Towards Provably Correct Hardware/Software Partitioning Using Occam

Edna Barros Augusto Sampaio
Depto. de Informatica - UFPE *
Caixa Postal 7851 - Cidade Universitaria
CEP 50732-970 Recife - PE - Brazil
{ensb,acas}@di.ufpe.br

Abstract

In this paper we present some ideas towards an approach to provably correct hardware/software partitioning. We use occam as the source programming language and perform the partitioning by applying a series of algebraic transformations on the source program. The result is still an occam program; its structure reflects the hardware and software components, and how they interact to achieve the overall goal. A simple case study is developed to illustrate the partitioning and to show how the transformations can be proved to preserve an algebraic semantics of occam.

1 Introduction

Hardware/Software codesign is the design of systems comprising two kinds of components: specific application components and general programmable ones. The latter kind is referred here as software component whereas the former is also called hardware component.

An essential aid for hardware/software codesign is the availability of approaches to hardware/software partitioning. Some partitioning approaches have been proposed by De Micheli [7], Ernst [3] and Barros [1]. One of the main challenges of approaches to hardware/software partitioning is the increasing number of implementation alternatives to be analysed. The approach proposed by Barros supports a better exploration of the design space permitting that the whole description be analysed. Additionally, distinct implementation possibilities of hardware components are considered during the partitioning process [1].

Although some approaches to hardware/software partitioning have been suggested, and even some of them have been implemented, an open problem remains for most approaches: the formal verification that the partitioning preserves the semantics of the original description. A provably correct design of a partitioning algorithm is a challenge in itself and to our knowledge none of the mentioned approaches has been formally developed.

This paper aims to propose some ideas towards a partitioning approach whose emphasis is correctness. The proposed method uses occam as a description language and performs the partitioning of an occam program by applying a series of transformations to obtain a set of parallel processes, one of which will be implemented in software and the others in hardware. The transformations as well as the strategy which guides their applications are based on the partitioning approach proposed by Barros [2, 1].

There are many reasons for choosing occam as the source programming language. It was developed from CSP [9], and like CSP occam obeys a large set of algebraic laws [11] which can be used to carry out program transformation with the preservation of semantics. Some interesting applications of program transformation using occam have been reported. In [6] a general-purpose system is described which allows mechanised transformations of occam programs. Basically, it implements the laws given in [11] and an algorithm which reduces programs to a restricted syntax through the application of laws.

Reduction to a restricted syntax (or a *normal form*) has proved useful for many applications. For example, a provably correct compiler for a sequential subset of occam is described in [10]. Another application is explored in [8] which presents a normal form approach to FPGA implementation of occam programs.

Hardware/software partitioning can also be regarded as an application of program transformation. The algebraic laws can be used to transform an arbitrary source program into a program whose structure will reflect the software and hardware components, and how they interact to achieve the overall goal. It is possible to express this in occam because it includes features to express parallelism and communication.

This paper is organised as follows: after a short description of a subset of occam and of some of its algebraic laws, an informal description of the partitioning process is given. The partitioning approach is then illustrated by a simple case study. This is followed by some conclusions and proposals for future work. Finally, in an appendix we briefly illustrate how some transformations are proved to preserve an algebraic semantics of occam.

*This work was partially supported by the brazilian research council CNPq

0-8186-6315-4/94 $04.00 © 1994 IEEE

2 A Language of Communicating Processes

The partitioning approach described in this paper should allow an arbitrary occam program as input. Nevertheless, at present it deals only with a subset of the language. For example, procedures and arbitrary loops are not considered, although we deal with a particular case of iteration through the use of *replicators*, as explained below.

For convenience we sometimes linearise occam syntax in this paper. For example, we may write `SEQ(p1,...,pn)` instead of the standard vertical style. The subset of occam used here is defined by the following BNF-style syntax definition, where [*clause*] has the usual meaning that *clause* is an optional item.

```
P ::= SKIP | STOP | x := e | ch ? x | ch ! e
    | IF(c1 p1,...,cn pn) | SEQ [rep] (p1,...,pn)
    | PAR [rep] (p1,...,pn) | TYPE x1,...,xn: p
    | CHAN OF TYPE ch1,...,chn: p
```

Informally, these processes behave as explained below.

