REPRINTED FROM:

PARALLEL COMPUTING 89

Proceedings of the International Conference
Leiden, 29 August - 1 September, 1989

Edited by

David J. Evans
Department of Computer Science
University of Technology
Loughborough, England

Gerhard R. Joubert
Corporate CAD Centre, Philips
Eindhoven, The Netherlands

Frans J. Peters
Corporate CAD Centre, Philips
Eindhoven, The Netherlands

1990

NORTH-HOLLAND
AMSTERDAM • NEW YORK • OXFORD • TOKYO

Parallel Computing 89
D.J. Evans, G.R. Joubert, F.J. Peters (Editors)
© Elsevier Science Publishers B.V. (North-Holland), 1990

PARALLELISM IN A DEFINITIVE PROGRAMMING FRAMEWORK

Meurig BEYNON

Department of Computer Science, University of Warwick,
Coventry CV4 7AL, United Kingdom

A method of programming based on the use of definitions is outlined and illustrated. Its potential merits as a medium for general-purpose parallel programming are examined with reference to criticisms of approaches based upon traditional programming paradigms.

1. INTRODUCTION

The problems of supporting general-purpose parallel programming are well-recognised. As the Parallel Architecture and Languages Europe project illustrates [11], many different programming styles have been advocated for parallelism, but the most appropriate choice - if such exists - remains controversial. An analysis of existing paradigms [2] suggests that multiprocessors cannot be programmed effectively without radical developments in programming language design. This paper discusses the application of a definition-based computational paradigm ("definitive programming") to general-purpose parallel computation.

2. DEFINITIVE PROGRAMMING

The key idea behind definitive programming is the representation of computational state by a set of definitions of variables and of a transition between states by a set of redefinitions. A simple application of this principle underlies the spreadsheet. By way of illustration (when augmented by definitions of voltage etc) the set of definitions:

```
  resistance = resistance_of_lamp + cable_length * coefft_of_resistance
     current = if switch_on then voltage / resistance else 0
    light_on = switch_on and current >= threshold
   switch_on = FALSE
```

can be interpreted as describing the state of a simple electrical circuit. In this context, an appropriate transition might involve the redefinition

```
   switch_on  = TRUE
```

or the redefinition of a parameter such as the length of the cable or the voltage.

The exploitation of this method of specifying states and transitions in general-purpose computation has been the focus of extensive research by the author over several years ([3,4,5,6,7]). The need to address applications that require sophisticated data representation motivates the introduction of complex data types and operators in definitions. Previous work has addressed the design of prototype definitive systems for interactive graphics requiring data types such as points, lines and shapes composed of families of points and lines, as in [4]. In such systems, the representation of state by a set of definitions provides a "generalised spreadsheet" well-adapted for a simple form of user-computer interaction in which all state changes are initiated by the user.

Generalising definitive principles to describe less restricted forms of computation leads directly to the consideration of concurrent action by several agents - in the first instance, the user and the computer - within the framework of a common set of definitions. A set of definitions is a powerful way to represent computational state, but arbitrary redefinition of variables is not in general an appropriate way to describe computationally useful transitions. A suitable state-transition model of computation is developed by regarding the user as a prototypical *agent* who carries out transitions from one computational state to another subject to certain privileges and protocols. The typical user of an electrical circuit can reset a switch or change a light bulb; an electrician can alter the length or electrical characteristics of a cable; the definition of the circuit current in terms of the voltage and resistance is invariant. The particular transitions available to the user may also depend upon the context, as when the switch is operated by a key. The roles of agents in a computation have to be circumscribed in an analogous way.

The *abstract definitive machine* (ADM) is an appropriate machine model within which to express context-sensitive parallel redefinition (cf.[3]). In the ADM, the computational state is represented by a set of definitions D that is dynamically modified through redefinition of variables and the creation or deletion of definitions. The transitions to be performed in executing an ADM program are specified by a set of guarded actions A to be executed in parallel as and when the guards allow. Each action is a sequence of instructions that either redefines a variable, or leads to the instantiation or deletion of an entity comprising a set of definitions and actions. The ADM outputs by redefining variables whose values model the state of an output device.

