Busy Time and Idle Time

C.A.R. Hoare

December 8, 1992

Summary: The introduction of timing considerations into programming theory is a potent source of complexity. This may be partially controlled by first treating time separately from all other features of a program. Two incommensurate notions of time, idle time and busy time, are first introduced separately and then combined. The mathematics is utterly trivial and not very useful; but the principles of separation and combination may be helpful in controlling the complexity of more realistic models.

Introduction

There are two reasons for introducing the concept of time into a programming language

1. To specify the consumption by a program of machine cycles or other unit of resource. This will be called *busy time*.

2. To specify an interval of real or simulated time which must elapse before the next command of the program is executed. This is called *idle time*, because it can be implemented without significant consumption of machine resources.

In a simulation program, the passage of real time is simulated by idle time, implemented efficiently by an event queue; and busy time is as irrelevant to the logic of the program as in conventional programming. In real time programming, idle time specifies a real time delay, which is needed to prevent the computer from interacting faster than its environment can tolerate — increasingly important as processing speeds increase. This kind of idle time can be efficiently implemented by a watchdog timer and an interrupt.

In real time programming there may be unfortunate interactions between busy time and idle time. These are controllable only if there is sufficient computing power. The mathematical theory introduced in this paper provides crude assistance in meeting this constraint.

1 Busy time

We represent the consumption of t machine cycles (or other unit of resource) by an explicit command in a programming notation:

$$\text{busy } (t).$$

This command may be inserted (by the programmer or by the compiler) just in front of any sequence of basic commands; it indicates that t is a safe upper bound (either desired or actual) on the real time consumed in executing the sequence.

Busy time costs; so any program which uses less of it is better than one which uses more:

$$\text{busy } (t) \sqsubseteq \text{ busy } (u), \qquad \text{if } u \leq t$$

where $\sqsubseteq$ is the merit ordering of programs. This has the valuable consequence that any program can be executed faster or more economically than specified, without invalidating its correctness. The worst program in this ordering is one that takes forever. We therefore define

$$\bot = \text{ busy } (\infty)$$

and state the law

$$\bot \sqsubseteq X, \qquad \text{for all } X.$$

When two or more sections of a program have to be executed to completion, the resources consumed are the sum of the resources consumed by the individual sections. For example, sequential composition can be defined:

$$\text{busy } (t);\ \text{busy } (u) = \text{ busy } (t+u).$$

The unit of addition is zero, so we define

$$\mathtt{SKIP} = \text{busy}(0)$$

and state the general law

$$\mathtt{SKIP}; X = \mathtt{SKIP} = X; \mathtt{SKIP}.$$

Similarly, associativity of sequential composition follows from the same property of addition.

Some combinators of a programming language (e.g. a conditional) require execution of a choice of only one of their operands. Let $\sqcap$ stand for a combinator (non-determinism) indicating that the programmer does not know or care which choice is made. The best that is known is that the resource requirement does not exceed that of the greater of the two alternatives. We therefore define

$$\text{busy } (t) \sqcap \text{ busy } (u) = \text{ busy } (\max\ (t, u)).$$

This operator inherits all the algebraic properties of max: it is idempotent, commutative and associative; it has SKIP as its unit and $\perp$ as its zero.

If there is a risk that one section of program may use too much resource, this can be limited by a timeout operator $\rhd_v$, where v is a strictly positive limit on the resources allocated to the left operand. It is defined

$$\begin{aligned} \text{busy } (t) \rhd_v \text{busy } (u) &= \text{ busy } (t) && \text{if } t < v \\ &= \text{ busy } (v + u) && \text{if } t \geq v. \end{aligned}$$

