Parallel computation in definitive models

Meurig Beynon, Mike Slade, Yun Wai Yung
Department of Computer Science, University of Warwick, Coventry CV4 7AL

Abstract

This paper describes an abstract machine model for parallel computation that exploits a programming paradigm based upon definitions. Proposed applications to the implementation of CAD systems, and to the modelling and simulation of concurrent systems are outlined. A brief comparison with alternative approaches to parallel computation is included.

§1 Introduction

The capability for parallel action exhibited by the spreadsheet has often been remarked. It may be interpreted as a particular illustration of a general principle: given a family of formulae defined in terms of a set of free parameters, the global effect upon the values of these formulae when new values are assigned to a distinct subset of the free parameters is independent of the order in which these assignments are made. Our purpose in this paper is

(1) to describe an abstract machine model for parallel computation that exploits this principle,

(2) to indicate its potential applications to the implementation of novel CAD systems [3,4], and to the simulation of concurrent systems [2,5].

Our model for parallel computation is based upon generalising spreadsheet principles in two respects. We need to extend the concept of "defining families of formulae" to richer semantic domains. We must also consider the implications of allowing several agents to act concurrently within the context of such a family of formulae.

The issues involved in generalising spreadsheet principles to richer domains have been dealt with extensively in previous work [4]. The approach we adopt is to choose an *underlying algebra* of data types and operators, and to introduce variables whose values lie in the underlying algebra and are in general defined in terms of other variables and constants by means of algebraic expressions using the operators. For the traditional spreadsheet, the underlying algebra is scalar arithmetic, and the variables are associated with the cells of the display. In a typical interaction, the user either declares a new variable, or (re)defines the value of a variable by supplying an appropriate formula for its value. The only restriction upon such definitions is that they should be free of cyclic reference - the value of a variable cannot be defined in terms of itself. At any stage, the current state of the interaction is then represented by a system of interrelated variable definitions that establishes an acyclic system of dependencies between variable values. The term "definitive", meaning "definition-based", has been adopted for programming notations based upon these principles.

Several examples of definitive notations have already been developed. By judiciously choosing the underlying algebra of variable values it is possible to address a wide variety of applications; our present repertoire includes notations for interactive graphics [3], for geometric modelling [4] and for screen layout in textual windows. The research we shall

describe in this paper is intended to provide a common fundamental basis for two superficially unrelated subsequent developments of work on "definitive notations for interaction":

(a) the application of definitive programming principles to the modelling and simulation of concurrent systems, as described in [2,5],

(b) the specification of special purpose software tools to support the implementation of definitive notations, as described in [3,4].

In extending the framework for user-computer interaction provided by a definitive notation to accommodate several agents, the key idea is to regard the user as a generic agent. To make this generalisation, it is necessary to examine the role of the user more carefully. When more than one agent is involved, it also becomes important to consider when and how each agent is privileged to act. Our discussion of these issues bears directly upon the design of the abstract machine model to be described in §2.

When considering a single user-computer interaction, it may be reasonable to refer to a family of definitions as an effective way of representing "the state of the interaction", but this is a simplification. The conceptual advantages of a definitive interface stem from the fact that it makes explicit both the parameters that the user can change and the consequences of such changes. In effect, a family of definitions expresses a particular user-view, incorporating knowledge not only of the present state but of latent transitions (c.f. [4]). This is particularly relevant in a multi-agent system, when it is simplistic to speak of the state of the interaction, and it becomes necessary to think rather of representing the views of the participating agents.

