MACNET: a language for situated AI systems

Howarth, Richard John

QMW Technical Report 684

MACNET: a language for situated AI systems

Richard Howarth
Department of Computer Science, Queen Mary and Westfield College, Mile End Road, London E1 4NS, UK.
howarth@dcs.qmw.ac.uk

Abstract

MACNET is the name of a language developed by Agre and Chapman as part of their programs PENGI [2], BLOCKHEAD [7], and SONJA [8]. Here we explore some of the background that lies behind this language and address some of the problems encountered while re-implementing MACNET. The re-implementation necessitated finding out more about how MACNET really works, the results of which are given here.

1 Introduction

In this paper we describe one interpretation of the MACNET language designed by Agre and Chapman [1, 2, 7, 8] that is used for specifying routines in terms of loosely connected rules. The description given by Agre and Chapman does not fully explain how MACNET can be implemented and the purpose of this paper is to fill in some of the details, so that the performance of MACNET can be understood. The descriptions given here may not be identical to those of Agre and Chapman because, although Chapman [8] provides a syntax and a semantics, given in an informal manner with examples, there is ambiguity in how what is described might be implemented and what it means.

MACNET is on the one hand intuitively simple and on the other surprisingly complex to specify and implement, requiring a much larger amount of code than initial considerations would envisage. To begin with, one useful "coarse" interpretation is to think of MACNET as being a language for expressing an expert system's set of rules together with a built in conflict resolution strategy. Notice we have two things here: the specification of the rules and its run-time execution.

Agre and Chapman's work is frequently referenced and in some respects their PENGI paper [2] is important for initiating the "reactive planning" field. However, the operation of the underlying system (i.e., MACNET) is not always understood, for example, see Vera and Simon [39]. This lack of understanding illustrates the need for some attempt at clarifying the MACNET approach. This description advances the current state of understanding by making more accessible the underlying foundations of an AI approach based upon deictic representation, and which can be used for developing situated AI systems (such as the HIVIS-WATCHER system described in Howarth [17, 18]).

This paper begins with a discussion of background details providing the context for the MACNET language. Following this, general details of the execution form called GATE and the specification form called MACNET are given.

2 Background

The motivation behind the development of MACNET is to provide a knowledge representation with which to describe the behaviour of a prototypical everyday agent. In this section we begin by describing how routines, those repreated elements of everyday behaviour, provide a basis for developing the agent model. Following this we consider the various properties of a computational model that are used to support routines.

2.1 Routines

MACNET is used to describe the rules performed by an agent who is engaged in some activity that is situated in the environment, for example, drinking coffee, driving home, etc. We are making the assumption that a system's behaviour and its understanding of the world can be defined by the program designer (see Dreyfus [11, page *xxxiii*] for discussion of this problem). Also, more details about situated activity are given by Norman [28] in his introduction to a recent special issue of the journal Cognitive Science. Basically it provides a link between social science and AI which is important in research concerning how an agent interprets the world. Winograd and Flores [41, pages 27–37] describe how the interpretation of the perceiver is not neutral, it is fundamentally social. The activities of the agents are not planned out in detail, instead they are in a state of what Heidegger calls "thrownness" (for details see [41]). When interacting it is not possible to step back, reflect and plan. This has been investigated by Suchman [35] who identifies plans as something that can evolve out of situated activity, so that previous experience can be used to structure future activity. This distinction between thrownness and planning is similar to the difference between deictic temporal representation (e.g., previous, now, next) and McDermott's [23] useful observation that in most AI models of temporal reasoning we reason from the side, taking a step back and representing the past and the future, which allows us to give names to time points, such as dates, and measure temporal durations. van Benthem [38, pages 6–7] discusses why deictic representation is not suitable for temporal reasoning, in summary, this is because the policy being studied is typically to formalise only those notions about which questions are to be answered and for which results are to be proven. This does not mean that deictic representation cannot be formalised, for example, Subramanian and Woodfill [34] discuss one approach that converts deictic-forms to the situation calculus [22]. One advantage of using a deictic representation is that it allows a propositional theory to be developed that is proportional to the number of properties of interest (e.g., previous, now, next), as opposed to the number of propositional objects in the world (e.g., all time points). Deictic representation is an important component of modelling routine behaviour and the MACNET language.

As identified by Winograd and Flores, an important influence upon our understanding of the environment and the use to which it is put comes from our social context. Garfinkel (see [14] and also Heritage [16]) has performed experiments that demonstrate the presence of "normal" behaviour or "maxims of conduct", which help keep an agent's perceivedly normal conduct "on the rails" because it enables the agent to anticipate some of the interpretations that its exercise of the options will give rise. Garfinkel's work provides a framework in which to describe routine and situated behaviour. Most activity is routine in nature, being that regular, practiced and unproblematic activity that makes up most of everyday life. We can approximate this by using MACNET to describe simple local models that express the normal or typical routine behaviour of our system that is embedded in the social world.

2.2 Embedded systems

An agent is an example of an embedded reasoning system, i.e., one that is situated in the world and which operates effectively given the real-time constraints of its environment. Georgeff and

Ingrand [15] describe an embedded reasoning system called the Procedural Reasoning System (PRS) which uses means-ends reasoning to govern future behaviour. PRS explicitly represents attitudes of belief, desire and intention, allowing them to be manipulated and reasoned about, providing complex goal-directed and reflexive behaviour. The system consists of a database holding current beliefs and facts about the world, a set of current goals to be realised, a set of procedures or rules, and an interpreter for manipulating these components. At any one moment, the system also has a process-stack containing all currently active plans, which can be viewed as the system's current intentions for achieving its goals or reacting to some observed situation. The rules describe how certain sequences of action and tests may be performed to achieve given goals, how to react to particular situations, and also includes meta-level knowledge that enables manipulation of the system's own beliefs, desires and intentions.

Although PRS is embedded in the world it is not really situated in the sense of thrownness. This is because PRS uses means-end analysis which involves stepping back, reflecting and planning. Agre's [1] program RUNNING ARGUMENTS provides a more situated approach. This technique is difficult to describe because there are at least two intertwined theories at its core. The first is to do with planning, which we describe here, and the second is about the separation of program components, which we describe below in section 2.3. The RUNNING ARGUMENTS technique does not develop plans as such, although they do exist in the form of hardwired "action-descriptions". These action-descriptions are written in MACNET[1] and expresses what the program is to do given the data of the current and previous clock tick. The rule based form used to define the action-descriptions makes a comparison to the standard production rule form inevitable. The main difference is in how conflict resolution[2] is addressed. In the RUNNING ARGUMENTS system conflict only occurs when two or more "proposals" try and "fire" the same operator, and any occurrence is resolved by assigning rule precedence to the rule definitions. This approach allows a number of operators to be fired on each clock tick as opposed to the usual one per clock tick in production systems. This language is not suitable for planning, but can be used to describe routines (those plan like elements introduced in section 2.1).

Rosenschein and Kaelbing [32, 33] present a more complex representation language called REX that is similar to MACNET in that it too is compiled to provide a combinatorial logic circuit. The REX language is attractive because it is based on a formal logic that is similar to that of Moore [25], however, the implementation details provided in [33] are difficult to understand. In [32], Rosenschein describes how an additional layer of compilation can be added to enable proof correctness to be performed upon the supplied rule specification. This involves re-expressing the rules as clauses in a new language $\mathcal{L}$. The rules are now generated as a side-effect of performing a proof analysis on the clauses written in $\mathcal{L}$, with this proof analysis indicating the completeness of the specification, at the cost of an additional layer of compilation.

Nilsson [27] describes an alternative to MACNET and REX called "teleo-reactive" (T-R) programs, which like the original version of the RUNNING ARGUMENTS program [1] delays constructing the required circuitry until it is needed.

MACNET, REX and T-R are all more appropriate than PRS for developing a situated AI system because they do not use means end analysis. Next we investigate an alternative architecture to means end analysis which also provides the second part of RUNNING ARGUMENTS.

---

[1]The language used by RUNNING ARGUMENTS is really a forerunner of MACNET. We discuss the differences in section 4.

[2]See Charniak and McDermott [9, pages 439-440] for a description of how conflict resolution operates.

2.3 Modularity

Fodor [13] describes the traditional separation made in cognitive science between input-/visual-/peripheral- systems and the central-system (see figure 1). This view is not held by all researchers, for example, Brooks [5, 6] provides a different view that uses an orthogonal separation based on task-achieving behaviours. Agre [1] and Chapman [8] are both proponents of Fodor's input/central split, (see their descriptions for further details). A brief description follows. On the input side we have a collection of perceptual and motor processes, each of which are to a large part innate, localised to specific brain areas, and task- and domain-independent. Each element of this collection is a module of the input-system. Fodor argues that the central side is different, saying it is not modular, being instead a single homogeneous central-system. The justification for this is that anything you know can potentially be used in any cognitive task. Agre uses this split in RUNNING ARGUMENTS, with the central-system holding the rules and the visual-system holding a collection of information gathering operators.

