Validating microcode algebraically

J. M. FOSTER
Royal Signals and Radar Establishment, St Andrew's Road, Malvern, Worcs, WR14 3PS

This paper describes systematic algebraic methods used by a program to help in validating microcode. The methods are formal, mathematical and generally applicable. Examples are given of the kind of property of microcode which can be found, including checking for timing constraints, ensuring that interrupts are polled frequently, checking against expression stack overflow and ensuring the absence of certain sequences of instruction. The method separates into a large part which deals only with the control structure of the microcode, and a small part which deals with the operations performed by the micro-instructions. It has been used to check many properties of the implementation of Flex on Perq, which involves more than 5000 microinstructions.

Received March 1985

1. INTRODUCTION

Many authors have described the use of algebraic methods to discover properties of programs, for example Aho,[1] Backhouse and Carre,[2] Bramson and Goodenough,[3] Cousot and Cousot,[5] Kam and Ullman,[7] Mycroft,[9] Mycroft and Nielson,[10] Rosen,[11] Schaeffer,[13] Tarjan[14] and Wegbreit.[15] These methods have most often been used to check the criteria which permit particular optimisations in compilers. It is also possible to use them to prove the absence of wide classes of error in programs, or if errors are present it may be possible to locate them.

Microcode is a particularly suitable object for such studies. The correctness of the whole computer depends on the correctness of the microcode, so the expense of guaranteeing the absence of certain errors can easily be borne. Microcode is often written in a very dense and compact fashion, with many **goto**s and much sharing of code because of the overriding need for speed and space economy. There are also often unusual constraints on the timing, and there may be sequences of instructions which are impermissible. It is therefore particularly prone to errors which are difficult to find by visual inspection but can be found by algebraic methods. Furthermore, microcode usually has a fairly simple control structure. Conditional jumps, some kinds of switch jumps including one for instruction entry, subroutine calls, returns and exceptions account for most microcode control structure. Procedures with parameters are usually absent and there are a fixed number of registers. Main store is regarded as a peripheral device. All this makes the algebraic method easy to apply, though subroutines and exception handling do need an extension of the usual technique.

The algebraic methods take account of the control structure of the program, without usually being able to consider the particular values that are being manipulated. It is not possible, for example, to know which way a conditional jump will go, we merely know that it will go one way or the other. Hence the methods are pessimistic. If they say that an optimisation is possible, then that is certainly so, but the optimisation might be possible without being detected, since the actual values which can occur might rule out some combinations of paths. Likewise, if the program is said to be free from a certain sort of error, then that is so, but the presence of an error might be suggested when in fact it could be ruled out by a more detailed examination of the semantics.

The method factors into two parts. In one, which is independent of the particular property being investigated, we calculate an algebraic expression corresponding to the control structure of the program. In the other we calculate a function of that algebraic expression which gives us the property we require. So the control structure of the program is entirely processed in the first component and is divorced from the particular property under investigation, which belongs entirely to the second component.

This paper describes the method, shows how it may be used to find various kinds of error and gives an account of a program which implements the method for microcode based on the AMD 2910 microcontroller. It should be adapted for other controllers. The program deals with subroutines and exceptions and in a limited way with computed jumps. It operates in time $O(ek)$, where e is the number of edges in the graph of the program and k is the maximum loop in-degree, which is normally small. The edges, e, are counted only once for each subroutine. The details of the algorithm and a justification of the time bound are given in Foster.[6]

We start by giving an outline of the method in the rest of this section. The second section illustrates the method using the AMD 2910 microcontroller. In the third section are examples of the method applied to finding a number of useful properties of microcode which are apparently quite different. These include ensuring that an expression stack cannot overflow, finding the maximum time between polling for interrupts, checking that timing constraints on the use of store are met and making sure that the interrupt routine does not disturb values in machine registers. The fourth section contains some conclusions. An appendix gives formal definitions.

We use what is known as a regular algebra. We can think of this as being defined abstractly, by giving the operations of the algebra and saying what laws they must obey. Such a definition is given in the appendix. We can also think of various concrete realisations of the algebra. For one such realisation the values manipulated by the algebra are sets of routes starting at one point of a program and ending at another. The operations are ways of combining these to make larger sets of routes and ultimately the whole program.

