ParTS
A Partitioning Transformation System

Juliano Iyoda, Augusto Sampaio, and Leila Silva

Departamento de Informática - UFPE
Caixa Postal 7851 - Cidade Universitária
CEP 50740-540 Recife - PE - Brazil
{jmi,acas,lmas}@di.ufpe.br

Abstract. This paper describes a system (ParTS) for automatic hardware/software partitioning of applications described in the concurrent programming language occam. Based on algebraic transformations of occam programs, the strategy guarantees, by construction, that the partitioning process preserves the semantics of the original description. ParTS has been developed as an extension of OTS — a tool implemented at Oxford University which allows one to apply basic algebraic laws to an occam program in an interactive way. ParTS extends OTS with elaborate transformation rules which are necessary for carrying out partitioning automatically. To illustrate the partitioning methodology and our system, a convolution program is used as a case study.

1 The Hardware/Software Partitioning Problem

The specification of a computer system is usually fully implemented as a software solution (executed in a general hardware like a microprocessor). On the other hand, some strong requirements (like performance or size) demand an implementation completely in hardware. Nevertheless, in between these two extremes, there are applications that favour a combined implementation with software and hardware components. This has become a recent trend in Computing called Hardware/Software Codesign, which has been widely adopted in the design of embedded systems.

The problem of how to divide a specification into hardware and software components, the hardware/software partitioning problem, raises at least two major and orthogonal problems: 1) How can the partitioning be done so that the result satisfies the efficiency requirements? 2) Does the final system execute its tasks according to the original specification?

The first question can be solved by heuristic algorithms and the second by formal verification that the partitioned system preserves the semantics of the original description.

Several approaches to hardware/software partitioning have been developed, as described, for example, in [2, 8, 12, 13]. All the approaches above emphasise the algorithmic aspects of hardware/software partitioning. More recently, some works have suggested the use of formal methods in the partitioning process, as

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. II, LNCS 1709, pp. 1400–1419, 1999.

reported, for example, in [1, 6, 7]. Although these approaches use formal methods to hardware/software partitioning, neither of them includes a formal verification that the partitioning preserves the semantics of the original description.

In [3] Barros and Sampaio presented some initial ideas towards a partitioning approach whose emphasis is correctness. This work was the seed of the PISH project, a co-design environment which is being developed by four Brazilian universities [4]. The project comprises all the steps from the partitioning of (an initial description of) the system into hardware and software components to the layout generation of the hardware.

Silva *et al.* [18, 19] further develop the ideas presented in [3] by giving a precise characterisation of the partitioning process as a program transformation task. These works apply algebraic rules to guarantee that the partitioned system has the same functionality of the original description.

The main purpose of this paper is to present an environment which implements the strategy described in [3, 18, 19] to provide automatic hardware/software partitioning. This environment, the Partitioning Transformation System — ParTS, is an extension of the Oxford occam Transformation System (OTS) [10] — a tool developed at Oxford University constructed to perform transformations of occam programs [16]. While the basic algebraic laws implemented in OTS are useful for program transformation in general, they express only simple transformations, and are not suitable to capture the partitioning problem. ParTS extends OTS with new transformation rules specific for the partitioning strategy adopted. Also, ParTS deals with new language constructs not addressed by OTS (see Section 3) and provides a new graphical user interface. The transformation rules are coded as functions in the SML [15] functional language and the strategy is also a function that applies the rules in an appropriate order. Then the final system generated by ParTS is derived from the application of several semantic-preserving rules which guarantees the correctness of the solution by construction.

The next sections are organised as follows. Section 2 presents a brief description of the occam language and some of its laws. Section 3 explains the strategy adopted to carry out the partitioning. The implementation issues of ParTS are described in Section 4 and a case study of a hardware/software partitioning (of a convolution program) is shown in Section 5. Finally, Section 6 summarises the contribution of this paper and discusses topics for further research.