The visual-system is based upon Ullman's argument [37] for the integration of multiple visual operators that perform particular sorts of perceptual work such as tracking, representing shape properties and spatial relations. We discuss this further in sections 2.4 and 2.5 below.

The central-system contains rules of the form described in section 2.2, which are used to select when an operator is to be used and what arguments are to be supplied to an activation. Crafting the rule-operator pairs into sequences (constructing routines) is done by making the result produced by one operator fulfill the input requirement of the next rule. However, this is not the only way a particular rule can be fulfilled, thus allowing the mechanism to react to similar situations that arise via a different route. A routine provides an abstraction for a common pattern of interaction between an agent's central-system and visual-system. This reduces "planning what to do next" to a matter of deciding what to do "now" based upon how the world is "now". Only the operators have access to the "world" data structures and the central-system only receives the results of the operators. This allows the central-system to use a simplified description of the world, that only needs to have the information necessary for making its action-selection. This restricted state ensures that the system can only reason about the current situation.

This separation of input- and central-system and the tight coupling between them provides one possible foundation for the situated approach, allowing us to address control without traditional planning or plan recognition (see Allen et al. [3] for example). Nilsson [27] discusses the importance of the continuous feedback provided by this tight coupling, pointing out why it seems to conflict with sequential execution.

2.4 Operators

As introduced above, the visual-system is composed of a collection of operators.

Definition 1 *An* ***operator*** op_i *is a function of the form*

$$op_i : args \rightarrow results \times side\text{-}effects$$

and is used to obtain the i^{th} *primitive property value.*

In general each operator takes a set of args and provides a set of results and/or side-effects to some global storage.

Previous work on operators also includes: Romanycia's [31] description of a programming language that uses visual operators to commute properties and relations present in 2D images of simple geometric shapes; Mahoney and Ullman's [21] description of low-level visual operators that operate on more complex shapes and curves to identify "image chunks"; and Chapman's [8] description of visual operators used in a video game context. No one has used visual operators in a natural task domain with access to camera image data in the way described by Chapman

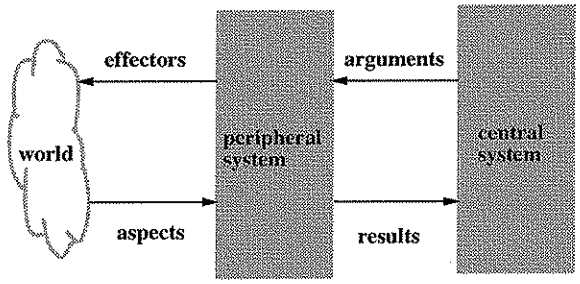

Figure 1: The input central split

(`warp-marker!` *m n doit?*) move marker *m* to location of marker *n* .

(`track!` *m n doit?*) move marker *m* to location of marker *n* and track the object at this location.

(`marker-`*m*`-assigned?`) is true when *m* is unassigned. There is an operator for each marker that inspects the data "under" the marker to see if it is "on" an object.

(`markers-coincident?` *m n doit?*) tells when the distance between two markers is zero.

Figure 2: Example operators.

(such as those which use "activation planes"[3]). However, there is related work, including that on tracking[4], and work on "active contour models"[5]. Chapman's work can be seen as an initial step towards the objective of defining a set of appropriate visual operators for use in real world application domains. On their own, these visual operators do not do much, but combined via rules held in the central-system, enable the system as a whole to respond to changes in the "world".

The collection of operators that we describe here are not able to store information, although they can change external global memory by side-effect. To perform perceptual reasoning deictic references are used called "markers". In the central-system associated with each marker is a set of rules that select which operators to run on the object indexed by that marker. Different markers may use different operators which obtain different object properties. This provides an example of how expectation can affect interpretation. Some example operators are given in figure 2. In addition to markers, Chapman [8] describes other primitive elements, such as, lines, rays and activation-planes, and has operators for things like: follow a line; shade a region; pick out the red bit and put a marker on it; and tell whether the red bit is moving.

2.5 *Peripheral-system*

The peripheral-system contains the operator processor (OP) which is a collection of operators (see definition 1) that are all given arguments and executed in parallel (i.e., there is no dependence on sequential execution). We will call this "OP-execution". Each operator performs a simple operation that in itself does not do much, so that all the operators in OP should complete quite quickly. The usefulness of the OP becomes apparent when it is coupled to another system that repeatedly and sequentially performs OP-executions, with the operators in OP given new arguments for each OP-execution.

As described in definition 1 the general form of each operator in OP is $(\mathtt{op}_i \ arg_1 \ldots arg_n)$, where $\mathtt{op}_i$ and arg_j denote domain relevant symbol names. The last argument position can be given a special meaning if it has the name `doit?` which denotes a boolean flag. When present, it states that $\mathtt{op}_i$ is selectable and is only to be executed when `doit?` is set to `true`. When the `doit?` flag is not present as the last argument then $\mathtt{op}_i$ is always run on each OP-execution.

[3] Activation planes are used to keep track of interesting regions of the image, as in Ullman's [37] routine for computing containment.

[4] See for example Murray et al. [26].

[5] Active contour models have been used to define outlines (Kass et al. [20], Cohen [10]) and dynamic regions distinguished by a particular visual property, e.g., texture and/or colour (Ivins and Porrill [19]).

The operator $\texttt{op}_i$ is defined by a self contained set of instructions, and does not return a result. Instead $\texttt{op}_i$ side-effects the execution environment by setting specially allocated memory addresses as well as common areas of memory. Operators use a specially allocated memory, called "aspects", where each selectable operator typically has two addresses called "$\texttt{*op}_i\texttt{*}$" and "$\texttt{*register-op}_i\texttt{*}$". The address called "$\texttt{*op}_i\texttt{*}$" is given the return value from operator $\texttt{op}_i$'s execution, usually an integer or a boolean, and "$\texttt{*register-op}_i\texttt{*}$" is set to say that "$\texttt{*op}_i\texttt{*}$" has been given a value. In a truly parallel implementation a "$\texttt{*op}_i\texttt{-ready*}$" would also be needed to say when the operator has completed. The set of aspects from the peripheral-system have an injective (one-to-one) mapping to the set of result wires registered by the central-system (this can be described by the function B : $\texttt{aspects} \mapsto \texttt{wires}$). There is also a corresponding mapping from central-system effector-commands to peripheral-system operators.

These operators do not form a nice functional language because of their use of side-effects, which can also make them difficult to define and debug. The OP-execution environment is in effect shared by all the operators with this environment passed onto future executions and leaving a thread of environments as the OP-execution history. As discussed in section 2.3 we need this continuous flow of execution to allow situation driven processing.

There are two constraints upon the description of operators. The first constraint concerns the amount of computation that needs to be performed by each OP-execution. This is dependent upon the selected rate chosen for the clock-ticks, which like a metronome "marks time". If we hold the total amount of computation performed by the system constant, then the amount of computation that needs to be performed for each OP-execution is proportional to the time between clock ticks. A large temporal interval between clock-ticks means that more computation needs to be performed by the operators and more assumptions and interpolations made about the data. The second constraint is the parsimony of having a small set of operators that can act upon the available data.

The solution to these constraints is application domain specific, however, in general, if we minimise the time between clock-ticks we can help reduce the number of operators by encapsulating common functionality in an often used operator. This is rather like the idea behind risc chips.

2.6 Summary

This has covered the general principles that lie behind MACNET. We have covered: routines, which is what MACNET is used to express; RUNNING ARGUMENTS, the initial system methodology from which MACNET was extracted; the input-/central-system split, the role each plays, and the position of MACNET within this system. Next we provide a more detailed description of MACNET starting with a discussion of the basic language primitives and then building up to the full version of the language.