A piece of microcode turns a state of the machine into another state. By a state we mean the contents of all the registers and whatever extra information is needed to

specify the machine completely. A single microinstruction is a route of one step, which transforms states into states. It consists of two parts. One, the control part, determines what instruction will be processed next, and the other specifies what changes are to be made to registers, what peripheral operations start and so forth. A particular execution of a microcode program consists of a sequence of instructions which thus perform a sequence of state changes. The simplest program is to do nothing, obey no instruction, and make no state change. This is a path of zero steps.

Usually there will be a number of possible ways from the start to the end of a program, because programs ordinarily contain conditional jumps of some sort. All the potential sequences of microinstructions which might be performed by a program, considered together, constitute the generalised notion of a path corresponding to that program. There may well be infinitely many such sequences. If we consider the collection of state changes corresponding to each of the individual instructions in a program, then each evaluation is a sequence of zero or more of these elementary state changes and a generalized path is a set (possibly infinite) of such sequences. So the 'carrier' of the path algebra, that is the values which are manipulated by the operations of the algebra, consists of sets of sequences of elementary state changes.

A regular algebra has three operators, denoted here by '.', '+' and '*', and two constants, written '1' and '0'. We shall describe them as they are used in the algebra of paths, which is a realisation of regular algebras.

The operation '.', read 'followed by', takes two sets of sequences and makes a new one consisting of the sequences composed from a member of the first parameter of the operation followed by a member of the second. The operation '+' takes two sets of sequences and makes a new one which is just the union of the two sets, that is routes from either one parameter or the other. The constant '1' corresponds to one sequence of zero length, in other words a short circuit between the starting and finishing points. The third operator, '*', takes a set of sequences, r, and produces

$$1+r+r.r+r.r.r+\ldots$$

that is, the paths in the result are those obtained by going zero times round r, or once or twice, etc. We may sometimes also use the constant '0', which corresponds to an empty set of sequences, but we do not need this for any of our calculations, just for some of the formal manipulations.

There are various laws which are obeyed by this algebra of sets of sequences. For example

$$a.(b+c) = a.b+a.c$$

which is a distributive law and

$$a.(b.c) = (a.b).c$$

which is the associative law for 'followed by'. These are examples of the laws necessary for regular algebras.

It is possible to take microcode and produce from its control structure a regular expression which describes the possible sets of routes through it. This interpretation of the control structure of the microcode can be done independently of its later use for checking, which depends on the actual state changes made by the individual steps. We therefore need just one program to do it.

As well as the realisation of the operations and laws of regular algebras in terms of paths, we could realise them in terms of other values and operations. Such a realisation would again define values to be manipulated, operations $+'$, $.'$ and $*'$ and constants $1'$ and $0'$, all of which obey the laws. A homomorphism of regular algebras, H, is a mapping which obeys the rules

$$H(1) = 1'$$
$$H(0) = 0'$$
$$H(a.b) = H(a).'H(b)$$
$$H(a+b) = H(a)+'H(b)$$
$$H(a*) = H(a)*'$$

In order to obtain a property of the program we will define a homomorphism of regular algebras from the path algebra to an algebra devised especially for determining the particular property. The implication of the rules for homomorphisms above is that we can calculate the property of a composite program from the properties of its parts.

A property of a program will be some sort of value. It may be as simple as a truth value, saying that the program is good or bad, or an integer giving the maximum value of something, or it may be quite a complex structure. The path algebra is such that if we choose how the one-step values are mapped by the homomorphism, that is if we specify the property for individual instructions, then there is only one way of extending this to a homomorphism. If we introduce a function, atom, which takes the one-step values to their properties, then the rule $H(a) = \text{atom}(a)$, when a is a single instruction together with the rules given above, defines a recursive program for evaluating the property.

Let us take a very simple example to illustrate this. Suppose that some of the elementary instructions can be wrong. We wish to find out whether any route through a program contains a wrong instruction. We take the type of our property values to be truth values, true being interpreted as meaning that a set of paths has no wrong instruction in it and false as meaning that it does contain a wrong instruction. A short circuit has no error so $1'$ is true. The function atom is that which says for each elementary instruction whether it is good or bad. For $.'$ we choose 'and' so paths will only be good of both components are good. For $+'$ we also choose 'and', since we want to detect an error if there is one in either component. The operation $*'$ can be deduced to be the identity operation on truth values, and the constant $0'$ is true. We must check that these operations obey the laws for regular algebras which are set out in the appendix. Then we can obtain the truth value which tells us whether there was a wrong instruction on any of the routes through the program by evaluating the homomorphism for the regular expression produced from the microcode.

