

Abacus: a Notation for Describing Concurrent Computations*

O.M. Nierstrasz

Abstract

Abacus is an experimental notation for specifying concurrent computations, to be used as a semantic target for defining and prototyping concurrent language constructs. We present the current implementation and its underlying computational model, and we illustrate its computational power and expressiveness through examples and by demonstrating equivalence with other models.

1 Introduction

During our previous work on defining and prototyping a concurrent, object-oriented programming language, *Hybrid* [Nierstrasz, 1987; Konstantas, et al. 1988], we observed that, although standard denotational techniques [Gordon 1980] for specifying the semantics of programming languages exist, and although a significant body of literature exists on formal models of concurrency, there is no commonly accepted approach to defining the semantics of concurrent programming languages. This, we believe, is partly due to the fact that denotational techniques, while eminently successful when applied to sequential languages, inherently fail to capture the essence of a concurrent computation (even when an interleaving semantics is used), and partly due to the widely varying concerns of the designers of concurrent languages. In an effort to remedy this situation, we are experimenting with a notation for describing concurrent computations loosely based on CCS [Milner, 1980] and CSP [Hoare, 1985], to be used eventually as a target for operational specifications of concurrent programming languages.

Our primary goal in developing a new notation is to arrive at a computational model that can be used not only for defining new programming languages, but can mainly serve as a standard for *comparing* languages, approaches and mechanisms. Although we do not claim to have reached

*In *Object Oriented Development*, ed. D. Tsichritzis, CUI, University of Geneva, 1989, pp. 247-275.

this goal, it has influenced the design in our search for simplicity and generality. Our interest in object-oriented languages and systems has led us towards a model of *behaviour nets* that seems to naturally support encapsulation and behavioural abstraction. We have sought to learn as much as possible from CCS, CSP, and Actors [Agha, 1986], and adapt their contributions to our purposes.

As specific requirements, we seek to:

- provide an operational, machine-independent computational model of concurrency
- support the dynamic instantiation of concurrent threads
- distinguish between the roles of non-determinism and concurrency in a computation
- support the description of either medium-grain or fine-grain concurrency

The version of Abacus we present in this paper exists as a Prolog implementation, and can thus serve as a basis for prototyping and experimenting with concurrent language constructs. To be a generally useful prototyping tool, however, we require some extensions to the computational model, and we need a means for mapping syntactic patterns to behavioural patterns described in Abacus. In the long term, we plan to characterize formal properties of behavioural patterns in order to facilitate the study and analysis of constructs specified in terms of Abacus.

In section 2 we informally introduce the basic concepts and the syntax of Abacus through a small example, and we introduce *event graphs* as a means of graphically depicting a computation. We then discuss the semantics of Abacus in terms of *behaviour nets*, a formal model of concurrent computation. We illustrate how the roles of non-determinism and concurrency are confused in language-theoretic (i.e., interleaving) approaches to modeling concurrency, and we show how event graphs can expose this confusion. Section 4 presents our main results, a formal comparison of Abacus and behaviour nets with finite state automata, Petri nets and Turing machines. Section 5 gives a hint as to how we intend to use Abacus for modeling computation and prototyping language constructs. We conclude with a few remarks on similarities and differences between our approach and those of CCS, CSP and Actors, and some observations on our future work.

2 Abacus Syntax and Basic Concepts

As in both CCS and CSP, we model computations as interactions between a systems of communicating processes, or *agents*. Each agent makes a set (possibly empty) of simultaneous input and output *offers* to participate in events. An offer is like a guarded command. An event may only take place between two agents when the input offer of one matches the output offer of the other. (An agent may not match its own offer.) The *behaviour* of an agent determines not only what events

it may participate in right now, but also what its replacement behaviour (or continuation) will be. In order for us to be able to model dynamic creation of new agents, we permit the replacement behaviour to be, in general, that of a *system* of agents, rather than simply a single agent. In this respect we depart from CSP and adopt an actor-like approach to agent instantiation.

In the example that follows, readers familiar with CCS and CSP will recognize the operational semantics of CCS and the syntax of CSP.

Let us consider the following declaration in Abacus:

```
true := isTrue!true + setTrue?true + setFalse?false .
```

Here `true` and `false` are agent names, and `isTrue`, `setTrue` and `setFalse` are event names. This declaration binds the agent name `true` to the behaviour expression

```
isTrue!true + setTrue?true + setFalse?false
```

(The terminating period is an artifact of the Prolog implementation, indicating that this declaration is a “fact.”)

The subexpression `isTrue!true` says that the agent `true` is willing to output the value `isTrue`, and replace itself with the behaviour named by `true`. The symbols `!` and `?` stand, respectively, for output and input guards on the values (event names) preceding them. The replacement behaviour follows immediately after the guard symbol.

The symbol `+` indicates an exclusive choice of input and output offers. The `true` behaviour will either output `isTrue`, or input `setTrue` or `setFalse`. An agent may participate in at most one event at a time.

The declaration of `true` is incomplete, since we do not know what its behaviour will be after a `setFalse` event. Here is the declaration named by `false`:

```
false := isFalse!false + setTrue?true + setFalse?false .
```

We can now interpret these declarations as follows: an instance of `true` will output the value `isTrue` to any agent wanting to know its current value. Alternatively, it will accept as input the request `setTrue` (which has no effect), or `setFalse`, which causes it to change its behaviour to that of `false`. The `false` behaviour is similar, except it outputs `isFalse` instead of `isTrue`. `true` and `false` therefore implement the two states of a primitive Boolean variable.

We can now use our Boolean variable in a simple computation that samples the state of the variable and toggles it. We need the help of a `negate` agent:

```
negate := isTrue?setFalse!nil + isFalse?setTrue!nil .
```

An instance of `negate` will input `isTrue` and output `setFalse`, or input `isFalse` and output `setTrue`. The final behaviour in either case, is the special behaviour `nil`, which makes no offers, and participates in no events. Since a `nil` behaviour contributes nothing to a computation, we normally delete it.

To perform a computation, we take the behaviour expression of a system of agents representing an initial configuration, and pass it to the Abacus interpreter. (We shall use the term *system* to refer in general to collections of agents, and *configuration* to refer to the collection of all agents visible at some stage in a computation.) For example:

```
true & negate
```

denotes a pair of agents, one with the `true` behaviour and one with the `negate` behaviour. The symbol `&` indicates parallel composition of agents or systems. This can be evaluated by Abacus with the statement:

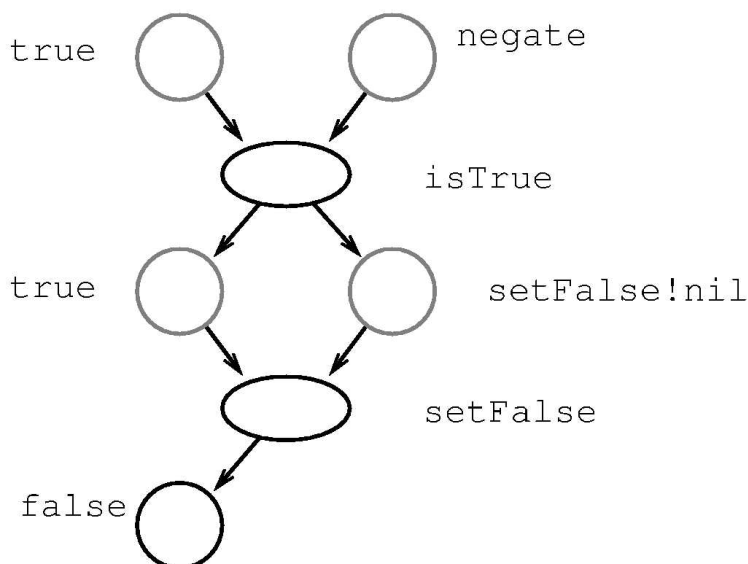
```
:- abcpath(true & negate).
```

which causes Abacus to print the first execution path (i.e., sequence of events) that it finds, namely:

```
Initial configuration: true & negate .
First execution path: (isTrue) -> (setFalse) ->
Final configuration: false .
```

That is, `negate` samples `true`, issues `setFalse`, and disappears, leaving behind `false`.