3 GATE language

The GATE description language provides the user with a collection of functions for constructing various data-structures that can be recursively combined with one data-structure providing an argument for another. Figure 3 provides an example of an adder, showing how sub-circuit forms can be assigned to variables that are used in more than one place. The important gates in this example are `andg`, `org` and `invert`, which correspond to the usual combinatorial logic meanings of $\wedge$, $\vee$ and $\neg$ respectively. The general GATE language syntax is given by tuples of the form $(\texttt{foog}\ arg_1 \ldots arg_n)$ which defines the gate `foo`. Each gate takes one or more input wires depending upon the gate's meaning and all gates have a single output wire. Figures 4 and 5 express the adder code in pictorial terms, making clearer the relationships between the components and sub-circuits. In figure 5 the labels `A` and `B` represent the binary

```
(defun make-adder (*a* *b* *cin* *cout* *sum*)
  (labels
   ((halfadder (in1 in2 out1 out2)
               (let ((x (andg in1 in2)))
                    (sequ (set-wire! out1 (andg (invert x) (org in1 in2)))
                          (set-wire! out2 x)))))
   (let ((w1 (gentemp "WIRE")) (w2 (gentemp "WIRE"))
         (w3 (gentemp "WIRE")) (w4 (gentemp "WIRE")))
        (sequ (halfadder (interface-node *b*) (interface-node *cin*) w1 w2)
              (halfadder (interface-node *a*) (interface-wire w1) w3 w4)
              (set-node! *sum* (interface-wire w3))
              (set-node! *cout* (org (interface-wire w4)
                                     (interface-wire w2)))))))
```

Figure 3: Adder circuit given in the Lisp implementation.

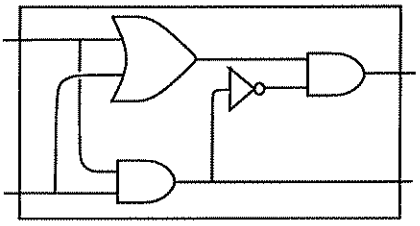

Figure 4: A half-adder unit.

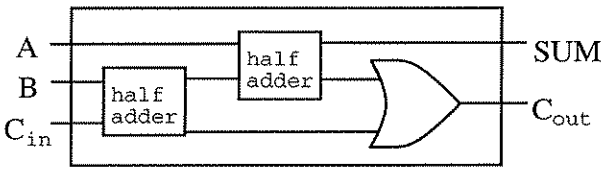

Figure 5: A full adder unit.

bits at corresponding positions in the two numbers to be added, and C_{in} is the carry bit from the addition one place to the right. The circuit generates SUM, which is the sum bit in the corresponding position, and C_{out}, which is the carry bit to be propagated to the left. We call the boxes that enclose a sub-circuit a "unit".

The tree structure that is created by the Lisp functions is shown in figure 6, even for this small example, it is quite large, and not very illuminating. However, it does give some idea of how the gate language can be composed to produce a tree of gates.

3.1 GATE primitives

In addition to the andg, org and invert gates already described above, other gates include (interface-node *input-symbol*) which "reads" the current value held by the specified input symbol. (set-node! *output-symbol input*) which "writes" the value of the input wire to the specified output symbol. (interface-wire *name*) which accesses the named wire, this wire acts like a variable holding its set value during the clock tick. At compile time a check is made to ensure that the wire is set before it is used via the (set-wire! *name input*) gate which sets the named wire to a new value. The jobs performed by these node and wire functions could be combined into one pair of functions, however keeping them distinct clarifies the interval wire operations from the node links of the circuit with the external world. A (latch *input*) gate that provides a one clock tick delay function which is used to supply the value from the previous clock tick and hold the current value. (saym *&rest input*) is used to print warnings and error messages. To allow switching between inputs there are three gate forms based on an ifm primitive. An *if-then* form (ifm *test input*), an *if-then-else* form (ifm *test input1 input2*) and also a condm macro that has the usual Lisp cond syntax and expands into a collection of nested ifm gates. The condm is to make specifying complicated ifm's more understandable and is used in the construction of arbiters (which we describe in section 4). In the adder example

```
#S(SEQU :NAME 3 :INPUT
  ( #S(SEQU :NAME 1 :INPUT
      ( #S(SET-WIRE! :NAME 1 :INPUT
          ( WIRE470
            #S(ANDG :NAME 2 :INPUT
              ( #S(INVERT :NAME 1 :INPUT
                  ( #S(ANDG :NAME 1 :INPUT
                      ( #S(INTERFACE-NODE :NAME 1 :INPUT (*B*))
                        #S(INTERFACE-NODE :NAME 2 :INPUT (*CIN*))))))
                #S(ORG :NAME 1 :INPUT
                  ( #S(INTERFACE-NODE :NAME 1 :INPUT (*B*))
                    #S(INTERFACE-NODE :NAME 2 :INPUT (*CIN*)))))))
      #S(SET-WIRE! :NAME 2 :INPUT
        ( WIRE471
          #S(ANDG :NAME 1 :INPUT
            ( #S(INTERFACE-NODE :NAME 1 :INPUT (*B*))
              #S(INTERFACE-NODE :NAME 2 :INPUT (*CIN*)))))))
    #S(SEQU :NAME 2 :INPUT
      ( #S(SET-WIRE! :NAME 3 :INPUT
          ( WIRE472
            #S(ANDG :NAME 4 :INPUT
              ( #S(INVERT :NAME 2 :INPUT
                  ( #S(ANDG :NAME 3 :INPUT
                      ( #S(INTERFACE-NODE :NAME 3 :INPUT (*A*))
                        #S(INTERFACE-WIRE :NAME 1 :INPUT (WIRE470))))))
                #S(ORG :NAME 2 :INPUT
                  ( #S(INTERFACE-NODE :NAME 3 :INPUT (*A*))
                    #S(INTERFACE-WIRE :NAME 1 :INPUT (WIRE470)))))))
        #S(SET-WIRE! :NAME 4 :INPUT
          ( WIRE473
            #S(ANDG :NAME 3 :INPUT
              ( #S(INTERFACE-NODE :NAME 3 :INPUT (*A*))
                #S(INTERFACE-WIRE :NAME 1 :INPUT (WIRE470)))))))
    #S(SET-NODE! :NAME 1 :INPUT
      ( *SUM*
        #S(INTERFACE-WIRE :NAME 2 :INPUT (WIRE472))))
    #S(SET-NODE! :NAME 2 :INPUT
      ( *COUT*
        #S(ORG :NAME 3 :INPUT
          ( #S(INTERFACE-WIRE :NAME 3 :INPUT (WIRE473))
            #S(INTERFACE-WIRE :NAME 4 :INPUT (WIRE471)))))))
```

Figure 6: The tree structure formed from the adder example. Note that for simplicity the slots for `LEVEL` (and also for `OUTPUT` and `RESULT` from the gates `set-wire!` and set-node! respectively) have been removed, because they are not used until we create the linear form.

we also use (`sequ` *&rest input*) to collect together a sequence of subtrees written in the GATE language. The gate `sequ` acts like a "progn" in Lisp, allowing sequential execution, but does not return any value.

3.2 GATE evaluation

Standard theorem proving techniques are not used to evaluate a circuit, instead it is run by a program based on a Digital LSI Design simulator (Terman [36]). The simulator uses the observation that a circuit of gates forms a tree structure that can be post-order tree-walked to evaluate the values held at each node, starting from the *root* (or final output) gate. This approach is fine for a single network execution however, there is a more efficient way of evaluating

a circuit tree that is to be repeatedly run with new *leaf* (or input gate) values.

This second method separates out from the "evaluation phase" work that can be done as a pre-process, which we will call, "network analysis". Network analysis performs a tree-walk, from the output node, not descending any further when an input node or dead end is reached. During the tree-walk all switches are assumed to be on, since the tree-walk is performed before any node values are calculated. During this tree-walk a linear runtime structure, called the "code-array", is constructed, which holds the gate nodes in post-order, ensuring that when we iterate through the code-array beginning from address 0, all arguments for each gate node are evaluated before they are used.

This constructs our network ready for simulation. A simple approach to running the simulator is to have two node-value arrays; one to hold the current values of each node, the other to collect new values as they are computed. Each node is assigned an index which can be used to access its current value in the first array or to store its new value in the second array. The algorithm for this is:

1. For each input node, set its current-value array entry to the designated input value.
2. Execute the simulation subroutine. This fills the new-value array.
3. Compare the current-value and new-value arrays. If their contents are identical the network has settled and the simulation step is over. Otherwise copy new-value array to old, and goto step 1.

This method can be simplified by taking advantage of the post-order present in code-array. Instead of using two value arrays, we can make do with a single value array, because the values of a node's inputs are calculated before the value of the node itself is calculated. The new algorithm, called Cascade is:

1. Set counter i to 0.
2. If i > length of array, exit.
3. Execute the simulation subroutine for node[i] using its specified input addresses from the value-array (or elsewhere). Put the result in value-array[i].
4. Goto step 2.

