Update Plans for Parallel Architectures*

Hugh Osborne
Department of Computer Science
University of York
Heslington
York YO1 5DD
United Kingdom

Abstract. This paper proposes Update Plans as a specification formalism for abstract machines for parallel architectures. Update Plans are a formal specification language for abstract and concrete machines. First results in using Update Plans to specify parallel architectures are illustrated, and some suggestions for further research are made.

1 Introduction

While there are formalisms for high level specification of abstract machines, such as transition systems [10, 6, 4, 3], and for low level specifications of machine architectures, both from the viewpoint of instruction sets [29] and from that of the circuitry [14, 20], there seems to be no such formalism for an intermediate level. Update Plans are such a formalism. This paper gives a brief introduction to Update Plans, and discusses their application to the specification of parallel architectures.

Update Plans are a formalism for the description of (abstract) machines and algorithms. With respect to data structures descriptions are concrete, detailed and low-level (although it is easy to abstract away from irrelevant details). With respect to algorithmic structures they are abstract and high-level. This makes Update Plans particularly suitable as a specification language for the description of large classes of machine architectures. Since a formal semantics for Update Plans exists [23, 24] it is possible to reason formally about the languages specified.

The aim of this paper is to propose Update Plans as a specification formalism for parallel architectures. It is beyond the scope of the paper to give a full definition of Update Plans. Section 2 contains a brief introduction, recapping on information available elsewhere [19, 25, 24, 26]. Section 2.1 covers basic Update Plans [19, 25, 24, 26], and section 2.2 introduces a macro-like mechanism known as *archetypes* [26]. In section 3 the use of Update Plans in specifying parallel architectures is illustrated. Section 3.1 contains a specification of simple asynchronous architectures. Section 3.2 repeats this for synchronous architectures. Though both of these illustrations are of fairly concrete

*In: *Proceedings of the 3rd Abstract Machines Workshop,* Abstract Machine Models for Parallel and Distributed Computing, *M. Kara, J.R. Davy, D. Goodeve and J. Nash (Eds.),* IOS Press, *1996*

machines, these methods could also easily be applied to abstract machines. Update Plans have already been used for the specification of abstract machines for functional languages [22, 25, 26] and functional logic languages [21].

Conclusions are drawn in section 4, which also includes some proposals for further research. Some other approaches to (parallel) machine specification are discussed in section 5.

2 Update Plans

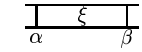
Update Plans constitute a language for the high level description of low-level structures and mechanisms. Originally [19], they were used to describe the machine-code generated by a compiler, in a way that was abstract and relatively machine-independent, but at the same time very close to machine language. They were deliberately designed in such a way that an update plan resembles a set of rewrite rules or, by application of syntactic sugar, function definitions in an applicative language. More recently, Update Plans have been used for the abstract as well as very concrete description of machine architectures. Update Plans bridge the gap between existing high-level and low-level formalisation languages and methods.

The basic concept underlying the Update Plan formalism is that of an *update* of a machine configuration, each possible update being specified by an *update rule*. A set of update rules is specified by an *update scheme*, the update rules being obtained by instantiation in the usual way. Each update scheme will represent one conceptual class of possible updates of machine configurations — e.g. the effect of one machine instruction. A full machine specification — an *update plan* — will consist of a set of update schemes. When combined with a specification of an initial machine configuration this will give a complete programme, known as an *update script*.

2.1 Basic Update Plans

An update plan is a set of *update schemes*, each of which may contain unspecified values. A update scheme containing no unspecified values is called an *update rule*. Update schemes yield update rules by instantiation. A scheme consists of a left-hand side and a right-hand side, both being sets of *locator expressions*.

A locator expression is a triple, written $\alpha[\xi]\beta$, where α and β are addresses or *locators* in one of a set of stores in an underlying machine model. Each store is a linear countably infinite sequence of memory cells (e.g. bytes or machine words). The above locator expression expresses the fact that $\alpha \leq \beta$ and that the cells between the addresses α and β contain (a particular representation of) the value of ξ.

