Serialising Parallel Processes in a Hardware/Software Partitioning Context

Leila Silva[1], Augusto Sampaio[2] and Geraint Jones[3]

[1] Departamento de Ciência da Computação e Estatística - UFS
Campus Universitário "Prof. José Aloísio Campos"
CEP 49000-100, Aracaju - SE - Brazil
{lmas@di.ufpe.br, leila@ufs.br}
[2] Depto. de Informática - UFPE
Caixa Postal 7851 - Cidade Universitaria
CEP 50740-540 Recife - PE - Brazil
acas@di.ufpe.br
[3] Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, UK, OX1 3Q
Geraint.Jones@comlab.ox.ac.uk

Abstract. In this paper we present a strategy to serialise parallel processes in occam, in the context of a hardware/software partitioning approach. The strategy takes as input two parallel processes to serialise and applies algebraic rules to perform the serialisation. These rules are derived from the semantics of occam, which guarantees that the serialisation strategy preserves the semantics of the original parallel processes. In particular, the strategy ensures that deadlock is not introduced during the serialisation procedure.

1 Introduction

Embedded systems are dedicated to a specific target application and the majority of them are implemented using general programmable components (software components) and specific application components (hardware components). They can be found in a large variety of applications such as telecommunication systems and in systems for the defence of territory and the environment. Hardware/Software co-design is a new design paradigm for the joint specification, design and synthesis of mixed hardware/software systems.

In the last years several methodologies and tools supporting hardware/software co-design for embedded systems have been published (among them, [5, 6, 8, 16]). A crucial point in the co-design flow is how to perform the partitioning of a system into hardware and software components. In addition to the approaches mentioned above, many other algorithms have been proposed (as for example, [1, 18]) to achieve this goal. Those approaches validate the partitioned system by simulation (for instance, [6, 8, 16]) or by using formal methods to prove that some desired properties of the original system are preserved after partitioning [5, 11].

As embedded systems become increasingly complex and are often used in life critical situations, the validation of the partitioned systems is no longer sufficient to guarantee safety. Thus, the formal verification of the partitioning process is essential to ensure that the behaviour is preserved throughout the co-design flow.

In [2] Barros and Sampaio presented some initial ideas towards a partitioning approach with emphasis on correctness. This work was the seed of the PISH co-design system [3], which 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. The PISH system attempts to address some key points not covered by the approaches mentioned before: the characterisation of the partitioning process as a program transformation task; the use of a same specification and reasoning mechanism (the occam language [15]); the use of formal verification techniques to assure the correctness of the partitioned system, and the automatic generation of the interface among hardware and software components, which is designed to be correct by construction.

The partitioning approach in PISH accepts as input an occam description (according to the grammar given in Section 2) and applies transformation rules to derive the description of the partitioned system. The main reason for choosing occam is that the algebraic laws of occam [17] can be used to carry out program transformation with the preservation of semantics. Furthermore, occam includes features to express parallelism and communication. Note that these are essential to express the result of the partitioning in the programming language itself: the hardware and software components generated by the partitioning are represented as communicating processes.

The PISH approach for partitioning verification comprises two main phases: *splitting* and *joining*. The aim of the splitting phase [20] is to transform the original description of the system into a description which is a set of *simple* parallel processes. These processes have a suitable form for the partitioning analysis regarding its implementation either in hardware or in software. Then, an algorithm is applied [1] to decide which processes will compose each component and the way they should be combined (in series or in parallel). By considering this decision, in the joining phase [21] these processes are effectively combined and the description of the partitioned system is generated. Both in the splitting and in the joining phases, algebraic rules are applied to perform the relevant transformations. These rules are all proved using the laws of occam, as shown in [22].

During the partitioning analysis, it may be required that processes originally in parallel be combined in series, due to economy of resources. Thus, a very important step of that approach is the elaboration of a strategy to serialise parallel processes; this is the central aim of this paper. Although [21] presents the main ideas of the joining strategy, it does not include any result about the serialisation of processes. Thus, the work presented here is the closing step of the methodology for partitioning verification published in [20] and [21].

