \{R\}\ \mathcal{C}) & = & |\mathcal{E}| + 1
\end{array}
$$

For example, $\mathsf{Time}(\mathtt{out}\ \mathtt{:=}\ \mathtt{out} + \mathtt{y};\ \ \mathtt{x}\ \mathtt{:=}\ \mathtt{x} - 1)$ simplifies to 8 and $\mathsf{Time}\ \mathsf{MultProg}$ (where $\mathsf{MultProg}$ is the command in box 1 on page 2) simplifies to $15 + (13 \times x)$.

When the verifier is invoked with a goal $\{P\}\ \mathcal{C}\ \{Q\}\ [I]$, it tries to prove the specification $\{P\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle\mathsf{Time}\ \mathcal{C}\rangle$ using a set of derived rules 'backwards'. It matches the specification with the conclusions of these rules (which are given below) and then generates subgoals consisting of the hypotheses of the (unique) rule that matched. [3] This process is repeated on the subgoals until they are all reduced to purely logical formulae. These formulae are then mechanically simplified and those that do not reduce to true are returned as the verification conditions. It is clear that if the verification conditions are proved then the rules may be applied in the 'forward' direction to establish the original goal. A more detailed discussion of this process can be found elsewhere [4, 5].

The following rules generate the verification conditions; they can be derived from the axioms and rules given in 1.5.

1.6.1 Assignments

$$\frac{\vdash\ P\ \Rightarrow\ Q\,[\mathcal{E}/\mathcal{V}] \qquad\qquad Q \lor Q\,[\mathcal{E}/\mathcal{V}]\ \Rightarrow\ I}{\vdash\ \{P\}\ \mathcal{V}\texttt{:=}\mathcal{E}\ \{Q\}\ [I]\ \langle\mathsf{Time}(\mathcal{V}\ \texttt{:=}\ \mathcal{E})\rangle}$$

1.6.2 Sequencing

$$\frac{\vdash\ \{P\}\ \mathcal{C}_1\ \{R\}\ [I]\ \langle\mathsf{Time}\ \mathcal{C}_1\rangle \qquad \vdash\ \{R\}\ \mathcal{C}_2\ \{Q\}\ [I]\ \langle\mathsf{Time}\ \mathcal{C}_2\rangle}{\vdash\ \{P\}\ \mathcal{C}_1\texttt{;}\ \{R\}\ \mathcal{C}_2\ \{Q\}\ [I]\ \langle\mathsf{Time}(\mathcal{C}_1\ \texttt{;}\ \mathcal{C}_2)\rangle}$$

$$\frac{\vdash\ \{P\}\ \mathcal{C}\ \{Q\,[\mathcal{E}/\mathcal{V}]\}\ [I]\ \langle\mathsf{Time}\ \mathcal{C}\rangle}{\vdash\ \{P\}\ \mathcal{C}\texttt{;}\ \mathcal{V}\ \texttt{:=}\ \mathcal{E}\ \{Q\}\ [I]\ \langle\mathsf{Time}(\mathcal{C}\ \texttt{;}\ \mathcal{V}\ \texttt{:=}\ \mathcal{E})\rangle}$$

1.6.3 Conditionals

$$\frac{\vdash\ \{P \land \mathcal{E}\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle\mathsf{Time}\ \mathcal{C}\rangle \qquad \vdash\ P \land \neg\mathcal{E}\ \Rightarrow\ Q \qquad \vdash\ P\ \Rightarrow I}{\vdash\ \{P\}\ \mathbf{if}\ \mathcal{E}\ \mathbf{then}\ \mathcal{C}\ \{Q\}\ [I]\ \langle\mathsf{Time}(\mathbf{if}\ \mathcal{E}\ \mathbf{then}\ \mathcal{C})\rangle}$$

$$\frac{\vdash\ P\ \Rightarrow\ \neg\mathcal{E} \qquad \vdash\ P\ \Rightarrow\ Q \qquad \vdash\ P\ \Rightarrow I}{\vdash\ \{P\}\ \mathbf{if}\ \mathcal{E}\ \texttt{[F]}\ \mathbf{then}\ \mathcal{C}\ \{Q\}\ [I]\ \langle\mathsf{Time}(\mathbf{if}\ \mathcal{E}\ \texttt{[F]}\ \mathbf{then}\ \mathcal{C})\rangle}$$

[3]Matching is currently done by an *ad hoc* procedure which has the definition of Time built in. A more general approach would use Prolog-style metavariables to synthesize running times by unification. Theorem provers such as Isabelle [12] provide built-in facilities to support this.