The notation of a locator expression is chosen such that it looks like the picture 

. An update scheme states that if it is applicable (i.e. if all locator expressions in its left-hand side are satisfied), the memory may be minimally updated such that thereafter all locator expressions in its right-hand side are satisfied. The left and right-hand side of an update scheme are separated by an arrow ($\Longrightarrow$) which optionally carries a *guard* ($=\!\![\ \gamma\ ]\!\!\Rightarrow$) which is an additional applicability condition. An *update script*

consists of an *initial configuration* and an update plan. The update plan is executed by repeatedly choosing an applicable update scheme from the plan, and applying it, until the configuration is such that no scheme is applicable. This *final configuration* is the result of the script.

For instance, the two-scheme update plan in example 1(a) computes the GCD of the number initially between `A` and `B` and that initially between `B` and `C`.

Example 1 (a)

$$\texttt{A[x]B B[y]C} \;=\![\; \texttt{x} < \texttt{y} \;]\!\Rightarrow\; \texttt{B[y} - \texttt{x]C}.$$
$$\texttt{A[x]B B[y]C} \;=\![\; \texttt{x} > \texttt{y} \;]\!\Rightarrow\; \texttt{A[x} - \texttt{y]B}.$$

Capitalised words denote constants: `A`, `B` and `C` are fixed locators and `x` and `y` are unspecified values. In fact, if at any stage of the computation the machine configuration contains `A[9]B` and `B[6]C`, (only) the update rule in example 1(b) is applicable, whereupon the `9` is replaced by a `3`.

Example 1 (b)

$$\texttt{A[9]B B[6]C} \;\Longrightarrow\; \texttt{A[3]B}$$

An unspecified value on the left hand side of an update scheme can be considered a variable which obtains its value by means of instantiation. This view will be particularly useful in understanding the macro like mechanism introduced in section 2.2.

Unspecified values may also be used as locators and obtain their actual value by instantiation, as illustrated by the (non-deterministic) update plan in example 2 which, in the context of a set of sensible typing rules, sorts the sequence initially between `A` and `C`.

Example 2

$$\texttt{A[xs]a a[x]b b[y]c c[ys]C} \;=\![\; \texttt{x} > \texttt{y} \;]\!\Rightarrow\; \texttt{a[y]b b[x]c}.$$

By a simple notational convention, acknowledging the existence of a *programme counter* at a fixed locator (by convention `PC`) but hiding it, certain update schemes may be written as so-called *commands*. Other such conventions allow the omission of irrelevant addresses and the combination of adjacent locator expressions. Repeated left hand sides are indicated by the '`"`' symbol. See example 3(a), which may be part of the description of some zero-address machine.

Example 3 (a)

$$\texttt{PUSH x S[q]} \;\Longrightarrow\; \texttt{S[p] p[x]q}.$$
$$\texttt{ADD S[q] [x y]q} \;\Longrightarrow\; \texttt{S[p] p[x} + \texttt{y]q}.$$

The expressive power of Update Plans is greatly increased by the use of a macro like mechanism known as *archetypes*, and in fact the "programme counter" convention is an instance of the archetype mechanism.

2.2 Archetypes

The archetype mechanism greatly increases the expressive power of Update Plans. Using the archetype mechanism complicated pointer structures, families of such structures, and even infinite classes of arbitrarily large structures may be replaced by a single archetype call, thus making it possible to express many update schemes as one.

Archetypes are inspired by macro mechanisms. Their parameter resolution system is purely "macro" in flavour, though their expansion may be context driven, i.e. dependent on the configuration in which the macro is expanded. Archetypes bear a degree of similarity to *syntax macros* [17].

The body of an archetype definition closely resembles a command form update scheme. See, for example, example 4(a) which could define a predecrement addressing mode. The (output) parameter `v` obtains its value by means of instantiation against the current configuration.

Example 4 (a)

$$\texttt{predec(v)} \;=\; \texttt{PREDEC r r[b] a[v]b} \;\Longrightarrow\; \texttt{r[a]}.$$

Conceptually this archetype returns the value `v`, while updating the contents of `r`.

Both the left and right hand side of the body of an archetype definition consist of a possibly empty 'textual' part (e.g. `PREDEC r`) known as the *expansion*, and a set of locator expressions (e.g. `r[b] a[v]b`) called the *context*. Archetype calls occur in indexed pairs, with one element of the pair on the left hand side of the update scheme or archetype definition in which it occurs, and the other on the right hand side. Archetypes are expanded by replacing the left and right hand side calls by the corresponding expansions, and by adding the contexts to the respective sides. Variables will be renamed, if necessary. There is a resolution mechanism for determining replacement expressions for an archetype's parameters. This is demonstrated in example 4(b), which gives a possible expansion of $\texttt{ADD predec}_1\texttt{(x) predec}_2\texttt{(y)} \Longrightarrow \texttt{predec}_1\texttt{(x) predec}_2\texttt{(y) r[x + y]}$.