The serialisation problem is extremely difficult in general, especially when the major concern is conceiving an automatic strategy which preserves correctness by construction. This has forced the approaches to serialisation reported in the literature to impose strong restrictions on the form of processes which are dealt with. As far as we are aware, the work closest to ours is that presented in [23], where Welch and Justo regard serialisation as an optimisation technique, applied automatically at the final stage of the development of parallel occam programs.

The occam processes considered in [23] are on the form of pre-defined templates. A template defines a particular kind of behaviour, that is, a reusable pattern of communication and synchronisation between the process and its environment. Assuming that these templates are deadlock-free, the strategy proceeds by combining them using the occam operators and ensuring that no deadlock is introduced. Our serialisation strategy accepts as input an arbitrary occam description (according to the grammar defined in Section 2) and the only restrictions we impose to the system are: it must be deadlock-free and each channel appears only twice in the description of the system (once for input and once for output).

This paper is organised as follows. Section 2 briefly describes the subset of occam adopted in this work, as well as presents some laws of occam and useful definitions. The approach to partitioning is briefly described in Section 3. Then, the serialisation strategy is presented in Section 4 and in Section 5 we discuss some implementation features of the strategy. A small example is presented in Section 6, for illustration purposes. Finally, in Section 7 we give the conclusions and some directions for future work.

2 A Language of Communicating Processes

The goal of this section is to present the subset of occam adopted in this work, defined by the BNF-style syntax given below. For convenience, we sometimes linearise occam syntax in this paper. For example, we may write `SEQ(P1, P2, ..., Pn)` instead of the standard vertical style.

$$
\begin{array}{rl}
P ::= & \texttt{SKIP} \mid \texttt{STOP} \mid \texttt{x := e} \mid \texttt{ch ? x} \mid \texttt{ch ! e} \\
\mid & \texttt{IF} \ (\mathtt{c_1}\ \mathtt{P_1},\ \mathtt{c_2}\ \mathtt{P_2},\ \ldots,\ \mathtt{c_n}\ \mathtt{P_n}) \mid \texttt{ALT} \ (\mathtt{c_1\&g_1}\ \mathtt{P_1},\ \mathtt{c_2\&g_2}\ \mathtt{P_2}, \ldots,\ \mathtt{c_n\&g_n}\ \mathtt{P_n}) \\
\mid & \texttt{SEQ} \ (\mathtt{P_1},\ \mathtt{P_2},\ \ldots,\ \mathtt{P_n}) \mid \texttt{PAR} \ (\mathtt{P_1},\ \mathtt{P_2},\ \ldots,\ \mathtt{P_n}) \\
\mid & \texttt{VAR x: P} \mid \texttt{CHAN ch: P}
\end{array}
$$

In what follows we give a short description of these commands. 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`). 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 among them, and cannot share variables. The constructs `VAR` and `CHAN` declare local variables and channels, respectively. Here we avoid mentioning a particular type for the declared variables or channels. A more detailed description of these commands can be found, for example, in [15].

To conduct the partitioning strategy we have extended occam syntax with some new constructs, among them, `PARhw`, `PARsw`, `PARpar`, `PARser`, `PARparFail` and `PARserFail`. These constructs have no semantic effect and can be regarded as annotations added to the occam description. The aim of these constructs is to indicate a hardware (`PARhw`) or a software (`PARsw`) component and the way the processes should be combined in each component, that is, in parallel (`PARpar`) or in series (`PARser`). The two last constructs indicates that a parallelisation (`PARparFail`) or a serialisation (`PARserFail`) of processes, required by the partitioning algorithm, cannot be implemented, as violates the semantics of the original description.

2.1 Some Algebraic Laws and Useful definitions

As mentioned in Section 1 occam obeys a set of algebraic laws [17] which can be used to carry out program transformation with the preservation of semantics. Here we present a very small subset of them. Each law is given a number and a name suggestive of its use, and the operational justification for each law is taken from [17].

The `SKIP` process is the identity of sequential composition.

Law 1 (*SEQ-SKIP unit*) `SEQ(SKIP, P) = SEQ(P, SKIP) = P`

The `SEQ` operator runs a number of processes in sequence. If it has no arguments it simply terminates. 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 (*SEQ assoc*) $\texttt{SEQ(P}_1\texttt{, P}_2\texttt{,..., P}_n\texttt{) = SEQ(P}_1\texttt{, SEQ(P}_2\texttt{, P}_3\texttt{,..., P}_n\texttt{))}$.