How are we to interpret this value? Let us consider the operator $+$, both in the path algebra and in the truth-value algebra. From the laws in the appendix we see that it is associative, commutative, idempotent and has 0 as its unit, that is

$$0+x = x \quad \text{and} \quad x+0 = x$$

It is always the case that for such an operator we can define a partial ordering, written $\leqslant$, by defining $x \leqslant y$ to

mean $x+y=y$, in which 0 is less than or equal to each element. In terms of the path algebra, any subset of a set of sequences is less than or equal to the set, for the + operator is just set union and the $\leqslant$ operator is set inclusion. In terms of the truth-value algebra, + is 'and', $\leqslant$ is 'is implied by' and true < false. For any homomorphism, H, $x \leqslant y$ implies $H(x) \leqslant H(y)$. Hence if $H(p)$ is the property of some set of sequences then $H(q)$ for any subset, q, of p satisfies $H(q) \leqslant H(p)$. Consider the truth-value algebra. If the value of the homomorphism applied to some set of sequences is true, then we know that the homomorphism applied to each of the sequences that makes up the set is true, since true is the least value possible. So if we get the value true we know that every route through the program is free from bad instructions.

If each member of a set of values is less than or equal to a particular value b, then we say that b is an upper bound to the set, and if b is the least such value, then it is the least upper bound of the set. Homomorphisms from the path algebra preserve least upper bounds. The set of sequences for a path is by definition the least upper bound of the component sequences. So its homomorphic image is the least upper bound in the algebra of truth values. If the value is false this implies that false is the last value that will do and so, for this algebra, some sequence must attain the value false and thus contain a wrong instruction. It is not always the case that the least upper bound is attained on some path, as will be shown in a later example.

In terms of our simple example, this analysis seems perhaps to be labouring a point which is sufficiently obvious, but is necessary in general to be accurate about the meaning of the value of a property, especially when we are using the property of leastness to make a statement about the value attained on some path.

2. EVALUATION OF THE HOMOMORPHISMS

2.1 Structure of the program

A program has been written in Algol68 to implement the method for microcode written for the ICL Perq computer. The Perq microcode is based on the AMD 2910 microcontroller and this provides the control structure of the microcode. Since the production of the regular expression depends only on the control structure, this part of the program depends on the properties of the AMD 2910. The individual homomorphisms will depend on the other parts of the microinstructions. So the major part of the program is dependent on the AMD 2910 rather than Perq. Methods used by other authors are described in Refs 2 and 14.

The program first assembles the text of the microcode into an in-store representation. This, of course, depends on the form of written microcode for Perq. It then applies a given homomorphism

hom(assemble(text), unit, dot, plus, star, atom)

the result of which is the value of the homomorphism for the whole program. The value, unit, and the functions, dot, plus, stat and atom define the particular homomorphism which is to be used and the procedure, hom, knows the control structure of the machine and evaluates the regular expression and its homomorphism. So assemble, unit, dot, plus, star and atom understand about the operations of the Perq microcode, but hom only knows about the control structure provided by the AMD 2910. It would have been possible to evaluate the regular expression first, and then the homomorphism afterwards, but that would have meant keeping a representation of the regular expression which for a large microcode would have been very bulky. The program therefore evaluates the homomorphism as it goes, as this usually takes very much less space.

Since different homomorphisms will map to regular algebras in terms of different base types (modes in Algol68), the types of unit, dot, plus, star, aroma and hom will be different when calculating different properties. Algol68 does not permit modes to be parameterised, so it is necessary to recompile in order to generate a program which applies a new homomorphism. The structure is schematically,

```
begin
  mode M = definition of mode of values produced by
hom
  proc atom = (instruction i)M: body;
  proc dot = (Ma, b)M: body;
  proc plus = (Ma, b)M: body;
  proc star = (Ma)M: body;
  M unit = value;
  proc hom = (microcode m,
    Mu,
    proc (M, M)Md,
    proc (M, M)Mp,
    proc (instruction)M atom)
    M:
  body;
  hom(assemble(text), unit, dot, plus, star, atom)
end
```

The first six lines of definition are special to a particular homomorphism, the remaining lines are always the same.

Of course it is possible for the control structure of the microcode to be illegal itself. For example, the AMD 2910 chip has only five levels of subroutine entry available, so exceeding this number must be wrong, excluding any consideration of regular expressions. Also, if we start to interpret a new macroinstruction when the subroutine stack is not empty this is likely to be a mistake. These and other errors are detected by hom while it is evaluating the regular expression, and are indicated as errors to the user.