$(X \ \rhd_v Y)$ can be executed by first running X. If this terminates before consuming v units of resource, the whole construction terminates. Otherwise an interrupt occurs, and Y is executed to completion. This operator has SKIP as left unit, and obeys an associative law:

$$X \ \rhd_v (Y \ \rhd_w Z) = (X \ \rhd_v Y) \ \rhd_{v+w} Z.$$

In a multiprocessor system, some components of a program may be executed on machines of differing power, with the effect that they apparently use only some (strictly positive) factor α of the resources consumed by other components. To model this, we define

$$\alpha \times \text{ busy } (t) = \text{ busy } (\alpha \times t), \qquad \text{for } \alpha > 0.$$

Multiplication by a positive scalar is linear and monotonic, so it distributes through sequential and non-deterministic composition; furthermore

$$\begin{aligned} \alpha \times (X \rhd_v Y) &= (\alpha \times X) \rhd_{\alpha \times v} (\alpha \times Y) \\ \alpha \times \texttt{SKIP} &= \texttt{SKIP} \\ \alpha \times \bot &= \bot. \end{aligned}$$

Multiplication may be used to model execution of parallel processes on separate processors; for example we could define

$$\text{busy } (t) \parallel \text{busy } (u) = \text{ busy } \left(\frac{t+u}{2}\right).$$

Of course, if the two components are timeshared on a single processor, the resources consumed are still the sum of those consumed by the components; as far as busy time is concerned, there is no difference between timeshared parallelism and sequential composition.

This concludes the range of combinators which can be reasonably defined in such a simple algebra of busy time. In principle, any monotonic operator on nonnegative numbers could be defined for busy time, but most of them are either useless (for example, exponential) or impossible to implement (for example, square root).

2 Idle time

We specify the passage of p units of elapsed time by the command

$$\text{idle } (p).$$

The only effect of this command is to separate its start and its termination by exactly p units of idle time. The properties of idle time are expressed at a sufficient level of abstraction that it can be implemented in radically different ways, for example in a simulation language by an event queue or in a real time language by a watchdog timer.

Following standard algebraic practice, we redefine for idle time exactly the same operator symbols as we have for busy time. In the case of sequential composition, the right hand side of the definition is also the same, and for the same reason:

$$\begin{aligned} \text{idle } (p);\ \text{idle } (q) &= \text{ idle } (p+q) \\ \texttt{SKIP} &= \text{ idle } (0). \end{aligned}$$

But parallel composition is quite different. We require it to terminate exactly when the second of its two operands has terminated:

$$\text{idle } (p) \parallel \text{idle } (q) = \text{ idle } (\max(p,q)).$$

$\parallel$ is also associative and has unit SKIP.

Change in speed of processing should not have any effect on idle time. In fact, this is exactly the reason why idle time has to be so clearly distinguished from busy time. So scalar multiplication is defined as the identity function:

$$\alpha \times \text{ idle } (p) = \text{ idle } (p).$$

The identity function commutes with and distributes through every operator; and has everything as fixed point.

An interrupt operator $\rhd$ for idle time is taken from Sifakis' ATP; the first operand is selected for execution only if this involves no delay:

$$\begin{aligned} \text{idle } (p) \rhd \text{ idle } (q) &= \text{idle } (0) && \text{if } p = 0 \\ &= \text{ idle } (q) && \text{if } p > 0. \end{aligned}$$

This operator is clearly associative, and

$$\texttt{SKIP} \rhd Y = \texttt{SKIP}.$$

3 Combination

The previous sections have defined separate algebras for busy time and idle time, in each case using the same collection of notations for the combinators and the constants. It is therefore trivial to combine the two algebras into a Cartesian product algebra:

1. each element of the carrier set is a pair (s, p) where s is a single number representing an upper bound on busy time, and p is a single number representing a lower bound on idle time;
2. each combinator of the algebra is defined by applying the same combinator pointwise to the two components; and
3. the algebraic laws are just the intersection of the laws valid for the component algebras.

A summary of definitions is given in Table 1 and the laws in Table 2.

