A Method to Generate Occam Skeletons from Formal Specifications

I.M. Ikram

Computer Science Department, Rhodes University,
P.O.Box 94, Grahamstown, 6140.
E-mail: *csii@cs.ru.ac.za*

Abstract

We consider the problem of automatically generating parallel programs in the occam language. In particular, we are concerned with the generation of program *skeletons* that define just those computations that are common to a number of similar systems, leaving empty or 'stub' functions or processes in the code which are to be filled in later in an application-specific manner.

Rather than attempt to solve the problem for arbitrary parallel systems, we focus on *data-parallel* systems composed of iterative processes. In this paper, we propose an implementation strategy for a cellular automaton. Only uniprocessor occam code is considered.

1 Introduction

In this paper we consider the problem of programming a simulation of such systems as cellular automata (CA) in the concurrent programming language occam [2]. Implementations of CA are quite common, for instance the one proposed by Brinch-Hansen [1]. Our motive here is to demonstrate the viability of a certain formal programming method ('formal' in the sense of being amenable to automation) by applying it to the task of implementing CA in occam.

The programming method under investigation attempts to generate code for inter-process communication, given a high-level specification or description of a concurrent system (the form and content of the specification are explained in the next section). The rest of the code, we assume, is developed manually. Communication code is analogous to a skeleton, in that it provides a fixed foundation upon which the main computational code of a concurrent system is build. Changes in the computational component normally do not require modification to the communications skeleton.

According to the occam model of communication, data is transmitted between processes though unidirectional, point-to-point and synchronous channels. For nontrivial systems, planning and managing inter-process communication can contribute significantly to the 'difficulty' of parallel programming. Thus the skeleton approach aims firstly at the establishment of a working inter-process communication network, usable, perhaps, by a variety of applications. Thereafter code for application-specific computations may be added. These ideas have been extended in the parallel-programming method known as 'algorithmic skeletons.'

The work presented here is directed towards the development of an *algorithmic skeleton* [3] based programming system for the class of so-called 'complex systems' applications of which CA are a particularly simple example (by virtue of their regular structure and simple component processes). We work with occam because of the close correspondence between the occam programming model and the typical features of 'complex systems' such as replicated components and purely local communications.

In Section 2 we present the theoretical foundation of the programming method and the associated formal notation. In section 3 we briefly introduce CA and proceed to develop an occam implementation of a simple instance. Finally in section 4 we conclude with a description of our current research in this area.

2 Model

2.1 Introduction

We proceed to describe the adopted model of concurrent systems. The model has been designed to aid in decomposing a system of interest into its component objects, to specify the possible states in which those objects may exist, the operations that transform their state, and their communications topology. A final aspect of the model allows sequential dependencies between operations to be specified. It will be seen in section 3 that this last aspect is central to the proposed code generation mechanism and permits us to reason about the system's mutual exclusion properties.

2.2 Objects

A *system* is a collection of *objects*. Informally, an object corresponds directly to a unique entity in the problem domain, therefore a unique *identifier* may be associated with each object. Formally, a system composed of N objects having identifiers $o_1, o_2, \ldots, o_N$ is denoted by the set $\{o_1, o_2, \ldots, o_N\}$. For convenience, we will hereafter refer to an object by mentioning its unique identifier.

Objects are only capable of communicating with other objects via occam-style channels. Disjoint subsets of a system may be identified and each one referred to as a *class*.

2.3 State

Objects encapsulate *state*. At any time, an object exists in a unique state. If a class is said to have *state-sets* $S_1, S_2, \ldots, S_N$, then the *instantaneous state* of any of its members is represented by an element belonging to the product $S_1 \times S_2 \times \ldots \times S_N$.

In the ensuing discussion, frequent reference will be made to the notion of instantaneous state of objects. The notation used to denote an object at a particular instant will be as follows: we associate the object o (belonging to some class with state-sets $S_1, S_2, \ldots, S_N$ within some system) with its instantaneous state as the tuple $(o, (s_1, s_2, \ldots, s_n))$, where $s_i \in S_i$ for all i, when its state is $(s_1, s_2, \ldots, s_n)$. Individually, $s_1, s_2, \ldots, s_n$ are referred to as the *attributes* of o.