An ADM program consists of a set of abstractly specified entities. Execution is initiated by instantiating appropriate entities. On each machine cycle the guards associated with actions in A are evaluated in the context specified by the definitions in D. If there is no interference, those actions that are associated with true guards are then executed in parallel. Evaluation required in a redefinition - as in "fixing the exchange rate" for purposes of a currency transaction - is performed in the same context as guard evaluation. Autonomous computation terminates when no action in A has a true guard.

Actions can interfere in several ways. The same variable may be redefined independently in concurrent actions, or the set of parallel redefinitions may introduce cyclic dependency. Such interference is detected during computation and the execution is suspended. In one possible mode of execution of the ADM, the programmer can act as an auxiliary agent to resolve conflicts as they arise. A similar technique can be used to handle input. These issues will be illustrated with reference to an ADM program to simulate a simple concurrent system.

3. AN ILLUSTRATIVE EXAMPLE

Suppose that the blocks L and R are under the independent control of two agents. For simplicity, assume that the blocks move in 1-dimension, have unit length, are centred at integral points pL and pR and are always moved by steps of 1 unit in discrete actions. Assume also that an inelastic string of integral length d-1 connects L and R.

An outline of the ADM simulation program appears in Figure 1. (The annotations on the right are mnemonic labels for actions.) The given skeleton must be complemented by adding a control() entity that provides the correct synchronisation between actions. The actions of the handler() entities for instance, must be sequential: since actions [^], [<] and [>] are simultaneously enabled, these must be made mutually exclusive. A simple method to ensure this is to generate an element from the set {< , ^ , >} at random within the control() entity, and to select the appropriate action accordingly. In addition, unless a suitable control mechanism is introduced, the definition invoked by an action such as [?][<] persists, as though the blocks became glued together on touching.

Only some of the interference between actions of the blockmover() entity has been resolved in the program. A static analysis shows that at most two actions of the blockmover() entity can be performed in each execution cycle: at most one from each of the sets {[<]~, [<]--, [>][?], [>]..} and {~[>], --[>], [?][<], ..[<]}. Certain combinations of action are impossible - e.g. the enabling conditions for [<]-- and ~[<] are incompatible. Actions [<]-- and --[>] interfere on parallel execution: they correspond to a situation in which the string is taut and the handlers are pulling in opposite directions. There is a conflict between actions [<]-- and ..[<] in so far as concurrent action is only possible because actions specify movement through the same distance. The actions [>].. and ..[<] are in conflict when this entails a collision of the blocks at a single location.

The possible patterns of singular behaviour are summarised in Figure 2. Most will be dynamically detected as instances of interference. For instance, the actions [<]-- and --[<] interfere; they invoke an inconsistent system of definitions if executed in parallel. The conflict between the actions [>].. and ..[<] that arises when pR-pL=2 is not detected as interference; in another model, co-location of blocks might be possible. The way in which the conflict between the actions [<]~ and ~[>] when pR-pL = d-1 is resolved in the model illustrates one possible method for resolving exceptional behaviour. The conflict arising from the parallel execution of [<]-- and --[>] could be resolved by permitting no movement, by allowing one handler to dominate the other - whether arbitrarily or otherwise, or by deeming that the string snap.