Example 4 (b)

$$\begin{aligned}&\texttt{ADD r PREDEC r}_1\ \texttt{PREDEC r}_2\\ &\quad \texttt{r}_1\texttt{[b}_1\texttt{]}\ \texttt{r}_2\texttt{[b}_2\texttt{]}\ \texttt{a}_1\texttt{[v}_1\texttt{]b}_1\ \texttt{a}_2\texttt{[v}_2\texttt{]b}_2 \Longrightarrow \texttt{r[v1 + v2]}\ \texttt{r}_1\texttt{[a}_1\texttt{]}\ \texttt{r}_2\texttt{[a}_2\texttt{]}.\end{aligned}$$

Syntactic sugar allows one of the elements of an archetype call pair to be omitted if the corresponding expansion is empty. The indices may then also be omitted, as they are superfluous. The archetype calls in example 4(b) can then be written as in example 4(c).

Example 4 (c)

$$\texttt{ADD r predec(x) predec(y)} \implies \texttt{r[x + y]}.$$

Syntactic sugar also makes it possible to share left hand sides of archetype definitions, and to omit the guard and right hand side from the body of an archetype definition.

3 Parallelism

The following section shows how the archetype mechanism can be applied to the specification of asynchronous parallelism (in section 3.1), and introduces a further extension for the specification of synchronous parallelism (in section 3.2).

3.1 Asynchronous Parallelism

An update scheme in which the programme counter has been "hidden" on both the left and right hand sides is in fact the body of an archetype definition. Each such scheme is considered to be the body of a definition of the archetype pc(), where pc is some archetype name not used elsewhere in the update plan.

Example 3 (b)

The two update schemes in example 3(a) can be rewritten as the archetype definitions:

$$\begin{aligned} \texttt{pc()} &= \texttt{PUSH x S[q]} &&\implies \texttt{S[p] p[x]q.} \\ \texttt{pc()} &= \texttt{ADD S[q] [x y]q} &&\implies \texttt{S[p] p[x + y]q.} \end{aligned}$$

The "programme counter" convention is then equivalent to rewriting each command as such an archetype definition and adding the update scheme

$$\texttt{PC[pc] pc[pc()]qc} \Longrightarrow \texttt{PC[pc']\ pc'[pc()]qc}.$$

to the update plan.

This makes it very simple to specify asynchronous parallel processors with shared memory and identical instruction sets. If there are n such processors it suffices to define n programme counters, all containing a call of the pc archetype.

$$\begin{aligned} \texttt{PC}_1\texttt{[pc] pc[pc()]qc} &\Longrightarrow \texttt{PC}_1\texttt{[pc']\ pc'[pc()]qc}. \\ &\vdots \\ \texttt{PC}_n\texttt{[pc] pc[pc()]qc} &\Longrightarrow \texttt{PC}_n\texttt{[pc']\ pc'[pc()]qc}. \end{aligned}$$

3.2 Synchronous Parallelism

In section 3.1 nondeterminism in Update Plans was exploited for the specification of asynchronous parallel processors. An adaptation of nondeterminism can be used to specify synchronous parallelism. This adaptation is described in section 3.2.1. An application — pipelining in a partial specification of the Berkeley RISC II CPU — is presented in section 3.2.2.

Update Plans intuitively "work" by instantiating all applicable update schemes and then making a nondeterministic choice which of the update rules thus obtained to apply. An alternative would be to apply all of the update rules simultaneously. This is the idea behind a *parallel block*. A parallel block is a set of update schemes all applicable instantiations of which will be applied at the same time, if possible. The caveat is that some of these instantiations may have conflicting right hand sides. If this is the case, then none of the schemes are applied, the reasoning being that the update rules are applied as if they all formed one update rule, and update rules with inconsistent right hand sides may not be applied.

3.2.1 Syntax

Parallel blocks are delimited by the *open parallel block symbol*, '(||', and the *close parallel block symbol*, '||)'. Alternatively, if the document preparation system in use allows it, the whole parallel block may have a pipeline symbol placed next to it. This is shown in example 5.