- `SKIP` has no effect and always terminates successfully. `STOP` is the canonical deadlock process which can make no further progress. `x := e` is executed by evaluating the expression `e` and then assigning its value to the variable `x`.

- `ch ? x` and `ch ! e` are the input and the output commands, respectively. They allow parallel processes communicate through channels. The communication occurs when a given process `p` is willing to receive a message and another process `q` is ready to send a message through the same channel (*synchronous* communication).

- `IF` is a conditional command which takes a list of arguments of the form `ci pi`, where `ci` is a boolean expression and `pi` is a program. It is the first (that is, lowest index) boolean guard to be true that activates the corresponding `pi`.

- `SEQ(p1,p2,...,pn)` denotes the sequential composition of processes `p1,p2,...,pn`. If the execution of `p1` terminates successfully then the execution of `p2` follows that of `p1`, and so on, until `pn` is executed.

- `PAR(p1,p2,...,pn)` stands for the parallel composition of processes `p1,p2,...,pn`. These processes run concurrently, with the possibility of communication between them. Communication is the only way two parallel processes can affect one another, so one parallel process cannot access a variable that another one can modify.

- *TYPE* `x1,...,xn: p` declares the local variable `x1,...,xn` for use in the process `p`. *TYPE* may be any type available in occam; here we will use only integer (`INT`) variables.

- `CHAN OF` *TYPE* `ch1,...,chn: p` declares the local channels `ch1,...,chn` for use in the process `p`. Like variables, channels are also typed, and here we will be confined to integer channels.

The optional argument *rep* which appears in the `SEQ` and in the `PAR` constructs stands for a replicator of the form `i = m FOR n` where `m` and `n` are integer expressions. Replicators allow the construction of array of processes, and they are especially useful to deal with array variables. For example,

```
SEQ i = 0 FOR 2 (a[i]:=b[i])
```

is the same as

```
SEQ(a[0]:=b[0], a[1]:=b[1], a[2]:=b[2])
```

A similar idea applies to the use of replicators in parallel composition.

Algebraic Laws

In order to carry out program transformation we need a semantics in addition to the above syntax of the occam operators. A set of laws which completely characterise the semantics of `WHILE`-free occam programs is given in [11]. In the following we select a very small subset of these laws which will be used to illustrate that hardware/software partitioning in the style suggested in this paper may be proved to be a semantic-preserving transformation process.

The `SEQ` operator runs a number of processes in sequence. If it has no arguments it simply terminates.

```
L1 SEQ( ) = SKIP
```

Otherwise it runs the first argument until it terminates and then runs the rest in sequence. Therefore it obeys the following associative law.

```
L2 SEQ(p1,p2,...,pn) = SEQ(p1,SEQ(p2,...,pn))
```

It is possible to use the above two laws to transform all occurrences of `SEQ` within a program to binary form. Thus the next laws for `SEQ` are cast in binary form.

Evaluation of a condition is not affected by what happens afterwards, and therefore `SEQ` distributes leftward through a conditional.

```
L3 SEQ(IF(c1 p1,...,cn pn), q) =
   IF(c1 SEQ(p1,q),...,cn SEQ(pn,q))
```

Assignment distributes rightward through a conditional, changing occurrences of the assigned variables in the condition. We use `c[e/x]` to stand for the substitution of `e` for every occurrence of `x` in `c`.

```
L4 SEQ(x:=e, IF(c1 p1,...,cn pn)) =
   IF(c1[e/x] SEQ(x:=e,p1),...,
      cn[e/x] SEQ(x:=e,pn))
```

A `PAR` command terminates as soon as all its components have; the empty `PAR` terminates immediately.

```
L5 PAR( ) = SKIP
```

PAR is an associative operator.

```
L6 PAR(p1,p2,...,pn) = PAR(p1,PAR(p2,...,pn))
```

As with SEQ, we can use the above laws to reduce PAR to a binary operator; thus the rest of the laws deal only with this case.