No output has been specified in Figure 1. The most appropriate form of output is a graphical animation that depicts the position of the blocks and the status of handlers throughout the simulation. Such an animation can be programmed within the ADM using a definitive notation for line-drawing within which parametrised figures composed of points and lines can be specified by a set of definitions. For instance, block L can be depicted by adding a set of definitions to D that describes a square centred at a point with coordinates parametrised by pL. Details of such a specification are given in [4].

```
entity handler(block)
{
    definition
        driving[block] = drivingL[block] or drivingR[block],
        drivingL[block] = holding[block] and pushingL[block],
        drivingR[block] = holding[block] and pushingR[block],
        pushingL[block] = false,
        pushingR[block] = false,
        holding[block] = false

    action
        not holding[block] -> holding[block] = true,
        holding[block] and not driving[block] -> holding[block] = false,     [^]
        holding[block] and not driving[block] -> pushingL[block] = true,     [<]
        holding[block] and not driving[block] -> pushingR[block] = true,     [>]
        drivingL[block] -> pushingL[block] = false,
        drivingR[block] -> pushingR[block] = false
}

entity blockstate()
{
    definition
        pL, pR, d,
        stringtaut = not stringsnap and (pR-pL)==d,
        touching = (pR-pL)==1,
        stringsnap = false

    action
        not stringsnap and (pR-pL)>d -> stringsnap = true
}

entity blockmover(blockL, blockR)
{
    action
        drivingL[blockL] and not stringtaut -> pL = |pL|-1,                  [<]~
        drivingL[blockL] and stringtaut -> pR = pL+d; pL = |pL|-1,           [<]--
        drivingR[blockR] and not stringtaut -> pR = |pR|+1,                  ~[>]
        drivingR[blockR] and stringtaut -> pL = pR-d; pR = |pR|+1,           --[>]
        drivingR[blockL] and not touching -> pL = |pL|+1,                    [>]..
        drivingR[blockL] and touching -> pR = pL+1; pL = |pL|+1,             [>][?]
        drivingL[blockR] and not touching -> pR = |pR|-1,                    ..[<]
        drivingL[blockR] and touching -> pL = pR-1; pR = |pR|-1              [?][<]
}

blockstate(); blockmover(L,R); handler(L); handler(R)
```

FIGURE 1: Blocks - a skeleton ADM program for the block moving simulation

[<]----[>]	String under tension: conflict to be resolved
[>][<]	Agents pushing against each other: conflict to be resolved
[<]----[<]	Conflicts unless the agents cooperate ([>]----[>] , [<][<] , [>][>] are similar)
[<]~~[>]	String can snap - and will under this model - if pR-pL = d-1
[>]....[<]	Blocks collide if pR-pL = 2
[<]~~[<]	Never generates interference ([>]~~[>] is similar)

FIGURE 2: An analysis of interference and anomalous behaviour

4. DEFINITIVE GENERAL-PURPOSE PARALLEL PROGRAMMING?

In the ADM, states and transitions are explicitly described. The description of a complex state-transition system is made tractable by using definitions to represent that part of the computation concerned with maintaining the relationships that characterise a transition (cf [7]). This is in contrast to the use of constraints or logical predicates for specifying information about the current state. For instance, in Blocks, the definition pR=pL+d is invoked when the string is taut if block L is driven to the left, but is inappropriate if block L is driven to the right. The subtlety of such a representation is best appreciated by considering variants of Blocks where the interconnection is (e.g.) an elastic string, a rigid rod or an umbrella handle. Comparison with [9] - an application of "qualitative reasoning" methods to a similar simulation problem - is also instructive.

Conventional computer programs typically use what is in essence a batch-processing paradigm. The programmer specifies the sequence of transitions that the computer is to perform, making provision for the user to supply input in a preconceived fashion. If an exceptional state is encountered during program execution, the program is edited and re-compiled. The ADM can be programmed for autonomous computation of this kind, but may also execute in such a way that the user is privileged to contribute actions at each machine cycle, cooperating directly with other agents - e.g. the preprogrammed computer - in directing the computation. Dynamic resolution of conflicts in Blocks is an application of this principle; yet more forceful user intervention can be viewed as blurring the distinction between the programmer and the user (cf. the design of software for modelling and animation in [8]).