In figure 1 we see the *event graph* depicting this computation. Agents are represented by circles and events as ellipses. Every event has exactly two in-edges from the agents participating in the event and a number of out-edges to the replacement agents. Every agent has at most one out-edge (since it participates in at most one event, and then replaces itself). We start with agents `true` and `negate` which are consumed in an `isTrue` event and produce `true` and `setFalse!nil` agents. These, in turn, are consumed in a `setFalse` event, leaving only a `false` agent in the final configuration.

Figure 1: The Event Graph of `(true & negate)`

Agents with no out-edge are free to participate in future events, should any be enabled. In this case, only the `false` agent is left, so no further events can take place.

This particular computation has only one possible course. In section 4 we shall see that event graphs can model concurrency within a computation, and that multiple event graphs can be useful for capturing non-determinism during the course of a computation.

The syntax of Abacus declarations is given in table 1 in an extended BNF. Non-terminals are in Times Roman and identifiers in italics. Double colon (`::`) defines a production. An or-bar (`|`) indicates a choice of productions. Optional items are in square brackets (`[]`), and repetitions of zero or more items are in brace brackets (`{}`). All other symbols (`:=`, `(`, `)`, `&`, `+`, `?`, `!`, `nil` and `.`) are literals.

Note that `+` binds more closely than `&`, so that:

$$a!p + b!q \ \& \ c!x$$

Is parsed as:

$$(a!p + b!q) \ \& \ (c!x)$$

Not all declarations define agent behaviours, for example, the declaration:

```

declarations :: { behName := behExpr . }
behName     :: identifier [ eventTuple ] | nil
eventTuple  :: ( eventName { , eventName } )
eventName   :: identifier | eventTuple
behExpr     :: seqBeh | parBeh
parBeh      :: seqBeh & behExpr
seqBeh      :: choice [ + seqBeh ]
choice      :: eventName ? replBeh
              | eventName ! replBeh
              | behName
replBeh     :: choice | ( parBeh )

```

Table 1: Abacus Syntax (extended BNF)

```
p := true & negate .
```

binds `p` to the behaviour of a system (i.e., a collection of agents). The agent behaviour declarations are exclusively of the form:

```
behName := seqBeh .
```

For this reason, when dealing with Abacus expressions, we shall typically speak of *behaviour* names, rather than simply agent names.

Parallel behaviour expressions are most useful for modeling dynamic process creation. An agent participating in an event may replace itself by a collection of agents, rather than just a single agent.

One can always tell from the context whether an identifier is an event name or a behaviour name. It is consequently permissible to overload the use of such names. For example, we could have declared `true` as:

```
true := true!true + setTrue?true + setFalse?false .
```

overloading `true` as both a behaviour name and an event name.

Event names, in general, may be tuples rather than simple identifiers, so we might define:

```
true := (is,true)!true
```

```
+ (set,true)?true
+ (set,false)?false .
```

A tuple may be used to index a behaviour name, so we might have:

```
bool(true) := (is,true)!bool(true)
            + (set,true)?bool(true)
            + (set,false)?bool(false) .
```

Although parameterized behaviours are not currently supported, we see this as a fundamental extension necessary for Abacus to be a truly useful notation. At present we can make use of Prolog's uninstantiated variables to achieve a similar effect, but we must leave our computational model to do so.

Finally, for any given set of Abacus behaviour declarations, there may or may not be a valid interpretation. For example,

```
p := q .
q := p .
```

defines no behaviour at all. We say that a behaviour expression is (recursively) *well-guarded* if it makes no use of circularly defined behaviour names. This is implemented by the Prolog predicate `wg()` given in appendix A.

3 Abacus Semantics

In order to precisely describe the operational semantics of Abacus, and in order to study the formal properties of Abacus programs, we provide here a formal description of the computational model behind behaviour expressions. The mapping between Abacus expressions and behaviour nets, and the evaluation of behaviour nets is accomplished by the Prolog implementation listed in the Appendix.

3.1 Basic notation

We shall make use of the following basic definitions:

\mathcal{N} stands for the set of natural (i.e., non-negative) integers.

The *power set* of a set S is denoted by $\mathcal{P}(S) \equiv \{X \mid X \subseteq S\}$

For sets A and B , B^A stands for the set of functions $A \rightarrow B$.

$\mathcal{F}(X)$ stands for the *finite multi-sets* of X , i.e.,

$$\mathcal{F}(X) \equiv \{f \in \mathcal{N}^X \mid \sum_{x \in X} f(x) < \infty\}$$

That is, for $f \in \mathcal{F}(X)$, $f(x) = n$ is the number of copies of x in the multi-set (or “bag”) f . In examples, we shall represent multi-sets in a set-like notation using square brackets, e.g., $[a, b, b, c]$ represents the multi-set with one copy each of a and c , and two of b .

For any subset $X \subseteq S$, χ_X denotes the *characteristic function* of X with respect to S , defined by

$$\chi_X(x) \equiv \begin{cases} 1 & x \in X \\ 0 & \text{otherwise} \end{cases}$$

Typically the set S will be understood.

2^S denotes the set of characteristic functions of subsets of S :

$$\begin{aligned} 2^S &\equiv S \rightarrow \{0, 1\} \\ &= \{\chi_X \mid X \in \mathcal{P}(S)\} \end{aligned}$$

Where X contains a single element a , we may abbreviate χ_X by χ_a (provided that a is not itself a set).

3.2 Behaviour nets

We define $\langle A, E, \beta, \sigma_0 \rangle$ to be a *behaviour net* where

1. A is the finite set of *agent names*
2. $S_A = \mathcal{F}(A)$ is the finite set of *systems* of agents, (i.e., if $\sigma \in S_A$, then $\sigma(a)$ yields the number of copies of a in σ)
3. E is the set of *event names*
4. $G = \{?, !\}$ is the set of *guards*, respectively, input and output guards

5. $C = E \times G \times S_A$ is the set of *choices* (i.e., each choice specifies an input or output offer and a replacement system)
6. $\beta : A \rightarrow \mathcal{P}(C)$ is the *behaviour mapping* (i.e., the behaviour of an agent is a set of possible choices)
7. $\sigma_0 \in S_A$ is the *initial configuration*

If $(e, \gamma, \rho) \in \beta(a)$, then we call ρ the *replacement* of a with respect to this choice. Note that replacements are, in general, systems rather than simply agents. This permits us to model dynamic creation of new agents.

In our earlier example,

$$\begin{aligned}
A &= \{ \text{true}, \text{false}, \text{negate}, \text{setFalse!nil}, \text{setTrue!nil} \} \\
E &= \{ \text{isTrue}, \text{isFalse}, \text{setTrue}, \text{setFalse} \} \\
\beta(\text{true}) &= \{ (\text{isTrue}, !, [\text{true}]), \\
&\quad (\text{setTrue}, ?, [\text{true}]), \\
&\quad (\text{setFalse}, ?, [\text{false}]) \} \\
\beta(\text{false}) &= \{ (\text{isFalse}, !, [\text{false}]), \\
&\quad (\text{setTrue}, ?, [\text{true}]), \\
&\quad (\text{setFalse}, ?, [\text{false}]) \} \\
\beta(\text{negate}) &= \{ (\text{isTrue}, ?, [\text{setFalse!nil}]), \\
&\quad (\text{isFalse}, ?, [\text{setTrue!nil}]) \} \\
\beta(\text{setFalse!nil}) &= \{ (\text{setFalse}, !, []) \} \\
\beta(\text{setTrue!nil}) &= \{ (\text{setTrue}, !, []) \} \\
\sigma_0 &= [\text{true}, \text{negate}]
\end{aligned}$$

Note that the set of agent names A is not identical to the set of behaviour names found in an Abacus program. This is because (1) some agents are named by their behaviour expressions (such as `setFalse!nil`), and (2) some behaviour names may be bound to systems rather than agents. For a given Abacus program, the set of agent names A in the corresponding behaviour net contains all sequential behaviour expressions (`seqBeh`) without behaviour names bound to them, plus all behaviour names bound to such expressions. The translation of a behaviour expression into a set of agent names is accomplished by the Prolog predicate `normal()` given in Appendix A.