A simple - if fanciful - illustration will help to indicate the type of activity that our abstract machine model must support. Imagine that the choke and the accelerator of a car engine are under the independent control of two agents. The speed of the engine (s) may be defined by a functional relationship in terms of how far the choke piston is out (c) and how far the accelerator pedal is depressed (a), so that $s=F(c,a)$. Notice that the values of a and c can be changed simultaneously without interference. Suppose now that a fault develops in the choke linkage, so that the degree of choke is independent of c. The functional relationship defining s then takes the form $s=F(\mathbf{k},a)$,where **k** is a constant, and the value of c is no longer significant. A more serious fault may now develop, whereby the choke and accelerator linkages become intertwined in such a way that the positions of the choke piston and the accelerator pedal are interdependent. In such a case, there may be a constraining relationship between a and c such that $a=G(c)$ and $c=G^{-1}(a)$. Now the agents controlling a and c can each act independently in accordance with a perceived definitive context, but they will in general interfere if they act simultaneously. This is reflected in the cyclic nature of the dependencies if both definitions $a=G(c)$ and $c=G^{-1}(a)$ are present.

Our illustration indicates the need in general to maintain a system of variables definitions that must change dynamically. It also shows that such a system can remain free from cyclic dependency only if agent actions are restricted. Interference between agents is generally associated with the co-existence of two incompatible views - to prevent this we shall need to place guards upon agent actions, specifying the preconditions that must pertain before variables can be redefined. As a very simple illustration of how the use of guards can provide a framework for constraining actions, consider a spreadsheet that is set up in such a way that a user redefinition leading to a critical value for a particular vari-

able (e.g. too small a profit) is revoked. In such a context, there is effectively an agent other than the user whose role is to undo the user definition subject to an appropriate guard being true. By implication, there is also a complementary guard to constrain the user's actions, for further interaction with the spreadsheet is suspended until the violated condition is first restored.

Within the above framework, it is only possible to capture agent actions in terms of "permissions" rather than "obligations" (c.f. [9]). If we elaborate our illustrative example further, we may suppose that the choke and accelerator linkages become so enmeshed that it is impossible to effect any change in the values of a and c without cooperative action between agents. To model this, it becomes necessary to insist that the action of assigning the value **v** to the variable a is precisely synchronised with that of assigning G(**v**) to c. Our abstract machine model enables synchronisation of this kind to be specified, but can also be programmed to support asynchronous activity (c.f. §3).

The four sections of the paper comprise: an account of the abstract machine model; a discussion of the specific applications to which we hope to apply our model; a brief comparison with previous work on parallelism within the functional, logic and object-oriented programming paradigms; a concluding section indicating outstanding issues, and some directions for further research.

§2 The abstract machine model

The illustrative example above motivates an abstract machine model whose operation is described in terms of a dynamically changing system of variable *definitions* together with a system of associated *actions* in the form of guarded commands. The state of the machine execution at any time is determined by the definitions and actions that are extant. (This may be compared and contrasted with a snapshot of the execution of a procedural program.) Each transition from state to state is effected by redefinition of variables, and creation or deletion of variables and actions, generally comprising many such operations performed in parallel. The underlying algebra over which definitions are framed will depend upon the semantics of the application - in fact, as illustrated in §3, there will in general be definitions in many different semantic categories.

To provide effective support for dynamic reconfiguration of definitions and actions, it is important to be able to associate groups of definitions and actions that interact. For instance, in an application, it may be appropriate to introduce a new variable together with an action designed to maintain a constraint upon its value. More generally, the definition of a variable will make reference to other variables whose instantiation must be correlated in time. Creation and deletion of variables and actions is accordingly achieved by introducing and deleting *entities*, each of which comprises a family of variable declarations or definitions together with a set of associated actions.

Formally, our abstract machine model consists of three components: a program store P, comprising a set of entities, a store D of variable definitions, and a store A of actions. Each entity comprises an abstractly specified block of definitions and actions, perhaps parametrised, that is superficially analogous to the declaration of a procedure in a conventional procedural language, or of an object in an object-oriented language. The variables whose definitions appear in D can have values of a variety of different types,

depending upon the application. At all times, the system of functional dependencies between the variables in D is acyclic, and the value of each variable is consistent with its definition. An action takes the form of a guarded sequence of instructions, each of which either redefines a variable, or invokes the introduction or deletion of a block of new definitions and actions into the stores D and A through the instantiation or elimination of an entity.