A `PAR` command terminates as soon as all its components have; the empty `PAR` terminates immediately. Furthermore, `PAR` is an associative operator.

Law 3 (*PAR-SKIP unit*) `PAR(P, SKIP) = PAR(P) = P`

Law 4 (*PAR assoc*) $\texttt{PAR(P}_1\texttt{, P}_2\texttt{,..., P}_n\texttt{) = PAR(P}_1\texttt{, PAR(P}_2\texttt{, P}_3\texttt{,..., P}_n\texttt{))}$

It is possible to use the previous laws to transform all occurrences of `SEQ` and `PAR` operators within a program to binary form. This is why the next laws are cast in binary form. The `PAR` operator is also commutative.

Law 5 (*PAR sym*) $\texttt{PAR(P}_1\texttt{, P}_2\texttt{) = PAR(P}_2\texttt{, P}_1\texttt{)}$

The declaration of variables obeys a set of laws. Here we illustrate only one of them, the one which distribute `VAR` over the first argument of a `SEQ`.

Law 6 (*VAR-SEQ 1*)
`SEQ(VAR x: P, Q) = VAR x: SEQ(P, Q), provided x does not occur in Q.`

As we have mentioned, the new constructs have no semantic effect and this fact is captured by algebraic laws, not given in [17]. Thus, for example,

Law 7 (*PARser unit*) `PARser P = PAR P`

Law 8 (*PARserFail unit*) `PARserFail P = PAR P`

Throughout this text we use the following definitions.

Definition 1 (Free and Bound Variables) *If* `P` *is an occam process 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.*

Definition 2 (Variable (Channel) Disjoint) *Two processes* $\mathtt{P}_1$ *and* $\mathtt{P}_2$ *are* variable (channel)-disjoint *if they have no free variable (channel) in common.*

Definition 3 (Variable-independent Processes) *Two processes* $\mathtt{P}_1$ *and* $\mathtt{P}_2$ *are* variable-independent *if no variable assigned by* $\mathtt{P}_1$ *is free in* $\mathtt{P}_2$*, and, on the other hand, no variable assigned by* $\mathtt{P}_2$ *is free in* $\mathtt{P}_1$*.*

Definition 4 (Independent and Disjoint Processes) *Two processes are* independent *if they are variable-independent and channel-disjoint. Two processes are* disjoint *if they are variable- and channel-disjoint.*

Definition 5 (Subprocess and Elementary Subprocess) *Let* $\mathtt{P_i}$ *be a process included into a sequential, parallel, conditional or* `ALT` *process* `P`*. We say that* $\mathtt{P_i}$ *is a* subprocess *of* `P`*. The subprocess* $\mathtt{P_i}$ *is* elementary *if either it includes at most one communication command or it includes at most one* `ALT` *construct, having communication commands only as its input guards.*

3 The Partitioning Approach

The general structure of the approach to partitioning in which this work is inserted is depicted in Figure 1. The shaded boxes represent the relevant phases to partitioning verification and the phase of definition of components is related to the non-functional issues of the partitioning (the efficient mapping of the system into hardware/software components). The implementation phase comprises the tasks of the co-design flow performed after partitioning, among them, hardware synthesis, software compilation, prototyping and co-simulation.

The splitting phase accepts as input an arbitrary occam description, according to the grammar given in Section 2, and transforms this description into a description in the following form. As mentioned in Section 1, this description must satisfy two restrictions: to be deadlock-free and to allow only one point of synchronisation, for each channel.

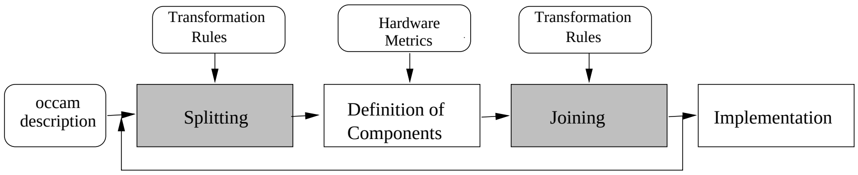

Fig. 1. The partitioning phases.