Terman [36] describes a more complex version of the Cascade algorithm that can also cope with loops such as would be present in a flip-flop however, the gate language does not need this form of feedback, allowing us to ignore such issues.

3.3 Summary

This outlines an informal description of the GATE language, we present a more formal description is given in Howarth [17] appendix E. Here we have covered the format of the GATE language primitives and described the Cascade algorithm that is used to evaluate a circuit composed from these gates. Next we describe the relationship between the GATE language and MACNET.

4 MACNET language

In this section we describe how the GATE language can be enriched to allow the expression of rules. These rules could be written directly using the GATE language however, this task is likely to be complex due to the number of gates involved. The MACNET language is designed

to make this task easier by providing functionality that replaces repeatedly used collections of gates. The MACNET language is based upon Agre's RUNNING ARGUMENTS which involves putting forward proposals and objections that can support and override one another reflecting the dynamics of the knowledge they represent. The RUNNING ARGUMENT rules themselves take two forms, `if` rules and `unless` rules, that correspond to combinatorial logic with an `if` being an `AND` and `unless` being a `NOT-AND`. Figure 7 depicts two example rules which in addition to providing examples of the `if` and `unless` form also show two different ways in which the argument structure of the RUNNING ARGUMENT rules affects the circuit created.

The MACNET language does not use the RUNNING ARGUMENT's rule form instead it uses arbiters. The rules expressed in MACNET form the central-system, which operates in conjunction with the set of operators in the peripheral-system which perform actions and obtain new values. This top-level-loop does one iteration of

1. `run peripheral-system`
2. `run central-system`

each clock tick. At any one time a subset of *selected* operators are run providing input values for the MACNET rules that are used during `run central-system`. When the MACNET rules are run they provide output that selects which operators should be run on what values for the next clock tick. Figure 8 shows the interface between operators and MACNET, showing the `ready` and `enable` flags that indicate when an operator result or MACNET rules result is available, respectively. Note that a rule consists of an arbiter combined with other MACNET components, and that an arbiter's result is the argument input to an effector, which in most cases is an operator that has the same name as the arbiter.

Arbiters are similar to a combined form of the `if` and `unless` rules although not as intuitively simple. The `if` rule in figure 7 captures an essential arbiter property concerning how proposed wire values are selected. In figure 7 we have the choice from three values of `foo` (`a`, `b` or `c`) that are to be passed onto the wire `bar`. The arbitration language provides a declarative way of stating how an "agent" in the world may react when given a certain input. The values given to an agent do not need to be boolean, we can also use integers and symbols. Inside the network there are two types of wire: wires that hold input signals, and wires that hold the internally generated boolean values. The circuit language primitives that primarily operate on the input signals are gates like `gtm` and `eqm`, that can compare boolean and non-boolean (e.g., integer) values producing a boolean result that can be used by the other gates (e.g., `andg`, etc). These primitives provide some flexibility in the form of the declarations made.

Although MACNET uses a simple value set, the `operators` that it works with can use a more complicated model of the world. The MACNET language has four main components: registrars, arbiters, conditions and proposers. In addition to these main components are `operator` and `pkg` (which is short for package) components. `operators` were described in section 2.4 and are really outside MACNET being those functions that a MACNET rule is designed to select, and who also provide the input that MACNET runs upon.

`pkgs` are used as a structure within which we construct a collection of MACNET rules. A `pkg` allows MACNET components to be added incrementally, to build a single MACNET identified by its `pkg` name *P*. The three main `pkg` language functions are: (`make-pkg`) which creates and returns an empty `pkg` structure, (`with-pkg` *P &rest body*) which opens `pkg` *P* allowing MACNET language declarations to be added in the `with-pkg` *body*. (`compile-pkg` *P*) which compiles the arbiter language definition held in *P* into an executable form which it returns.

4.1 Language components

We begin by describing the various components that make up the MACNET language. This description is based on that given by Chapman [8] and to which the reader is directed for further

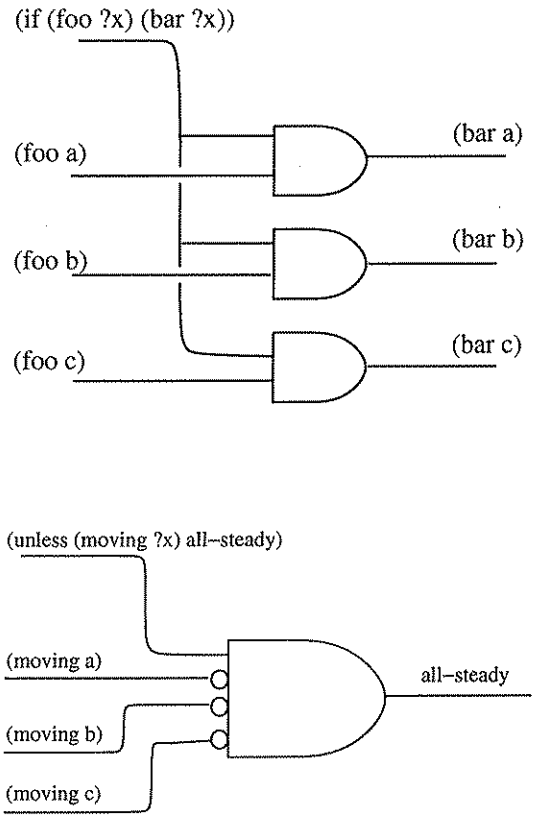

Figure 7: An example of different gate constructions based upon different rule patterns and rule types. From Agre [1].

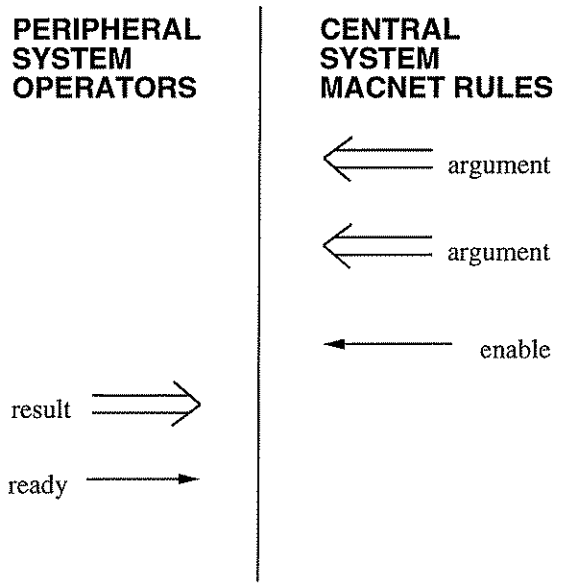

Figure 8: The generic operator-MACNET interface.

details. The description given here differs a little from Chapman's where we have attempted to make things more explicit by the introduction of the arbiter construct. In the main, these are superficial differences, just a little syntactic sugar.

4.1.1 Registrars

Registrars are used to provide an interface between aspects, the output from operators, and the internal circuitry of a `pkg`. These components have the form (`registrar` *register-name wire*). The input values from the outside world are fed through registrars, which perform any initial calculations, to produce a result that can then be used by more than one arbiter. This provides a useful pre-processing stage, making rules written in the MACNET language clearer by allowing the use of each *register-name* instead of the set of gates it represents. We can also use the *register-name* in the definition of other registrars, as long as, no dependency loop is created.

4.1.2 Arbiters

Arbiters are important elements in this language because they facilitate the selection of which set of values are to be assigned to a given set of wires. There are two forms of arbitration: **(1)** the `abstract-arbiter` which allows internal value selections to be made that can be used in further arbitration, and **(2)** the `arbiter` which selects which set of values should be given to the `operator` that it represents. Figure 9 shows the ordering between `abstract-arbiters` and