2.2 From control structure to regular expressions

From the point of view of rapid calculation, the most important properties of regular algebras are the associative and distributive laws. These imply that we can calculate the regular expression corresponding to a part of the program and slot it into place in a larger expression, without any problem about the order in which these things are done. So we can calculate the regular expression starting from a label in the code up to, say, the points at which we start to decode the next instruction, and store it, or rather its homomorphic image, together with the label. Then when we find another jump to that label we can use the already computed value. Something similar can be done for subroutine calls by storing the value associated with the subroutine up to the returns and exception jumps, and re-using this. This is made complex by the need to treat

exceptions (Jump-pop in the AMD 2910) and because of the explicit manipulations of the subroutine link stack which are possible. Subroutine calls, exceptions and the efficiency of the calculation of regular expressions are dealt with in another paper.[6]

Clearly a section of program which continues to another using 'next' or an unconditional jump is to be composed using the 'followed by' operator.

A conditional jump brings the + operator into play. Notice that the operation performed by the instruction before jumping has to be dealt with, so a conditional jump yields

$$(\text{effect of instruction}).(\text{dest1}+\text{dest2})$$

were dest1 and dest2 are the regular expressions calculated from the continuation and the (labelled) destination. Switch jumps are dealt with similarly.

Microcode typically has a structure

$$\text{initiate}.(\text{ins0}+\text{ins1}+\text{ins2}\ldots)*$$

where initiate is microcode to set up the system, and ins0, ins1, etc. are microcode to interpret the various macroinstructions. This, therefore, is how we treat the switch jump to decode the next instruction.

The AMD 2910 has a register S. This is used both for counting, which can be treated by the same method as conditional jump, and also to hold a destination for a jump. This opens the possibility of computed jumps, but if the values which are loaded into S are constants of the microcode it is possible to calculate the values which can be in S at any moment and treat a jump using S as if it were a switch jump to all these places.

We have still to deal with loops caused by jumps. Suppose we take a note when we start to process code starting at a label. If, while processing it, we arrive at a jump to the same label we have in effect the situation

$$x = a+b.x$$

were a is the rest of the regular expression, and b is the set of paths leading up to the jump to the label. For this we produce the regular expression

$$b*.a$$

It can be shown that this substitution can always be made (see Appendix), and that although doing things in a different order may lead to different regular expressions, these are always equivalent under the laws of regular algebras.

The rest of the features of the AMD 2910 can be dealt with in ways which are easily deduced from those which have been treated above.

3. EXAMPLES OF HOMOMORPHISMS

3.1 Some regular algebras

We start by considering a few examples of regular algebras in order to help with the homomorphisms to follow.

Consider the non-negative integers with infinity

$$\begin{aligned} a.b &\to a+b \quad \text{(addition on integers)} \\ a+b &\to \min(a,b) \\ a* &\to 0 \\ 1 &\to 0 \\ 0 &\to \infty \end{aligned} \tag{3.1}$$

Here $\leqslant$ is 'greater than or equal' for integers.

Take the non-negative integers with plus and minus infinity

$$\begin{aligned} a.b &\to a+b \quad \text{(addition on integers)} \\ a+b &\to \max(a,b) \\ a* &\to \text{if } a>0 \text{ then} \infty \text{ else } 0 \\ 1 &\to 0 \\ 0 &\to -\infty \end{aligned} \tag{3.2}$$

This will satisfy the laws for regular algebras with a zero (see Appendix) provided that we take $-\infty+\infty=-\infty$. The relation $\leqslant$ is the normal 'less than or equal' for integers.

Another simple example is to take truth values, T and F with

$$\begin{aligned} a.b &\to a \text{ and } b \\ a+b &\to a \text{ or } b \\ a* &\to T \\ 1 &\to T \\ 0 &\to F \end{aligned} \tag{3.3}$$

or its obvious dual. Here $\leqslant$ is 'implies' and in the dual case it is 'is implied by'.

Several of the examples of homomorphism below use regular algebras derived in the following way. Given a regular algebra with a zero defined on a set R, we can obtain a regular algebra defined on $R \times R$ by

$$\begin{aligned} (a1,b1).(a2,b2) &\to (a2+a1.b2, b1.b2) \\ (a1,b1)+(a2,b2) &\to (a1+a2, b1+b2) \\ (a,b)* &\to (a.b*, b*) \\ 1 &\to (0,1) \end{aligned} \tag{3.4}$$

We can see that $(0,0)$ is a right zero for 'followed by', but it is not a left zero. However $(0,0)$ is a unit for the plus operation, which is all that is needed to define the bottom element for the $\leqslant$ relation. Indeed $\leqslant$ for the pair is just $\leqslant$ in both the components. If a proper zero is needed one may be added consistently to the base set, but it will not be needed in any example below.