2 A Language of Communicating Processes

The goal of this section is to present the language which is used both to describe the applications and to reason about the partitioning process itself. This language is a representative subset of occam. For convenience, we sometimes linearise occam syntax in this paper. For example, we may write `SEQ(`P_1, P_2,..., P_n`)` instead of the standard vertical style. The subset of occam adopted 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 [ rep](c1 P1, c2 P2,..., cn Pn)
    | ALT [ rep] (c1&g1 P1, c2&g2 P2,..., cn&gn Pn)
    | SEQ [ rep] (P1, P2,..., Pn)
    | PAR [ rep] (P1, P2..., Pn)
    | WHILE c P
    | VAR x: P
    | CHAN ch: P
```

Informally, these processes behave as explained in what follows. The `SKIP` construct has no effect and always terminates successfully. `STOP` is the canonical deadlock process which can make no further progress. The commands `x := e`, `ch ? x` and `ch ! e`, are assignment, input and output commands, respectively; the communication in occam is synchronous. The commands `IF` and `ALT` select a process to execute, based on a condition (`IF`) or on a guard (`ALT`). While `IF`'s conditions are always boolean expressions, `ALT`'s guards involve input commands. `IF`'s selection is deterministic; the lowest index boolean condition to be true activates the corresponding process. If none of the conditions is `TRUE` it behaves like `STOP`. On the other hand, `ALT`'s selection is non-deterministic and randomly activates the process corresponding to the first guard to be satisfied. If more than one guard is satisfied at the same time, `ALT` activates non-deterministically one of the corresponding processes. If none of the guards is no satisfied `ALT` behaves like `STOP`. The commands `SEQ` and `PAR` denote the sequential and parallel composition of processes, respectively. Processes within a `PAR` constructor run concurrently, with the possibility of communication between them. Communication is the only way two parallel processes can affect one another, so (when combined in parallel) one process cannot access a variable that another one can modify. The command `WHILE` denotes a loop which executes a process until the `WHILE`'s condition becomes false. The constructs `VAR` and `CHAN` declare local variables and channels, respectively. Here we avoid mentioning a particular type for the declared variables or channels. The optional argument **rep** which appears in the `IF`, `ALT`, `SEQ` and `PAR` constructors stands for a replicator of the form `i = m FOR n` where `m` and `n` are integer expressions. A more detailed description of these commands can be found in [16].

As shown in [17], there are many algebraic laws which hold of the occam constructs. Such laws change the syntax of a program but preserve its semantics. A set of algebraic laws which completely characterises the semantics of WHILE-free occam programs is given in [17]. In this section we present only a few of these laws for the purpose of illustration.

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

Law 2.1 (*SEQ-SKIP unit*) `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.

Law 2.2 (*SEQ-assoc)* `SEQ(P1, P2,..., Pn) = SEQ(P1, SEQ(P2, P3,..., Pn))`

It is possible to use the above laws to transform all occurrences of `SEQ` within a program to binary form.

`PAR` is an associative operator.

Law 2.3 (*PAR-assoc)* `PAR(P1, P2,..., Pn) = PAR(P1, PAR(P2, P3,..., Pn))`

As with `SEQ`, we can reduce all occurrences of `PAR` to a binary form. The next law shows that the order in which the processes are combined in parallel is not important (`PAR` is symmetric).

Law 2.4 (*PAR-sym)* `PAR(P1, P2) = PAR(P2, P1)`

3 The Partitioning Approach

The hardware/software partitioning approach considered in this work performs the partitioning by applying a set of algebraic rules to the original system description. This is carried out in four major phases, as captured by Figure 1 and explained below.

Splitting The initial description of the system (written in occam) is transformed into the parallel composition of a number of *simple* processes. The formal definition of a simple process is given in [18], but it is enough to think of it as a process with granularity of a primitive command, possibly as a branch of a conditional (`IF`) or choice (`ALT`) statement.

Classification A set of implementation alternatives for each simple process is established by considering some features such as concurrent behaviour, data dependency, multiplicity, non-determinism and mutual exclusion.

Clustering Among the implementation alternatives of each process, one is chosen based on the minimisation of an area-delay cost function. The simple processes are grouped in clusters according to the similarity of functionality and the degree of parallelism. Each cluster groups simple processes that will be implemented in hardware or in software (this is determined by annotations). The fact that the simple processes generated by the splitting are in parallel gives full flexibility for this phase: as `PAR` is symmetric (Law 2.4) all possible permutations can be analysed.

Joining The processes in each cluster are effectively combined (either in sequence or in parallel), as determined by the result of the clustering process.

It is worthwhile mentioning that the phases that use algebraic transformations are splitting and joining. It has been proved that the use of algebraic rules in these phases preserves the semantics of the system while the program is being transformed [18, 19]. The classification and clustering phases implement heuristics to produce an efficient final system and the produced output is a mere permutation of the simple processes inside the `PAR` construction. Note that this

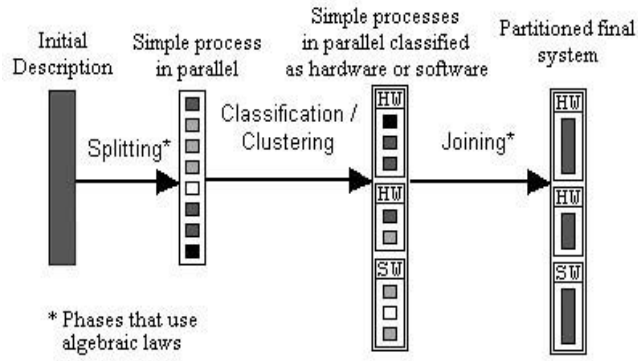


Fig. 1. The partitioning approach

procedure also does not affect the system behaviour once the `PAR` is associative and symmetric, as captured by laws 2.3 and 2.4.

ParTS is concerned only with splitting and joining since these are implemented by program transformation. In the rest of this section we describe these two phases in more detail. More information about the other two phases and about the tool which implements them can be found in [4]. In any case we will make clear how ParTS interacts with this tool to generate the partitioned system.

3.1 The Splitting Strategy