Definition 6 (*Splitting Normal Form*) *A process is in splitting normal form if it has the structure below:*

```
CHAN ch1, ch2,..., chm: PAR(P1, P2,..., Pr)
```

where each $\mathtt{P_i}$*, for* $1 \leq \mathtt{i} \leq \mathtt{r}$*, is simple.*

Here we will not detail the form of simple processes, neither discuss their suitability to hardware/software partitioning (see [20, 22]). For this work it is relevant to mention that `ALT` constructs are in a *simple* form if they include communication commands only as input guards. If a conditional includes communication commands and is in a *simple* form, it has two branches, of the form `IF(c Q, TRUE SKIP)`. If `Q` involves communication it can be: an input command, an output command or an `ALT` construct in the simple form.

To achieve the normal form, a reduction strategy is necessary [20]. This strategy applies algebraic rules (derived from the basic laws of occam) in two main steps. In the first step, `ALT` and `IF` constructs are transformed into the simple form mentioned above. In the second step, basically the sequential processes are put in parallel. As original sequential processes may be data-dependent and as parallel processes in occam cannot share variables, we have to introduce communication to perform this transformation. The splitting strategy can be summarised by the following theorem, proved in [20, 22]:

Theorem 1 (*Splitting Strategy*) *An arbitrary program* `P` *(according to the syntax defined in Section 2), deadlock-free and allowing only one point of synchronisation for each channel, can be reduced to the splitting normal form*

```
CHAN ch1, ch2,..., chm: PAR(P1, P2, ..., Pr)
```

where each $\mathtt{P_i}$ *is simple.*

The description generated by the splitting phase is the input of the phase of definition of components, where heuristics are applied to decide which simple processes will compose the hardware and the software components, as well as in which way these processes should be combined (in series or in parallel). This decision is guided exclusively by implementation metrics, which consider various features of the description, such as non-determinism, mutual exclusion, communication costs, concurrent behaviour, data-dependency and functional similarity among processes. Moreover, the tradeoff between the area and the delay of the architecture implementing the partitioned system is also appraised.

The result of this phase is expressed by using the associativity and commutativity of the PAR operator and the new constructors PARhw, PARsw, PARpar and PARser. As these constructors have no semantic effect, the description generated by this phase can be seen as a permutation of the splitting description rearranged and extended with some useful annotations for the joining phase.

It is important to notice that this phase is responsible only for determining which processes should be combined (in series or in parallel) to form the components, but it does not actually carry out the necessary transformations to combine the processes. These transformations are performed by the joining phase. Like the splitting phase, it is applied algebraic rules to effectively combine the processes of the components, according to the way established by the phase of definition of components. During the combination procedure, the communication introduced by the splitting phase is eliminated, whenever possible. It is important to remark that if the phase of definition of components requires a transformation which violates the semantics of the original description, this transformation is not performed by the joining phase and an error is annotated for future designer analysis. The joining strategy can also be summarised by a theorem, proved in [22].

Theorem 2 (*Joining Strategy*) *A program* P *in the splitting normal form*

$$\texttt{CHAN } \texttt{ch}_1,\ \texttt{ch}_2,\ldots,\ \texttt{ch}_m\texttt{: PAR(}\texttt{P}_1,\ \texttt{P}_2,\ \ldots,\ \texttt{P}_r\texttt{)},$$

*with some annotations (*PARhw, PARsw, PARpar *and* PARser*), can be reduced to the joining normal form*

$$\texttt{CHAN } \texttt{ch}_1,\ \texttt{ch}_2,\ldots,\ \texttt{ch}_s\texttt{: PAR(}\texttt{Q}_1,\ \texttt{Q}_2,\ \ldots,\ \texttt{Q}_t\texttt{)},$$

where $s \leq m$*,* $t \leq r$ *and each* P_i*,* $1 \leq i \leq r$*, belongs to exactly one* Q_j*,* $1 \leq j \leq t$*. Moreover, the reduced description has two characteristics: (1) either follows what was established by the phase of definition of components or includes an indication of an error of the requirements, and (2) the communication and variables introduced by the splitting phase are eliminated, whenever possible.*

4 The Serialisation Strategy

As mentioned before, during the phase of definition of components it may be required that processes originally in parallel be serialised, due to economy of resources of the final chip.