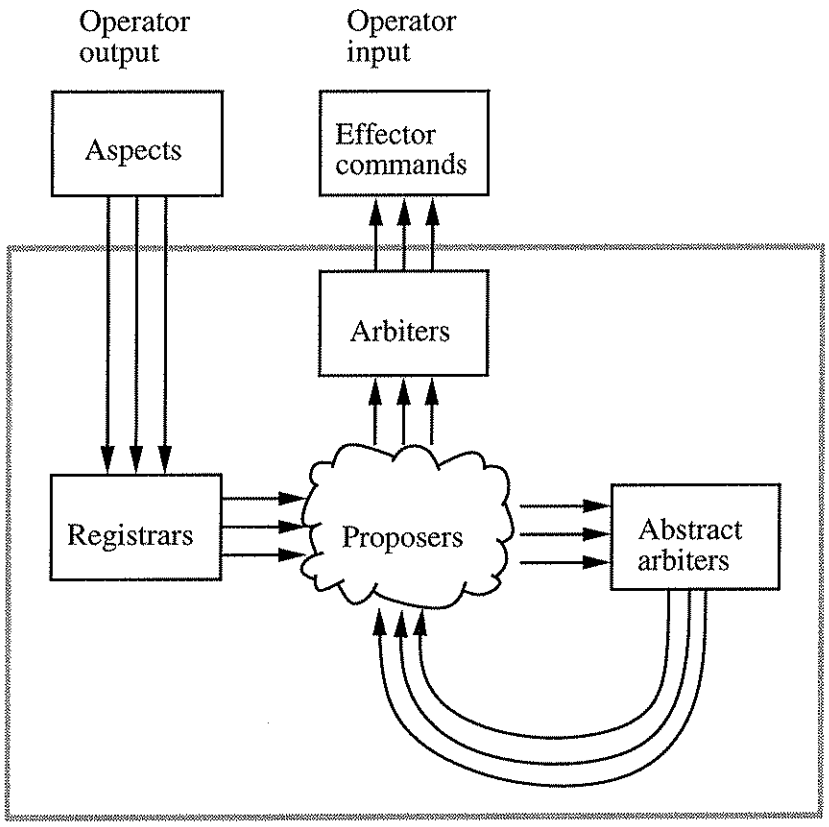

Figure 9: Control flow.

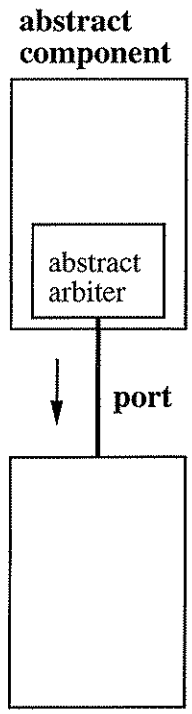

Figure 10: The abstract component.

`arbiters`. The abstract form (`abstract-arbiter` *name arglist &rest body*) produces a result that can be used inside the `pkg` it is defined in. The argument list of the abstract arbiter defines the names of the arbiter's ports which can be accessed by using (`port` *abstract-operation-name port-name*). As shown in figure 10, `port` returns the bus that is the port named *port-name* of the abstract operation named *abstract-operation-name*.

An arbiter (`arbiter` *name arglist &rest body*) produces a result that is exported as the result for the operator, *name*, which is given the selected values for *arglist*. Once either arbiter form has been created it can later be added to, at a later date, by using the form (`with-arbiter` *name &rest body*). All these arbiters have the same form and we describe next how they allow proposals to be put forward and overridden.

4.1.3 Proposals

Each proposer has an identifying name and gives the result variables of its arbiter a proposed value or (more correctly) a circuit that will evaluate to a value. A *proposal* is the set of actual parameters that a proposer thinks the named `operator` should be given at a particular time. For example: (`propose marker-behind :marker *nearest-marker* :testfor (constant :overlap) :doit? (constant *t*)`) A proposal can take one of two forms, a default (`propose-default` *proposer-name &rest key-arglist*) or a general proposal (`propose` *proposer-name &rest key-arglist*) which can be supported by the use of one or more `conditions`. The difference between these two forms of proposal lies in their precedence. Propose-default has zero presidence and is overridden by any other proposal. If more than one `propose` is satisfied

at any one time, the proposal with the highest presidence is selected. This precedence ordering is defined by a declaration using the function (`override-proposer` *overriding-proposer &rest overridden-proposers*) which ensures that overriding *proposal* has a higher presidence than the other named *overridden-proposers*. `override-proposer` ensures that the named proposer is preferred to its *overridden-proposers*. The ability to override is determined by assigning presidence to the proposals that interact. Sometimes proposals are mutually exclusive, so no additional ordering is required.

To ensure that one proposer is valid for an arbiter at any one time, runtime error checking is performed. This error checking takes two forms which produce a message whenever: **(1)** more than one proposer is satisfied, and **(2)** no proposer is satisfied. The first case can be solved by using `override-proposer` to better define the precedence order or by using (`indifferent` *&rest proposers*) which removes error checking for the given case and instead arbitrarily chooses one of the satisfied *proposers*.

4.1.4 Conditions

The condition is a separate clause stating the name of the proposal that it is a condition of and the circuit that is to be evaluated to produce a boolean result. Proposals are satisfied by their conditions being met. Conditions are defined separately from a `propose` function by using (`condition` *proposer-name wire*). A condition specifies the situation under which the named proposer is satisfied, e.g., (`condition marker-behind (andg *registered-nearest* *nearest-behind-me*)`). If more than one condition is present they act conjunctively, so that if any of them are false the proposer does not propose anything.

If a proposal is not given a condition it may never be fired so we give a compile time error, however, there are two exceptions to this rule: **(1)** arbiters that only have one proposal, and **(2)** the default-proposal's condition test which is always true and so does not need a `condition`.

4.2 Interpretation

Now that we have outlined the main components of the MACNET language, our next step is to describe how they fit together, providing a computational model of the central-system.

The rules represented by the two forms of arbiter shown in figure 9, are defined off-line before runtime and fixed during execution. The set of rules, R, represents the action of a prototypical object to a given situation, S, and recent history, H, (internal state of one time step). The rules express relationships between observed situations and effector commands, and these effector commands are used to select which operators, O, to use on the next clock tick. The recent history for H, provided by the `latch` gate makes the circuit context dependent. The relationship between these elements can be summarised by: $R : S \times H_{t-1} \rightarrow O \times H_t$, which shows the correspondence of MACNET to the "behavioural component" described by Whitehead and Ballard [40].

In the remainder of this section we briefly consider four different interpretations of MACNET. The first two work at a more intuitive level, while the third one provides a connection to the GATE language of section 3 by trying to summarize the more formal operational semantics given in Howarth [17] appendix E. The fourth interpretation discusses parallelism.

4.2.1 Component level

Let us begin by re-defining the language components of section 4.1 with a slightly more concise syntax: $\mathcal{P}$ is a set of `pkgs`, and $\forall M \in \mathcal{P}, M \subseteq \mathcal{R}$ where $\mathcal{R}$ is the set of rules, and an arbiter is defined as one of:

`arbiter` $\rho_r\ (\delta_1, \ldots, \delta_n)$ **c**

`abstract-arbiter` $\rho_r\ (\delta_1, \ldots, \delta_n)$ **c**

where $\forall r,\ \rho_r \in \mathcal{R}$; ρ_r is the arbiter name and also the name of the operator to which the

values of arguments $\delta_1, \ldots, \delta_n$ are supplied; and **c** is the arbiter's body, a set of the MACNET components `proposer`, `condition`, `override-proposer`, `indifferent`. These components are defined as follows. A default proposer is written

$$\texttt{propose-default}\ \alpha_h[\delta_1 := \gamma_1, \ldots, \delta_n := \gamma_n]$$

with the following implicit declarations

$$\beta_h = \texttt{true} \wedge (\forall \alpha_j \in \mathcal{A} \wedge j \neq h, (\texttt{override-proposer}\ \alpha_j\ [\alpha_h]))$$

where α_h is a proposer name; β_h is the corresponding GATE language predicate; $\mathcal{A}$ is the set of processor names of ρ, $\forall i$, $\alpha_i \in \mathcal{A}$. Also n_ρ is the number of arbiter names in the pkg M, $n_{\alpha r}$ is the number of proposer names in the arbiter ρ_r, where $n_\rho, n_{\alpha r} \in \mathbb{Z}_\infty$ (the set of positive integers including 0), and $h \in n_{\alpha r}, r \in n_\rho$.

In the more general case we explicitly state both proposer and condition

$$\texttt{propose}\ \alpha_i[\delta_1 := \gamma_1, \ldots, \delta_n := \gamma_n]$$
$$\texttt{condition}\ \alpha_i\ \beta_i$$

and if necessary the indifferent and override-proposer

$$\texttt{indifferent}\ \boldsymbol{\alpha}$$

where $\boldsymbol{\alpha} \subseteq \mathcal{A}$

$$\texttt{override-proposer}\ \alpha_j\ \boldsymbol{\alpha}$$

where $\boldsymbol{\alpha} \subseteq (\mathcal{A} - \alpha_j)$.

Also $\mathcal{O}$ is the set of all combinations of elements from $\mathcal{A}$ (i.e., $\boldsymbol{\alpha} \in \mathcal{O}$). $\mathcal{T}$ is the set of tests, $\forall i$, $\beta_i \in \mathcal{T}$. Note that the sets $\mathcal{A}$, $\mathcal{O}$ and $\mathcal{T}$ are scoped by arbiter ρ_r, and that $i, j \in n_{\alpha r}$, such that i might be equal j, or be a member of $\boldsymbol{\alpha}$ describing its relationship to the other members of $\mathcal{A}$.