A computation consists of a sequence of parallel executions of appropriate actions. In a single computational step, the guards of all actions in the action store are evaluated, and the actions associated with true guards executed in parallel. This in general has the effect of changing the contents of the store D by modifying the definitions of variables (possibly including those whose value is implicitly defined by a formula), and may also lead to the introduction or deletion of blocks of definitions and actions. To admit redefinitions involving the evaluation of implicitly defined variables (as is appropriate for instance when referring to the current exchange rate for the purposes of a financial transaction), there is a mechanism for the evaluation of specified expressions in the same context in which the evaluation of guards is carried out.

The full implications of programming within this abstract machine model have yet to be examined in detail. In this paper, our main emphasis is upon outlining the practical applications for our abstract machine model that we are currently pursuing, and that have so far informed its design. Some general issues deserve closer consideration however.

It is clear that the use of a definitive framework for actions lends itself to parallel execution resembling parallel updating of a spreadsheet. In many respects, the identification of potential interference between parallel actions appears to be simpler than in conventional programming paradigms. There is a very important distinction between supplying a definition for a variable, and assigning a current value based upon the evaluation of a given expression, as is typical of procedural methods. This is indicated by the fact that a system of definitions that involves no expression evaluation can be interpreted non-sequentially. In as much as definitions within our programming model may involve the evaluation of variables, we must of course address the traditional issues of read/write interference. In our abstract machine, the elimination of such interference on evaluation corresponds to eliminating evaluation during machine transitions. A fundamental question is the extent to which general algorithmic problems can be expressed in such a way as to take advantage of our machine model; in particular, whether they can be formulated in terms of the implicit definitions of variables that are required to differentiate between our model and purely procedural models for concurrent computation that are notoriously difficult to analyse.

Other issues of interference are peculiar to our machine model. As we illustrated in §1, the avoidance of cyclic dependency in definition is closely connected with ensuring compatibility between the definitive views of the agents acting in any transition. At present, it is not clear how to identify potential interference of this nature in an abstract machine program, but it is in general easier to ensure in the context of a specific application (c.f. §3).

Another important issue is that of guaranteeing the consistency of variable references in definitions and actions. An action that involves the evaluation or redefinition of a variable

requires that the variable is currently extant. The motivation behind the *entity* is that it provides a means of associating variables and actions sharing a common extent in time, thereby alleviating some of the problems of maintaining consistency. It is of interest to contrast the use of entities with the organisation of variables and actions by common sequentially acting agent as in the notation LSD outlined in §3. It is also instructive to compare the problems of formally analysing variable references within an object-oriented programming paradigm (c.f. [1]). Notice in particular that the use of implicit variable definitions has advantages over procedural methods for maintaining functional relationships based upon repeated re-evaluation. In effect, the recipe that defines the value of a variable may be recorded even when evaluation is at present impossible, so that the problems of maintaining consistent variable referencing are potentially less critical.

§3 Applying the machine model

The motivation for developing our abstract machine model is twofold, and derives from divergent strands of research generalising the use of definitive notations for interaction.

The first application is concerned with modelling and simulating the behaviour of concurrent systems, and with the techniques that can be used to describe the parallel actions of processes participating in loosely synchronised system. An appropriate notation for modelling the interaction between agents in a concurrent system using definitive principles (LSD) was first described in [2], and we hope that our abstract machine model can provide a more satisfactory basis for its operational semantics.