To motivate the serialisation problem, consider the fragment of program PAR(ch_1?x, ch_2?y). To serialise these processes, it is necessary to know how the channels ch_1 and ch_2 interact with the environment (the rest of the program). If the environment has the form SEQ(ch_1!e, ch_2!f), ch_1 synchronises before ch_2, and, in this case, the parallel command can be transformed into SEQ(ch_1?x, ch_2?y). On the other hand, if the environment has the form

$\texttt{SEQ(ch}_2\texttt{!f, ch}_1\texttt{!e)}$, the transformation must be $\texttt{SEQ(ch}_2\texttt{?y, ch}_1\texttt{?x)}$. Furthermore, if $\texttt{ch}_1$ and $\texttt{ch}_2$ are in parallel in the environment, the order of serialisation is irrelevant. In general, the serialisation problem can be stated as follows:

> *Let* P *be an occam program. Given two parallel processes of* P*, say* $\texttt{P}_\texttt{x}$ *and* $\texttt{P}_\texttt{y}$*, we want to find a sequential order of execution of the processes* $\texttt{P}_\texttt{x}$ *and* $\texttt{P}_\texttt{y}$ *that preserves the semantics avoiding the introduction of deadlock in* P*.*

Before introducing the serialisation rules (Section 4.1), as well as the algorithm which guides the rules application (Section 4.2), we will give a more precise definition of environment. The environment is the context into which the parallel processes to serialise are immersed. In [22, Chapter 8] it is shown that it is always possible to express the environment as a process E which runs in parallel with the fragment of program P we want to serialise. Thus, the serialisation rules which depend on the context have the form:

```
PAR(P, E) = PAR(Q, E),    provided E can satisfy S.
```

where S specifies, as defined in what follows, a list of restrictions on the order of execution of communication commands inside P, in order to guarantee that the transformation, from P to Q, does not introduce deadlock. The condition of the above equation expresses the fact that the environment E (running in parallel with P and Q) must offer the opportunity of a synchronisation which satisfies the synchronisation restrictions S, given in the sequel. For conciseness reasons, we abbreviate rules in that form to $\texttt{P} =_{(\texttt{E},\texttt{S})} \texttt{Q}$.

Definition 7 (*Synchronisation Restriction*) *Let* P *be an occam program. Let* $\texttt{P}_\texttt{j}$ *and* $\texttt{P}_\texttt{k}$ *be two independent processes of* P*. We call by restriction* $\texttt{P}_\texttt{j}\texttt{P}_\texttt{k}$ *the following constraint in the order of synchronisation of the channels used by* $\texttt{P}_\texttt{j}$ *and* $\texttt{P}_\texttt{k}$ *with the environment* E*: the channels used by* $\texttt{P}_\texttt{j}$ *must synchronise before any channel of* $\texttt{P}_\texttt{k}$*. In particular, if either* $\texttt{P}_\texttt{j}$ *or* $\texttt{P}_\texttt{k}$ *does not include a communication command, we say that the restriction* $\texttt{P}_\texttt{j}\texttt{P}_\texttt{k}$ *is void.*

Considering the previous example, let $\texttt{P}_1$ be $\texttt{ch}_1\texttt{?x}$ and let $\texttt{P}_2$ be $\texttt{ch}_2\texttt{?y}$. If the environment has either the form $\texttt{SEQ(ch}_1\texttt{!e, ch}_2\texttt{!f)}$ or $\texttt{PAR(ch}_1\texttt{!e, ch}_2\texttt{!f)}$ it satisfies restriction $\texttt{P}_1\texttt{P}_2$, and thus $\texttt{P}_1$ and $\texttt{P}_2$ can be serialised in this order. Observe that when the environment is a parallel process, it offers as one of the possibilities the synchronisation which satisfies $\texttt{P}_1\texttt{P}_2$. In the sequential case, this is the only way the environment can synchronise.

4.1 The Serialisation Rules

The serialisation rules can be derived from the basic laws of occam given in [17] and from Lemma 1 (*PAR-SEQ result*) (proved in [22, Appendix A]), which is a generalisation of a similar result for disjoint processes presented in [10, Chapter 7].