An event e may *fire* in σ , yielding σ' if $\sigma' \in \delta(\sigma, e)$, where $\delta : S_A \times E \rightarrow \mathcal{P}(S_A)$. Specifically, $\sigma' \in \delta(\sigma, e)$ iff $\exists a, b \in A$ such that:

1. $x \in \{a, b\} \Rightarrow \sigma(x) - \chi_a(x) - \chi_b(x) \geq 0$

2. $(e, \gamma_a, \rho_a) \in \beta(a)$
3. $(e, \gamma_b, \rho_b) \in \beta(b)$
4. $(\gamma_a, \gamma_b) \in \{(?,!),(!,?)\}$
5. $\sigma' \equiv \sigma - \chi_a + \rho_a - \chi_b + \rho_b$

That is, if a and b are agents in σ with matching offers on event name e , then e may fire, yielding σ' , with a and b substituted by their replacements. (Note condition 1 above is not equivalent to $\sigma(a) > 0$, $\sigma(b) > 0$: if $a = b$, we permit two copies of a to communicate with each other, but a single copy of a may not communicate with itself.)

Going back to our example:

$$\begin{aligned} \delta([\mathbf{true}, \mathbf{negate}], \mathbf{isTrue}) &= \{ [\mathbf{true}, \mathbf{setFalse!nil}] \} \\ \delta([\mathbf{true}, \mathbf{setFalse!nil}], \mathbf{setFalse}) &= \{ [\mathbf{false}] \} \end{aligned}$$

The transition function δ is implemented by the Prolog predicate `step()` (see Appendix A).

We can now define the *language* $\mathcal{L}(\sigma)$ of a system σ to be the set of strings in E^* , such that

$$\mathcal{L}(\sigma) = \{e \cdot \mathcal{L}(\sigma') \mid \sigma' \in \delta(\sigma, e)\}$$

The language of a behaviour net is the language $\mathcal{L}(\sigma_0)$ of its initial configuration.

In some cases we shall find it convenient to introduce an *event labeling function* $\lambda : E \rightarrow \Sigma$ from event names to some alphabet Σ . In this case,

$$\mathcal{L}_\lambda(\sigma) = \{\lambda(e) \cdot \mathcal{L}_\lambda(\sigma') \mid \sigma' \in \delta(\sigma, e)\}$$

For example, if $\Sigma = E$, we can choose a λ that simply deletes events we are not interested in.

3.3 Non-determinism vs. Concurrency

Although the notion of the language of behaviour nets will turn out to be useful in gauging their computational power, it turns out that this is inadequate for measuring non-determinism and concurrency within a computation.

Let us consider the system σ represented by the Abacus expression $\mathbf{x\&u\&y\&v}$, where

```

x := a!nil .
u := a?u + c?u .
y := b!nil + c!nil .
v := b?v .

```

Agents x and y each want to output in an event and then go away. Agents u and v represent passive resources, repeatedly accepting input requests.

The corresponding behaviour net $\langle A, E, \beta, \sigma_0 \rangle$ is:

$$\begin{aligned}
A &= \{ x, u, y, v \} \\
E &= \{ a, b, c \} \\
\beta(x) &= \{ (a, !, []) \} \\
\beta(u) &= \{ (a, ?, [u]), (c, ?, [u]) \} \\
\beta(y) &= \{ (b, !, []), (c, !, []) \} \\
\beta(v) &= \{ (b, ?, [v]) \} \\
\sigma_0 &= [x, u, y, v]
\end{aligned}$$

The language $\mathcal{L}(\sigma_0) = \{ ab, ba, ac, ca \}$. This suggests that there are four possible computations proceeding from our initial configuration. This view, however, fails to expose the structure of the possible computations, in particular confusing the roles of concurrency and non-determinism. There are, thus, in our view *three* and not four possible computations (represented as event graphs), one of which contains concurrent events.

In figure 2 we can see more clearly the actual progress of the possible computations by showing the event graphs of all partial computations and the flow between them. This two-level diagram we call a *computation chart*. Graph A shows the initial configuration, with three enabled events indicated by thick grey lines (i.e., between x and u , between u and y , or between y and v).

Although we can clearly see here that events a and b may occur concurrently (they do not share any agents, and so do not interfere) it is not obvious what all the possibilities are. If we consider just one event at a time, we are led to event graphs B, C and D. (The agents that have been consumed are indicated in light grey, to clearly distinguish them from those that remain.)

Only in B is there now a choice between two enabled events. Now it is clear that partial computations B and C both lead to F. This exposes the fact the choice in $\mathcal{L}(\sigma)$ between ab and ba is due to concurrency, whereas that between ac and ca is due to non-determinism.

The computation chart distinguishes between non-determinism, which is the result of choosing a particular traversal to a leaf event graph, and concurrency, which occurs purely within an event

graph. In effect, in order not to lose information, we must model concurrency and non-determinism in terms of a partial order of partial orders. At the level of event graphs, we have a partial order of potentially concurrent events, and at the level of computation charts, we have a partial order of non-deterministically chosen event graphs.

4 Computational Power

Abacus in its present form is not computationally complete. We shall show that, with a simple restriction, behaviour nets are equivalent to non-deterministic finite state automata (FSA), and that, without the restriction, they are equivalent to Petri nets. There are several slight extensions which would suffice to achieve Turing machine completeness. We shall consider some of these possibilities and their ramifications.

For the purpose of the discussion, we shall make use of the following auxiliary definition:

$$\mathcal{F}_n(X) = \{f \in \mathcal{N}^X \mid \sum_{x \in X} f(x) \leq n\}$$

That is, $\sigma \in \mathcal{F}_n(X)$ is a multi-set of at most n elements from X .

Let us define a *conservative behaviour net* to be one in which:

$$C_1 = E \times G \times \mathcal{F}_1(A)$$

and

$$\beta \subset A \rightarrow \mathcal{P}(C_1)$$

That is, the replacement of an agent is always at most one other agent. The effect of this is to forbid the dynamic instantiation of new agents and, consequently, the number of agents can never increase during a computation.

Abacus behaviour declarations can be forced to be conservative by applying the following restriction to the syntax:

```

declarations :: { behName := seqBeh . }
replBeh      :: choice

```

that is, by dropping `parBeh` expressions.

4.1 FSA equivalence

Theorem 1 *Conservative behaviour nets are computationally equivalent to non-deterministic finite state automata.*

Lemma 1.1 *Every conservative behaviour net can be simulated by a non-deterministic finite state automaton.*

Proof:

Suppose $\langle A, E, \beta, \sigma_0 \rangle$ is a conservative behaviour net, where

$$n = \sum_{a \in A} \sigma_0(a)$$

is the number of agents in the initial configuration σ_0 . We then construct the FSA $\langle S, I, \delta_S, s_0, F \rangle$ with

1. finite state space $S = \mathcal{F}_n(A)$
2. input symbols $I = E$
3. state transition function $\delta_S = \delta$
4. initial state $s_0 = \sigma_0$
5. and final states $F = \mathcal{F}_n(A)$

It is now easy to see that the language of the FSA is identical to that of the system σ_0 . For example, the FSA of the system $\mathbf{x} \& \mathbf{u} \& \mathbf{y} \& \mathbf{v}$ is depicted in figure 3 (with all unreachable states in $\mathcal{F}_4(A)$ deleted). Note that concurrency present in the behaviour net must be modelled by non-determinism in the corresponding FSA.