A full description of LSD is beyond the scope of this paper, but the principal concepts will be outlined. (Note that, as in [5], we prefer to substitute the term "agent" for the original LSD term "process" imported from SDL.) The basic framework for an LSD model of a concurrent system is similar to the interaction of agents as described in §1. In setting up an LSD model, certain templates for agents are first abstractly declared, and certain specific agent instances are initially instantiated. Each agent has a view of the system in which we distinguish three types of variable: **derivate**s, whose value is implicitly defined by a formula (e.g. the speed of the engine in our illustrative example), **state**s, whose value is explicitly known and conditionally under the agent's control (e.g. the extent to which the accelerator is depressed), and **oracle**s, whose value is explicitly known, but in general subject to change via an external agent (e.g. how far the choke is out). The role that each agent plays is specified by its **protocol**; this takes the form of a set of guarded commands, each consisting of a sequence of actions enabled by an appropriate boolean precondition expressed in terms of variables known to the process. Each action is either redefines a state variable, or leads to the instantiation or removal of an agent.

The interpretation of an LSD model differs from our abstract machine model in significant respects. Though the prior specification and subsequent instantiation of entities is superficially similar to that of agents, the organisation of actions within entities and agents serves an entirely different function. Whilst several actions in an entity may be performed simultaneously in a transition of the abstract machine, an LSD agent operates sequentially, and at any time is either in a waiting mode, pending commitment to action, or in the process of executing a particular enabled sequence of actions specified in its protocol. The guarded commands in a protocol are intended to capture the framework of permissions within which an agent acts, and an asynchronous mode of execution of agent ac-

tions is assumed. As illustrated in [5], reasoning about the behaviour of an LSD model in general involves additional assumptions about synchronisation of duration of action that are beyond the scope of this discussion. Our purpose here is to observe that our machine model has suitable characteristics to support an LSD simulator. These include: means of representing derivates directly, mechanisms to support the creation and deletion of definitions and actions, and a control structure within which agent actions can be programmed to perform asynchronously.

The second application concerns the development of an appropriate software tool for the implementation of definitive notations. A prototype interpreter - EDEN ("an evaluator for definitive notations") has already been implemented and successfully used to support further software development. The EDEN interpreter includes built-in support for definitions resembling that provided by the definition store D in our abstract machine model, but relies primarily upon traditional procedural methods for the specification of actions. For example, it is easy to specify that a particular screen location in a spreadsheet is to be updated whenever the corresponding variable value is changed. In effect, EDEN provides a practical programming tool that makes it readily possible to link complex procedural actions and intricate systems of definitions: a mixed programming paradigm that has proved to be potentially very powerful but can also be difficult to use and analyse. Our abstract machine model is intended as a framework within which to realise the capabilities of such an interpreter without compromising clarity.

It will be helpful to first examine the philosophical issues associated with representing programs in terms of definitions. By way of illustration, consider how a simple definitive notation for interactive graphics might be implemented in the abstract machine model (c.f. [3]). In using such a definitive notation, the user typically constructs an abstract description of a geometrical object in terms of the data types of the underlying algebra. In EDEN, changes to the internal structure of this object trigger procedural actions that are interpreted as changes to the screen display. To describe this updating process satisfactorily in the abstract machine model would require that the screen display itself behaved as a variable, whose value was at all times defined implicitly in terms of the internal representations of the geometrical objects currently on display. Though in principle we could express the relationship between the state of the screen and that of the internal data structures used to construct the display entirely through definitions, we shall in practice prefer to arbitrarily decide what is the appropriate level of abstraction at which to define the screen display and ignore the low-level definitions required for physical realisation.

There is no difficulty in interpreting the definitions that establish the geometric relationships between objects - these can form a part of the definition store D. We shall typically need to handle constraints that cannot be expressed as definitions however: to express the fact that the total area of an object is too large, for instance. In such a case, there are several appropriate responses. It might be that we wish to monitor the violation of such a constraint, displaying an error message whilst it pertains. We might wish to prevent any violation, and revoke a user redefinition leading to such a violation. A more complex solution would be to invoke an appropriate constraint management routine to eliminate the error automatically.