We can now define an informal semantics that describes how the rules are interpreted at run-time:

$$M \in \mathcal{P}, \forall \rho \in M, \forall \beta_i \in \mathcal{T}, \quad \begin{array}{l}\text{if } \beta_i \text{ then } [\delta_1 := \gamma_1, \ldots, \delta_n := \gamma_n] \\ \text{unless (overridden } \alpha_i) \vee \text{(other-indifferent-selected } \alpha_i)\end{array} \tag{1}$$

where

$$\text{overridden } \alpha_i = \begin{array}{l}\exists \alpha_j \in \mathcal{A}, \exists \boldsymbol{\alpha} \in \mathcal{O} \mid (\texttt{override-proposer}\ \alpha_j\ \boldsymbol{\alpha}) \wedge \alpha_i \in \boldsymbol{\alpha} \\ \wedge\ \beta_j \text{ evaluates to true}\end{array} \tag{2}$$

$$\text{other-indifferent-selected } \alpha_i = \text{if} \quad \begin{array}{l}(\texttt{indifferent}\ \boldsymbol{\alpha}) \wedge \boldsymbol{\alpha} \in \alpha_i \wedge \\ (\text{compile-time-random-selected-option } \boldsymbol{\alpha}) \neq \alpha_i \\ \wedge\ \forall \alpha_j \in \boldsymbol{\alpha}, \beta_j \text{ evaluates to true}\end{array} \tag{3}$$

Equation 1 describes how for each rule ρ_r in the current pkg M we run each test β_i, a collection of gates providing a boolean result, which if true, causes the set of values γ in the corresponding proposer α_i to be assigned to the arbiter arguments $\boldsymbol{\delta}$, unless α_i is overridden in some way. Equations 2 and 3 describe how a valid proposition can be overridden. In equation 2, α_i is overridden if there exists another α_j that overrides a set of processor names of which α_i is a member and that this overriding member α_j has its conditions met. In equation 3, α_i is overridden if it is a member of a set of an indifferent declaration and has not been randomly selected as the valid proposition from that set and that all the members in the indifferent set $\boldsymbol{\alpha}$ have had their various conditions met.

This provides an informal semantics for the MACNET language, but does not really bridge the gap to the GATE language, except for their role as the β predicates. Equation 1 shows the relationship to the `if` and `unless` rules in RUNNING ARGUMENTS as described at the beginning of section 4.

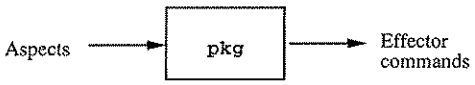

Figure 11: The pkg.

4.2.2 Default logic

Here we explore the similarity between MACNET and default logic[6] (Reiter [30]) where the elements of the MACNET language can be seen as mapping onto default rule form as follows: `propose` being the prerequisite, the `conditions` and `override-proposers` being the justification, and the `arbiter` value allocation being the consequent. Using the syntax of section 4.2.1 this can be described as:

$$\frac{\beta_i : \neg(\text{overridden } \alpha_i)}{[\delta_1 := \gamma_1, \ldots, \delta_n := \gamma_n]}$$

Although this mapping fits well in the AI field, the proofs of equivalence are difficult and we do not follow this route here.

4.2.3 Gate language level

In this subsection we describe the equivalence between MACNET and the GATE language. To do this we use the box-like unit form introduced in section 3. This is not just the case of saying what goes on inside each of the unit descriptions, it also involves how the various units are composed within a `pkg`, which is effected by the declarations for `indifferent` and `override-proposer` detailed in equations 2 and 3. Figure 11 gives the top-level description of inputs and outputs, which is looked at in more detail in figure 12. As shown in figure 9, the `registrars` provide the input wires used by all `conditions` and `proposers`. The form these connections can take is illustrated in figure 13 with `registrars` taking input from aspects and `arbiters` giving effector commands in the form of a tuple (operator-name args). In the figure, the internal component level mapping to the GATE language is shown. At the top of figure 13, we have the `arbiter` unit where proposal-wires carry the γ values from the proposer to the arbiter; $\triangle$ generates the run-time constant operator name ρ_r and the job of the arbiter is just to pass on the γ values. In the `abstract-arbiter`, we use the arbiter name to resolve `port` indirections, and link up the wires of the port bus to the gates that use them. In the `registrar` we just pass on the value. In `conditions` we use the gate structure given to produce the boolean result. The proposer looks more complex, with the set of `andg` gates used to select whether the bus of args should be passed on or not.

The wire connections within a `pkg` are sketched in figure 14. To simplify things in figure 14, only the first letter of the various components has been used. It is interesting to note the similarity of this figure with figure 1 in Nilsson [27], showing the underlying commonality of the two approaches. Also the figures 13 and 14 show the relationship of the proposers to the `if` gate, and the overriders to the `unless` gates in RUNNING ARGUMENTS. Figure 15 gives a more detailed view of the and gate used to express overriders. Also note the correspondence

[6] A default inference rule in default logic is written in the form:

$$\frac{\alpha(x) : \beta(x)}{\gamma(x)}$$

where $\alpha(x)$, $\beta(x)$ and $\gamma(x)$ are well-formed formula called the "prerequisite", the "justification" and the "consequent" of the default respectively. The interpretation of this rule is as follows. If $\alpha(x)$ is known, and $\beta(x)$ is consistent with what is known, then $\gamma(x)$ may be concluded.

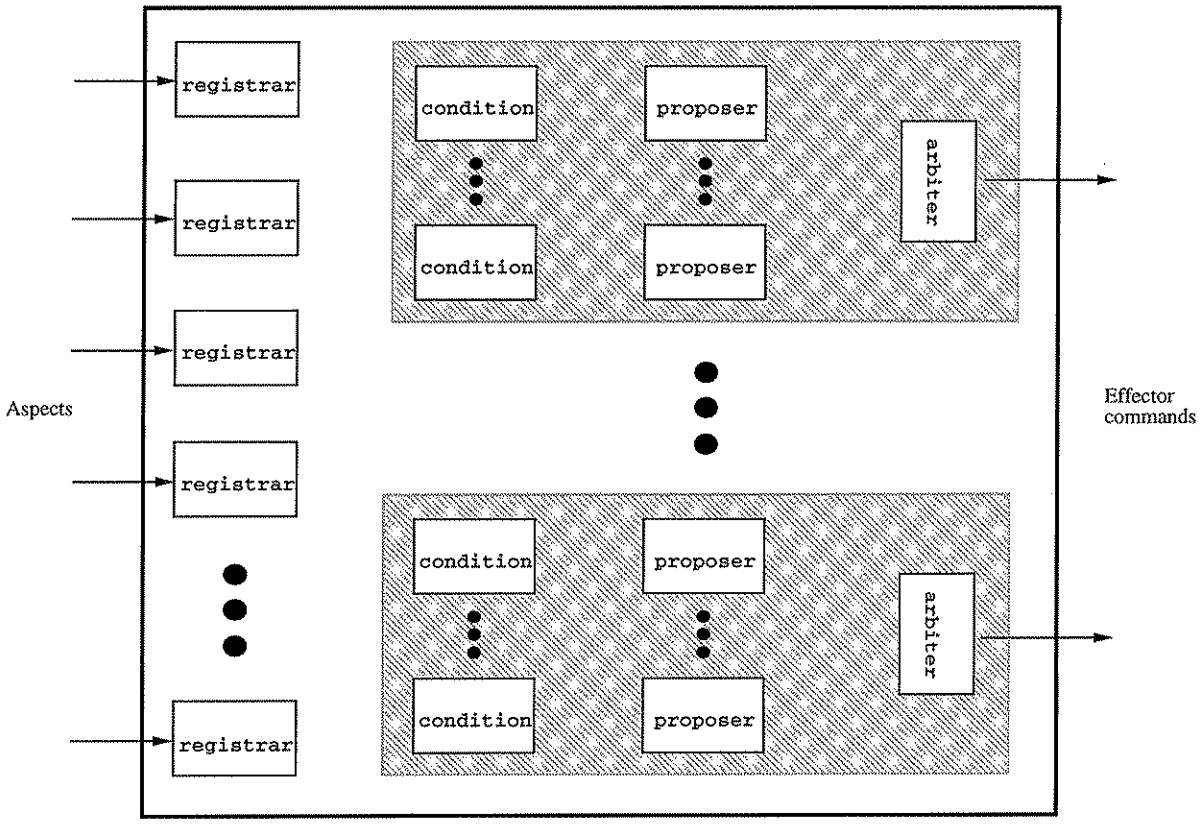

Figure 12: An outline of the contents of a pkg.

to equation 1 of section 4.2.3, which is what this collection of gates and wires is implementing.