Lemma 1.2 *Every finite state automaton can be simulated by a conservative behaviour net.*

Proof:

Given a FSA $\langle S, I, \delta_S, s_0, F \rangle$, we construct a behaviour net $\langle A, E, \beta, \sigma_0 \rangle$ where

1. $A = S \cup \{\ell\}, \ell \notin S$
2. $E = I$
3. $\beta(\ell) \equiv \{(e, !, \chi_\ell) \mid e \in E\}$
4. $\beta(\sigma) \equiv \{(e, ?, \chi_p) \mid p \in \delta_S(\sigma, e)\}$
5. $\sigma_0 = \chi_{\{\ell, s_0\}}$

Such a behaviour net is clearly conservative (there are never more than two agents at any time, one representing the state of the FSA, and the other representing the observer ℓ).

The Abacus declaration of the behaviour net generated from the FSA in figure 3 would look like:

```

lang := a!lang + b!lang + c!lang .
state(1,1,1,1) := a?state(0,1,1,1)
                + b?state(1,1,0,1)
                + c?state(1,1,0,1) .
state(0,1,1,1) := b?state(0,1,0,1)
                + c?state(0,1,0,1) .
state(1,1,0,1) := a?state(0,1,0,1) .
state(0,1,0,1) := nil .

```

with $\sigma_0 = \text{lang} \ \& \ \text{state}(1,1,1,1)$.

The theorem follows from lemmas 1.1 and 1.2. \square

4.2 Petri net equivalence

Theorem 2 *Behaviour nets are equivalent to Petri nets.*

Lemma 2.1 *Every behaviour net can be simulated by a Petri net.*

Proof:

Given a behaviour net $\langle A, E, \beta, \sigma_0 \rangle$, we construct a Petri net $\langle P, T, I, O \rangle$ with marking μ_0 , where

1. $P = A$ is the set of *places*
2. $T = \delta \cap (\mathcal{F}_2(A) \times E \rightarrow \mathcal{P}(S_A))$ is the set of *transitions*
3. $I : T \rightarrow P$ where $I((\sigma, e, \sigma')) = \{x \mid \sigma(x) > 0\}$ is the *input function*
4. $O : T \rightarrow P$ where $O((\sigma, e, \sigma')) = \{y \mid \sigma'(y) > 0\}$ is the *output function*
5. $\mu_0 = \sigma_0$ is the *initial marking*
6. and $\lambda : T \rightarrow E$ where $\lambda((\sigma, e, \sigma')) = e$ is the *transition labelling function*

We create one transition for every *pair* of agents that might participate in an event (thus we focus our attention on systems in $\mathcal{F}_2(A)$ rather than all of S_A). Configurations are represented by markings, that is, each agent in σ_0 is represented by a token in μ_0 . When a transition fires, the tokens representing the participating agents are consumed, and those representing the replacement behaviours are generated. Since an agent is represented by a token, we preserve the property that an agent may participate in at most one event at a time. Since the number of agents in a Petri net is potentially unbounded, we are able to model non-conservative behaviour nets.

The Petri net representation of `x&u&y&v` with its initial marking is depicted in figure 4. There, in contrast to the FSA representation, we preserve the distinction between concurrency and non-determinism. In general, however, there may be a combinatorial explosion of transitions in the Petri net representation of a behaviour net, since potentially all pairs of agents may participate in potentially all possible events.

Lemma 2.2 *Every Petri net can be simulated by a behaviour net.*

Proof:

Given an arbitrary Petri net $\langle P, T, I, O \rangle$ with marking μ_0 , we construct a behaviour net $\langle A, E, \beta, \sigma_0 \rangle$, where

1. $A = \{\ell\} \cup P \cup 2^P$
2. $E = T \cup \{\text{notify}(p) \mid p \in P\}$
3. $\beta(\ell) \equiv \{(t, !, \chi_\ell) \mid t \in T\}$
4. $\beta(p) \equiv \{(\text{notify}(p), !, \chi_{\text{nil}})\}$
5. $\beta(\sigma) \equiv \{(\text{notify}(p), ?, \chi_{(\sigma + \chi_p)}) \mid \sigma(p) = 0\} \cup \{(t, ?, \chi_{(\sigma - \chi_{I(t)})} + \chi_{O(t)}) \mid p \in I(t) \Rightarrow \sigma(p) > 0\}$

$$6. \sigma_0 = \chi_\ell + \mu_0 + \chi_{\chi_\emptyset}$$

The initial configuration consists of a single observer agent ℓ , a multi-set of agents representing tokens of the initial marking, and a single net agent in 2^P representing the state of the Petri net, initially empty. (Recall that 2^P is the set of characteristic functions of P , each representing some subset of P .) The net agent is interested in at most one token at any place at any time, since more tokens will not effect the enabling of transitions. The token agents attempt to deliver themselves to the net agent by outputting a `notify(p)`. The net agent refuses to accept more than one token per place, causing backlogs of tokens to wait until a place becomes free. When a transition fires, the net agent is updated, and the instances of token agents are generated to represent the output. Since the `notify` events are not part of the original Petri net language, we must adopt an event labelling function that eliminates them:

$$\lambda(e) = \begin{cases} e & e \in T \\ \epsilon & \text{otherwise} \end{cases}$$

where ϵ represents the empty string.

The Abacus declaration of the behaviour net generated from the Petri net in figure 4 would look like:

```
lang := a!lang + b!lang + c!lang .
token(x) := notify(x)!nil .
token(u) := notify(u)!nil .
token(y) := notify(y)!nil .
token(v) := notify(v)!nil .
net(1,1,1,1) := a ? ( net(0,0,1,1) & token(u) )
               + b ? ( net(1,1,0,0) & token(v) )
               + c ? ( net(1,0,0,1) & token(u) ) .
net(1,1,1,0) := notify(v) ? net(1,1,1,1)
               + a ? ( net(0,0,1,0) & token(u) )
               + c ? ( net(1,0,0,0) & token(u) ) .
...
```

with $\sigma_0 = \text{lang} \ \& \ \text{token}(x) \ \& \ \text{token}(u) \ \& \ \text{token}(y) \ \& \ \text{token}(v) \ \& \ \text{net}(0,0,0,0)$.

Note that there is one `net()` behaviour for every element of 2^P , and that we dynamically create new `token` agents in response to events (transitions) firing.

The theorem follows from the two lemmas. \square

4.3 Turing machine equivalence

Since behaviour nets are equivalent to Petri nets, they are clearly not computationally complete. Although we have an infinite state space (through the dynamic creation of new agents), the fact that Abacus restricts us to a finite set of event names and behaviours means that we cannot distinguish between multiple instances of the same agent behaviour. As a consequence, we cannot model (say) the infinite tape of a Turing machine, since we can neither instantiate an infinite behaviour, nor can we model a finite tape to which new, distinguishable “cell” agents can be indefinitely appended.

There are (at least) two ways to achieve computational completeness through minor extensions to Abacus. The first way is to provide for the definition of *event classes*, and to use these to parameterize behaviour declarations:

```
var(n:names,x:values) := (n,is,x)!var(n,x)
                      + (n,set,y:values)?var(n,y) .
```

We have declared a class of agent behaviours by allowing **n** and **x** to range over the event classes **names** and **values**, and we have declared a class of choices by letting **y** range over **values**. The event classes **names** and **values** could be declared as follows:

```
names := { a, b, c } .
values := { red, green, blue } .
```

Macros and recursive definitions would be used to declare infinite event classes:

```
list(X) := { (x):X, (x:X,y:list(X)) } .
valuelist := list(values) .
```

Infinite event classes could then be used to declare infinite classes of distinguishable agents. This would, of course, require us to drop the restriction on behaviour nets that the sets of agent and event names be finite.

Another approach to distinguishing agents is to provide for encapsulation of subsystems. The idea is to declare explicitly which offers are to be exported from a subsystem to external agents. The notation: $[\sigma::\kappa]$ would indicate that the agents in σ may only participate in those events with external agents that lie in the event class κ . This mechanism is sufficient for us to model a stack, or any other unbounded data structure. The following example is adapted from [Milner, 1980]:

```

e := { z,i,d,yes,no } .
ea := { a, x:e } .
eb := { b, x:e } .
zero := z?yes!zero + i?[ link & a?zero :: e ] .
link := [ pos & b?a!nil :: ea ] .
pos := z?no!pos + d?b!nil + i?[ link & a?pos :: eb ] .