$$\frac{\begin{array}{l}\vdash\ \{P \wedge \mathcal{E}\}\ \mathcal{C}_1\ \{Q\}\ [I]\ \langle \mathsf{Time}\ \mathcal{C}_1\rangle \\ \vdash\ \{P \wedge \neg\mathcal{E}\}\ \mathcal{C}_2\ \{Q\}\ [I]\ \langle \mathsf{Time}\ \mathcal{C}_2\rangle \\ \vdash\ P \Rightarrow I\end{array}}{\vdash\ \{P\}\ \textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2\ \{Q\}\ [I]\ \langle \mathsf{Time}(\textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2)\rangle}$$

$$\frac{\vdash\ \{P \wedge \mathcal{E}\}\ \mathcal{C}_1\ \{Q\}\ [I]\ \langle \mathsf{Time}\ \mathcal{C}_1\rangle \qquad \vdash\ P \Rightarrow \mathcal{E} \qquad \vdash\ P \Rightarrow I}{\vdash\ \{P\}\ \textbf{if}\ \mathcal{E}\ \texttt{[T]}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2\ \{Q\}\ [I]\ \langle \mathsf{Time}(\textbf{if}\ \mathcal{E}\ \texttt{[T]}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2)\rangle}$$

$$\frac{\vdash\ \{P \wedge \neg\mathcal{E}\}\ \mathcal{C}_2\ \{Q\}\ [I]\ \langle \mathsf{Time}\ \mathcal{C}_2\rangle \qquad \vdash\ P \Rightarrow \neg\mathcal{E} \qquad \vdash\ P \Rightarrow I}{\vdash\ \{P\}\ \textbf{if}\ \mathcal{E}\ \texttt{[F]}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2\ \{Q\}\ [I]\ \langle \mathsf{Time}(\textbf{if}\ \mathcal{E}\ \texttt{[F]}\ \textbf{then}\ \mathcal{C}_1\ \textbf{else}\ \mathcal{C}_2)\rangle}$$

1.6.4 While loops

$$\frac{\begin{array}{l}\vdash\ \forall v.\ \{R(v) \wedge \mathcal{E} \wedge \texttt{v} = v\}\ \textbf{while}\ \mathcal{E}\ \textbf{do}\ \texttt{[v]}\ \{R(v)\}\ \mathcal{C}\ \{R(v) \wedge \texttt{v} < v\}\ [I]\ \langle \mathsf{Time}\ \mathcal{C}\rangle \\ \vdash\ \forall v.\ P(v) \Rightarrow R(v) \wedge \texttt{v} = v \\ \vdash\ \forall v.\ R(v) \Rightarrow I\end{array}}{\vdash\ \forall v.\ \{P(v)\}\ \textbf{while}\ \mathcal{E}\ \textbf{do}\ \texttt{[v]}\ \{R(v)\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle \mathsf{Time}(\textbf{while}\ \mathcal{E}\ \textbf{do}\ \texttt{[v]}\ \{R(v)\}\ \mathcal{C})\rangle}$$

$$\frac{\vdash\ P \Rightarrow \neg\mathcal{E} \qquad \vdash\ P \Rightarrow Q \qquad \vdash\ P \Rightarrow I}{\vdash\ \{P\}\ \textbf{while}\ \mathcal{E}\ \texttt{[F]}\ \textbf{do}\ \texttt{[v]}\ \{R\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle \mathsf{Time}(\textbf{while}\ \mathcal{E}\ \texttt{[F]}\ \textbf{do}\ \texttt{[v]}\ \{R\}\ \mathcal{C})\rangle}$$

1.6.5 Example

Consider the goal in box 6 on page 4. Using the definition of Time the verifier computes that 11 cycles are needed. It thus tries to show:

```
{x = x ∧ y = y ∧ x = 0}
out := 0;  {out = 0  ∧  x = x  ∧  y = y ∧ x = 0}
 if ¬(x = 0 ∨ y = 0) [F]
  then while  x > 0
         do [x] {out + (x × y) = x × y  ∧  y = y}
          out := out + y;
          x := x − 1
{out = x × y }
[y = y]
⟨11⟩
```

Backchaining with the first rule in 1.6.2 yields two subgoals:

```
(1) {x = x ∧ y = y ∧ x = 0}
     out := 0
    {out = 0  ∧  x = x  ∧  y = y ∧ x = 0}
    [y = y]
    ⟨2⟩
```

```
(2) {out = 0  ∧  x = x  ∧  y = y ∧ x = 0}
     if ¬(x = 0 ∨ y = 0) [F]
      then while  x > 0
                do [x] {out + (x × y) = x × y  ∧  y = y}
                 out := out + y;
                 x := x − 1
    {out = x × y }
    [y = y]
    ⟨9⟩
```

The assignment rule in 1.6.1 reduces (1) to two purely logical verification conditions; both are trivial.

(3) $(\mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0) \Rightarrow$
$(0 = 0 \wedge \mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0)$

(4) $(0 = 0 \wedge \mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0) \vee$
$(\mathtt{out} = 0 \wedge \mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0) \Rightarrow$
$(\mathtt{y} = y)$

The second conditional rule 1.6.3 reduces (2) to three purely logical verification conditions; all three are trivial.