To improve flexibility concerning user interaction, the subset of occam presented in Section 2 was extended to consider new constructors: `BOX`, `HBOX`, `SBOX` and `CON`.

The syntax of these constructors is `BOX P`, `HBOX P`, and so on, where `P` is a process. The introduction of these constructors in occam has no semantic effect; they can be regarded just as annotations, useful not only for the splitting, but also for the other phases.

A process included into a constructor `BOX` is not split and its cost is analysed as a whole at the clustering phase. The `HBOX` and `SBOX` constructors denote a `BOX` which must be implemented in hardware and in software, respectively. They are used to raise the granularity level of the splitting phase when this happens to be convenient for a given application.

The constructor `CON` is an annotation for a *controlling process*; this is further explained in this section.

The goal of the splitting phase is to transform any initial description into a set of simple parallel processes by the application of a reduction strategy. This strategy applies algebraic rules and has two main steps. The first step transforms all `IF`'s and `ALT`'s commands into simple processes. As a simple process has at most one statement in its internal level, `IF`'s and `ALT`'s commands with multiple

branches must be broken. Moreover, if a branch of a conditional command is a SEQ, PAR or WHILE process, it is necessary to distribute the conditional over these processes. Rule 1 and Rule 2 are examples of the rules employed in this step.

Rule 1:

$$
\begin{aligned}
&\texttt{IF}(\texttt{b}_1\ \texttt{P}_1, ...,\ \texttt{b}_n\ \texttt{P}_n)\\
=\ &\texttt{VAR}\ \texttt{c}_1, ...,\ \texttt{c}_n\colon\ \texttt{SEQ}(\texttt{c}_1, ...,\ \texttt{c}_n := \texttt{FALSE}, ...,\ \texttt{FALSE},\\
&\quad \texttt{IF}(\texttt{b}_1\ \texttt{c}_1 :=\ \texttt{TRUE}, ...,\ \texttt{b}_n\ \texttt{c}_n :=\ \texttt{TRUE}),\\
&\quad \texttt{IF}(\texttt{c}_1\ \texttt{P}_1,\ \texttt{TRUE SKIP}), ...,\ \texttt{IF}(\texttt{c}_n\ \texttt{P}_n,\ \texttt{TRUE SKIP}))
\end{aligned}
$$

provided each $\texttt{c}_k$ is a fresh variable (occurring only where explicitly shown).

This rule transforms any conditional process into a sequence of IF's to allow the analysis of each subprocess of the original conditional.

Note that the first IF of the right-hand side makes the choice (and saves the result in one of the fresh variables) allowing the subsequent conditionals to be carried out in sequence.

Rule 2:

$$
\begin{aligned}
&\texttt{IF}(\texttt{b VAR x}:\ \texttt{SEQ}(\texttt{P}_1, ...,\ \texttt{P}_n),\ \texttt{TRUE SKIP})\\
=\ &\texttt{VAR c}:\ \texttt{SEQ}(\texttt{c}\ :=\ \texttt{b},\\
&\qquad \texttt{VAR x}:\texttt{SEQ}(\texttt{IF}(\texttt{c P}_1,\ \texttt{TRUE SKIP}), ...,\ \texttt{IF}(\texttt{c P}_n,\ \texttt{TRUE SKIP})))
\end{aligned}
$$

provided that c is a fresh variable.

This rule distributes IF over SEQ. Note that after exhaustive application of this rule, no IF will include any SEQ in its internal process. Similar rules distribute IF over ALT and over WHILE.

The second step of the splitting strategy transforms the intermediary description generated by the first step in the *normal form* of the splitting phase, which is a set of parallel (simple) processes. Two crucial transformations of this step are: 1) To turn simple processes closed in the sense that all variable used and assigned in the process are local. 2) To introduce a *controlling process* between every two simple processes. The controlling process acts as the interface between the processes under its control and the environment.

To understand the usefulness of a controlling process, consider two processes $\texttt{P}_1$ and $\texttt{P}_2$ with data-dependency and originally in sequence. To put $\texttt{P}_1$ and $\texttt{P}_2$ in parallel, as required by the normal form, communication must be introduced between them, as occam does not allow parallel processes to share variables. The purpose of the controlling process is to manage this communication. Except for communication commands of the original description, each $\texttt{P}_i$ interacts with the environment through the controlling process.

Rule 3 shows how sequential processes can be combined in parallel.

Rule 3:

$$
\begin{aligned}
&\texttt{VAR z : SEQ}(P_1,\ P_2)\\
=\ &\texttt{CHAN ch}_1,\ \texttt{ch}_2,\ \texttt{ch}_3,\ \texttt{ch}_4\texttt{: PAR}(\texttt{VAR x}_1\texttt{: SEQ}(\texttt{ch}_1\ ?\ x_1,\ P_1,\ \texttt{ch}_2\ !\ x_1'),\\
&\quad \texttt{VAR x}_2\texttt{: SEQ}(\texttt{ch}_3\ ?\ x_2,\ P_2,\ \texttt{ch}_4\ !\ x_2'),\\
&\quad \texttt{VAR z : CON}(\texttt{SEQ}(\texttt{ch}_1\ !\ x_1,\ \texttt{ch}_2\ ?\ x_1', \texttt{ch}_3\ !\ x_2,\ \texttt{ch}_4\ ?\ x_2')))
\end{aligned}
$$

provided $x_i = \texttt{USED}(P_i) \cup \texttt{ASS}(P_i)$ and $x_i' = \texttt{ASS}(P_i)$ and $\texttt{ch}_1, \texttt{ch}_2, \texttt{ch}_3$ and $\texttt{ch}_4$ are not free in P_1 or P_2.

It is denoted by ASS(P) the list of free[1] variables that are assigned in process P and by USED(P) the list of free variables used in expressions of P (either on the right-hand side of an assignment or in a boolean expression or in an output command).

Observe that although P_1 and P_2 are in parallel on the right-hand side of the rule above, in fact their behaviour are sequential. Process P_2 can executes only after the controlling process synchronises with P_1 through channel ch2.

3.2 The Joining Strategy

To indicate the result of the clustering phase, other new constructors are introduced: PARhw, PARsw, PARser and PARpar. These constructors have the same semantics of the standard PAR. The constructors PARhw and PARsw serve as annotations to denote the hardware and the software cluster, respectively. The constructors PARser and PARpar denote that the sub-processes included in each of them must be serialised and parallelised, respectively.

The goal of the joining strategy is to combine the processes that belong to the same cluster with the aim to implement the decisions taken by the clustering phase. Basically the joining phase applies algebraic rules to parallelise and serialise arbitrary simple processes. The parallelisation and serialisation must eliminate the communication introduced during the splitting phase, as well as the introduced variables on the case of IF's and ALT's recomposition.

As an example of the rules employed in this phase, consider Rule 4 below:

Rule 4:

```
CHAN ch,ch1,ch2,ch3,ch4,ch5,ch6:
PAR
  Q1
  F(PARpar
      VAR x1: SEQ(ch1? x1, P1, ch2! x1')
      VAR x2: SEQ(ch3? x2, P2, ch4! x2')
      Q2)
  VAR x:CON(SEQ(ch5? x,VAR z:SEQ(ch1!x1,ch2?x1',ch3!x2,ch4?x2'),
                ch6! x'))
```

[1] 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.

```
=
CHAN ch,ch5,ch6:
PAR
  Q1
  F(PARpar VAR x:SEQ(ch5?x, PAR(VAR z1:P1,VAR z2:P2), ch6!x')
      Q2)
```

provided that $x_1' \cap x_2 = \emptyset$ and $x_2' \cap x_1 = \emptyset$
where $x = x_1 \cup x_2$, $x' = x_1' \cup x_2'$, $x_i = USED(P_i) \cup ASS(P_i)$, $x_i' = ASS(P_i)$ and $z_i = z \cap x_i$, for i = 1,2.

To understand this rule, observe that the process P_1 and P_2 on the left-hand side of the rule are executed in sequence and their execution is controlled by the controlling process annotated with the construct CON. Note also that P_1 and P_2 are included in a PARpar constructor which means that they should be parallelised. The side conditions of the rule requires that P_1 and P_2 do not have data-dependency. The effect of the rule is to combine P_1 and P_2 in parallel, with the elimination of the controlling process, as can be noticed from the right-hand side of the rule.

4 ParTS Implementation

This section describes some implementation issues of ParTS such as its architecture, the programming languages used and the system it extends, OTS.

ParTS comprises two software layers: the transformation system in SML [15] and a graphical user interface in Java [5]. The core of ParTS is the SML module which implements the strategy to perform the hardware/software partitioning. This module extends the OTS environment including the specific rules of the splitting and the joining phases.

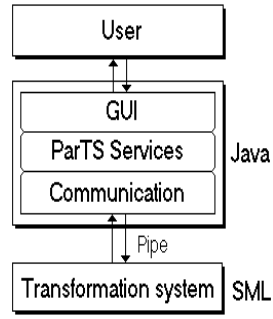


Fig. 2. The ParTS architecture

As shown in Figure 2 the Java module comprises three sub-modules which are concerned with communication with the SML module via a pipe (Commu-

nication module), concealment of the SML functions (ParTS Services module) and interface presentation (GUI module).

This architecture properly separates the system functionality from its graphical interface.

4.1 The Transformation Layer in SML

The OTS is implemented using the Edinburgh SML which is an interactive programming environment for a strongly-typed strict functional language. A functional program is defined as a set of values and functions. The SML also includes some imperative programming features which enables the construction of input/output commands and side-effect operations (assignment).

Collections of items are processed in SML using lists, a pre-defined type of the language. The lists are denoted by `[]` (the empty list) or by enumeration of its elements (such as `[1,2,3]`). The infix operator `::` (pronounced 'cons') constructs a new list by adding an element in front of an existing list (if `l` is the list `[x1,...,xn]` and `x` is a value of the correct type then `x::l` is the list `[x,x1,...,xn]`).

New types are defined by the `datatype` declaration which allows the creation of heterogeneous classes (a class constructed from several distinct subclasses). A simple example of `datatype` declaration is shown below:

```
datatype process = SKIP
                 | STOP
                 | seq of process list;
```

This example defines a very small subset of the occam language. The new type `process` and the *constructors* `SKIP`, `STOP` and `seq` are created. Constructors are regarded as functions which create values of a datatype. The constructors `SKIP` and `STOP` receive no arguments and returns a `process` and the constructor `seq` receives a `process` list and returns a `process`.

A function is defined as a set of equations containing a pattern as parameter and an expression as result. The argument passed is compared with the patterns and if some pattern matches then the corresponding expression is evaluated.

```
fun binary_seq (seq (p1::p2::p)) = seq (p1::[seq (p2::p)])
  | binary_seq p = p;
```

The first equation uses on its left-hand side the pattern `(seq (p1::p2::p))` — a sequence with at least two processes — and, on its right-hand side, the expression `seq (p1::[seq (p2::p)])` which constructs a binary sequential process. The second equation performs no transformation on the argument. Whenever the argument does not match the pattern stated in the first equation, it will always match the second equation which uses a variable `p` to stand for a general pattern. For example, the evaluation of `binary_seq(SKIP)` reduces to `SKIP`.

The last version of OTS (released in 1988) was implemented by Goldsmith [10] in the SML functional language. An abstract syntax for occam was defined

in SML as a set of recursive datatypes. The basic algebraic laws of occam are implemented as functions. A parse function is used to input a text file containing an occam process and translates it to the abstract syntax.

A sample of how an abstract syntax can be implemented using SML datatypes is shown below.

An identifier is represented as a string.

```
datatype identifier = ident of string;
```

Variables and channels are identifiers.

```
datatype variable = var of identifier;
datatype channel = chan of identifier;
```

Each operator of the language is a constructor of the type `process` with the relevant arguments. For example, an assignment statement is represented by the `assign` constructor and has as arguments a list of variables and a list of expressions.

```
datatype process = assign of (variable list) * expression list
                 | input_proc of channel * (variable list)
                 | output_proc of channel * (expression list)
                 | SKIP
                 | STOP
                 | dec of declaration * process
                 | seq_con of process list
                 | par_con of process list
                 | if_con of conditional list
                 | ...
and declaration = var_dec of variable list
                | chan_dec of channel list
and conditional = sim_cond of expression * process
                | if_cond of conditional list
and expression = TRUE
               | FALSE
               | num of int
               | varexp of variable
               | ...
```

As an example, the parser of OTS reads a file containing the following process

```
SEQ
  x := y
  ch ? y
```

and translates it to

```
seq_con [ assign( [var(ident ‘‘x’’)],[varexp(var(ident ‘‘y’’))] ),
          input_proc( chan(ident ‘‘ch’’),[var(ident ‘‘y’’)] ) ]
```

ParTS implements the transformation rules for the partitioning as functions. Nevertheless, these rules usually express much more complex transformations than the basic algebraics laws implemented in OTS.

As an example, we discuss the implementation of Rule 2 which will be called `distIF()`. The implementation of this rule has some auxiliary definitions. The function `freshVar()` receives a process `P` and returns a fresh variable (a variable that does not occur free in `P`). The function `map()` receives a function `f` and a list `l` and applies `f` to each element of l.

We also need to construct a function that builds each `IF` of the right-hand side of Rule 2. The `oneIF()` function receives as parameters a boolean expression and a process and returns a conditional process.

We also use the `let` expressions facility of SML. A `let` expression has the general form `let D in E end`. `D` is a declaration of values that is evaluated first. Then the expression `E` is evaluated inside the context of names declared in `D`.

Now we can define the `distIF()` function that implements Rule 2.