To support constraint monitoring in our abstract machine model, it is necessary to introduce definitions into D that link the content of the screen display (e.g. the text in a special-

purpose error monitoring window) to the characteristics of the objects currently defined. In effect, we must establish a functional relationship between a particular textual window and the prevailing geometric error conditions. For this purpose, we introduce a definitive notation in which it is possible to describe the current content of the screen display: manipulation of the display interface can then be represented through redefinitions and actions within the abstract machine model.

To maintain constraints in other ways, whether by revoking a user redefinition, or by more complex constraint management techniques, requires the introduction of agents other than the user. The simplest form that an agent to impose a constraint might take is described in §1. For more complex methods of constraint maintenance, more sophisticated families of interacting agents may be required e.g. to handle value propagation, or implement iterative solution of a constraint. As a trivial illustration, we may implement an algorithm to find a root by the bisection method by creating an entity comprising a single definition: $X=f(x)$, together with a single action: $x-X > \varepsilon \rightarrow x = |x+X|/2$, where $|...|$ is used to denote expression evaluation. As even this simple example indicates, in using iterative constraint management techniques of this kind, we must beware infinite behaviour.

A fuller discussion of how the implementation of a sophisticated user-interface to a complex interactive system can in principle be supported by the abstract machine model appears in [4]. Notice in particular how the natural association of clusters of definitions and actions motivates the concept of entity. For instance, to a generic object there may correspond the definitions needed to specify its internal structure, and the actions required to monitor or impose associated constraints upon this structure. Declaring an instance of a generic object then corresponds to creating an entity instance. There are also indications that other higher-level abstractions may be helpful in programming the abstract machine: for instance, it is useful to be able to specify a guard of the form "**changed**(x)" that has the value **true** provided that the value of x was changed on the previous machine transition.

§4 Alternative models for parallel computation

The abstract machine model we have presented supports parallel computation within the context of a general purpose "definitive programming paradigm". A proper evaluation of our approach is premature at this stage, and we are continuing to investigate both its potential applications and its relationship to other parallel programming models. Research on parallelisation has very clearly indicated the relevance of intrinsic characteristics of algorithmic problems - as for instance in developing algorithms for systolic array architectures. In this context, it may be significant that making effective use of our abstract machine model requires problem-specific programming ingenuity over and above trivial reformulation of an algorithmic problem, viz the identification of an appropriate family of functional relationships. The thesis that parallelisation of computation is connected with the identification of functional relationships is plausible in as much as the maintenance of functional relationships entails synchronised changes in variable values. There are also connections with procedural programming using invariants [8], and with "intelligent views" as introduced in §1 and [4].

Our approach to parallelism may be contrasted with declarative programming paradigms, based on functional or logic programming, with procedural methods of which object-ori-

ented programming is the most sophisticated, and with special-purpose models for parallelism, such as systolic array or data flow architectures.

It has often been argued that functional programming methods are particularly well-suited for parallel programming, on the basis that multiple function evaluations can be efficiently performed in parallel. In its naive form, this thesis fails to reflect the need in general for problem-specific knowledge when developing parallel algorithms. Much recent research in the functional programming arena has focussed on the problems of formulating functional programs for efficient parallel evaluation, perhaps using special-purpose hardware [10].

Parallelism has also had a significant role to play in logic programming. The limitations of interpreters that perform inferences sequentially - as in standard Prolog implementations - are well-recognised, and theorem provers that exploit parallel inference are needed for pure logic programming methods to succeed. The problem of achieving efficient parallelisation of unification is a well-known obstacle for resolution-based theorem provers [6]. It is of interest to observe the close resemblance between the representation of relationships between variables within logical clauses, as commonly introduced in pragmatic logic programming, and the use of definitions in our abstract machine framework [12].

A convincing case for declarative methods as a basis for parallel programming has yet to be made. It is not clear to what extent work on logic and functional programming offers an insight into the characteristics of algorithmic problems that favour efficient parallelisation. In our approach we implicitly assume that a concept of program state is appropriate where interaction between several processing agents is concerned, and our abstract machine model is in this respect more akin to a procedural programming model.