```

Here we have declared event classes `e`, `ea` and `eb`. The agents `zero` and `pos` model a counter by simulating a stack. In response to `z!`, `zero` will respond `yes` and `pos` will respond `no`. If we start with a `zero` agent, we can increment it with `i!` requests, and decrement it with `d!` requests. (We can only decrement the counter when it is in a non-zero state.)

The events `a` and `b` occur only within the counter to manage the stack of `pos` agents. The `link` behaviour is basically a macro for an encapsulated subsystem that links together `pos` agents. When the `zero` agent accepts a `i!` request, it hides a copy of itself inside an encapsulated system in which only a copy of `pos` can communicate with outside agents. When `pos` receives `i` requests, it, in turn, splits off encapsulated copies of itself in the same manner. Decrement requests (`d!`) successively “pop off” `pos` agents until the original `zero` is exposed again.

Consider, for example, the initial configuration:

```
[i!i!d!d!nil & zero]
```

If we step through the (only possible) computation, removing the `nils` left behind, we obtain:

```

      [i!i!d!d!nil & zero]
i -> [i!d!d!nil & [[pos & b?a!nil :: ea] & a?zero :: e]]
i -> [d!d!nil & [[[pos & b?a!nil :: ea] & a?pos :: eb]
      & b?a!nil :: ea] & a?zero :: e]]
d -> [d!nil & [[[b!nil & b?a!nil :: ea] & a?pos :: eb]
      & b?a!nil :: ea] & a?zero :: e]]
b -> [d!nil & [[[nil & a!nil :: ea] & a?pos :: eb]
      & b?a!nil :: ea] & a?zero :: e]]
a -> [d!nil & [[[nil & nil :: ea] & pos :: eb]
      & b?a!nil :: ea] & a?zero :: e]]
    = [d!nil & [[[pos :: eb] & b?a!nil :: ea] & a?zero :: e]]
d -> [nil & [[[b!nil :: eb] & b?a!nil :: ea] & a?zero :: e]]
b -> [nil & [[[nil :: eb] & a!nil :: ea] & a?zero :: e]]
a -> [nil & [[[nil :: eb] & nil :: ea] & zero :: e]]
    = [[zero :: e]]

```

```
= zero
```

The last equality holds because the behaviour of `zero` exports only offers from `e` in any case.

For those familiar with the CCS example upon which this is based, note that no relabelling operator is necessary to make our version work. Instead, the local agent with the behaviour `b?a!nil` is used to simulate such an operator.

Note also that infinite event classes are not required here to model an unbounded stack.

It is now a trivial exercise to model a Turing machine using two stacks to represent the infinite tape. (See also [Milner 1989, p. 135].) Thus either an encapsulation mechanism or infinite event classes suffice to make Abacus computationally complete.

5 Behavioural Patterns

Our stated goal for this work was to develop a semantic target for the specification and prototyping of concurrent language constructs. Until now, we have concentrated on the notation itself and on the underlying computational model. Since we do not yet have a computationally complete notation, the modeling power of Abacus is until now somewhat limited. Nevertheless, we can give a flavour of how we plan to use Abacus with a simple example.

Consider a small language for evaluating Boolean expressions of the form:

```
boolExp :: true
         | false
         | boolExp or boolExp
         | boolExp and boolExp
         | ( boolExp )
```

We would like `and` to bind more closely than `or`, and we would like independent subexpressions to be evaluated concurrently.

Let us consider some behavioural patterns that can be used to implement our Boolean expressions. The following is a template for temporary Boolean variables:

```
bool(N,X) := (N,X)!nil
```

```

+ (N,and,true)?bool(N,X)
+ (N,and,false)?bool(N,false)
+ (N,or,true)?bool(N,true)
+ (N,or,false)?bool(N,X) .

```

In lieu of event classes, we have made use of Prolog's uninstantiated variables to define the class of `bool()` behaviours. `N` stands for names of `bool()` variables, and `X` for values. We declare an instance, for example, `bool(a,true)` by binding `N` and `X`. This variable either outputs its value as `(a,true)` and terminates, or accepts to update its value through an `and` or an `or` request.

The template:

```

op(O,B1,Op,B2) := (B2,X) ? (B1,Op,X) ! O ! nil .

```

stands for an operator agent with name `O`, that obtains the value of Boolean variable `B2`, and applies it with operator `Op` (either `and` or `or`) to variable `B1`. It announces completion by outputting its name and terminating.

The `join` behaviour waits for two (concurrent) operator agents `E1` and `E2` (i.e., executing subexpressions) to complete, in either order, and then applies operator `Op` on the variables now containing the values of the subexpressions:

```

join(O,E1,B1,Op,E2,B2) := E1 ? E2 ? op(O,B1,Op,B2)
                        + E1 ? E2 ? op(O,B1,Op,B2) .

```

`done()` simply waits for the last operator to terminate:

```

done(O) := O?nil .

```

We would now like to map the syntactic patterns of Boolean expressions to the behavioural patterns of Abacus. Consider, for example, the expression

```

( true or false ) and ( false or true )

```

This could be mapped to the behaviour net `bexp` declared as follows:

```

bexp := bool(t1,true) & op(o1,t1,or,t2) & bool(t2,false)
      & join(o3,o1,t1,and,o2,t3)
      & bool(t3,false) & op(o2,t3,or,t4) & bool(t4,true)
      & done(o3) .

```

The event graph of one of the two possible resulting computations is shown in figure 5 (either of `o1` or `o2` could terminate first). In either case the final configuration is `bool(t1,true)`.

At present we include no support for encapsulating behaviour patterns as syntactic patterns, though we view such support as essential for Abacus to realize its potential as a language prototyping tool. We envisage the definition of semantic functions as a set of rules based either on attribute grammars [Madsen, 1980; Kastens, et al. 1982] or on unification, as in TXL [Cordy, et al. 1988] or Prolog.

6 Concluding Remarks

We have presented our first results on a computational model and notation for specifying concurrent language constructs. The notation as it stands is computationally equivalent to Petri nets. We are presently considering the addition of event classes, parameterized behaviour declarations and encapsulation of subsystems to extend the power and expressiveness of the notation.

Our computational model draws heavily from CCS [Milner 1980]. The main difference is that Milner is interested primarily in formalizing notions of behavioural equivalence, whereas we wish to use of our model as a way of capturing the operational semantics of various concurrent language constructs. As a consequence, our approach is inherently more operational rather than algebraic. For example, it is hard to see how the relabeling operator of CCS could be realized in an implementation. Although CCS ostensibly supports dynamic creation of new processes, this issue is all but ignored (with the exception of the “chaining” example referred to earlier). Milner hints at the possibilities of using CCS as a semantic target in chapter 9 of [Milner 1980] on “Translating into CCS,” and also in chapter 8 of [Milner 1989] on “Defining a Programming Language.”

Hoare’s CSP [Hoare 1985] also bears comparison to Abacus (and to CCS), however Hoare forbids dynamic process creation, as this poses difficulties for his proof system, and is, in any case, not central to his discussion. Implementability of CSP has been considered and is demonstrated through fragments of Lisp code. The synchronization mechanism of CSP differs from that of Abacus or CCS, since multi-party synchronization is supported, rather than simply two-party communication. It is not clear to us that this is either desirable or necessary, though we see no other way of modeling broadcast protocols.

Actor models, as described in [Agha 1986], differ from both CSP and CCS in that communication is

asynchronous rather than synchronous. Actors may send messages to other actors and continue to execute independently of when the messages are delivered. An event in an actor system constitutes the receipt of a message by an actor, the creation of a set of new messages to be delivered, and the replacement of the actor by its new behaviour. Dynamic creation of new actors is emphasized. As a consequence, an infinite supply of new actor names is needed. Given infinite event classes, we can simulate actors within Abacus by representing both actors and messages as agents: the job of an actor agent is to accept a message and replace itself by a set of new actors and message agents; the job of a message agent is to deliver itself to an actor, and then terminate. We believe that the synchronous behaviour net model we have presented is simpler than the actor model since it comprises fewer concepts (just agents and events, instead of actors, messages and events), and more suitable for general modeling of concurrency constructs, since synchronous mechanisms can be directly represented, and asynchronous mechanisms are, in any case, inherently indirect. Nevertheless, the Actor model has had a profound influence on the design of concurrent object-oriented languages, so the fact that we can capture actor models is an encouraging sign.