```
fun distIF (proc as
            if_con [
              sim_cond(b,
                dec(var_dec x,
                seq_con Pn)),
              sim_cond(TRUE,
                SKIP) ]) =
            let val c = freshVar(proc)
                val c_exp = varexp c
            in dec(var_dec [c],
               seq_con [
                 assign([c],[b]),
                 dec(var_dec x,
                 seq_con (map (oneIF c_exp) Pn)) ])
            end;
```

The `proc as` clause before the pattern creates the name `proc` that is bound to the conditional process received as argument. Then, `proc` is used as the argument of `freshVar()` function to generate a fresh variable (`c`); `c_exp` is just the fresh variable transformed into an expression type. The expression `(map (oneIF c_exp) Pn)` applies the `(oneIF c_exp)` function to each element of the process list `Pn`.

Clearly, the abstract syntax of occam (and the auxiliary functions) makes the implementation less readable. Even so, each rule is implemented in an elegant and abstract way as an SML function.

In a similar way, ParTS implements all the rules of the splitting and the joining phases. These new functions form the main code of ParTS. The splitting and the joining strategies are also implemented as functions. Each one is defined as the composition of the transformation rules (coded as functions) for the relevant phase. These rules are applied in an appropriate order to produce the desired result. The application of a rule is achieved through a higher-order function that takes the rule as argument and applies it to the current process.

4.2 The Graphical User Interface

A great improvement with respect to OTS is that the interface of OTS was specific for the Sun View environment and requires the user to interact at the level of SML functions. We have also implemented some facilities not available in OTS; this is further discussed below.

The GUI of ParTS implemented in Java allows users to manipulate several occam processes in different windows. The portability of Java makes possible the implementation of different versions of ParTS, for Unix SunOS and Windows95. The Windows95 version uses the Moscow SML instead of Edinburgh SML without any loss of functionality.

Figure 3 shows the interface of ParTS. A brief description of some elements of the screen is given below.

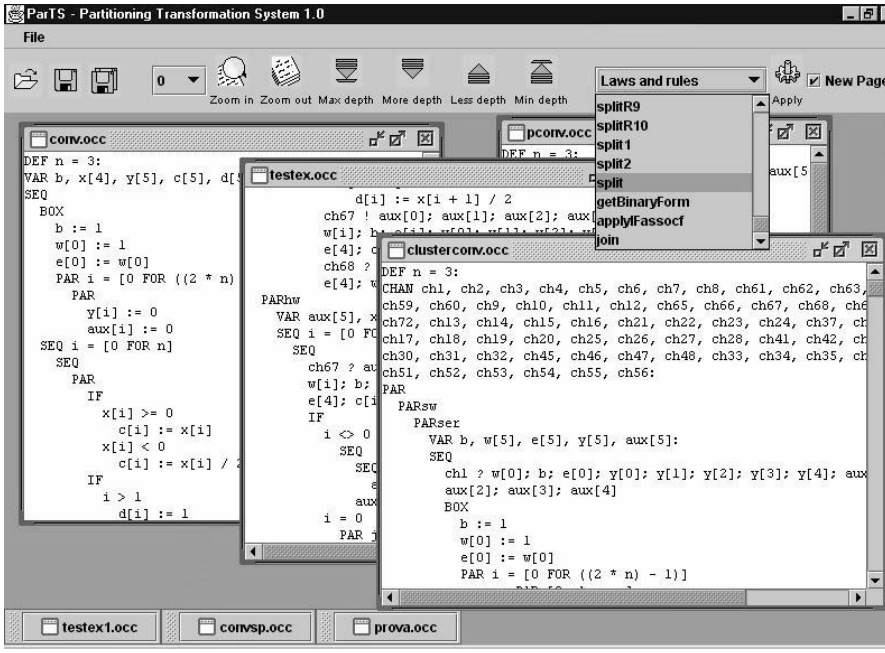
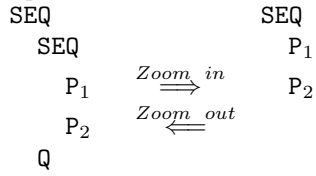


Fig. 3. The ParTS interface

File Menu The file menu provides commands to load occam files and save them (in general after performing transformations). It is possible to open various different files at the same time.

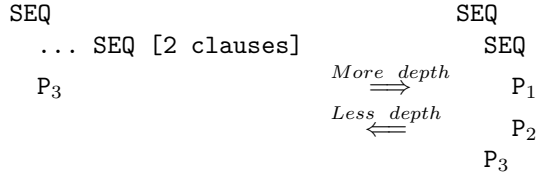
Zoom in Using this facility one can focus on any internal subprocess of the process, allowing the user to apply laws whose effect is restricted to internal parts of the process.

Zoom out This button has the opposite effect of the *Zoom in* button. For example:



Max depth / More depth ParTS allows the user to visualise a process partially. The *Max depth* button shows the process completely, without hiding any subprocess. The effect of the *More depth* button is to show the hidden subprocesses incrementally.

Less depth / Min depth The opposite effect of *More depth* is achieved with *Less depth* button. There is also the equivalent *Min depth* button that hides all subprocesses. For example,



Laws and rules This combo box is used to select the name of law/rule that will be applied to the current process (it contains rules named as `split` and `join` which perform the transformations to carry out the hardware/software partitioning). Also there exist all the laws that construct the split and the join strategies, allowing the user to do the partitioning step by step, if desired.

Apply The apply button must be used after choosing a law. It will apply that law and transform the current process accordingly.