Example 5

parallel block symbols — pipeline symbol

```
(||
    IR+S[src(x)]     ⟹RX[x].          ‖ IR+S[src(x)]     ⟹RX[x].
    IR+D[dst(ea,y)]⟹RY[y] MAR[ea].    ‖ IR+D[dst(ea,y)]⟹RY[y] MAR[ea].
||)
```

In keeping with the macro character of archetypes, archetype expansion conceptually takes place before interpretation of parallel blocks, and any expansions remain within the parallel block.

3.2.2 An Example: The Berkeley RISC II CPU

In the following example a subset of the Berkeley RISC II instruction set will be specified, first at the instruction level, to provide a general view; then at the instruction cycle level, but without pipelining; and then, finally, with pipelining, first without internal forwarding, and then with internal forwarding. The specification is based on the description of the Berkeley RISC II CPU given by Katavenis [15].

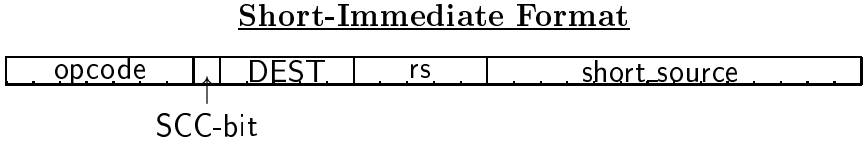


Figure 1: Format of a RISC II arithmetic instruction.

```
ADD OFF r s (IMM y) CWP[cwp] cwp+s[x]
    =[ r ∈ GLBL ∧ r ≠ GBASE ∧ s ∈ LOC ∧ cwp+s ≠ GBASE ]⇒
        GBASE+r[y+x].
```

Figure 2: A typical Berkeley RISC II arithmetic instruction. This instruction takes the value y, adds it to the contents of local register s, and places the result in register r. The condition codes are not set.

The Instruction Set. The subset that will be specified is that of the arithmetic commands. An arithmetic instruction on the RISC II has a *short-immediate format*, as shown in figure 1 in which: DEST is the destination, which must be a register; rs, which must also be a register, is the first operand; and short_source is the second. Archetypes can be defined [26] for the elements of such an instruction: `arith(x,y,r)` for the arithmetic operations; `scc(flag)` for the condition codes; `aflg(r)` for the condition codes flag; `reg(ea)` for the destination, which must be addressed via a register; `rs(v)` for the first operand, a "register source"; and `short(v)` for the second, which must be a "short immediate" operand. The first operand is, for example, defined as

```
rs(0)   = reg(GBASE).
rs(val) = reg(ea) ea[val]  =[ ea ≠ GBASE ]⇒ .
```

The value of register 0 (here represented by the constant `GBASE`) is always zero, and no result is ever written to register 0.

All arithmetic instructions are now defined by

```
arith(x,y,r) scc(aflg(r)) reg(ea) rs(x) short(y)
        =[ ea = GBASE ]⇒ .
   "    =[ ea ≠ GBASE ]⇒ ea[r].
```

A possible concrete example of this update scheme, with the archetypes fully expanded, is given in figure 2. Some expressions derived during the expansion have been simplified.

The Instruction Cycle. The instruction cycle consists of three phases; instruction fetch, execute, and write. In the instruction fetch phase the instruction currently addressed by the `PC` is copied to the instruction register `IR`. The execute phase accesses the operands and performs the operation, setting the condition codes, if necessary, and placing the result in the `RES` register. In order to keep all accesses of the internal structure of an instruction within one phase, the destination address is also copied to the `DST` register, where it will be used by the write phase. The execute phase also updates the `PC` to

contain the address of the next instruction to be executed. The PC is only updated at execution, rather than when the instruction is fetched, since addressing may be relative to the PC for some instructions. Finally, the write phase copies the result from the RES register to the destination. A clock is introduced to ensure that the phases take place sequentially. In the next step, pipelining, the clock will be eliminated.

```
FETCH() =
  PC[pc] pc[instruction]⟹IR[instruction].
EXEC() =
  IR[arith(x,y,r) scc(aflg(r)) reg(dst) rs(x) short(y)] PC[pc]
      ⟹RES[r] DST[dst] PC[pc+WORD].
WRITE()
  =  DST[dst] RES[r]=[ dst = GBASE ]⇒.
  =        "       =[ dst ≠ GBASE ]⇒dst[r].
```

WORD is the length of an instruction. The instruction cycle is defined by:

```
tick() = FETCH() ⟹ EXEC.
       = EXEC()  ⟹ WRITE.
       = WRITE() ⟹ FETCH.
```

The only update scheme in the update plan is

```
CLOCK[tick()]⟹CLOCK[tick()].
```

It is easy to establish that the instruction cycle specification is a correct translation of the higher level specification.