In addition to improving the functionality and convenience of the Prolog implementation, we are studying the formal properties of behaviour expressions. In particular, we would like to characterize behavioural patterns according to:

- standard patterns: how should data structures, control statements, synchronization mechanisms, objects, actors, etc. be represented?
- control flow: how do “threads” of events flow between agents throughout a computation?
- efficiency: what kinds of patterns are easy to implement on various machine architectures?
- fairness: what kinds restrictions guarantee fair computations?
- equivalence: can notions of behaviour equivalence (as in [Milner, 1980]) help in compiling specifications?

Finally, as our approach is decidedly operational, it will be interesting to explore the applicability of design methods, such as [Chandy and Misra 1988].

Acknowledgements

I would like to thank Laurent Dami, Peter Wegner and Barbara Pernici for suggestions that led to improvements in the presentation of this paper.

A Abacus Implementation

```

/*
    Working version #3 of prolog-abacus
*/

% Load standard predicates: ~, in, cat, rev, deal, ord, writeln
:- [-plib].

/*
    Re-define operator precedence to conform to Abacus syntax:
*/
:- op(840,xfx,needs).
:- op(690,xfx,:=).
:- op(680,xfy,&).
:- op(460,xfy,[!,?]).
/*
    valid(B) -- B is valid if it is:
        1. uniquely defined
        2. well-guarded
        3. well-formed
*/
valid(B) :-
    uses(B,N),
    % unique(N), % relax uniqueness to allow variables in RHS
    wg(B,N),
    wf(B,N).

/*
    A behaviour expression is uniquely defined if all
    the behaviour names used have a unique definition.
*/
unique(N) :-
    P in N,
    P := D1, P := D2, D1 \== D2, !,
    writeln(['Error: ',P,' multiply defined']), fail.
unique(_).
/*
    A behaviour expression is (recursively) well-guarded
    if it uses no circularly defined behaviour names.
    E.g., p := q. q := p. -- can't generate guards.
*/
wg(B,N) :- defloop(B,N,C), !,
    writeln(['Error: ',B,' uses circular definition ',C]), fail.
wg(_,_).
/*
    A behaviour expression is well-formed if all
    sequential behaviours (formed with "+")

```

```

do not contain any unguarded parallel behaviours.
E.g., ((p&q) + r) is not well-formed.
*/
wf(B,N) :- malformed(B,N,0), !,
    writeln(['Error: ',0,'] is both sequential and parallel']),
    fail.
wf(_,_).

/*
    uses(B,N)      -- N is the list of behaviour names
                   used (recursively) within B
*/
uses(B,N) :- bnames([B],[],N), !.
bnames(_!B|L,T,N) :- bnames([B|L],T,N).
bnames(_?B|L,T,N) :- bnames([B|L],T,N).
bnames([B1 + B2|L],T,N) :- bnames([B1|[B2|L]],T,N).
bnames([B1 & B2|L],T,N) :- bnames([B1|[B2|L]],T,N).
bnames([nil|L],T,N) :- bnames(L,T,N).
bnames([P|L],T,N) :-
    P in T, !,          % already in T, so must be defined
    bnames(L,T,N).
bnames([P|L],T,N) :-
    P := B,            % new name, so add to list
    bnames([B|L],[P|T],N).
bnames([],N,N). % stop when no more behaviours to traverse
bnames([P|L],T,N) :-
    writeln(['Error: ',P,' undefined']), fail.

/*
    defloop(B,N,C) -- C is a sequence of circularly-defined behaviours
    The needs relation defines a digraph. We look for a circuit in
    the digraph by trying all possible walks from all possible nodes
    till we find a circuit + fail. (The circuit is extracted from
    the walk, since we may not loop to the beginning of the walk.)
*/
defloop(B,N,C) :-
    P in N,
    P needs Q,
    findloop(~P~Q,W~S),
    hascircuit(S,W,C).
findloop(I~S,I~S) :- S in I, !.
findloop(I,W) :-
    I = J~Q,
    Q needs S,
    findloop(I~S,W).
% P~X has a circuit C if X occurs in P
hascircuit(X,~X,~X).
hascircuit(X,P~X,~X).

```



```

hascircuit(X,P~Y,C~Y) :- hascircuit(X,P,C).
/*
    P needs Q      -- P needs Q to generate its guards
    NB: this is used in the backtracking step of defloop()
*/
P needs Q :-
    P := D,
    [D] needs Q.
[_!B|L] needs Q :- L needs Q.
[_?B|L] needs Q :- L needs Q.
[B1 + B2|L] needs Q :- [B1|[B2|L]] needs Q.
[B1 & B2|L] needs Q :- [B1|[B2|L]] needs Q.
[nil|L] needs Q :- L needs Q.
[Q|L] needs Q :- Q := _, !.

/*
    Search for malformed behaviour expressions in B,
    or in any definition used by B.
*/
malformed(B,_,0) :- bad_seq(B,0), !.
malformed(B,N,0) :-
    P in N,
    P := D,
    bad_seq(D,0).

/*
    Recursively search all sequential behaviour
    expressions for the use of unguarded parallelism.
*/
bad_seq(B,0) :-
    or_expr([B],0),
    par_beh([0]).

/*
    or_expr(L,0)      -- 0 is of the form "B1 + B2" within
                       the behaviours in list L.
    NB: We do NOT expand behaviour names at this stage!!!
        (To do so would put Prolog in an infinite loop.)
*/
or_expr([_!B|L],0) :- or_expr([B|L],0).
or_expr([_?B|L],0) :- or_expr([B|L],0).
or_expr([B1 & B2|L], 0) :- or_expr([B1|[B2|L]],0).
or_expr([B1 + B2|_],B1 + B2). % found one!
or_expr([_|L],0) :- or_expr(L,0).

/*
    par_beh(L)       -- L contains a parallel behaviour
    NB: may only be applied to well-guarded behaviours
*/
par_beh([_!_|L]) :- par_beh(L). % guarded, so discard
par_beh([_?_|L]) :- par_beh(L).

```

```

par_beh([B1 + B2|L]) :- par_beh([B1|[B2|L]]). % not sure yet
par_beh([nil|L]) :- par_beh(L).
par_beh([P|L]) :-
    P := B,          % well-guarded, so must terminate!
    par_beh([B|L]).
par_beh([_&_|L]) :- !. % got it!

/*
    END OF SEMANTIC CHECKS
*/

/*
    normal(B,S)      -- S is a list of sequential agents whose
                      parallel composition gives B.
*/
normal(A,[A]) :- isseq(A), !.
normal(P, S) :- P := D, normal(D, S). % decompose
normal(nil&B, S) :- normal(B, S). % discard
normal(A&B, [A|S]) :- isseq(A), !, normal(B,S).
normal(P&B, S) :- P := D, normal(D&B, S).
normal((B1&B2)&B, S) :-
    normal(B1&B2&B, S).
normal(nil, []).

/*
    isseq(B)        -- B is a sequential behaviour expression
    NB: assumes that B is both well-guarded and well-formed!
*/
isseq(!_!).
isseq(!_?).
isseq(_ + _).
isseq(P) :- P := B, isseq(B).

/*
    compose(S,B)    -- construct a behaviour expression B
                      from list of parallel agents S
*/
compose([A],A) :- isseq(A), !.
compose([A|S],A&B) :- compose(S,B).
compose([],nil).

/*
    step(S,E,SR)    -- E may fire in system S, yielding SR
*/
step(S,E,SR) :-
    cat(S1,[P|S2],S), % select sender P
    output(P,E,PR),
    match(E,S1,S2,SR1,SR2), % find a matching receiver
    normal(PR,PN),