New page If it is set before the application of a law, the effect produced by applying the law is shown in a separate window (without changing the current process). This is useful when one is not sure whether the law will provide the desired transformation.

OTS already included facilities related to zoom, depth and application of laws. Nevertheless, the interaction with the user is at the level of SML syntax. In ParTS all the interaction is directly in the occam notation. Furthermore, all the facilities concerning file manipulation and multiple windows are entirely new.

5 A Small Case Study

This section illustrates the hardware/software partitioning process of a vector convolution program used as a case study. For conciseness reasons, the explanation will emphasise particular aspects of the partitioning process, instead of trying to address all the details of the transformations involved.

Figure 4a shows the original description of the convolution program, and Figure 4b the partial result of the splitting phase generated by ParTS. The system exhaustively applies the rules of the splitting phase coded as functions in SML, as explained in the previous section. Using an Intel Pentium II 300 MHz and 64 MB of RAM as the hardware platform, ParTS takes about 15 seconds to perform the splitting of this program, and transforms the original description into 37 simple processes, combined in parallel.

Observe from Figure 4b that each of these processes has at most one assignment in its most internal level. The only exception is Process 1, where all subprocesses included into a `BOX` constructor are considered as an atomic process and therefore it has not been split.

Another point to notice is that all simple processes have been turned *closed* (their variables are local). Moreover, each original process is encapsulated (preceded and followed by communication commands).

The application of Rule 1 to Process 2 of Figure 4a transforms it into four simple processes (see Process 2.1, 2.2, 2.3 and 2.4 of Figure 4b) and the application of Rule 3 introduces communication between each pair of these simple processes. Process 3 in Figure 4b is the controlling process of Process 2.3 and 2.4.

After the splitting phase, the classification and the clustering phases take place. As we have mentioned before, these phases are related to the efficiency issue of partitioning process. The classification and clustering phases are being implemented as a separate tool which is under development and communicates with ParTS via shared files.

The classification phase defines for each simple process a set of implementations alternatives such as parallel, sequential, independent, etc. (see Figure 5a). The clustering phase builds a clustering tree which defines the clusters and how their processes must be combined (Figure 5b). Observe that the processes 2.1 – 2.4 of Figure 4b are grouped in the same cluster and must be combined in sequence. The *cut line* shown in Figure 5b separates the hardware and software clusters based on the heuristics defined in [2]. The clustering phase is responsible only for determining which processes should be combined to form the clusters, but do not carry out the transformations to effectively combine them.

Figure 6a shows the program after the classification and the clustering phases. This program reflects the design decision of the clustering phase. Note that the only changes concerning the occam program of Figure 4b are the annotations to identify the software (`PARsw`) and the hardware (`PARhw`) clusters, and whether each group of processes must be combined in sequence (`PARser`) or in parallel (`PARpar`).

Regarding the preservation of semantics, the transformation of the program in Figure 4b into the one in Figure 6a is immediately justified by the associativity and symmetry of parallel composition (see laws 2.3 and 2.4 of Section 2). This emphasises the fact that classification and clustering are concerned with the efficiency of the partitioning process, and have very little to do with program transformation.