3.2 Examples of homomorphisms

Consider the following simple example, useful in what follows. We wish to find whether a path expression contains a path of zero length, that is, whether there is a possible short-circuit between start and finish. Note that the path expressions that we produce from microcode are slightly different from the sequence of instruction steps through the program. For the single instruction

$$\text{label: op, } \textbf{goto} \text{ label}$$

translates into the path expression

$$\text{op}.\text{op}*$$

since the operation is always obeyed at least once. A zero-length path in a path expression is quite possible. Let us take R to be truth values, with

$$\begin{aligned} \text{atom}(a) &= \text{false} \\ 1 &= \text{true} \\ a.b &= a \text{ and } b \\ a+b &= a \text{ or } b \\ a* &= \text{true} \\ 0 &= \text{false} \end{aligned} \tag{3.5}$$

It can easily be seen that this computes the correct answer and satisfies the laws for regular algebras with a zero. It just uses the algebra of Formula 3.3. In view of the remarks about least upper bounds in the first section, we can see that if the result is false, then there is no zero-length path, and if the result is true there must be a zero-length path somewhere in the expression.

Now let us design a homomorphism which will decide whether an X instruction is immediately followed by a Y instruction. Though it appears that this might be easy to detect by eye, in fact to do so in 4000 instructions with all the possibilities of interaction which are permitted by the AMD 2910, and to be certain that no example has been missed, is no light task. It is something which is useful for Perq microcode, since there are instructions which can occasionally, because of interrupts, spoil the effect of the following one. We will let R consist of quadruples of truth values (x, y, t, e), in which x will tell us if any of the paths ends with an X instruction, y will say whether any path starts with a Y instruction, t will say whether there is a path of zero length, and e will tell us whether there is an example of XY in any of the paths, which is the error we are looking for. Let

$$\begin{aligned}
\text{atom}(X) &= (T, F, F, F) \qquad (3.6)\\
\text{atom}(Y) &= (F, T, F, F)\\
\text{atom}(a) &= (F, F, F, F) \text{ otherwise}\\
1 &= (F, F, T, F)
\end{aligned}$$

$(x1, y1, t1, e1).(x2, y2, t2, e2) =$

$(x2$ or $(t2$ and $x1)$,

$y1$ or $(t1$ and $y2)$,

$t1$ and $t2$,

$e1$ or $e2$ or $(y1$ and $x2))$

$(x1, y1, t1, e1)+(x2, y2, t2, e2) =$

$(x1$ or $x2$,

$y1$ or $y2$,

$t1$ or $t2$,

$e1$ or $e2)$

$(x, y, t, e)* = (x, y, T, e$ or $(x$ and $y))$

The t component is just the same as in Formula 3.5. The pair x and t as well as the pair y and t are examples of Formula 3.4. The laws for e can also be easily checked. Inspection will verify that these definitions are a correct interpretation of what was required. Note that it might be the case than an instruction X contains a conditional jump which, because of extra facts which we know, cannot jump to the Y instruction. In this case we shall be told that there is an error when there is none. But certainly if this homomorphism says there is no error, we can be sure that this is so. In practice we would like, not only to know that the program is free from this error if it is so, but also if there is an error, where the fault occurred. Such error location can be done in this case without much difficulty by adding extra components to R, but it has not been done here, in order to keep the example uncluttered. However, in general it is not so easy because the error is not usually easily attributed to a particular place in the program.

Another example, closely related to Formula 3.5, will give us the shortest path through a program in terms of the number of instructions. This could easily be modified to give the minimum time. Take R to be integers and let

$$\begin{aligned}
\text{atom}(a) &= 1 \qquad (3.7)\\
1 &= 0\\
a.b &= a+b \text{ (integer addition)}\\
a+b &= \min(a, b)\\
a* &= 0\\
0 &= 0
\end{aligned}$$

This is just the regular algebra of Formula 3.1. Again the resulting value is attained on some path.