Pipelining. The first step in introducing pipelining is eliminating the clock, and introducing a parallel block in order to synchronise the phases of the instruction cycle. The tick() archetype bodies are elevated to update schemes within the parallel block.

```
PC[pc] pc[instruction]⟹IR[instruction].
IR[arith(x,y,r) scc(aflg(r)) reg(dst) rs(x) short(y)] PC[pc]
    ⟹RES[r] DST[dst] PC[pc+WORD].
DST[dst] RES[v]=[ dst = GBASE ]⇒ .
      "       =[ dst ≠ GBASE ]⇒ dst[v].
```

For most configurations this specification will have the same semantics as the specification of the non-pipelined machine. For example, given the following initial configuration (the locator expression GBASE[0 1 2 ...] specifies that register 0 contains zero, register 1, one, etc; the expression DST[GBASE] initialises the destination to REG 0, ensuring no unwanted write access will take place before the pipeline is full; the last two instructions are included as null operations, giving the pipeline time to clear):

[3.1] pc_1[ADD OFF 1 2 (IMM 1)]pc_2[ADD OFF 3 4 (IMM 1)]~
pc_3[ADD OFF 0 0 (REG 0)]pc_4[ADD OFF 0 0 (REG 0)]pc_5

PC[pc_1] IR[ADD 0 0 (REG 0)] RES[0] DST[GBASE] GBASE[0 1 2 ...]
CLOCK[FETCH].

both the pipelined and non-pipelined specifications will have the net effect:

$$\texttt{PC}[\texttt{pc}_1] \Longrightarrow \texttt{PC}[\texttt{pc}_5]\ \texttt{GBASE}{+}1[3]\ \texttt{GBASE}{+}3[5].$$

However not all configurations are as "well behaved" as 3.1. In the pipelined machine care must be taken if an instruction uses the result of the previous instruction. Since the operands of the next instruction are determined before the result of the previous instruction has been written to its destination, a stale value may be used. For example, if the second instruction in 3.1 were to be `ADD OFF 3 1 (IMM 1)` rather than `ADD OFF 3 4 (IMM 1)` the net effect of the sequential specification would be

$$\texttt{PC}[\texttt{pc}_1] \Longrightarrow \texttt{PC}[\texttt{pc}_5]\ \texttt{GBASE}{+}1[3]\ \texttt{GBASE}{+}3[4].$$

while that of the pipelined specification would be still be

$$\texttt{PC}[\texttt{pc}_1] \Longrightarrow \texttt{PC}[\texttt{pc}_5]\ \texttt{GBASE}{+}1[3]\ \texttt{GBASE}{+}3[5].$$

This problem is avoided by using *internal forwarding*. The execution phase must check if the result waiting to be written should be used instead of a stale value. This can be done by revising the specification of a register source.

$$\begin{aligned}
\texttt{rs}(0) &= \texttt{reg}(\texttt{GBASE}).\\
\texttt{rs}(\texttt{res}) &= \texttt{reg}(\texttt{dst})\ \texttt{DST}[\texttt{dst}]\ \texttt{RES}[\texttt{res}] =\!\!\mid\ \texttt{dst} \neq \texttt{GBASE}\ \mid\!\!\Rightarrow .\\
\texttt{rs}(\texttt{val}) &= \texttt{reg}(\texttt{ea})\ \texttt{DST}[\texttt{dst}]\ \texttt{ea}[\texttt{val}] =\!\!\mid\ \texttt{ea} \neq \texttt{GBASE} \wedge \texttt{ea} \neq \texttt{dst}\ \mid\!\!\Rightarrow .
\end{aligned}$$

The definition of $\texttt{rs}(\cdot)$ has been extended. The second declaration "intercepts" a result of the previous instruction. `DST` and `RES` contain the result register and value of the previous instruction, which are still in the pipeline. If $\texttt{rs}(\cdot)$ requires the value in that result register it should take the "new" value from `RES`, rather than the "old" value at `ea`, since this will not yet have been updated.