```

```

        cat(PN,SR2,T),          % replace P
        cat(SR1,T,SR).
% receiver might be in either S1 + S2
match(E,S1,S2,SR1,S2) :- fire(E,S1,SR1).
match(E,S1,S2,S1,SR2) :- fire(E,S2,SR2).
fire(E,S,SR) :-
    cat(S1,[Q|S2],S),          % select receiver Q
    input(Q,E,QR),             % match output E
    normal(QR,QN),
    cat(QN,S2,T),              % replace Q
    cat(S1,T,SR).

/*
    output(B,E,R)  -- E is an output with replacement R in agent B
*/
output(E!R,E,R).
output(B + _,E,R) :- output(B,E,R).
output(_ + B,E,R) :- output(B,E,R).
output(P,E,R) :- P := B, output(B,E,R).
/*
    input(B,E,R)   -- E is an input with replacement R in agent B
*/
input(E?R,E,R).
input(B + _,E,R) :- input(B,E,R).
input(_ + B,E,R) :- input(B,E,R).
input(P,E,R) :- P := B, input(B,E,R).

/*
    The main predicate: compute and display the
    first execution path.
*/
abcpath(B) :-
    valid(B),
    normal(B,S),
    write('Initial configuration:'), nl,
    writeagents(S),
    write('First execution path:'), nl,
    firstpath(S,F),
    write('Final configuration:'), nl,
    writeagents(F).

/*
    Compute first execution path & stop
*/
firstpath(S,F) :-
    step(S,E,SR), !,
    compose(SR,B),
    writeln([ '      ', '(,E,) -> ' ]),

```

```

    firstpath(SR,F).
firstpath(F,F).

writeagents([A|[S|R]]) :-
    writeln([ '      ',A,' &' ]),
    writeagents([S|R]) .
writeagents([A|[]]) :-
    writeln([ '      ',A,' .' ] ) .

/*
    List all events and replacements for the current step.
*/
allevents(S) :-
    step(S,E,SR),
    compose(SR,B),
    writeln([ '(,E,) -> ',B ]),
    fail.
allevents(_).

abcall(B) :-
    valid(B),
    normal(B,S),
    write('Initial configuration:'), nl,
    writeagents(S),
    allpaths(S).

/*
    List all execution paths.
    (May not terminate...)
*/
allpaths(S) :-
    write('Searching for all execution paths:'), nl,
    findpath(S,P,R),
    compose(R,B),
    writeln([ P,' -> ',B ]),
    fail.
allpaths(_).

/*
    Find a complete execution path P.
    Warning: assumes all paths terminate!
*/
findpath(S,P,R) :-
    step(S,E,SR),
    cont(~E,SR,P,R).
cont(I,S,P,R) :-
    step(S,E,SR),
    cont(I~E,SR,P,R).

```

```
cont(P,R,P,R) :- not(step(R,E,_)).
```

B Standard Predicates

```
/*
    PLIB : library of useful Prolog predicates.

    CONTENTS:

    X in L          -- X is in list L
    X in S          -- X is in sequence S

    cat(L1,L2,L)   -- L is the concatenation of L1 and L2
    rev(X,Y)       -- list X is Y reversed
    deal(X,Y,Z)    -- dealing X yields Y and Z

    ord(C,N)       -- N is ascii for char C
    writeln(L)     -- write list of names with newline

*/

/*
    LIST PREDICATES
*/
:- op(850,xfx,in).
:- op(550,yfx,~). % lower priority than =
:- op(540,fx,~).
/*
    Sequences are like lists, except they grow to the right.
    A sequence MUST start with a ~.
    Examples:
                ~a      -- sequence containing a
                ~a~b    -- sequence (a,b)
                S~x     -- append x to S
                x~S     -- garbage
                a~b     -- garbage

*/
isseq(~_). % single element sequence
isseq(S~_) :- isseq(S). % multiple sequence

/*
    Two forms -- if X is instantiated or not
    X in L          -- selects X from L
    X in L,!       -- test if X is in L
*/
```

```

*/
X in [X|_] .
X in [_|Y] :- X in Y.

/*
    X in S          -- select X from sequence S
    X in S,!        -- test if X is in L
*/
X in ~X .
X in _~X .
X in S~_ :- X in S .

cat([],L,L).
cat([A|X],Y,[A|Z]) :- cat(X,Y,Z).

/*
    deal(X,Y,Z)     -- dealing X yields Y and Z
                    e.g., [a,b,c] -> [b,c],[a] etc.
    NB: will not terminate if solution deal(X,[],Y) is rejected.
*/
rev(X,Y) :- deal(X,[],Y), !.
deal(X,X,[]) .
deal(X,Y,[I|Z]) :- deal(X,[I|Y],Z).

/*
    I/O PREDICATES
*/
% Look up ascii code or vice versa:
ord(C,N) :- name(C,[N]).

writeln([]) :- nl.
writeln([X|L]) :- write(X), writeln(L).

```

References

- [1] G.A. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Massachusetts, 1986.
- [2] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.
- [3] J.R. Cordy, C.D. Halpern and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects," Proceedings of The International Conference of Computer Languages, pp. 280-285, Miami, FL, Oct 9-13, 1988.

- [4] M.J.C. Gordon, *The Denotational Description of Programming Languages*, Springer-Verlag, 1979.
- [5] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.
- [6] U. Kastens, B. Hutt and E. Zimmermann, *GAG: A Practical Compiler Generator*, Lecture Notes in Computer Science 141, Springer-Verlag, 1982.
- [7] D. Konstantas, O.M. Nierstrasz and M. Papathomas, "An Implementation of Hybrid, a Concurrent Object-Oriented Language," in *Active Object Environments*, ed. D.C. Tschritzis, pp. 61-105, Centre Universitaire d'Informatique, University of Geneva, June 1988.
- [8] O.L. Madsen, "On Defining Semantics by Means of Extended Attribute Grammars," in *Semantics-Directed Compiler Generation*, ed. N.D. Jones, Lecture Notes in Computer Science 94, pp. 259-299, Springer-Verlag, 1980.
- [9] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, 1980.
- [10] R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.
- [11] O.M. Nierstrasz, "Active Objects in Hybrid," ACM SIGPLAN Notices, Proceedings OOPSLA 1987, vol. 22, no. 12, pp. 243-253, Dec 1987.