Lemma 1 (*PAR-SEQ result*)

$$\mathtt{PAR(SEQ(Q_1,\ Q_2),\ SEQ(T_1,\ T_2))} =_{(\mathtt{E},(\mathtt{Q_1T_2},\mathtt{T_1Q_2}))} \mathtt{SEQ(PAR(Q_1,\ T_1),\ PAR(Q_2,\ T_2))}$$

This result changes the way of execution of the processes $\mathtt{Q}_1$ and $\mathtt{T}_2$ and also $\mathtt{Q}_2$ and $\mathtt{T}_1$. Observe, for example, that $\mathtt{Q}_1$ and $\mathtt{T}_2$ execute in parallel on the left-hand side of the equation and in sequence on the right-hand side. To do this transformation, $\mathtt{Q}_1$ and $\mathtt{T}_2$, as well as $\mathtt{Q}_2$ and $\mathtt{T}_1$ must be independent processes. Moreover, the environment must offer the opportunity of a synchronisation with all channels of $\mathtt{Q}_1$ before synchronising with any channel of $\mathtt{T}_2$ and with all channels of $\mathtt{Q}_2$ before synchronising with any channel of $\mathtt{T}_1$. Three immediate corollaries of this result are given in what follows.

Corollary 1 (*PAR-SEQ derived - 1*)

$$\mathtt{PAR(Q_1,\ SEQ(T_1,\ T_2))} =_{(\mathtt{E},\mathtt{Q_1T_2})} \mathtt{SEQ(PAR(Q_1,\ T_1),\ T_2)}$$

Corollary 2 (*PAR-SEQ derived - 2*)

$$\mathtt{PAR(SEQ(Q_1,\ Q_2),\ T_1)} =_{(\mathtt{E},\mathtt{Q_1T_1})} \mathtt{SEQ(Q_1,\ PAR(Q_2,\ T_1))}$$

Corollary 3 (*PAR-SEQ derived - 3*)

$$\mathtt{PAR(Q_1,\ T_1)} =_{(\mathtt{E},\mathtt{Q_1T_1})} \mathtt{SEQ(Q_1,\ T_1)}.$$

In this text we implicitly use Law 7 (*PARser unit*) and, therefore, all results applied to the `PAR` operator can immediately be applied to the `PARser` construct. Thus, for example, we can refer to Lemma 1 (*PAR-SEQ result*), when the `PAR` construct is, in fact, a `PARser` construct.

Now, let $\mathtt{P_x}$ and $\mathtt{P_y}$ be two parallel processes for which the serialisation is required. The easy case of the serialisation procedure happens when the processes $\mathtt{P_x}$ and $\mathtt{P_y}$ are independent and satisfy either the restriction $\mathtt{P_xP_y}$ or $\mathtt{P_yP_x}$. In this case, Corollary 3 (*PAR-SEQ derived - 3*) (applied to a `PARser` construct) is enough to serialise $\mathtt{P_x}$ and $\mathtt{P_y}$. (This is the case of the fragment of program shown previously.)

When applying Corollary 3 (*PAR-SEQ derived - 3*) to serialise processes $\mathtt{P_x}$ and $\mathtt{P_y}$ in $\mathtt{PARser(P_x,\ P_y)}$, the symmetry of the `PARser` operator should be applied to the parallel construct, with the aim of exploring both possibilities of serialisation, that is $\mathtt{P_x}$ either before or after $\mathtt{P_y}$. Similar comments will be omited in the remainder of this text.

If this result cannot be applied, the proposed strategy performs a *gradual serialisation* of $\mathtt{P_x}$ and $\mathtt{P_y}$. Figure 2 shows a simple example of a gradual serialisation of processes $\mathtt{P_u}$ and $\mathtt{P_v}$, where the environment is represented by process $\mathtt{P_z}$. Observe that as the channels of $\mathtt{P_u}$ and $\mathtt{P_v}$ synchronise in an interleaved way with $\mathtt{P_z}$, Corollary 3 (*PAR-SEQ derived - 3*) cannot be applied (the restriction $\mathtt{P_uP_v}$ (or $\mathtt{P_vP_u}$) is violated). Nevertheless, we can perform the serialisation of $\mathtt{P_u}$ and $\mathtt{P_v}$ by exhaustively serialising their subprocesses, according to the order of execution of the channels in $\mathtt{P_z}$, as shown in Figure 2(b) and (c).