The rest of the specification is unchanged. The techniques of [26] could be applied to this specification and that of the sequential machine in order to prove their semantic equivalence. The correctness of optimalisers which take advantage of synchronous parallelism could also be shown.

4 Conclusions

Update Plans provide a compact and lucid specification formalism for abstract machines. Since the same formalism can be used to specify lower level representations, such as the instruction set of some concrete machine, it is possible to prove transformations of abstract machine code to such instruction sets correct [26].

The archetype mechanism provides a natural method for specifying asynchronous parallel processes. The parallel block notation achieves the same for synchronous parallel processes. Further work is needed in classifying the different ways in which the concept of parallelism is used (e.g. co-operating processes, or systolic processes, or synchronous processes). Work is in progress[1] at the University of York on the specification of abstract machine architectures for LINDA [11, 5].

[1] As part of the project "Concurrent Architectures for Heterogeneous Knowledge Manipulation Systems", part of the EPSRC's AIKMS programme [9].

To date Update Plans have only been used for the specification of the *effect* of machine operations. It would be fairly easy to develop an annotation to indicate the *cost* of applying an update scheme or archetype. This would make it possible to apply Update Plans to questions of performance. The existence of a working implementation (some work has already been done on this [26]) would make this even more useful, since the cost of proposed solutions could then be estimated by means of a simulation.

Update Plans form an abstract rewrite system. Work on their relation to ARS's, and to graph rewrite systems in particular, should lead to insights which would allow the current well formedness conditions [26] to be relaxed while still guaranteeing, for example, finite ambiguity or one or more of the Church-Rosser properties. Update Plans also define a category having consistent configurations as its objects, and update rules as its morphisms. Further research may make a category theoretical description of Update Plans possible.

For a specification and programming language to be useful at all it is imperative that one may prove a programme correct (e.g. equivalent to a specification) and/or derive a programme from a specification and/or prove that a specification or programme has certain desirable properties. Therefore a verification, and possibly transformation method should be developed, together with appropriate heuristic rules. A first step in this direction has been taken [26].

The language also needs to be validated on a set of major sample applications, in order to develop the pragmatics of use. Among these applications are those in which *parallelism* plays a dominant rôle, e.g. the full specification of a 'real' processor, such as SUN's Sparc processor, Digital's Alpha [2] or IBM's PowerPC, or the (abstract) implementation of network protocols based on various communication primitives.

Update Plans can also serve as a useful didactic tool. They are used in the compiler construction course given at the university of Nijmegen [28], and in the machine architecture course at the university of Utrecht.

A specification of a possible implementation is available [26]. Such an implementation would make it possible to prototype a wide variety of systems.

5 Other approaches

Other specification formalisms have been proposed for parallel architectures. However many of these — for example the Γ language [3] and the Chemical Abstract Machine [4] — start at a higher level of abstraction than Update Plans. The lower level of the underlying machine model of Update Plans has deliberately been chosen to ensure that Update Plan specifications are reasonably simple to implement on concrete machines. In fact there is a specification of a possible implementation of an Update Plan compiler [26] which would do this automatically. The low level of the underlying machine model does not mean, however, that a high level of abstraction cannot be attained in Update Plans. In particular archetypes provide a powerful tool for abstraction.

The UNITY model [6] bears the most similarity to Update Plans. The assignment statement in the UNITY model is the counterpart of an update scheme. The computational models of the two formalisms are almost identical — execution starts in some given initial state, and at each step a nondeterministic choice is made of an applicable

update or assignment. It is however much easier in Update Plans than in the Unity model to express assignments to complex structures (updates of such structures), especially at machine level. The UNITY model, in common with the Γ language and the Chemical Abstract Machine, is more suited to the specification of parallel *algorithms*, while Update Plans are more suited to the specification of parallel *architectures*. Using the terminology of Banâtre and Métayer [3], the UNITY model, the Γ language and the Chemical Abstract Machine are aimed at the specification of *logical parallelism*, while Update Plans are more concerned with *physical parallelism*.