The instantaneous state of a system is the set of all the system's objects associated with *their* current states.

2.4 Topology

The *topology* of a system S is represented by a subset of the pairs in the product $S \times S$. If T is such a subset then the pair of objects (o_1, o_2) is an element of T if and only if o_1 passes messages to o_2. A topology may be visualized as a directed graph with nodes corresponding to objects and edges orientated in the direction of communication between pairs of objects.

2.5 Operations

2.5.1 Pre- and Post-state

Objects may change their state, perform input or output as part of an *operation*. Essentially, we adopt the same philosophy as the Z notation [4] in specifying an operation by indicating both its pre-state and post-state and then asserting a relationship between those two states. In our usage, an operation performed on the objects of system S will be given as a function mapping the instantaneous pre-state, written $S\prime$ to instantaneous post-state, written $S\prime\prime$.

For the sake of illustration, suppose that S consists of a number of objects belonging to the same class and possessing the single state-set $\{0, 1\}$. Then an operation O setting the state of $o \in S$ to some function f of the pre-state of o is written as the following mapping from $S\prime$ to $S\prime\prime$:

$$O : (o, a) \rightarrow (o, f.a) \qquad (o \in S, a \in \{0, 1\})$$

A more complicated operation may involve 3 distinct objects in S: o_1, o_2 and o_3, and set the state of o_3 to some function g of the pre-states of o_1, o_2 and o_3. We would write such an operation P as a mapping from $S\prime \times S\prime \times S\prime$ to $S\prime\prime$:

$$P : (o_1, a), (o_2, b), (o_3, c) \rightarrow (o_3, g.(a, b, c))$$
$$(o_1, o_2, o_3 \in S, o_1 \neq o_2 \neq o3, a, b, c \in \{0, 1\})$$

In cases such as P above, where the operation depends upon the pre-state of more than one object, it is conventional to write the pre-state of the object undergoing state-change as the last argument. The range-set of an operation on system S has to be $S\prime\prime$, and not a product, as is permissible for the domain-set. This is to say that all operations must be so defined as to alter the state of a single object.

Note that it is possible to examine such definitions of operations and deduce both the flow of data from object to object and also the identity of the single object whose state is altered by the operation. With O, the argument and result objects were identical (o) so no communication is involved. With P, the objects o_1, o_2 and o_3 are the arguments and o_3 the result, so we have implicitly stated a requirement that the states of o_1 and o_2 be communicated to o_3 prior to state-change.

2.5.2 Order of Actions

We will need to express formally the *order* in which operations are required to take place in a system of interest. Supposing the operation P defined in the previous section were to be performed firstly on the object z, taking inputs from x and y. Supposing thereafter that exactly the same operation is to be repeated and that finally, P is to be performed on x taking inputs from objects y and z. The following notation may be employed, for example, with the operator '$\longrightarrow$' standing for the binary precedence relationship:

$$P_{x,y,z} \longrightarrow P_{x,y,z}$$
$$P_{x,y,z} \longrightarrow P_{y,z,x}$$

$P_{x,y,z}$ and $P_{y,z,x}$ are called *instances* of operation P. Operationally, an instance, say $P_{x,y,z}$, is meant to signify that some parts of the current state of x and y are output to z, resulting in a change to the state of system S.

The above notation is unsatisfactory as the second line is ambiguous as to whether the instance $P_{y,z,x}$ should take place after the first or the second occurrence of $P_{x,y,z}$ (or even after both). We resolve this by 'time-stamping' each instance of an operation, by an occurrence-count, as follows:

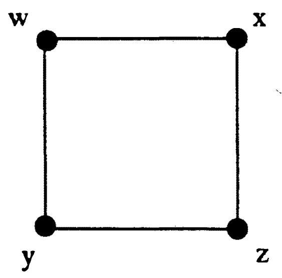

Figure 1: One-dimesional cellular automaton consisting of four cells.

$$P_{x,y,z,1} \longrightarrow P_{x,y,z,2}$$
$$P_{x,y,z,2} \longrightarrow P_{y,z,x,1}$$

Instances so written are called *actions*.