Figure 16 shows the form of the circuit that is used to check for errors described in section 4.1.3 and generate error messages when necessary. In the figure, we have a set of three input wires a, b and c, and the output wires w and e. This test for correct evaluation of an arbiter has two parts: (1) an org is used to determine if the arbiter has *any* inputs, which gives a value to w, and if w is false then we give a warning to say that there has been no call to this operation; and a pairing of all inputs (in the example this is ((a b), (a c), (b c))) is sent to another org to check whether *two* or more are true at once, giving the wire e a value, and if e is true then there has been an ambiguous call to the arbiter. Note that an "indifferent" declaration between input wires is equivalent to an org that replaces the inputs that are declared indifferent by the org's result. In figure 16 if a and b were declared indifferent then we would replace them by a joint a ∨ b input, so simplifying the check to operate on just two input wires, i.e., c and the new a ∨ b result.

What we end up with is a sequence of translations, from MACNET to GATE, and then GATE to its execution form, i.e., $ML \rightarrow GL \rightarrow EF$, which could be replaced by one translation $ML \rightarrow EF$.

4.2.4 Parallelism

Although the implementations described above provide a serial solution, some of the circuit instructions can be performed in parallel. However, there are constraints on parallel execution due to various orderings placed on the MACNET components.

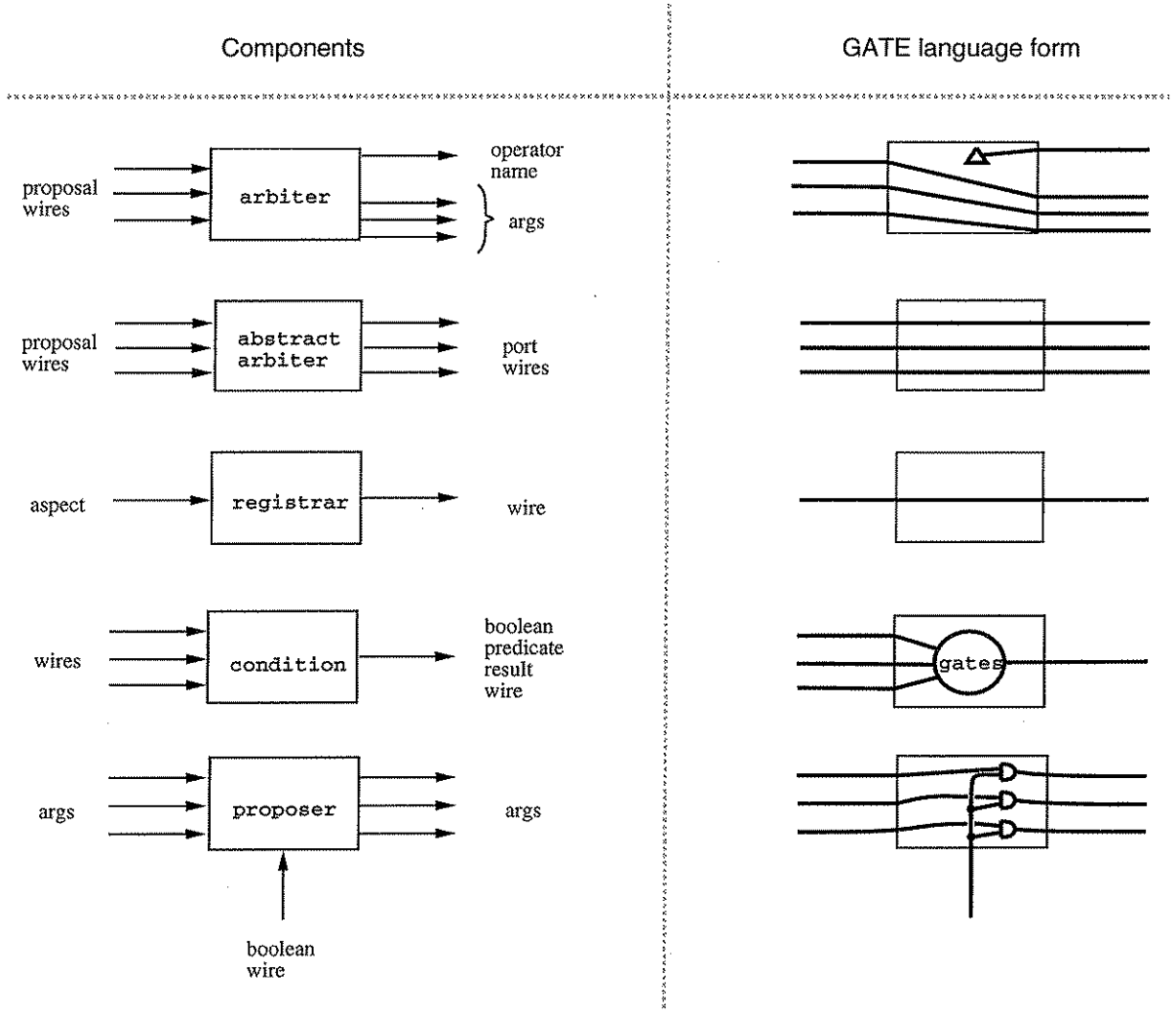

Figure 13: The various components and their GATE language equivalents.

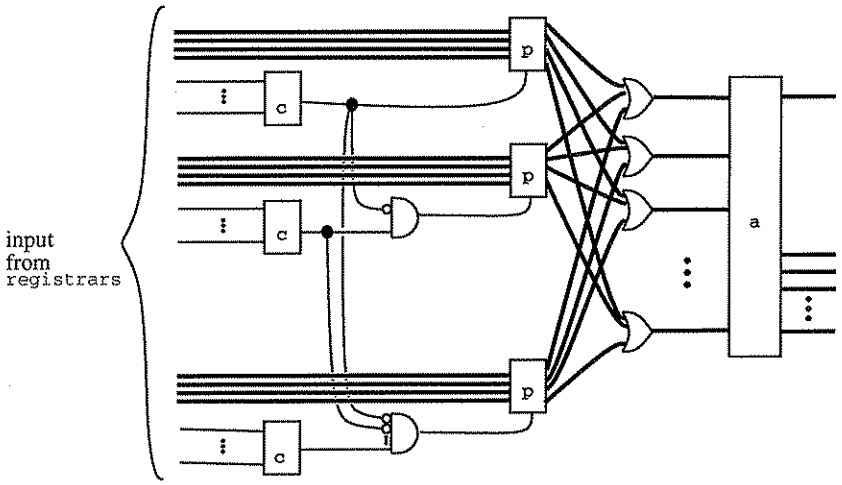

Figure 14: The wiring of gates, conditions, proposers and arbiters as defined by the overriders.

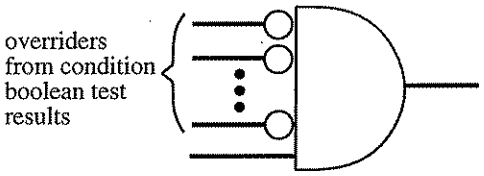

Figure 15: The overriders.

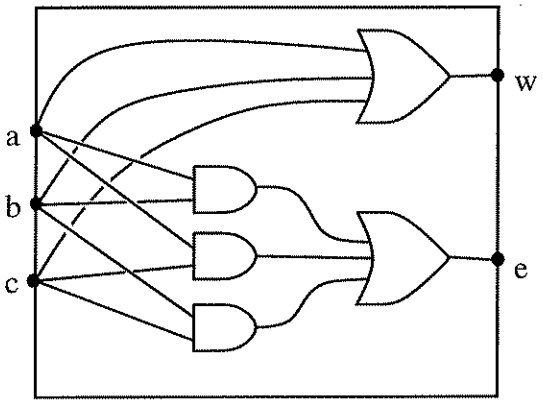

Figure 16: Check the inputs to the arbiter.

- There is an ordering on `arbiters` and `abstract-arbiters` such that all `abstract--arbiters` that produce a value used by another `arbiter` (including other abstract ones) must be calculated before it is used. This dependency relationship forms levels of arbitration, where all arbiters at the same level can be processed in parallel. We can calculate the levels as follows:

$$\text{level } \rho = \begin{cases} 0, & \text{if } \rho \text{ is an } \texttt{arbiter} \\ 1 + \max[(\text{level } x) \mid x \in \mathcal{R} \wedge \\ \quad (\text{gives-a-port-to } \rho\ x)], & \text{if } \rho \text{ is an } \texttt{abstract-arbiter} \end{cases}$$

  Here, (gives-a-port-to a b) is a predicate that returns true if a supplies b with a `port` (see section 4.1.2).