$\perp$	$=$	$(\infty, 0)$	
SKIP	$=$	$(0, 0)$	
$(s,p);(t,q)$	$=$	$(s+t,\ \ p+q)$	
$(s,p)\|\|(t,q)$	$=$	$(s+t,\ \ \max(p,q))$	
$(s,p) \sqcap (t,q)$	$=$	$(\max(s,t),\ \ \min(p,q))$	
$\alpha \times (s,p)$	$=$	$(\alpha \times s, p)$	
$(s,p)\ \rhd_v (t,q)$	$=$	(s,p)	if $s < v$ and $p = 0$
	$=$	$(v+t, q)$	otherwise.

Table 1. Definitions

$;$	is associative with unit SKIP
$\|$	is associative and commutative with unit SKIP
$\sqcap$	is associative, commutative, and idempotent with unit SKIP and zero $\perp$
$\alpha\times$	distributes through $;\ \|\ \sqcap$, and has fixed points SKIP and $\perp$
SKIP $\rhd_v X =$ SKIP	
$X \rhd_v (Y \rhd_w Z) = (X \rhd_v Y) \rhd_{v+w} Z$	
all the operators distribute through $\sqcap$	

Table 2. Laws

4 Application

The motive for the development of a theory of timing for programs is to provide a means of calculating how much processing power is needed to meet critical time constraints. Let (s, p) be the timing parameters for a typical reasonably-sized section of the program. Then p is the worst case real time deadline that must be met. So the necessary ratio of machine cycles to elapsed time is s/p. The maximum of this value over all sections of the program determines the speed of the processor that must be purchased to meet the real time constraint.

The calculus can also be used for fine tuning of the implementation of a parallel command

$$(s,p)\|(t,q),$$

where (say) q is larger than p. If $s+t$ is larger than q, this command must be executed on parallel processors, which is represented in this calculus by multiplication by fractional α. Otherwise, if $s+t$ is larger than p, the first process must be executed at higher priority, to avoid the risk that cycles expended by the other process may cause the deadline p to expire. Otherwise, if proportionate scheduling is implemented, the ratios s/p and t/q should be used.

These checks provide only a necessary condition of success. It is still possible for a parallel program which meets all global constraints and all sequential local constraints to fail as a result of temporary resource congestion, for example at the beginning of two parallel processes:

$$((1,1);(0,5)) \parallel ((1,1);(0,4)) = (1,6) \parallel (1,5) = (2,6).$$

These risks are intensified if the parallel processes engage in communication or mutual synchronisation, and so have to wait for each other.

The simplest way of averting the risk is to impose large safety factors; but in addition it will usually be necessary to conduct more sophisticated checks, at least on sections of the program identified as critical. This will need a much more sophisticated model than that presented here, for example using sets of sequences of pairs of times, rather than single pairs.

In the development of such a model, the lessons of this paper may help to avoid some of the complexities that have traditionally accompanied the introduction of time into programming theory. The following advice may be helpful.

1. First investigate each feature of the theory separately, and then put them together.

2. Do not be afraid of more than two concepts of time, for example to keep count of machine cycles on separate processors of a multiprocessor system.

3. Use non-determinism to model operators that cannot be exactly represented in each separate theory (for example, conditionals in this paper).

4. Construct a range of simple and more complex models, with clear embeddings, so that the engineer can choose the simplest model adequate to its purpose.

5. Ensure that all models enjoy roughly the same set of algebraic properties.

6. Give maximum assistance both to the engineer and to the theorist in the use of algebraic calculation rather than operational reasoning.

Appendix: Formal Semantics

The simple definitions given above ascribe a mathematical meaning or semantics to the concepts and constructions of a programming language. The semantics is called *specification-oriented*, because it relates each program text to a description or specification of relevant aspects of its observable behaviour, for example the machine cycles consumed in its execution. The algebraic laws derived from the definitions serve to confirm and enlarge our understanding of the concepts; they can also help to transform a program text from the form in which it is most clearly written to one in which it is most efficiently executed. Program transformation is so useful, that a complete list of laws is often taken as a formal definition of the semantics of the language, independent of the original definitions from which they were proved.