3 Cellular Automaton

3.1 Description

We approach the problem of generating communication code for a cellular automaton (CA) implementation by specifying the CA according to the model and notation presented in section 2, and going on to apply general rewriting rules according to information extracted from the specification.

The CA to be modelled here is a ring of four cells, connected so that each cell has two neighbouring cells (see figure 1). At any time each cell can be in a state represented by an element of some non-empty set A. All cells undergo a fixed number, G, of state-transitions according to a state-transition rule. The rule in this instance is given by some function f mapping both the received state of neighbouring cells and the current state of the cell onto the next state of the cell.

In contrast to most familiar cellular automata (e.g. the Game of Life), we do not insist that the state-transitions be globally synchronized among all the cells. State-transitions may be asynchronous, placing this CA in the class of asynchronous cellular automata.

3.2 Specification

The CA described in the previous section will be considered a system, C, composed of objects with identifiers w, x, y and z. Objects correspond directly to

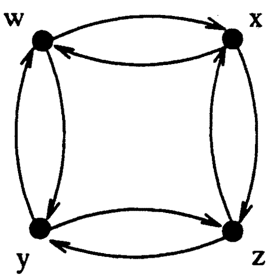

Figure 2: Toplogy of the cellular automaton system. Each edge represents a member of the set *Topology*.

cells. The objects are assigned two attributes: the first, taking values from A, gives the current output of the cell (this corresponds to what has been called the state of the cell, above); and the second, taking values from $A \times A$, gives the two values received from neighbouring cells.

The topology is a doubly-linked ring consisting of eight object-pairs (see figure 2).

We distinguish two operations upon the state of C, corresponding exactly to the updating of the two attributes of each object: the first applies f to the values received from the neighbours of an object and sets its new output to be the result:

$$\begin{aligned} O : (o, out, (in_1, in_2)) \; &\rightarrow (o, f \cdot (in_1, in_2, out), (in_1, in_2)) \\ &(o \in C, out, in_1, in_2 \in A) \end{aligned}$$

This is to be interpreted as saying that operation O, when performed on an object o in system C, sets the first attribute of o (its current output) to be the result of applying state-transition function f to both the received state of neighbours (in_1 and in_2) as well as o's previous output out.

The second operation upon C alters the second attribute of its objects according to the following definition:

$$\begin{aligned} I : (o_1, out_1, (a, b)), (o_2, out_2, (c, d)) \; &\rightarrow (o_2, out_1, insert \cdot (o_1, o_2, out_1, (c, d))) \\ &((o_1, o_2) \in Topology, out_1, out_2, a, b, c, d \in A). \end{aligned}$$

This operation takes two arguments, so it maps $C\prime \times C\prime$ to $C\prime\prime$, as explained in section 2. The above definition states that I takes object o_1 as argument and effects a state-change in object o_2, such that o_1's current output, out_1, is added to o_2's second attribute. The auxiliary function *insert* is employed to decide which obsolete input value, either c or d above, is to be replaced by *out*$_1$. To take an example, we may decide that the two components of the second attribute of objects are to be arranged thus: the first component will contain the value received from the left neighbour as represented in figure 1, while the second components will be similarly related to the right neighbour. Then $\textbf{\textit{insert}} \cdot (w, x, 1, (0, 0))$ would give $(1, 0)$ and $\textbf{\textit{insert}} \cdot (z, x, 1, (0, 0))$ would give $(0, 1)$.

Having defined the operations, we turn to task of enumerating the operation instances that may occur in the system, as described in section 2.5.2. For operation O we know that there are four possible instances, one corresponding to each object, thus we have O_w, O_x, O_y and O_z. For I there are eight instances, for each pair in *Topology* ($I_{w,x}, I_{x,z}, I_{z,y}; I_{y,w}$ and their reversed counterparts).

From the system description given in section 3.1 the number of state-transitions required of each cell is G. Since O is the operation effecting state-transition, there must be G occurrences of each of the four O-instances. We conclude that there will be $4G$ O-actions, namely $O_{w,i}, O_{x,i}, O_{y,i}$ and $O_{z,i}$ for all $1 \le i \le G$.

Similarly, there will be G occurrences of each I-instance because every O-instance must be preceded by the two I-instances corresponding to the input of state from two neighbours. Thus we have $8G$ I-actions.