PAR is symmetric because the order in which processes are combined in parallel is immaterial.

```
L7 PAR(p1,p2) = PAR(p2,p1)
```

If one of a pair of parallel processes is a conditional, then the choice represented by that conditional may be performed before the parallel construct is entered, provided the choices are exhaustives[1].

```
L8 PAR(IF(c1 p1,...,cn pn), q) =
   IF(c1 PAR(p1,q),...,cn PAR(pn,q))
        provided (c1 ∨...∨ cn) = true
```

If a parallel component starts by executing an assignment, it can proceed immediately. This is valid because occam does not allow processes communicate by using shared variables.

```
L9 PAR(SEQ(x:=e, p), q) = SEQ(x:=e, PAR(p,q))
```

As described earlier, communication in occam is synchronous. When two parallel processes synchronise to communicate through a given channel, the effect is to assign the value sent by the output process to the variable of the input process.

```
L10 PAR(SEQ(ch ? x, p), SEQ(ch ! e, q)) =
    SEQ(x := e, PAR(p,q))
```

The scope of a *bound* variable[2] may be increased without effect, provided it does not interfere with another variable with the same name. Thus each of the occam constructors has a distribution law with declaration. As illustration we give below the law for PAR.

```
L11 PAR(VAR x: p, q) = VAR x: PAR(p, q)
        provided x is not free in q
```

Similar laws are valid for channel declarations.

3 The Hardware/Software Partitioning Approach

In this section we describe the partitioning approach through its main concepts. Initially the underlying target architecture is presented; this is followed by a description of the main steps of the partitioning process.

[1] The reason for this restriction is that if no choice is true the conditional behaves like STOP, and this would prevent the PAR being entered.

[2] If p is some occam term and x is a variable, we say that an occurrence of x in p is *free* if it is not in the scope of any declaration of x in p, and *bound* otherwise.

3.1 The underlying target architecture

The target architecture underlying the hardware/software partitioning approach can be seen in Figure 1. The target architecture can be considered as pre-defined since there is only one software component. The hardware components on the other hand can exhibit distinct degrees of parallelism. Due to the fact that parallelism in occam is based on message passing, the memory will be distributed, unlike the target architecture taken in the original partitioning approach proposed by Barros [1]. Each set of processes kept in the same cluster will have its local variables stored in the component where it is allocated into.

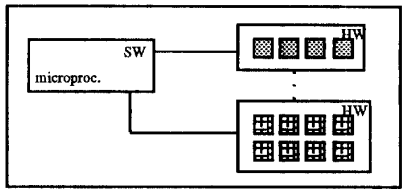

Figure 1: The underlying target architecture.

Variables declared by processes kept in different clusters will be transferred through message passing by using named channels. Each channel will be named according to the following rule: a channel named pi.pj interconnects process pi with process pj and the message flows from process pi to process pj.

3.2 The Approach

The hardware/software partitioning is based on the approach proposed by Barros [1]. This approach was developed considering UNITY specifications but it can be applied to other description languages, and to occam in particular. In order to take into account the target architecture presented earlier, some modifications were made in the cost functions guiding the clustering process. One of them is the consideration of communication cost due to message passing.

The main tasks associated with the partitioning approach are depicted in Figure 2. The tasks represented by normal boxes belongs to the original approach and will be explained shortly in this section. The tasks represented as dashed boxes are new and were included in the partitioning approach in order to allow the verification of the correctness of the partitioning process. Such tasks will be explained in Section 4.

According to Figure 2 before the clustering process takes place, the set of implementation alternatives is established during the *Classification* phase by considering distinct degrees of parallelism when implementing the original program. The set of implementation alternatives is represented by a set of class values concerning various features of the program, such as concurrent behaviour, data dependences, multiplicity, non-determinism and mutual exclusion.

The clustering process takes into account only one alternative. The choice of some implementation alternative as the current one can be made manually

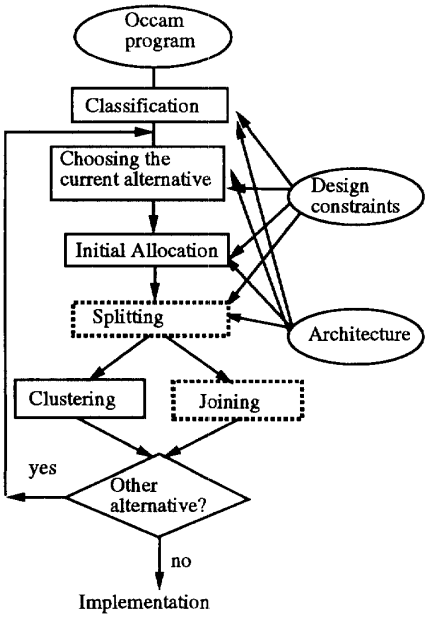

Figure 2: The main tasks of our partitioning approach.

or automatically. When choosing automatically, the alternatives leading to a balanced degree of parallelism among the various statements and minimising the area-delay cost function will be taken as the current one.