(5) $(\mathtt{out} = 0 \wedge \mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0) \Rightarrow$
$\neg(\neg(\mathtt{x} = 0 \vee \mathtt{y} = 0))$

(6) $(\mathtt{out} = 0 \wedge \mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0) \Rightarrow$
$(\mathtt{out} = x \times y)$

(7) $(\mathtt{out} = 0 \wedge \mathtt{x} = x \wedge \mathtt{y} = y \wedge x = 0) \Rightarrow$
$(\mathtt{y} = y)$

The verification conditions from the original goal are (3), (4), (5), (6) and (7). They can all be solved automatically.

1.7 State transition assertions

A state transition assertion (STA) has the form:

$$\mathcal{M} \models A \xrightarrow{\mathcal{I}} B$$

where $\mathcal{M}$ is a machine, A and B are predicates on states and $\mathcal{I}$ is a predicate on sequences of states. In general, a machine is a function from states and inputs to states [6], but as inputs are not considered here, a machine will be represented by a function from states to states. If $\mathcal{M} : \mathit{state} \to \mathit{state}$ is such a machine, then one cycle is a step $s \mapsto \mathcal{M}\ s$. A trace of $\mathcal{M}$ is a sequence:

$$s \frown (\mathcal{M}\ s) \frown (\mathcal{M}(\mathcal{M}\ s)) \frown (\mathcal{M}(\mathcal{M}(\mathcal{M}\ s)))\ \ldots$$

The STA above is true iff for all traces $s_0 \frown s_1 \frown \ldots$ of $\mathcal{M}$, if A is true of s_i then there is a later state s_j (where $i < j$) for which B is true and $\mathcal{I}$ is true of all subsequences $s_{i+1} \frown \ldots \frown s_k$, where $i < k \leq j$.

A machine code program $\mathcal{O}$ determines a machine $\mathsf{Machine}\ \mathcal{O}$ that maps a state $(\mathit{pc}, \mathit{stk}, \mathit{mem})$ to the state resulting from executing the instruction pointed to by the program counter pc. For example, the program in box 4 on page 3 determines a machine that maps the state $(8, \mathit{stk}, \mathit{mem})$ to the successor state $(22, \mathit{stk}, \mathit{mem})$, since the program counter (viz. 8) points to the instruction `JMP 22` (the program counter starts at 0, so 8 points to the 9th instruction).

The predicate $\mathsf{At}\ n$ is true of a state $(\mathit{pc}, \mathit{stk}, \mathit{mem})$ iff $\mathit{pc} = n$. The predicate $\mathsf{By}\ t$ is true of a sequence of states iff its length is at most t.

If the object code for a command $\mathcal{C}$ is loaded into program memory starting at location n, then the resulting machine code program, $\mathcal{O}$ say, will contain instructions $[\![\mathcal{C}]\!]n$ from position n to position $n + |\mathcal{C}|$. The notation $\mathsf{Loaded}(\mathcal{O}, \mathcal{C}, n)$ means that this is the case.

If $\mathcal{F}$ is a formula then $\{\mathcal{F}\}$ is the predicate on states defined by:

$$\{\mathcal{F}\}(\mathit{pc}, \mathit{stk}, \mathit{mem}) \ = \ \mathcal{F}[\mathit{mem}\ \mathtt{x}_1, \ldots, \mathit{mem}\ \mathtt{x}_n / \mathtt{x}_1, \ldots, \mathtt{x}_n]$$

where $\mathtt{x}_1$, …, $\mathtt{x}_n$ are the teletype font variables in $\mathcal{F}$ and $\mathcal{F}[v_1, \ldots, v_n/\mathtt{x}_1, \ldots, \mathtt{x}_n]$ denotes the result of textually substituting v_i for $\mathtt{x}_i$ (for $1 \leq i \leq n$) in $\mathcal{F}$. For example:

$$\begin{aligned}&\{\mathtt{out} + (\mathtt{x} \times \mathtt{y}) = x \times y \ \wedge\ \mathtt{y} = y\}(\mathit{pc}, \mathit{stk}, \mathit{mem}) \ = \\ &\quad (\mathit{mem}\ \mathtt{out}) + ((\mathit{mem}\ \mathtt{x}) \times (\mathit{mem}\ \mathtt{y})) = x \times y \ \wedge\ (\mathit{mem}\ \mathtt{y}) = y\end{aligned}$$

This notation formalizes the convention that teletype font variables are program variables (i.e. names bound by the memory) and italic variables are logical (or auxiliary) variables.

If $\mathcal{P}$ is a predicate on states then $[\mathcal{P}]$ is the predicate on sequences of states that is true iff $\mathcal{P}$ is true of each individual state in the sequence.