The characteristic problems of procedural programming stem from the need to associate logically related updating actions. Typically, when a procedural action is performed, it is also necessary to attend to the various side-effects. The purpose of using invariants is to make explicit the relationships between variables that ensembles of procedural actions are intended to maintain. The problems of parallel programming within a procedural paradigm are connected with the difficulty of ensuring that the states in which interaction between agents occurs are consistent with respect to these invariants. Although our abstract machine model is non-declarative in that the value assigned to a variable may change, there can nonetheless be a significant distinction between redefinition of variables and procedural assignment. Indeed, provided that the defining expression assigned to variable involves no *evaluation*, a variable redefinition is radically different from traditional reassignment. In principle, the idea behind programming in our abstract machine model is that, in the presence of appropriate definitions, the side-effects of parameter reassignments are handled automatically. The identification of functional relationships between variables referred to above corresponds to the rationalisation of side-effects through the use of definitions. The successful use of our programming paradigm must correspond to a disciplined use of procedural programming resembling programming with invariants in many respects.

The object-oriented programming (OOP) paradigm ostensibly offers an improvement over traditional procedural programming where parallelism is concerned. Each object in an OOP environment can readily be programmed in such a way as to maintain internal

consistency, and the maintenance of relationships between objects can be ensured by appropriate message passing. There are several problems in formally interpreting such a style of programming [1]. There is no explicit expression of the intended relationships between objects; these relationships are simply determined as consequences of the particular methods invoked within objects and message passing protocols employed. Effective use of the OOP paradigm requires discretion on the part of the programmer (e.g. to eliminate infinite behaviour in connection with the maintenance of relationships) and perhaps obscure assumptions about the implementation (e.g. the form of the relationships established may depend upon how message passing is synchronised) [1]. Other difficulties arise in connection with maintaining consistent variable references when objects are dynamically invoked and removed, and through the maintenance of relationships by repeated re-evaluation. As we have explained in §2, many of these problems are no less relevant within our programming paradigm, but may - we believe - be more effectively addressed within the context of our abstract machine model. For instance, it becomes important to distinguish between relationships between variables that can be expressed by an acyclic system of definitions from those that require auxiliary actions. In this way, programming for our abstract machine model may be related to guaranteeing appropriate synchronisation within an OOP model.

To show the versatility of our machine model, we outline the simulation of a standard systolic array algorithm. Our example is that of banded matrix multiplication, as described in [10]. The input to the algorithm consists of two banded n×n matrices A and B, where $A = (a_{ij})$, $B = (b_{jk})$, and $a_{ij} = 0$ unless $-p_A \le i-j \le q_A$, and $b_{jk} = 0$ unless $-p_B \le j-k \le q_B$. The output is the banded matrix $C = (c_{ik})$ where C=A·B, and $c_{ik} = 0$ unless $-(p_A + p_B) \le i-k \le q_A + q_B$. Following [10], we construct a hexagonal array of processors, indexed by pairs (α,β), where $-p_A \le \alpha \le q_A$, and $-q_B \le \beta \le p_B$, and pipe the entries of the matrices A and B, and the partially computed entries of the matrix C through the array (see Figure 1). A simple analysis shows that the processor indexed by (α,β) operates at time t only if t is congruent to $(\alpha+\beta)$ modulo 3, when it performs the assignment c := c + a·b, where a, b and c are the current values stored in its three registers. These three values are respectively: the inputs a_{ij}, b_{jk} and the partially computed output c_{ik}, where

$$i \equiv i_{\alpha\beta} = (t+2\alpha-\beta)/3,\ j \equiv j_{\alpha\beta} = (t-\alpha-\beta)/3 \text{ and } k \equiv k_{\alpha\beta} = (t+2\beta-\alpha)/3.$$