Once a current implementation alternative was chosen for each process, the clustering takes place. The first step is the allocation of some process to the control unit, which means the implementation of such process in software (see the underlying target architecture presented in Section 3.1. The allocation process can also be controlled by the user or can be made automatically guided by a cost function. The degree of parallelism of the current implementation as well as the minimisation of the area delay cost function will be considered when analysing the software implementation of each process.

The partitioning of an occam program into a process to be implemented in software and processes implemented directly in hardware is achieved by a multistage hierarchical clustering algorithm, which is based on the algorithm described in [1]. The cost functions have been slightly modified in order to take into account the communication overhead introduced for synchronisation purposes.

At the first stage clusters are built according to the similarity of the functionality of the processes and the similarity of the degree of parallelism exhibited by their current implementation alternative.

To build the cluster tree, a metric upon processes (and their implementation alternatives) were defined according to the following guidelines such as: processes exhibiting similar parallelism degree are kept closer to each other, processes with data dependency are kept close to minimise synchronisation cost, mutually exclusive processes exhibiting distinct degree of parallelism are kept separate.

The function $D(e_i, e_j)$ measuring the distance between the processes e_i and e_j is defined by Eq.(1). The term $f_{Hierarchy}(e_i, e_j)$ has been introduced in order to take into account the communication cost introduced when sequential processes with data dependences have been split into parallel ones.

$$\begin{aligned} D(e_i, e_j) &= D_{Ass.Type}(e_i, e_j) + D_{ClassVal.}(e_i, e_j) \\ &+ f_{Hierarchy}(e_i, e_j) \end{aligned} \quad (1)$$

Distances between each pair of processes build a distance matrix, from which a cluster tree can be built. The algorithm for building the cluster tree from a distance matrix can be found in [1]. For all clustering sequences it is analysed whether resource should be shared or not according to the minimisation of the area/delay cost function given by Eq.(2).

$$AreaDelay = \alpha \ln(Area) + \beta \ln(Delay) \quad (2)$$

The detection of the cut line at the clustering sequence s of the cluster tree c is guided by the cost function defined by Eq.(3):

$$\begin{aligned} f_{cut1}(s, c) = f_{PipelineFlag}(s, c) + f_{CUAlloc}(s, c) + \\ f_{AreaDelay}(s, c) \end{aligned} \quad (3)$$

The term $f_{PipelineFlag}(s, c)$ suggests that sequential processes not suitable for a pipelining implementation be kept in a cluster ($f_{PipelineFlag}(s, c) = 1$) while those entitled to pipelining and those assigned to different functional units be kept separate ($f_{PipelineFlag}(s, c) = \infty$). This function is calculated concerning the costs of a pipelining implementation against the obtained speedup.

The function $f_{CUAlloc1}(s, c)$ causes processes preallocated to the control unit (i.e. implemented in software) to be separate from processes assigned to hardware. As already mentioned, good candidates for software allocation are processes exhibiting least data dependences (to reduce the communication cost). Among these processes, those which include parallel assignments are more favoured since they could be executed in a single clock cycle in a super scalar processor. Eventually, processes conformed to both the above conditions and at the same time minimising the area/delay cost are allocated to the control unit.

Finally, $f_{AreaDelay}(s, c)$ evaluates the area/delay cost concerning both dynamic and static resource sharing returning ∞ when resources are not shared. The algorithm runs in a bottom-up fashion starting with the clustering sequence generated by the splitting phase until $f_{cut1}(s+1, c) > f_{cut1}(s, c')$.

At the second stage, a new distance matrix for the clusters (resulting from stage 1) is established and from it a cluster tree is built. The goal of the clustering at the second stage is to determine whether asynchronous clusters will be implemented as pipelining, since pipelining is also accompanied by additional delays. The generated clusters are finally allocated to the underlying target architecture.