- There is a similar constraint on registrars but this could be precompiled out using the result from a topological sort on `registrar` interdependences.

- At a finer grain there is gate sequencing in arbiters. as shown in figure 14. Fortunately there are no loops within the gate language.

Apart from these issues the central-system could be implemented on a MIMD parallel architecture. The peripheral-system is also a suitable candidate to a parallel implementation because, as described in section 2.5, the operators should be mutually independent.

4.3 Summary

In this section we have described the MACNET language from a number of different viewpoints that hopefully capture both the intuitive simplicity and the various complexities encountered when an implementation is attempted. We do not provide an example of how MACNET can be applied to a problem, this is covered expertly in Chapman's description of BLOCKHEAD [7] and SONJA [8], and discussed in a broader context by McDermott [24].

Both McDermott and Nilsson [27] explore some interesting avenues for developing MACNET-like approaches including the incorporation of probabilistic techniques and various learning techniques (also see Whitehead and Ballard [40]). These are necessary if we are to begin addressing the assumed involvement of the program designer discussed in section 2.1.

5 Conclusion

Having implemented MACNET and used it for developing deictic systems, it may be surprising to find that there is little in its definition that makes it a necessary element in the development of future deictic implementations. Wires and circuits are not needed for deictic reasoning. Their presence, however, limits the options available, guiding the process of implementation towards something that may support the deictic viewpoint. Most of the complexity in MACNET supports ways of taking environmental conditions into account when choosing between alternative courses of action.

The implementation of MACNET does not require the use of Lisp,[7] although it does make some aspects of the implementation easier. Also there is no reason that the "linear executable form" from the gate language could not be translated to a host architecture's machine code, should faster execution be required. After all, when you get the the bottom of it, what we describe here is a logic simulator that executes a given network of combinatorial logic gates.

Acknowledgement

This work has been funded by SERC under a CASE award with the GEC Marconi Research Centre. The development in section 4.2.3 fell out from conversations with Peter Landin as an appropriate short-hand instead of the lengthy VDL [4, 29] notation I had previously used. This operational semantics is available via www `ftp://ftp.dcs.qmw.ac.uk/pub/vision/pg-/howarth/THESIS/apE.274-306.ps.gz` and anonymous ftp (`ftp.dcs.qmw.ac.uk`) in the directory `vision/pg/howarth/THESIS/`.

References

[1] Philip E. Agre. *The Dynamic Structure of Everyday Life.* PhD thesis, MIT AI Lab., October 1988. AI-TR 1085.

[2] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth AAAI Conference*, pages 268–272, 1987.

[3] James F. Allen, Henry A. Kautz, Richard N. Pelavin, and Josh D. Tenenberg. *Reasoning about plans.* Morgan Kaufman Publ. Inc., 1991.

[4] Andrew D. McGettrick. *The Definition of Programming Languages.* Cambridge University Press, 1980.

[5] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[6] Rodney A. Brooks. Intelligence without reason. In *Proceedings of the Twelfth IJCAI Conference*, pages 569–595, 1991.

[7] David Chapman. Penguins can make cake. *AI Magazine*, 10(4):45–50, Winter 1989.

[8] David Chapman. *Vision, Instruction and Action.* The MIT Press, 1991.

[9] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, 1985.

[10] Laurent D. Cohen. On active contour models and balloons. *CVGIP: Image Understanding*, 53(2):211–218, 1991.

[11] Hubert L. Dreyfus. *What Computers Still Can't Do: A Critique of Artificial Reason.* The MIT Press, 1992. A revised version with new introduction of the New York: Harper & Row, 1972 edition.

[12] Martin A. Fischler and Oscar Firschein, editors. *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms.* Morgan Kaufman Publ. Inc., 1987.

[7] For example, the version of MACNET described here has been implemented in Lisp and GOFER.

[13] Jerry A. Fodor. *The Modularity of Mind: An Essay on Faculty Psychology.* The MIT Press, 1983.

[14] Harold Garfinkel. *Studies in Ethnomethodology.* Prentice-Hall, 1967.

[15] Michael P. Georgeff and François Felix Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh IJCAI Conference*, pages 972–978, 1989.

[16] John Heritage. *Garfinkel and Ethnomethodology.* Polity Press, 1984.

[17] Richard J. Howarth. *Spatial Representation, Reasoning and Control for a Surveillance System.* PhD thesis, Queen Mary and Westfield College, London, July 1994.

[18] Richard J. Howarth and Hilary Buxton. Selective attention in dynamic vision. In *Proceedings of the Thirteenth IJCAI Conference*, pages 1579–1584, August 1993.

[19] Jim Ivins and J. Porrill. Statistical snakes: active region models. In *Proceedings of the Fifth British Machine Vision Conference*, 1994.

[20] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: active contour models. *International Journal of Computer Vision*, 1(4):321–331, 1987.

[21] James V. Mahoney and Shimon Ullman. Image chunking defining spatial building blocks for scene analysis. In Zenon W. Pylyshyn, editor, *Computational Processes in Human Vision: An Interdisciplinary Perspective*, pages 169–209. Ablex Publishing Corporation, 1988.

[22] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4.* Edinburgh University Press, 1969.

[23] Drew McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6:101–155, 1982.

[24] Drew McDermott. Robot planning. *AI Magazine*, 13:55–79, Summer 1992.

[25] Robert C. Moore. A formal theory of knowledge and action. In Jerry R. Hobbs and Robert C. Moore, editors, *Formal Theories of the Commonsense World*, pages 319–358. Ablex Publishing Corporation, 1985.

[26] David W. Murray, Philip F. McLauchlan, Ian D. Reid, and Paul M. Sharkey. Reactions to peripheral image motion using a head/eye platform. In *Proceedings of the Fourth International Conference on Computer Vision*, pages 403–411, 1993.

[27] Nils J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

[28] Donald A. Norman. Cognition in the head and in the world: An introduction to the special issue on situated action. *Cognitive Science*, 17(1):1–6, 1993.

[29] Frank G. Pagan. *Formal Specification of Programming Languages: A Panoramic Primer.* Prentice Hall, 1981.

[30] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[31] Marc H.J. Romanycia. The composition and control of visual routines. In Adam Krzyzak, Tony Kasvand, and Ching Y. Suen, editors, *Computer Vision and Shape Recognition*, pages 291–315. Singapore: World Scientific, 1989.

[32] Stanley J. Rosenschein. Synthesizing information-tracking automata from environment descriptions. In R. Brachman et al., editor, *KR '89: Principles of Knowledge Representation and Reasoning*, pages 386–393. Morgan Kaufman Publ. Inc., 1989. Proceedings of the first conference.

[33] Stanley J. Rosenschein and Leslie Pack Kaelbling. The synthesis of digital machines with provable epistemic properties. In Joseph Y. Halpern, editor, *Theoretical Aspects of Reasoning about Knowledge*, pages 83–98. Morgan Kaufman Publ. Inc., 1986. Proceedings of the 1986 Conference.

[34] Devika Subramanian and John Woodfill. Making situation calculus indexical. In R. Brachman et al., editor, *KR '89: Principles of Knowledge Representation and Reasoning*, pages 467–474. Morgan Kaufman Publ. Inc., 1989. Proceedings of the first conference.

[35] Lucy Suchman. *Plans and Situated Actions: The Problems of Human-Machine Communication.* Cambridge University Press, 1987.

[36] Christopher Terman. *Simulation Tools For Digital LSI Design.* PhD thesis, MIT Laboratory for Computer Science, 1983. MIT/LCS/TR-304.

[37] Shimon Ullman. Visual routines. In Steven Pinker, editor, *Visual Cognition*, pages 97–159. The MIT Press, 1985. Also in Fischler and Firschein [12], pages 298–328, and in a special issue of *Cognition*, volume 18, 1984.

[38] Johan F.A.K. van Benthem. *The Logic of Time.* D. Reidel Publishing Company, 1983.

[39] Alonso H. Vera and Herbert A. Simon. Situated action: a symbolic interpretation. *Cognitive Science*, 17(1):7–48, 1993.

[40] Steven D. Whitehead and Dana H. Ballard. Learning to perceive and act by trial and error. *Machine Learning*, 7(1):45–83, 1991.

[41] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition: A New Foundation for Design.* Ablex Publishing Corporation, 1986.