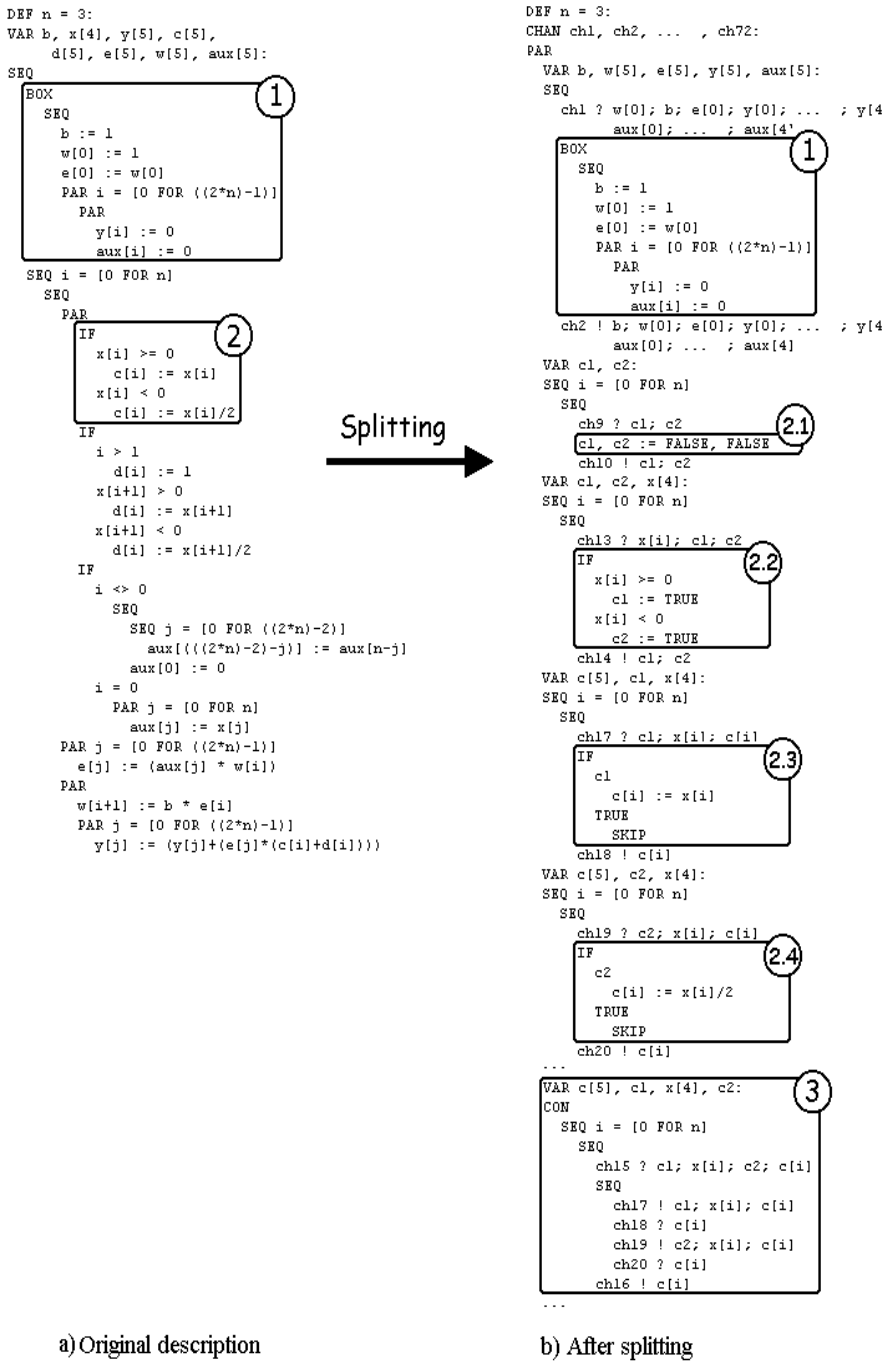


Fig. 4. The splitting phase

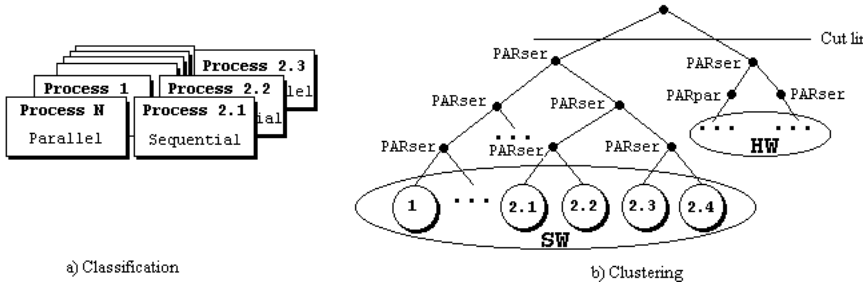


Fig. 5. The classification and the clustering phases

The final partitioned system is shown in Figure 6b. Observe that the `BOX` annotation has been eliminated. Since all subprocesses 2.1 – 2.4 belongs to the same cluster, the `IF` process is re-arranged and Process 3 is eliminated. ParTS combines the processes in the joining phase by applying the `join` strategy to the file generated by the clustering phase. This process takes about 2 seconds to complete (using the same hardware previously mentioned).

In fact, all controlling processes are eliminated and the only remaining communication is the one necessary for the synchronisation between the hardware and the software components (Observe channels `ch67` and `ch68` in Figure 6b).

6 Conclusions

We have used an intentionally small case study to illustrate an innovative approach to the implementation of hardware/software partitioning. The Partitioning Transformation System (ParTS) realises the partitioning process as a program transformation task, based on algebraic rules which ensure that the resulting program is correct (with respect to the original input) by construction.

The approach to partitioning is structured into four major phases, as summarised by Figure 7.

The first task performed by ParTS (Figure 7a-b) is to split a process in several simple processes operating in parallel. The phases of classification and clustering are concerned with the efficiency of the partitioned program. From the result of the splitting, a graph is constructed to allow a cost analysis to be carried out. However, this tree is suitable only as an intermediate representation, and, as already said, the implementation of classification and clustering are separate from ParTS which is exclusively concerned with program transformation.

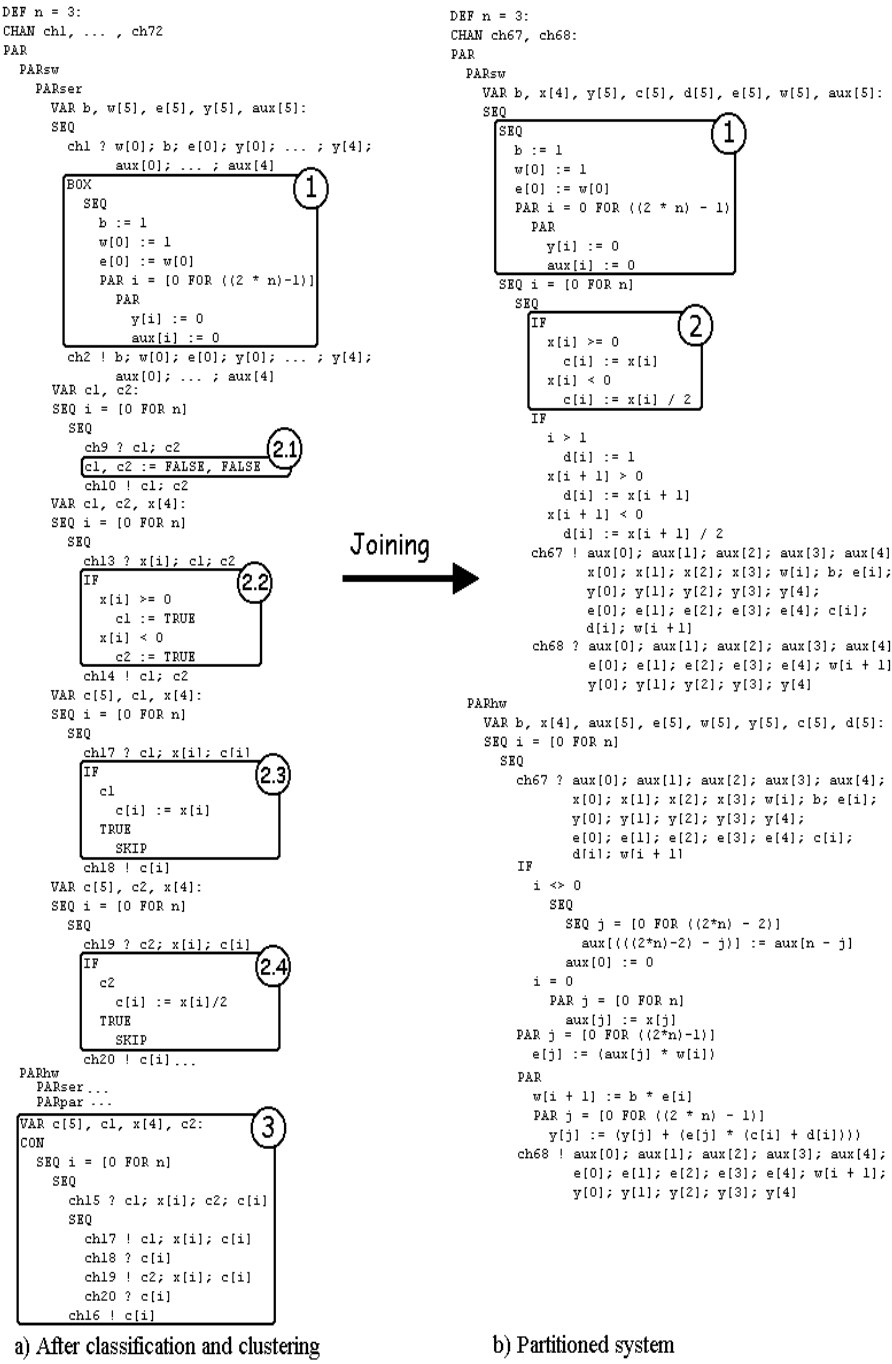


Fig. 6. The joining phase

The following step is to represent the clustering tree as a program with the form shown in Figure 7c, where annotations are used to determine which clusters are to be implemented in hardware and in software. Also note that we use an annotation for each cluster. This contains useful information (generated during the clustering) to guide the combination of the processes in each cluster; basically, this indicates whether process must be combined in sequence (`PARser`) or in parallel (`PARpar`).

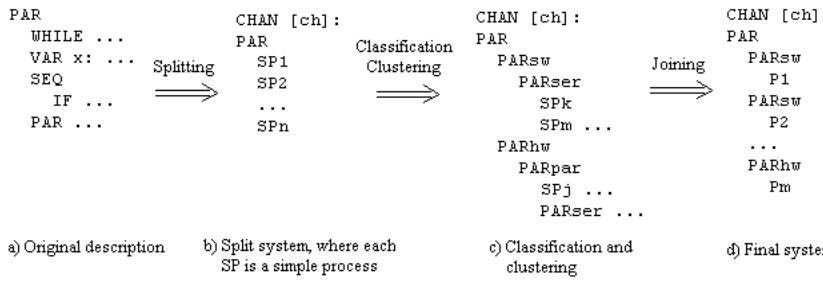


Fig. 7. The partitioning approach

Finally, the joining phases takes a program as in Figure 7c, carries out the necessary transformations to combine the processes in each cluster, and generates the final result, which is a precise abstract representation of our intended target architecture: with one software process and an arbitrary number of hardware processes (Figure 7d).

In terms of implementation, ParTS was built as an extension of the *Oxford occam Transformation System* (OTS), keeping all the original functionality of OTS, but adding specific notation and rules to allow a precise capture of the partitioning process, apart from a new user interface.

While the splitting has been completely formalised and implemented, the joining is still our major current focus of attention. A strategy for the joining phase is proposed in [19] based on transformation and reduction of configurations in a binary tree which represents the result of the clustering phase. While we have already implemented some general rules of the joining phase (which allows us to automatically partition some small examples like the one presented here) the full implementation of the joining strategy is still under development.

The integration between ParTS and the tool which implements the classification and clustering phases is also a topic for further research.