4 A Simple Case Study

The partitioning is carried out using a set of program transformation rules whose application sequence is controlled by the clustering process. Considering the block diagram described in Figure 2, one might see that the first step is the splitting of the description into a set of communicating processes. After that, the joining of processes in clusters takes place when building the clustering tree and placing the cut line at each clustering stage.

Here we illustrate this method through the partitioning of a simple occam program into hardware and software components. The strategy is to perform a series of algebraic transformations on the original program until we reach a program which models precisely the behaviour of our target architecture. The presentation below includes the macro steps of the development process. The aim is to give an overview of the kind of transformation employed. Strictly, every single transformation must be justified by one algebraic law; in Appendix A we show how some of the steps can be carried out formally.

Our source program implements the convolution given by Eq.(4) [4].

$$y_i = \sum_{j=1,n} x_{i-j} \times w_j \times \alpha_i, 1 \leq i \leq 2n-1 \qquad (4)$$

where w_i is given by $w_{j+1} = b \times x_1 \times w_j$, and α_i is given by $\alpha_i = c_i + d_i$ with

$$c_i = \begin{cases} x_i & \text{if } x_i \geq 0 \\ \frac{x_i}{2} & \text{if } x_i < 0 \end{cases} \qquad d_i = \begin{cases} x_{i+1} & \text{if } x_{i+1} \geq 0 \\ \frac{x_{i+1}}{2} & \text{if } x_{i+1} < 0 \end{cases}$$

The variables x_i, y_i and w_i are global, with the values of x_i and w_1 being given by the environment. The variable y_i will be used by the environment. The environment is not relevant here and for the sake of simplicity it is omitted in the occam description below. Recall that we sometimes linearise occam syntax, as explained in Section 2.

```
INT c, d :
[5] INT e :
SEQ i = 0 FOR 2
  PAR
    IF (x[i] >= 0 c := x[i],                    (1)
        x[i] < 0  c := x[i]/2)
    IF (x[i] >= 0 d := x[i+1],                  (2)
        x[i] < 0 d := x[i+1]/2)
    PAR j = 0 FOR 4
      e[j] := x[5 * (i/(j + ((j + 1)/(i + 1))))
              +(j - i)] * w
  PAR
    w := k * e[i]
    PAR j = 0 FOR 4
      y[j] := y[j] + e[j] * (c + d)
```

4.1 Splitting an Occam Program into a Set of Concurrent Processes

As mentioned earlier, the first step in the partitioning process is the splitting of an initial occam program into a set of processes. Before the splitting takes place the current implementation alternative must be chosen for each process in the original program. Furthermore, some process must be allocated to the control unit.

According to the criteria described in the previous section, either process (1) or process (2) could be implemented in software. This analysis was made by considering the degree of parallelism exhibited by the current implementation of each process and by analysing the data dependences among them. For both processes, the degree of parallelism is equal to one and the number of data dependences between these and other processes is minimal, leading to a minimisation of communication costs. To start with we choose process (1). The result of this transformation is given below. We assign informally an identifier to each process for further reference.

```
CHAN OF INT p1.p4, p2.p4, p4.p3 :
CHAN OF [5] INT p3.p4 :
PAR
  -- software process p1
  INT c :
  SEQ i = 0 FOR 2
    IF (x[i] >= 0 c := x[i], x[i] < 0 c := x[i]/2)
    p1.p4 ! c
  -- process p2
  INT d :
  SEQ i = 0 FOR 2
    IF (x[i] >= 0 d := x[i+1],
        x[i] < 0  d := x[i+1]/2)
    p2.p4 ! d
  -- process p3
  [5] INT e :
  INT w :
  SEQ i = 0 FOR 2
    p4.p3 ? w
    PAR j = 0 FOR 4
      e[j] := x[5*(i/(j+((j+1)/(i+1))))+(j-i)]*w
    p3.p4 ! e
  -- process p4
  INT c, d :
  [5] INT e :
  SEQ i = 0 FOR 2
    p4.p3 ! w
    PAR
```