From this homomorphism we can derive another which will check for the following kind of error. It happens on the Perq that if a certain sort of store instruction is followed within four instructions by another memory instruction then the memory instruction will go wrong. Ensuring that this does not happen is clearly both important and difficult to check. Let us devise a homomorphism to ensure that an instruction of class X is not followed within n steps by an instruction of class Y. We take R to consist of three integers and a truth value and interpret (x, y, t, e) by letting x be the smallest number of steps from an X to the end, Y the smallest number of steps from the start to a Y, t the shortest number of steps through the path expression not involving X or Y, and e the truth value telling whether an error has occurred. Let

$$\begin{aligned}
\text{atom}(X) &= (0, \infty, \infty, F) \qquad (3.8)\\
\text{atom}(Y) &= (\infty, 0, \infty, F)\\
\text{atom}(a) &= (\infty, \infty, 1, F) \text{otherwise}
\end{aligned}$$

$(x1, y1, t1, e1).(x2, y2, t2, e2) =$

$(\min(x2m\, x2+t2)$,

$\min(y1, t1+y2)$,

$t1+t2$,

$e1$ or $e2$ or $x1+y2 < n+1)$

$(x1, y1, t1, e1)+(x2, y2, t2, e2) =$

$(\min(x1, x2)$,

$\min(y1, y2)$,

$\min(t1, t2)$,

$e1$ or $e2)$

$(x, y, t, e)* = (x, y, 0, e$ or $x+y < 5)$

This is clearly closely related to Formula 3.6 and again is an example of the use of Formula 3.4.

We may find the maximum number of steps (or time) through a program by using the algebra of Formula 3.2 and taking

$$\text{atom}(a) = 1$$

The least upper bound argument now tells us, if the result is infinite, not that an infinite value is attained, but that there is no finite upper bound to the length of paths.

We can use this in a similar way to find the maximum distance between instances of a particular kind of instruction, X. This is useful, for example if we want to

ensure that polling for interrupts occurs sufficiently frequently to keep up with some peripheral. We take a quadruple of integers (a, b, t, m), a being the maximum distance after an X to the end, b the maximum distance from the start to an X, t the maximum distance through the path expression not involving an X, and m the maximum separation between Xs.

$$\begin{aligned}\text{atom}(X) &= (0, 0, -\infty, -\infty) \\ \text{atom}(a) &= (-\infty, -\infty, 1, -\infty) \\ 1 &= (-\infty, -\infty, 0, -\infty)\end{aligned} \tag{3.9}$$

$$\begin{aligned}&(a1, b1, t1, m1).(a2, b2, t2, m2) = \\ &\quad (\max(a2, a1+t2), \\ &\quad \max(b1, b2+t1), \\ &\quad t1+t2, \\ &\quad \max(m1, m2, a1+b2))\end{aligned}$$

$$\begin{aligned}&(a1, b1, t1, m1)+(a2, b2,)t2, m2) = \\ &\quad (\max(a1, a2), \\ &\quad \max(b1, b2), \\ &\quad \max(t1, t2), \\ &\quad \max(m1, m2))\end{aligned}$$

$$\begin{aligned}&(a, b, t, m)* = (\textbf{if } t > 0 \textbf{ then } a+\infty \textbf{ else } a, \\ &\quad \textbf{if } t > 0 \textbf{ then } b+\infty \textbf{ else } b, \\ &\quad \textbf{if } t > 0 \textbf{ then } \infty \textbf{ else } 0, \\ &\quad \textbf{if } t > 0 \textbf{ then } \max(m, a+b+\infty) \\ &\qquad \textbf{else } \max(m, a+b))\end{aligned}$$

We examine the t and m fields of the result to see whether we have good microcode. Since any loop not containing an X will give a value of infinity this will mean that we must look at every loop which does not poll for interrupts in order to check by hand that using it will not exceed the permitted interval. However, the homomorphism can be modified to find such loops, and it is in any case probably wise to look at them.

We will produce a homomorphism to check that any sequence of instructions starting with an X instruction and ending with a Z instruction does not contain any Y instructions. This might be because Y spoils something set up by X for Z to use. Typically Y is the interrupt poll. We use a septet of truth values

$$(x, y, z, xy, yz, t, e)$$

Let x mean that there is a path from an X to the end of the path expression not involving a Y or a Z, let y mean that there is a path from start to finish with one or more Y instructions on it but neither an X nor a Z, and let z mean that there is a path from the start to a Z instruction without X or Y. The truth value xy shall mean that there is a path with an X on it followed by some Ys and dually for yz. Let t mean that there is a path without X, Y or Z and e shall signify that an error has been detected.