To interpret the above construction within our abstract machine model, we must first instantiate a system of variables a[i,j], b[j,k] and c[i,k], subject to the initial conditions:

$$a[i,j] = a_{ij},\ b[j,k] = b_{jk} \text{ and } c[i,k] = 0.$$

(This corresponds to instantiating a single entity, comprising a system of definitions, with no associated actions.) Clocking in the systolic array is simulated by a clock entity that comprises a single variable t together with a single action:

not changed(t) **and** $t \ge 3n+\max(\min(p_A,q_B)+\min(p_B,q_A)\ ,\ p_A+q_B)+1 \rightarrow t = t+1.$

We can then represent each processor by an entity in the abstract machine model that incorporates the definitions of the three indices $i_{\alpha\beta}$, $j_{\alpha\beta}$ and $k_{\alpha\beta}$ above, together with a single action, viz.

changed(t) **and** $t \equiv (\alpha+\beta) \bmod 3 \rightarrow c[|i_{\alpha\beta}|, |k_{\alpha\beta}|] = |\ c[|i_{\alpha\beta}|, |k_{\alpha\beta}|]\ | + a[|i_{\alpha\beta}|, |j_{\alpha\beta}|]\cdot b[|j_{\alpha\beta}|, |k_{\alpha\beta}|].$

The use of the construction **changed**(t) in the guard ensures that incrementing the clock does not interfere with the evaluation of the indices in the associated assignment. In effect, successive clock cycles are associated with shifting the data through the array and

processing.

§5 Future research

The abstract machine model we have described is as yet little explored. Our present purpose is to use the model as the basis for a programming language with the practical advantages of EDEN that has a more satisfactory semantics. There are several unresolved issues at present concerning the design of entities, and the extent to which consistent variable referencing can be guaranteed through their disciplined use. Work is currently in progress on the design of a CAD system, and of an LSD simulation, to be implemented with reference to our machine model.

It is of particular interest to consider how our abstract machine model might be used to solve traditional algorithmic problems with parallelism. An example of an abstract algorithm that incorporates relevant ideas appears in [7]. We are also investigating ways in which other parallel machine models, such as data-flow machines, can be represented in our framework.

References

[1] P America *A proof theory for a sequential version of POOL,* ESPRIT 415A Doc. #188, Philips Res Lab, 1986
[2] W M Beynon *The LSD notation for communicating systems*, CS RR#87, Warwick Univ, 1986
[3] W M Beynon, Y W Yung, *Implementing a definitive notation for interactive graphics*, New Trends in Computer Graphics, ed N Magenat-Thalman, D Thalman, Springer-Verlag 1988, 456-68
[4] W M Beynon, A J Cartwright *A definitive framework for implementing intelligent CAD systems*, in Proc 2nd Eurographics Workshop on Intelligent CAD Systems 1988 (to appear)
[5] W M Beynon, M T Norris, M D Slade *Definitions for Modelling and Simulation of Concurrent Systems*, in Proc IASTED Applied Simulation and Modelling Conference 1988 (to appear)
[6] C Dwork, P Kanellakis, J Mitchell *On the sequential nature of unification*, Journal of Logic Programming (1984) 1, 35-50
[7] A M Gibbons, W Rytter *Optimal Parallel Algorithms for Dynamic Expression Evaluation and Context-free Recognition*, Information and Computation (to appear)
[8] D Gries *The Science of Programming*, Springer-Verlag,1981
[9] T Maibaum et al *A Logic for the Formal Requirements Specification of Real-Time Embedded Systems*, Alvey Project SE 015, Report R3.
[10] C Mead, L Conway *Introduction to VLSI Systems* , Addison-Wesley, 1980
[11] S Peyton-Jones *The implementation of functional programming languages*, Prentice-Hall, London 1987
[12] M Y Rafiq, I A McLeod *Logic Programming for Technical Design*, Workshop on AI in Civil Engineering, Edinburgh, November 1987