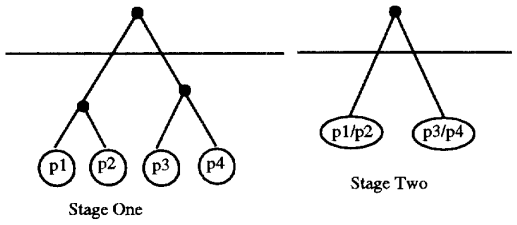

Figure 3: The clustering phase

```
  p1.p4 ? c
  p2.p4 ? d
  p3.p4 ? e
PAR
  w := k * e[i]
  PAR j = 0 FOR 4
    y[j] := y[j] + e[j] * (c + d)
```

According to this program, the synchronisation between processes is achieved by introducing channels; this allows message passing between processes with data dependences. For example, there is a data dependency between the processes (p1) and (p4). The synchronisation between the two processes is achieved by introducing a channel, p1.p4, in order to permit message passing between them. As one might see, the splitting of a description into concurrent processes leads to an increasing of the communication costs. During the clustering phase this cost can be minimised by keeping processes with data dependences in the same cluster. This is equivalent to apply rules for joining such processes.

4.2 The clustering guiding the joining of process

The clustering algorithm groups processes in clusters by building a clustering tree followed by the placement of a cut line at some level of the tree. As mentioned earlier, the criteria guiding the clustering algorithm include similarity of the functionality and of the implementation alternative (degree of parallelism) associated with each process. Communication and synchronisation costs are also taken into account by considering data dependences between processes. The parallelism improvement through pipelining implementations as well as the associated overhead are also criteria considered by the clustering approach. The clustering process of the occam description presented previously is depicted in Figure 3.

When building the clustering tree at stage one according to Eq.(1), the processes p1 and p2 are the first to be clustered (in this case they are serialised). The reason is that the current implementation alternative of both processes exhibits the same degree of parallelism and process p2 is also a good candidate to be implemented in software. Furthermore, keeping both processes in the same cluster minimises the area-delay cost function.

```
-- serialisation of p1 and p2
INT c, d :
SEQ i = 0 FOR 2
  IF (x[i] >= 0 c := x[i], x[i] < 0 c := x[i]/2)
  IF (x[i] >= 0 d := x[i+1],
      x[i] < 0  d := x[i+1]/2)
  p1.p4 ! c
  p2.p4 ! d
```

The process p3 is not affected by this transformation. The process p4 could be slightly changed to take the above transformation into account: as c and d are now produced in sequence, the commands p1.p4 ? c and p2.p4 ? d could be executed in sequence as well. This brings no significant gain and is not performed here.

The partitioning criteria described earlier also suggests that the processes p3 and p4 should be grouped into a single cluster. This eliminates the communication between these processes since the array e and the variable w can now be shared by p3 and p4. Furthermore, both processes execute serially and the similarity of the current implementation alternative of both processes (i.e. their degree of parallelism) as well as the similarity of their functionality (both processes performs multiplication and addition) suggests that resource can be shared between these two processes without delay increase.

The cluster tree is cut according to the minimisation of the cost function given by Eq.(3) resulting into two clusters: one to be implemented in software and another one to be implemented in hardware.

The result of the overall partitioning is given by the following occam program.

```
CHAN OF INT p1.p4, p2.p4, p3.p4 :
PAR
  -- software process (serialisation of p1 and p2)
  INT c, d :
  SEQ i = 0 FOR 2
    IF (x[i] >= 0 c := x[i], x[i] < 0 c := x[i]/2)
    IF (x[i+1] >= 0 d := x[i+1],
       x[i+1] < 0 d := x[i+1]/2)
    p1.p4 ! c
    p2.p4 ! d
  -- hardware process (serialisation of p3 and p4)
  INT c, d :
  [5] INT e :
  SEQ i = 0 FOR 2
    PAR
      p1.p4 ? c
      p2.p4 ? d
      PAR j = 0 FOR 4
        e[j] := x[5*(i/(j+((j+1)/(i+1))))+(j-i)]*w
    PAR
      w := k * e[i]
      PAR j = 0 FOR 4
        y[j] := y[j] + e[j] * (c + d)
```

5 Conclusions

In this paper we presented some ideas towards an approach to provably correct hardware/software partitioning using occam; this was illustrated by a case study. Although a simple example was considered, it illustrates several aspects of the partitioning process. It also allowed us to address correctness issues.