$$\begin{aligned}\text{atom}(X) &= (T, F, F, F, F, F, F) \\ \text{atom}(Y) &= (F, T, F, F, F, F, F) \\ \text{atom}(Z) &= (F, F, T, F, F, F, F) \\ \text{atom}(a) &= (F, F, F, F, F, T, F) \text{ otherwise} \\ 1 &= (F, F, F, F, F, T, F)\end{aligned} \tag{3.10}$$

$$\begin{aligned}&(x1, y1, z1, xy1, yz1, t1, e1).(x2, y2, z2, xy2, yz2, t2, e2) \\ &\quad = (x2 \text{ or } (x1 \text{ and } t2), \\ &\qquad (y1, \text{ and } t2) \text{ or } (t1 \text{and } y2) \text{ or } (y1 \text{ and } y2), \\ &\qquad z1 \text{ or } (z2 \text{ and } t1), \\ &\qquad xy2 \text{ or } (xy1 \text{ and } t2) \text{ or } (xy1 \text{ and } y2) \\ &\text{or } (x1 \text{ and } y2), \\ &\qquad yz1 \text{ or } (t1 \text{ and } yz2) \text{ or } (y1 \text{ and } yz2) \\ &\text{or } (y1 \text{ and } z2), \\ &\qquad t1 \text{ and } t2, \\ &\qquad e1 \text{ or } e2 \text{ or } (x1 \text{ and } yz2) \text{ or } (xy1 \text{ and } z2))\end{aligned}$$

$$\begin{aligned}&(x1, y1, z1, xy1, yz1, t1, e1)+(x2, y2, z2, xy2, yz2, t2, e2) \\ &\quad = (x1 \text{ or } x2, \\ &\qquad y1 \text{ or } y2, \\ &\qquad z1 \text{ or } z2, \\ &\qquad xy1 \text{ or } xy2, \\ &\qquad yz1 \text{ or } yz2, \\ &\qquad t1 \text{ or } t2, \\ &\qquad e1 \text{ or } e2)\end{aligned}$$

$$\begin{aligned}(x, y, z, xy, yz, t, e)* = {} & x \\ & y, \\ & z, \\ & xy \text{ or } (x \text{ and } y), \\ & yz \text{ or } (y \text{ and } z), \\ & T, \\ & e \text{ or } (x \text{ and } yz) \text{ or } (xy \text{ and } z) \text{ or} \\ & (x \text{ and } y \text{ and } z))\end{aligned}$$

Finally, we show how we may ensure that the limits of an expression stack are not exceeded. Suppose we have two instructions, push and pop, and a stack which is limited to n items. The instructions only change the number of items in a stack, so we work in terms of change relative to the start, and will assume that the stack is initialised at the start of the program.

Take a quartet of integers (d, i, s, g). Let the accumulated change in value of the stack height from start to finish through a single route in the path expression be c, which may be positive or negative. Let d be the least value of c for all the routes in a path and let i be the greatest value of c for all the routes. Let s be the least value of the stack height which occurred anywhere on any of the routes, relative to the start, and let g be the greatest.

$$\begin{aligned}\text{atom}(\text{push}) &= (1, 1, 1, 1) \\ \text{atom}(\text{pop}) &= (-1, -1, -1, -1) \\ \text{atom}(a) &= (0, 0, 0, 0) \textbf{ otherwise} \\ 1 &= (0, 0, 0, 0)\end{aligned} \tag{3.11}$$

$$\begin{aligned}&(d1, i1, s1, g1).(d2, i2, s2, g2) = \\ &\quad (d1+d2, \\ &\quad i1+i2, \\ &\quad \min(s1, s2+d1), \\ &\quad \max(g1, g2+i1))\end{aligned}$$

$(d1, i1, s1, g1) + (d2, i2, s2, g2) =$

$(\min(d1, d2),$

$\max(i1, i2),$

$\min(s1, s2),$

$\max(g1, g2))$

$(d, i, s, g)* =$ (**if** $d < 0$ **then** $-\infty$ else 0,

if $i > 0$ **then** ∞ **else** 0,

if $d < 0$ **then** $-\infty$ **else** s,

if $i > 0$ **then** ∞ **else** g)

If the final value of g is greater than n then there is an error, and if the final value of s is less than 0 there is an error. Pinpointing the error can be more difficult since it is not necessarily a localised mistake, but the area where the maximum or minimum occurred can be found. It is more complex to try to find the same kind of mistake in the presence of a stack-reset operation, because the natural extension of Formula 3.11 does not obey the distributive law. However, a modification of this method can be designed which computes both relative and absolute maxima and minima.