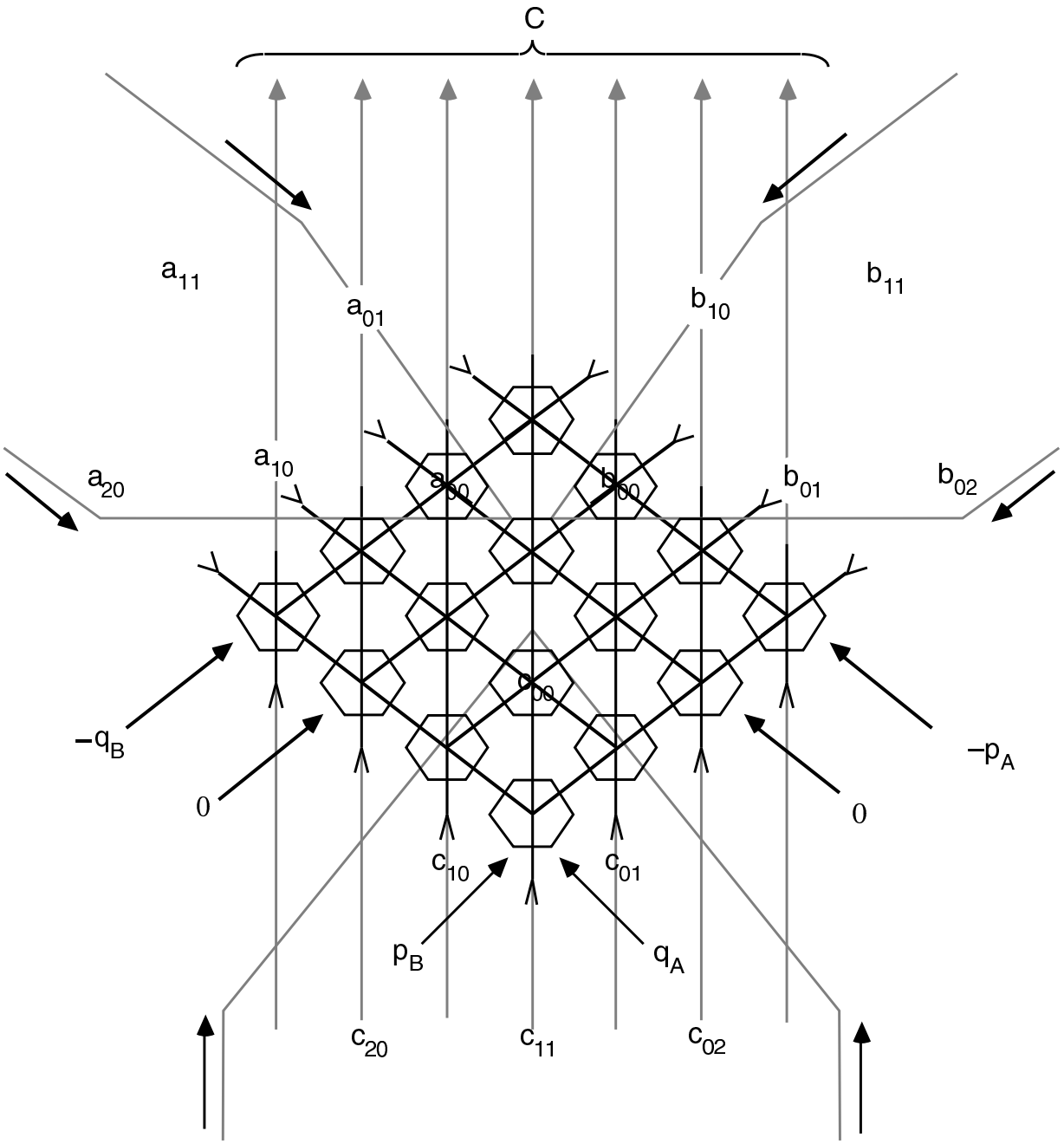
C
a11
a01
b10
b11
a20
a10
a00
b00
b01
b02
c00
−qB
0
−pA
0
c10
c01
pB
qA
c20
c11
c02

Figure 1