We are now equipped with sufficient notation to state the relative timing of actions in four simple rules: the first two state precedence relations that are true by definition:

$$O_{o,i} \longrightarrow O_{o,i+1} \qquad (\forall i : 1 \le i < G, \forall o \in C)$$

and

$$I_{o_1,o_2,i} \longrightarrow I_{o_1,o_2,i+1} \qquad (\forall i : 1 \le i < G, \forall (o_1, o_2) \in \textit{Topology})$$

More interestingly, we can state that:

$$I_{o_1,o_2,i} \longrightarrow O_{o_2,i} \qquad (\forall (o_1, o_2) \in \textit{Topology}, \forall i : 1 \le i \le G)$$

That is, object o_1 should output its current state to o_2 before o_2 may update *its* state. The final rule states that, after having undergone its i-th state-transition, o_1 is to output its new state to o_2:

$$O_{o_1,i} \longrightarrow I_{o_1,o_2,i+1} \qquad (\forall (o_1, o_2) \in \textit{Topology}, \forall i : 1 \le i < G)$$

Note that the occurrence count of the I-action in this last rule is one greater that the occurrence count of the preceding O-action. This is because we count a single iteration of a cell as consisting (firstly) of input of neighbouring cells' state and (secondly) update of local state.

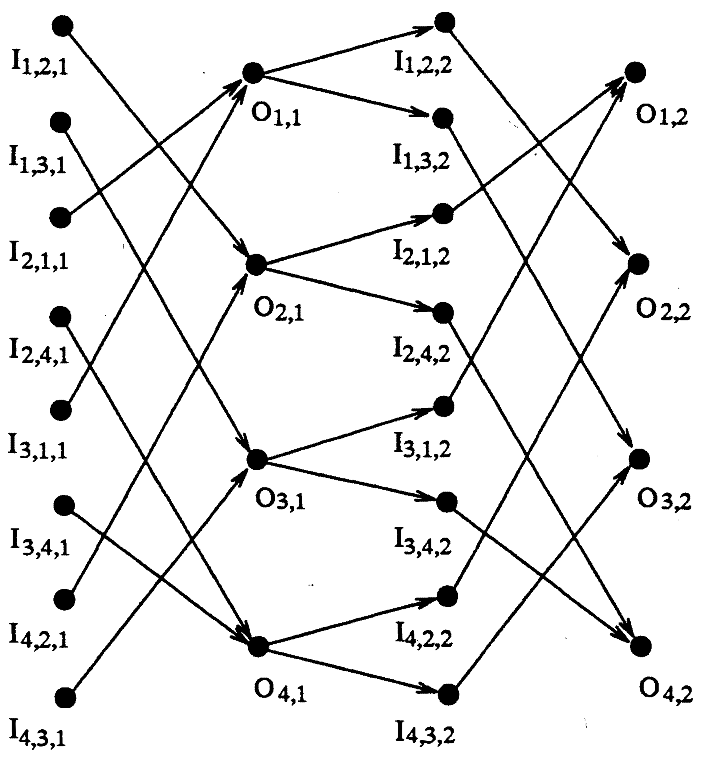

Figure 3: Directed acyclic graph of actions in system C, for $G = 2$. Labelled edges represent actions while edges represent action-precedence constraints.

The information stated in these rules may be presented alternatively as a directed acyclic graph (DAG) of actions, for example figure 3 (cf. *event-structures* [5]).

3.3 Program Skeleton

Using the precedence information derived in the above specification, we now decompose the system into a collection of concurrent processes. These processes have already been identified in the specification as cell objects. Thus in the case of the present CA we have four cell processes. The communication channels connecting these processes are apparent from Topology, in fact there is a one-to-one correspondence between the edges of that set and the required channels. Using meaningful identifiers, the top level occam skeleton for the CA is of the form (where P is the appropriate channel protocol):

```
PROC cell(CHAN OF P from.left, to.left, from.right, to.right)
  ... body
:
CHAN OF P w.to.x, x.to.w, ..., y.to.w, w.to.y :
PAR
  cell(y.to.w, w.to.y, x.to.w, w.to.x) – Cell w.
  cell(w.to.x, x.to.w, z.to.x, x.to.z) – Cell x.
  cell(z.to.y, y.to.z, w.to.y, y.to.w) – Cell y.
  cell(x.to.z, z.to.x, y.to.z, z.to.y) – Cell z.
```