The conjunction of predicates on states and sequences of states will be indicated by vertical stacking.

Armed with all this notation, the timed Hoare formula $\{P\}\ \mathcal{C}\ \{Q\}\ [I]\ \langle t\rangle$ can now be defined to mean:

$$\forall\,\mathcal{O}\ n.\ \mathsf{Loaded}(\mathcal{O},\mathcal{C},n)\ \Rightarrow\ \mathsf{Machine}\ \mathcal{O}\ \models\ \begin{array}{c}\mathsf{At}\ n\\ \{P\}\end{array}\ \xrightarrow[\quad]{\begin{array}{c}\mathsf{By}\ t\\ [\{I\}]\end{array}}\ \begin{array}{c}\mathsf{At}(n+|\mathcal{C}|)\\ \{Q\}\end{array}$$

In practice, the curley brackets { } are omitted. For example, if $\mathcal{O}$ is the sequence of instructions in box 4 on page 3 and $\mathsf{Loaded}(\mathcal{O},\mathsf{MultProg},0)$, then the meaning of the timed Hoare formula in box 3 on page 2 entails the STA in box 5 on page 3.

1.8 Mechanization

The prototype verifier is implemented in HOL [7]. First STAs are defined and various derived rules for them are proved. Next the stack machine, programming language and compiler are defined. Timed Hoare formulae are then defined in terms of STAs and the various laws in 1.5 are proved. Finally, tactics are programmed to generate verification conditions from goals; the justification part of these tactics are the derived rules [4].

1.9 Conclusions

The work described here is just the beginning of an attempt to provide theorem proving support for specifying and verifying timed reactive systems. Such systems alternate between waiting for input events in the environment and processing the data associated with these events. This paper concentrates on the data processing aspects (another paper [6] discusses waiting states and reaction times). Although some theorem-proving technology has been developed for reasoning about STAs, it is still not clear how useful the STA formalism is as a specification method for 'real' real-time systems.

1.10 Acknowledgements

This work was carried out as part of the safemos project; a SERC/DTI funded collaboration between inmos, SRI International, the Oxford University Programming Research Group and the Cambridge University Computer Laboratory. I am grateful to my collaborators on this project: Jonathan Bowen, Juanito Camilleri, Rachel Cardell-Oliver, Roger Hale, John Herbert and David Shepherd. In addition, I have had useful feedback on STAs from Victor Carreno, Nancy Day and Neil Viljoen.

References

[1] Apt, K.R., 'Ten years of Hoare's logic: a survey – part 1', ACM *Trans. on Programming Languages and Systems*, **3**, pp. 431-483, 1981.

[2] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, 1976.

[3] Good, D.I., 'Mechanical proofs about computer programs', in Hoare, C.A.R. and Shepherdson, J.C. (Eds), *Mathematical Logic and Programming Languages*, Prentice Hall, 1985.

[4] Gordon, M.J.C., 'Mechanizing Programming Logics in Higher Order Logic', G. Birtwistle and P.A. Subrahmanyam, (Eds), *Current Trends in Hardware Verification and Automated Theorem Proving*, Springer-Verlag, 1989.

[5] Gordon, M.J.C., *Programming Language Theory and its Implementation*, Prentice-Hall International Series in Computer Science, 1988.

[6] Gordon, M.J.C., 'State transition assertions: a case study', in Jonathan Bowen (Ed.), *Towards System Verification*, Real-Time Safety-Critical Systems series, Elsevier, 1993.

[7] Gordon, M.J.C. and Melham, T.F., *Introduction to HOL: a theorem-proving environment for higher-order logic*, Cambridge University Press, 1993.

[8] Hoare, C.A.R., 'An axiomatic basis for computer programming', *Communications of the ACM*, **12**, pp. 576-583, October 1969.

[9] Jones, C.B., *Systematic Software Development Using VDM*, Prentice Hall, 1986.

[10] J. Halpern, Z. Manna and B. Moszkowski., 'A Hardware Semantics based on Temporal Intervals', In the proceedings of the *10-th International Colloquium on Automata, Languages and Programming*, Barcelona, Spain, 1983.

[11] Nielson, H.R., 'A Hoare-like proof system for run-time analysis of programs', *Science of Computer Programming*, **9**, 1987.

[12] Paulson, L.C., 'Isabelle: The Next 700 Theorem Provers', in Odifreddi, P. (Ed) *Logic and Computer Science*, pp. 361–386, Academic Press, 1990.

[13] von Henke, F.W. and Luckham, 'Automatic Program Verification III: A Methodology for Verifying Programs', Stanford University Computer Science Department, Report No. STAN-CS-74-474, 1974.

[14] Wegbreit, B., 'The synthesis of loop predicates', *Comm. ACM*, **17**, pp 102–112, 1974.