The ADM was derived from research into implementing interactive definitive systems [6] and modelling and simulating concurrent systems [5]. Its wider application requires techniques for identifying and specifying the role of computational agents. The Blocks program illustrates a subtle model for parallel action, but does not realistically simulate block-handlers that act asynchronously, move blocks at different relative speeds and may observe protocols to avoid conflict (e.g. a block can move at most one unit distance whilst the other is released). This problem has been addressed by developing methods of specifying agent privileges and protocols independent of timing considerations [5]. A derived ADM simulation program can then express relative timing of agent actions as determined by communication ("the sprinter starts when the pistol is fired"), by speed of execution ("the gas ignites before the match can be extinguished"), or by definition ("the article is purchased when the document is signed"). Future research will examine the application of analogous techniques at a lower level of abstraction to the implementation of abstract algorithms on parallel architectures.

Baldwin [2] studies parallelisation in programming paradigms. He identifies "two deep flaws of existing languages: 1) reliance on side-effects, 2) use of iteration or recursion to express data parallelism ..." and describes "the ultimate goal for a parallel programming language" as "supporting a clear statement of the data dependencies".

Data dependency has a key role in definitive programming. A set of definitions expresses the dependencies between variable values explicitly; its use in representing states and transitions entails the rationalisation of side-effects. As scripts in functional programming environments [10] indicate, sets of definitions can have such power to represent states and transitions that complex procedural abstractions are unnecessary.

A potential source of interference in parallel procedural programming is "has this variable currently an appropriate value?" and in parallel evaluation in a declarative programming environment "is this variable currently defined?" [2]. Definitive principles alleviate these problems, since the order in which a set of definitions is introduced is immaterial and undefined values can be gracefully accommodated.

5. CONCLUDING REMARKS

Definitive programming has significant connections with several programming paradigms that have been studied in connection with parallel architectures. In many respects, it seems likely to meet the criteria for an appropriate programming medium for multiprocessors, as identified by Baldwin in [2]. Its combination of procedural and declarative features indicate good prospects for application at different levels of abstraction, potentially bridging the gap between application-oriented and architecture-oriented language concerns. In due course, it is to be hoped that it will contribute towards the objectives set out by Backus in [1], perhaps ultimately providing the foundation for alternative parallel computer architectures.

ACKNOWLEDGMENTS

I am much indebted to Mark Norris, Mike Slade and Edward Yung for significant contributions to the design and implementation of the abstract definitive machine.

REFERENCES

[1] Backus J, Can programming be liberated from the von Neumann style?, Comm ACM 21, 8 (August) 1978, 613-641
[2] Baldwin D, Why we can't program multiprocessors the way we're trying to do it now, CS Tech Rep 224, University of Rochester 1987
[3] Beynon W M, M D Slade, Y W Yung, Parallel computation in definitive models, CONPAR'88, British Computer Society Workshop Series CUP 1989, 359-367
[4] Beynon W M, Evaluating definitive principles for interactive graphics, New Advances in Computer Graphics, Springer-Verlag 1989, 291-303
[5] Beynon W M, Norris M T, Slade M D, Definitions for modelling and simulating concurrent systems, Proc IASTED conference ASM'88, Acta Press 1988, 94-98
[6] Beynon W M, Cartwright A J, A definitive programming approach to the implementation of CAD software, Intelligent CAD Systems II: Implementation Issues, Springer-Verlag 1989, 126-145
[7] Beynon W M, Norris M T, Slade M D, Yung Y P, Yung Y W, Software construction using definitions: an illustrative example, CS RR#147, University of Warwick 1989
[8] Chmilar M, Wyvill B, A software architecture for integrated modelling and animation, New Advances in Computer Graphics, Spinger-Verlag 1989, 257-276
[9] Ginsberg M L, Smith D E, Reasoning about Action I & II Artificial Intelligence 35, 1988, 165-195 & 311-342
[10] MIRANDA System Manual, Research Software Ltd 1987
[11] Parallel Architectures and Languages Europe Vol 1, Lecture Notes in Computer Science 258, Springer-Verlag 1987