There are several systems which perform automatic hardware/software partitioning based on different approaches. The COSYMA system [8] assumes an all-software implementation as initial solution. A simulated annealing algorithm moves software code to hardware until the time constraints are met. The VULCAN system [11] starts by an all-hardware solution and uses an iterative approach to move operations from hardware to software. The SpecSyn [9] supports

several partitioning algorithms and presents an approach combining clustering and greedy algorithms. The Lycos [14] achieves the partitioning by a dynamic-programming algorithm that uses the information of the profiling and the time and area estimation steps.

None of these systems is concerned with the formal correctness of the partitioning process. To our knowledge, ParTS is the only existing tool which implements hardware/software partitioning based on algebraic transformations which ensures the preservation of semantics.

Acknowledgements

We are grateful to Michael Goldsmith for making available the source code of OTS. We also thank the Brazilian Research Council (CNPq) for financial support through the grants 130264/98-9 and 521039/95-9.

References

[1] A. Balsoni, W. Fornaccari, D. Sciuto. Partitioning and Exploration Strategies in the TOSCA Co-Design Flow. In *Proceedings of Fourth International Workshop on HW/SW Codesign*, (1996) 62–69.

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

[3] E. Barros and A. Sampaio. Towards Probably Correct Hardware/Software Partitioning Using Occam. In *Proceedings of the Third International Workshop on HW/SW Codesign (CODES'94), Grenoble, France. IEEE Press,* (1994) 210-217.

[4] E. Barros *et al.* The PISH Methodology for Hardware/Software Codesign. In *Workshop of ProTem-CC, CNPq*, (1998) 65–98.

[5] M. Campione and K. Walrath. The Java Tutorial: Object-Oriented Programming for the Internet. Addison Wesley Pub Co., 1998.

[6] C. Carreras, J. C. López, M. L. López, C. Delgado-Kloos, N. Martinéz, L. Sánchez. A Co-Design Methodology Based on Formal Specification and High-level Estimation. In *Proceedings of Fourth International Workshop on HW/SW Codesign* (1996) 28–35.

[7] T. Cheung, G. Hellestrand and P. Kanthamanon. A Multi-level Transformation Approach to HW/SW Co-Design: A Case Study. In *Proceedings of Fourth International Workshop on HW/SW Codesign*, (1996) 10–17.

[8] 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.

[9] D. Gajski and F. Vahid. Specification and Design of Embedded Hardware-Software Systems. In *IEEE Design and Test of Computers*, Spring 1995, 53-67.

[10] M. Goldsmith. The Oxford occam Transformation System. *Technical report, Oxford University Computing Laboratory*, January 1988.

[11] R. Gupta, C. N. Coelho and G. De Micheli. Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components. In *Proceedings of 29th Design Automation Conference*,1992.

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

[13] P. V. Knudsen and J. Madsen. PACE: A Dynamic Programming Algorithm for Hardware/Software Partitioning.In *Proceedings of Fourth International Workshop on HW/SW Codesign*, (1996) 85–92.

[14] J. Madsen. J. Groge, P. V. Knudsen, M. E. Petersen and A. Haxthausen. Lycos: The Lyngby Co-synthesis System. In *Design Automation of Embedded Systems*, 1997,2(2):195-235.

[15] L. Paulson. ML for the working programmer. Cambridge University Press, 1991.

[16] D. Pountain and D. May. A tutorial introduction to occam programming. INMOS, BSP Professional Books, 1987.

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

[18] L. Silva, A. Sampaio and E. Barros. A Normal Form Reduction Strategy for Hardware/Software Partitioning. In *Formal Methods Europe (FME) 97. Lecture Notes in Computer Science 1313*, (1997) 624–643.

[19] L. Silva, A. Sampaio, E. Barros and J. Iyoda. An Algebraic Approach for Combining Processes in a Hardware/Software Partitioning Environment. In *Proceedings of the Seventh International Conference on Algebraic Methodology and Software Technology (AMAST)*, (1998) 308–324