Another very popular method of defining a programming language is by an operational semantics. The primitive concept is a transition

$$X \xrightarrow{s} Y$$

where X is a program in its initial state
s is an observation of the behaviour of the program started in state X
and Y is a possible final state of the program after observation of s.

For example, in dealing with busy time, s may be a count of the number of machine cycles consumed by the execution of the program in its passage from X to Y. Note that in a non-deterministic system, there may be more than Y corresponding to any given X and s.

An easy way to derive an operational semantics for timing is to start with an independent mathematical definition to the transition relation. For example, the transition for busy time is defined

$$\text{busy }(s) \xrightarrow{v} \text{busy }(t) \ = \ t + v \leq s.$$

The inequation represents the fact that a valid implementation of busy (t) may actually consume less resources than t, a benefit that is expressed in the trivial theorem

$$\text{busy }(s) \xrightarrow{0} \text{busy }(t), \qquad \text{if } t \leq s.$$

Similarly, for idle time we define an inequation in the opposite direction

$$\text{busy }(p) \xrightarrow[r]{} \text{idle }(q) \ = \ p \leq q + r.$$

The annotation below the arrow distinguishes idle time from busy.

A list of valid transitions for busy time is given in Table 3 and for idle time in Table 4. Their proofs are no more complicated than those of the algebraic laws of Tables 1 and 2. The purpose of these laws is to give guidance on simple and efficient methods of implementing the corresponding concepts of the language. In particular, they serve as a check on the feasibility of the implementation. For example, it is impossible to give a transition rule in the standard format for the unimplemenatable square root operator on busy time. The importance of avoiding unimplemenatable theories and languages has led many theorists to accept a collection of transitions as a formal definition of the meaning of the concepts of a programming language, independent of the algebraic laws from which they can be proved.

$\mathtt{SKIP}; X \xrightarrow{0} X$
If $X \xrightarrow{v} Y$ then $(X; Z) \xrightarrow{v} (Y; Z)$

$X \sqcap Y \xrightarrow{0} X$
$X \sqcap Y \xrightarrow{0} Y$
$\perp \xrightarrow{v} Z$

If $X \xrightarrow{v} Y$ then $\alpha \times X \xrightarrow{\alpha \times v} \alpha \times Y$

If $X \xrightarrow{v} Y$ then $(X \rhd_{v+w} Z) \xrightarrow{v} (Y \rhd_{w} Z)$
If $X \xrightarrow{v+w} Y$ then $(X \rhd_{v} Z) \xrightarrow{v} Z$

If $X \xrightarrow{v} Y$ and $Y \xrightarrow{w} Z$ then $X \xrightarrow{v+w} Z$

Table 3: Transitions for busy time.

$\mathtt{SKIP}; Z \xrightarrow[0]{} Z$
If $X \xrightarrow[v]{} Y$ then $(X; Z) \xrightarrow[v]{} (Y; Z)$

$X \sqcap Y \xrightarrow[0]{} X$
$X \sqcap Y \xrightarrow[0]{} Y$
$\perp \xrightarrow[v]{} Z$

If $X \xrightarrow[v]{} Y$ and $X' \xrightarrow[v]{} Y'$ then $(X \| X') \xrightarrow[v]{} (Y \| Y')$
$\mathtt{SKIP} \| Y \xrightarrow[0]{} Y$
$X \| \mathtt{SKIP} \xrightarrow[0]{} X$

If $X \xrightarrow[0]{} Y$ then $(X \rhd Z) \xrightarrow[0]{} (Y \rhd Z)$
If $X \xrightarrow[v]{} Y$ then $(X \rhd Z) \xrightarrow[0]{} Z$
$(\mathtt{SKIP} \rhd Z) \xrightarrow[0]{} \mathtt{SKIP}$

If $X \xrightarrow[v]{} Y$ and $Y \xrightarrow[w]{} Z$ then $X \xrightarrow[v+w]{} Z$

Table 4: Transitions for idle time.