In the presented approach the hardware/software partitioning of an occam program has been performed by applying a set of rules on an initial occam description to obtain a set of processes, one of which is implemented in software and the others in hardware. The rules and the way they were applied were based on the partitioning approach proposed by Barros [1, 2].

But the partitioning is only part of a system development process. The software components generated by the process should then be correctly compiled using, for example, the algebraic method presented in [10, 12]. The hardware parts need also be correctly compiled into some hardware description language; this can be achieved by using, for example, the method presented in [8]. The advantage of using the partitioning technique suggested in this paper together with the software and hardware compilation methods cited above is that they are all based on the same semantic framework which is relatively simple: a programming language and its algebraic laws.

Although we believe that the approach is very promising, there is much work to be done to turn it into a well-defined method. We need to define a *normal form* to represent the result of the partitioning, and discover a set of transformation rules which must be complete in the sense that their application must allow an arbitrary program be transformed into this normal form. In particular, we need transformation rules for loops and procedures. Each of these rules must be proved from the more basic laws of occam. Furthermore, a more general target architecture model should be considered in the partitioning process.

The ultimate goal is to code the transformations as rewrite rules in an algebraic system such as OBJ3 [5]. This would enable us to mechanically check the correctness of each transformation; but most importantly, the rewrite rules would serve as a prototyping of a system to perform the partitioning automatically.

A Formal Justification of Some Transformations

As explained in Section 2, the purpose here is only to illustrate how some of the transformations can be formally justified. We do not have the tools yet for a detailed justification of every single transformation involved in the partitioning task of the case study presented in Section 4. This will require the search for more algebraic laws to deal with the operators not addressed in [11].

The first step which splits the original program into a number of communicating processes comprises a large number of transformations and will not be addressed here. But in the following we give a more detailed justification of the serialisation of processes p1 and p2.

The splitting process generates four parallel processes. As we are concerned with the serialisation of the first two processes, we can use the associativity of PAR (Law 6) and focus on these two processes.

```
PAR
  INT c :
  SEQ i = 0 FOR 2
    IF(x[i] >= 0 c := x[i], x[i] < 0 c := x[i]/2)
    p1.p4 ! c
  INT d :
  SEQ i = 0 FOR 2
    IF(x[i+1] >= 0 d := x[i+1],
       x[i+1] < 0  d := x[i+1]/2)
    p2.p4 ! d
```

As the variable c is not free in the second process and d is not free in the first one, we can use Law 11 to move their declarations out of the PAR construct. Furthermore, note that each of these processes is in fact an array of three processes which execute in sequence (as the replicator index varies from 0 to 2). For each iteration, both p1 and p2 execute a conditional which determines the value of a variable which should be sent to process p4. Then they both synchronise with p4 to send this value. For any value assumed by the replicator index the two conditionals are executed in parallel. Therefore in this particular case we distribute SEQ over PAR[3].

```
INT c, d :
SEQ i = 0 FOR 2
  PAR
    SEQ
      IF(x[i] >= 0 c := x[i],
         x[i] < 0  c := x[i]/2)
      p1.p4 ! c
    SEQ
      IF(x[i+1] >= 0 d := x[i+1],
         x[i+1] < 0  d := x[i+1]/2)
      p2.p4 ! d
```

Then we can apply Law 3 to distribute the SEQ operator leftward through each of the conditionals:

```
INT c, d :
SEQ i = 0 FOR 2
  PAR
    IF(x[i] >= 0  SEQ(c := x[i], p1.p4 ! c),
       x[i] < 0   SEQ(c := x[i]/2, p1.p4 ! c))
    IF(x[i+1] >= 0 SEQ(d := x[i+1], p2.p4 ! d),
       x[i+1] < 0  SEQ(d := x[i+1]/2, p2.p4 ! d))
```

Now we can apply Law 8 to distribute PAR over the first IF. This will cause the first IF to be the top level command and each of its branches will be in parallel with the second IF. Then we can use Law 7 (symmetry

[3] A more detailed justification needs to take into account the text of the process p4.