Some other methods have been proposed for specifying low level activities. Abstract machines have been specified using transition systems [10], informal descriptions [30], an imperative programming style [18, 27] and functional languages [16], to name but a few. The best known contributions from the concrete side are probably ISPS [29] and register transfer languages, for example three address code [1]. None of these methods, however, is particularly suited to specifying low level machines, let alone reasoning about them.

At least two other researchers have developed specification languages aimed more specifically at instruction sets [12, 8, 7, 13]. In both cases the underlying model is applicative and the surface structure has a strong imperative flavour. Reversing this, as in update schemes, leads to a model in which it is easy to reason and yet in which essentially imperative machine primitives can easily be expressed and combined.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Communications of the ACM, 26(2), February 1993. (Special issue on the Alpha chip).

[3] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[4] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[5] N. Carriero and D. Gelertner. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[6] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[7] Todd A. Cook, Paul D. Franzon, Ed Harcourt, and Thomas K. Miller III. System-level specification of instruction sets. In *Proceedings of the 1993 IEEE International Conference on Computer Design*, pages 552–557, 1993.

[8] Todd A. Cook and Ed Harcourt. A functional specification language for instruction set architectures. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 11–19, 1994.

[9] Engineering and Physical Sciences Research Council. Architectures for Integrated Knowledge-Manipulation Systems. `http://cswww.essex.ac.uk/AIKMS/Welcome.html`.

[10] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*. Springer Verlag, 1987. Gilles Kahn (editor).

[11] D. Gelertner. Generative communications in LINDA. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[12] Robert Giegerich. A formal framework for the derivation of machine-specific optimizers. *ACM Transactions on Programming Languages and Systems*, 5(3):478–498, July 1983.

[13] Ed Harcourt, Jon Mauney, and Todd Cook. Functional specification and simulation of instruction set architectures. In *Proceedings of the International Conference on Simulation and Hardware Description Languages*, pages 3–8. The Society for Computer Simulation, 1994.

[14] IEEE, New York. *IEEE Standard VHDL Language Reference Manual*, 1988. IEEE Std. 1076–1987.

[15] Manolis G.H. Katevenis. *Reduced Instruction Set Architectures for VLSI.* ACM Doctoral Dissertation Awards. The MIT Press, 1985.

[16] Pieter Koopman. *Functional Programs as Executable Specifications.* PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1990.

[17] B.M. Leavenworth. Syntax macros and extended translation. *Communications of the ACM*, 9(11):790–793, November 1966.

[18] Hendrik C.R. Lock. An abstract machine for the implementation of functional logic programming languages. Technical report, ESPRIT Basic Research Action No. 3147 (the Phœnix Project), 1990.

[19] Hans Meijer. *Programmar: A Translator Generator.* PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1986.

[20] M.Seutter. *The development of semantic functions for a system description language with multiple interpretations.* PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1994.

[21] Hugh Osborne. An update plan for the JUMP machine. ESPRIT Basic Research Action No. 3147 (the Phœnix Project), 1991. Internal memo.

[22] Hugh Osborne. Update plans, an example: Flip. Technical report, ESPRIT Basic Research Action No. 3147 (the Phœnix Project), 1991.

[23] Hugh Osborne. Update plans, an introduction. Technical report, ESPRIT Basic Research Action No. 3147 (the Phœnix Project), 1991.

[24] Hugh Osborne. The semantics and syntax of update schemes. In *Code Generation — Concepts, Tools, Techniques (Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, 20–24 May 1991)*, Workshops in Computing, pages 210–223. Springer Verlag, 1992.

[25] Hugh Osborne. Update plans. In *Proceedings of the 25th Hawaii International Conference on System Sciences* (Volume II: *Software Technology*), pages 488–496. IEEE Computer Society Press, 1992.

[26] Hugh Osborne. *Update Plans — A High Level Low Level Specification Language.* PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1994. Copies available from the author, Department of Computer Science, University of York, Heslington, York Y05 5DD, UK. A partial version is also available at `http://www.cs.york.ac.uk/ hugh/update/thesis.html`.

[27] Simon L. Peyton Jones and Jon Salkild. The spineless tagless G-machine. University College, London, 1989.

[28] Janos Sarbo. Collegedictaat vertalerbouw. Technical report, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1991. (In Dutch).

[29] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell. *Computer Structures: Principles and Examples.* Computer Science Series. McGraw-Hill, 1982.

[30] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, Artificial Intelligence Center, SRI International, Menlo Park, California, USA, 1983.