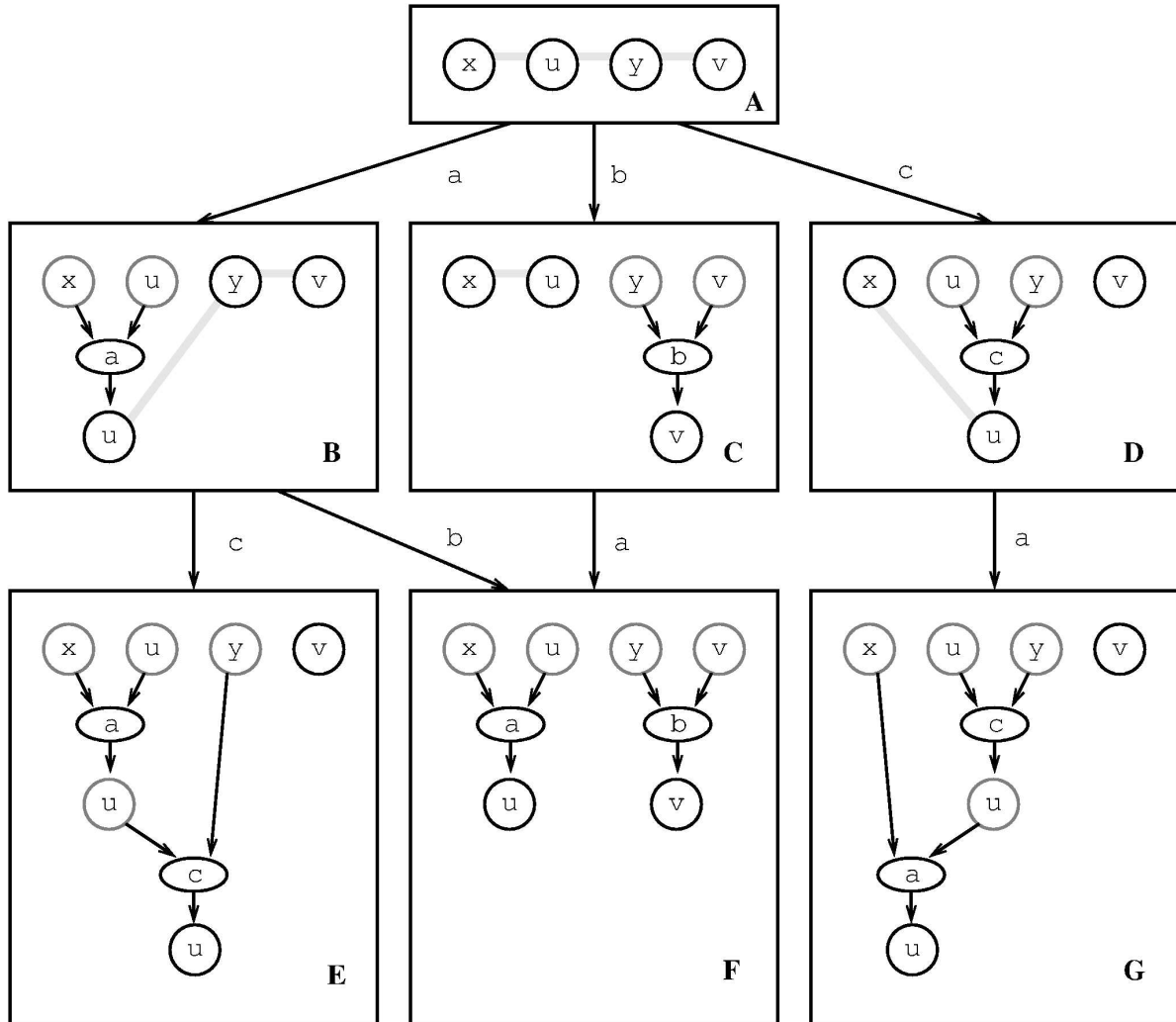


Figure 2: Non-determinism vs Concurrency

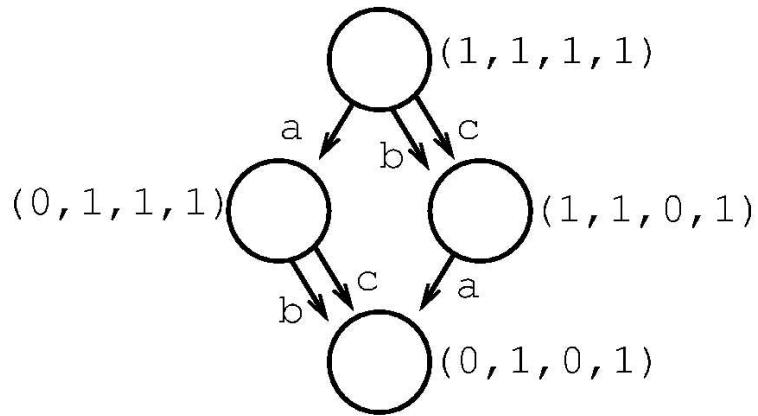


Figure 3: The Finite State Automaton representation of $x \& u \& y \& v$

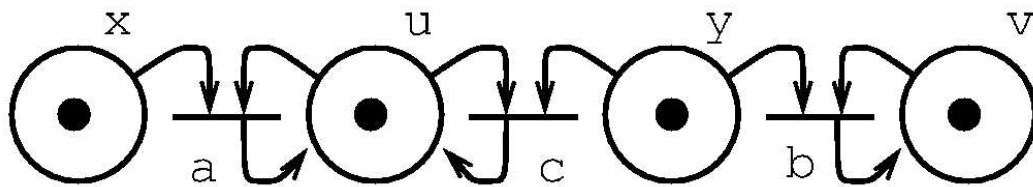


Figure 4: The Petri Net representation of $x \& u \& y \& v$

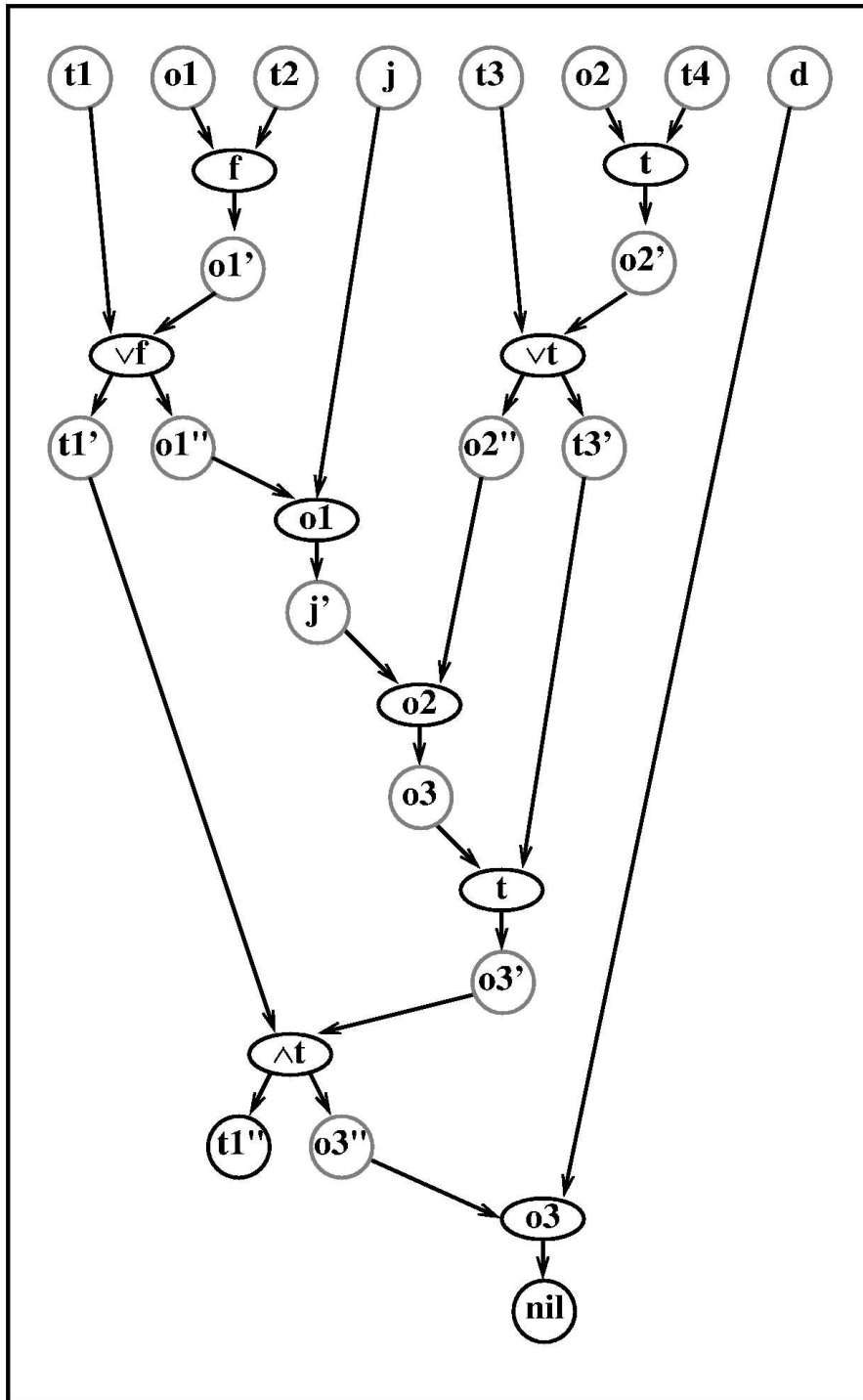


Figure 5: Concurrent Boolean Agents