of **PAR**) and Law 8 again to distribute the internal occurrences of **PAR**. The result is

```
INT c, d :
SEQ i = 0 FOR 2
  IF(x[i] >= 0 IF(x[i+1] >= 0
                  PAR(SEQ(c:=x[i], p1.p4 ! c),
                      SEQ(d:=x[i+1], p2.p4 ! d))
                x[i+1] < 0
                  PAR(SEQ(c:=x[i], p1.p4 ! c),
                      SEQ(d:=x[i+1]/2, p2.p4 ! d)
     x[i] < 0 IF(x[i+1] >= 0
                  PAR(SEQ(c:=x[i]/2, p1.p4 ! c),
                      SEQ(d:=x[i+1], p2.p4 ! d))
                x[i+1] < 0
                  PAR(SEQ(c:=x[i]/2, p1.p4 ! c),
                      SEQ(d:=x[i+1]/2,p2.p4 ! d)))
```

Law 9 allows the serialisation of the assignment `c:=x[i]`, and we can use Law 2 to move it out of the first internal **IF**. The same laws justify moving the assignment `c:=x[i]/2` out of the second internal **IF**. This will cause the two internal **IF** commands to be identical; therefore we can apply Law 3 to unnest the internal **IF**s. These transformations result in

```
INT c, d :
SEQ i = 0 FOR 2
  IF(x[i] >= 0   c := x[i],
     x[i] < 0    c := x[i]/2)
  IF(x[i+1] >= 0
       PAR(p1.p4 ! c,
           SEQ(d := x[i+1], p2.p4 ! d))
     x[i+1] < 0
       PAR(p1.p4 ! c,
           SEQ(d := x[i+1] / 2, p2.p4 ! d)))
```

Then we again serialise the assignments in the latter **IF** (using Law 9) and repeat the step above (Law 3):

```
INT c, d :
SEQ i = 0 FOR 2
  IF(x[i] >= 0 c:=x[i], x[i] < 0 c:=x[i]/2)
  IF(x[i+1] >= 0 d:=x[i], x[i+1] < 0 d:=x[i]/2)
  PAR(p1.p4 ! c, p2.p4 ! d)
```

The only remaining step is the serialisation of the two output commands above. Unlike assignment, there is no independent law to serialise a communication command, and a careless transformation may lead to *deadlock*. In the case study presented in Section 4 it is valid to transform the above parallel command into `SEQ(p1.p4 ! c, p2.p4 ! d)` or into `SEQ(p2.p4 ! d, p1.p4 ! c)`, because the corresponding input commands are in parallel (see process **p4** in Section 4).

References

[1] E. Barros. *Hardware/Software Partitioning using UNITY*. PhD thesis, Universität Tübingen, 1993.

[2] E. Barros, X. Xiong, and W. Rosenstiel. Hardware/Software Partitioning with UNITY. In *Handouts of International Workshop on Hardware-Software Co-design*, 1993.

[3] R. Ernst and J. Henkel. Hardware-Software Codesign of Embedded Controllers Based on Hardware Extraction. In *Handouts of the International Workshop on Hardware-Software Co-Design*, October 1992.

[4] G. Fox, M. Johson, and G. Lysenga. *Solving Problems on Concurrent Processors*. Prentice-Hall International Editions, 1988.

[5] J. Goguen *et al.* Introducing obj. Technical report, SRI International, 1993. To appear.

[6] M. Goldsmith. The oxford **occam** transformation system. Technical report, Oxford University Computing Laboratory, January 1988.

[7] R. Gupta and G. De Micheli. System-level Synthesis Using Re-programmable Components. In *Proceedings of EDAC*, pages 2–7, 1992.

[8] Jifeng He, I. Page, and J. Bowen. A provably correct hardware implementation of occam. Technical report, **ProCoS** Project Document [OU HJF 9/5], Oxford University Computing Laboratory, November 1992.

[9] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[10] C. A. R. Hoare, J. He, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.

[11] A. Roscoe and C. A. R. Hoare. The laws of **occam** programming. *Theoretical Computer Science*, 60:177–229, 1988.

[12] A. Sampaio. *An Algebraic Approach to Compiler Design*. PhD thesis, Oxford University Computing Laboratory, 1993.