As an object-based decomposition has been followed, the variables used by each process are simply the attributes possessed by the respective objects. Again, using meaningful identifiers, the following declarations may be made at the top of the body of procedure cell (where T is the appropriate variable type):

```
T left, right : – The received state of neighbours.
T state : – Local state
... rest of procedure cell's body
```

Before proceeding to expand the remainder of this procedure, further examination of the operations O and I is required. From the definition of O above, its effect on the state of a cell is to assign the local state attribute (the first attribute) to the result of applying the state-transition function. We can deduce that each O-action may be implemented as an assignment statement:

```
state := f(left, right, state) – Compute the next state.
```

The definition of operation I implies the input from one neighbour of its current state, followed by assignment to either one of the components of the second attribute. For each pair of outputting and inputting objects, depending on the definition of the auxiliary function *insert*, one of the following two processes is appropriate:

```
PAR
  to.right ! state – Performed by outputting cell.
  from.left ? left – Performed by inputting cell.
PAR
  to.left ! state – Performed by outputting cell.
  from.right ? right – Performed by inputting cell.
```

Having established the range of possible statements composing the procedure cell, we go on to order the statements using the action-precedence DAG. Seeing that the only actions without precedent in the DAG are those I-actions with occurrence-count 1, and that there are two such actions for each cell (representing input from each neighbour, respectively), we deduce that the first process in procedure cell should input neighbours' state (along with complementary outputs to neighbours):

```
PAR
    to.left ! state
    to.right ! state
    from.left ? left
    from.right ? right
```

As a general principle, actions not connected by a path in the DAG are permitted to execute concurrently, as there is no danger of breaking mutual exclusion from state attributes.

From the DAG, immediately following a pair of I-actions is an O-action effecting state-transition:

```
PAR
    to.left ! state
    to.right ! state
    from.left ? left
    from.right ? right
state := f(left, right, state)
...
```

This completes the first iteration of a cell. As the DAG has a regular and repeating structure, further expansion of the body of procedure cell reveals identical code to the above, repeated G times.

4 Conclusion

We have developed a prototype of a system that translates specifications such as given in section 3.2 into a set of precedence relationships. Work is under way to automate the next step (demonstrated in section 3.3) of actually generating occam skeletons from those precedence relationships and from further information given in specifications.

We have found that the specification model described in section 2 to be adequate for the task of specifying a number of systems within our domain of interest (notably various models of neural networks and parallel genetic algorithms). These systems are of a data-parallel nature, consisting of simple replicated, iterative processes.

The scheme presented in this paper has a number of undesirable properties and deficiencies that we are seeking to eliminate or correct. Firstly, operations refine to either assignment statements or input/output statements, but not to looping constructs. Thus, as seen in section 3.3, the size of generated code grows with the parameter G, the number of iterations required. A mechanism is required to detect regularities in the action-precedence DAG and deduce the possibility of 'rolling' repeated code into loop statements. Secondly, in practice it is often not possible to specify a value for G, as we would like the system to iterate until some termination condition is satisfied. Thirdly, due to considerations of execution speed, it may be desirable to sequentialize some of the code and thereby reduce the number of parallel processes occupying a single processor. We are therefore interested in integrating automated sequentialization strategies into this scheme.

A future research goal is to embed sufficient intelligence into the code generation system to exploit true parallel processing on a network of processes.

References

[1] Per Brinch Hansen. Parallel cellular automata: A model program for computational science. *Concurrency: Practice and Experience*, 5(5):425–448, August 1993.

[2] Alan Burns. *Programming in occam 2*. Addison-Wesley, 1988.

[3] Murray Cole. *Algorithmic Skeletons: a Structured Approach to the Management of Parallel Computation*. Pitman, 1989.

[4] J.M. Spivey. *The Z Notation*. Prentice Hall, 1989.

[5] Glynn Winskel. An introduction to event structures. In J.W. Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 364–397. Springer-Verlag, 1989.