4. CONCLUSIONS

We have given a number of examples of useful diagnostic properties of microcode that can be computed using the method of homomorphisms, and shown how a program has been written which separates this calculation into a part involving the control structure which depends only on the AMD 2910, and a part involving knowledge of the Perq microcode. This program runs fairly quickly and has proved its use in removing errors from about 4000 instructions of Perq microcode implementing the Flex architecture. Many of the tests which were run found errors, some of which were subtle, involving interrupts or unlikely combinations of circumstance which would have been difficult to remove by the usual methods of trial and error.

Acknowledgement

I would like to acknowledge many helpful remarks from B. D. Bramson on a draft of this paper.

APPENDIX

The definition of regular algebras can be expressed formally; see for example Ref. 12. A regular algebra, R, consists of an alphabet, A, of atomic values, a unit value, 1, and three operations '.', '+' and '*'.

$$. : R \times R \to R$$

$$+ : R \times R \to R$$

$$* : R \to R$$

The following laws are satisfied.

$(a.b).c = a.(b.c)$	associativity of .
$(a+b)+c = a+(b+c)$	associativity of +
$a+b = b+a$	commutativity of +
$a+a = a$	idempotence of +
$a.(b+c) = a.b+a.c$	left distribution of . over +
$(a+b).c = a.c+b.c$	right distribution of . over +
$1.a = a$	1 is a left unit for .
$a.1 = a$	1 is right unit for .
$a* = 1+a*.a$	
$a* = (1+a)*$	

and $a = b*.c$ is a solution of $a = b.a+c$.

A regular algebra with a zero, 0, also satisfies

$0.a = 0$	0 is a left zero for .
$a.0 = 0$	0 is a right zero for .
$0+a = a$	0 is a unit for +

In most cases we do not need a zero for the . operation, but we would like a unit for + to provide a bottom element in the ordering.

It is also true that

$$a* = \sum_{i=0}^{i=\infty} a.a... \quad (i \text{ factors})$$

The results showing that homomorphisms from the path algebra preserve least upper bounds can be found in Ref. 8.

REFERENCES

1. A. V. Aho, *Principles of Compiler Design*. Addison-Wesley, London (1977).
2. R. C. Backhouse and B. A. Carre, Regular algebra applied to path-finding problems. *J. Inst. Math. Applic.* **15**, 161–186 (1975).
3. B. D. Bramson and S. J. Goodenough, Data use analysis for computer programs. *Unpublished MOD(PE) report* (1982).
4. B. D. Bramson, Information flow analysis for computer programs. *Unpublished MOD(PE) report* (1982).
5. P. and R. Cousot, Abstract interpretation. A unified lattice model for static analysis of programs by construction of approximation of fixpoints. *4th ACM Symp. on Principles of Programming Languages, Los Angeles* (1977).
6. J. M. Foster, *Regular expression analysis of procedures and exceptions*. R.S.R.E. Report No. 85008 (1985).
7. J. B. Kam and J. D. Ullman, Monotone data flow analysis frameworks. *Acta Inf.* **7** 305–317 (1977).
8. J. Mezei and J. B. Wright, Algebraic automata and context-free sets. *Information and Control* **11**, 3–29 (1967).
9. A. Mycroft, The theory and practice of transforming call-by-need into call-by-value. *Proc. 4th Int. Symp. on Programming*. Lecture notes in Computer Science no. 83. Springer-Verlag, Heidelberg (1980).
10. A. Mycroft and F. Nielson, Strong abstract interpretation using power domains. *ICALP 1983*. Lecture notes in Computer Science no. 154. Springer-Verlag, Heidelberg (1983).
11. B. K. Rosen, Monoids for rapid data flow analysis. *SIAM J. Comput.* **9**, 159–196 (1980).
12. A. Salomaa, Two complete axiom systems for the algebra of regular events. *JACM* **13** (1), 158–169 (1966).
13. M. Schaeffer, *A Mathematical Theory of Global Program Optimisation*. Prentice-Hall, Hemel Hempstead (1973).
14. R. E. Tarjan, A unified approach to path problems. *JACM* **28** (3), 577–593 (1981).
15. B. Wegbreit, Property extraction in well-founded property sets. *IEEE Trans Software Eng.* **1**, 270–285 (1975).