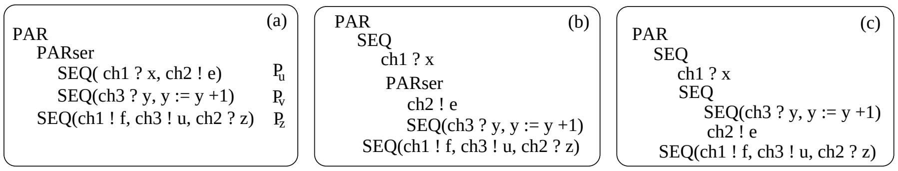

Fig. 2. An example of a gradual serialisation of $\mathtt{P_u}$ and $\mathtt{P_v}$.

To perform the gradual serialisation of processes, firstly laws 2 (*SEQ assoc*), 1 (*SEQ-SKIP unit*), 4 (PAR assoc) and 3 (*PAR-SKIP unit*) are used to transform into a binary form all occurrences of `SEQ` and `PAR` inside $\mathtt{P_x}$ and $\mathtt{P_y}$. (This is why the remainder of the rules of this section is cast in binary form.) In addition, to simplify the proposed serialisation rules, the laws of declaration (given in [17]) are used to increase the scope of the possible local variables of $\mathtt{P_x}$ and $\mathtt{P_y}$ to widest scope.

To avoid the introduction of deadlock when $\mathtt{P_x}$ and $\mathtt{P_y}$ are gradually serialised, all internal synchronisations of these processes must be solved first; this is why we need to introduce the following rule:

Rule 1 (*internal serialisations)*

$\mathtt{PARser(F(PAR(Q_1,\ Q_2)),\ P_2) = PARser(F(PARser(Q_1,\ Q_2)),\ P_2)}$,
provided $\mathtt{Q_1}$ and $\mathtt{Q_2}$ are not channel-disjoint.

The role of the previous rule is to indicate, for future serialisation, the processes that share channels ($\mathtt{Q_1}$ and $\mathtt{Q_2}$) into $\mathtt{P_1}$ (represented by $\mathtt{F(PAR(Q_1,\ Q_2))}$).

To solve internal serialisations of $\mathtt{P_x}$ and $\mathtt{P_y}$, the strategy is applied recursively. To guarantee that the internal serialisations are solved first, the remainder of the rules of this section requires that the processes to serialise do not include any `PARser` construct. This is also required when we apply Lemma 1 (*PAR-SEQ result*) and its corollaries.

After performing these preliminary transformations, the rules for effectively serialising $\mathtt{P_x}$ and $\mathtt{P_y}$ are applied. The proposed strategy follows an inductive approach and thus we have *base* and *intermediate* cases. Corollary 3 (*PAR-SEQ derived - 3*) can be considered the base case (0). In addition, we have the following base cases:

- (1) the first subprocess of $\mathtt{P_x}$ to serialise is channel-disjoint from $\mathtt{P_y}$ and, as a consequence, independent of $\mathtt{P_y}$;
- (2) the first subprocess of $\mathtt{P_x}$ to serialise synchronises with $\mathtt{P_y}$.

To solve the base case (1), Corollary 2 (*PAR-SEQ derived - 2*) and Rule 2 (*serialisation - 1*) can be applied.

Rule 2 (*serialisation-1*)

$$\texttt{PARser(PAR(Q}_1\texttt{, Q}_2\texttt{), T}_1\texttt{)} =_{(\texttt{E},(\texttt{Q}_1\texttt{Q}_2,\texttt{Q}_1\texttt{T}_1))} \texttt{SEQ(Q}_1\texttt{, PARser(Q}_2\texttt{, T}_1\texttt{))}$$

provided the processes involved do not include any PARser process.

Rule 2 (*serialisation-1*) is applied when one process of the PARser construct is a parallel process. The first subprocess to execute is $\texttt{Q}_1$ and the environment must offer the opportunity of a synchronisation with $\texttt{Q}_1$ before synchronising with any channel of $\texttt{T}_1$ and $\texttt{Q}_2$. In fact, the transformation captured by this rule can be achieved by applying Corollary 3 (*PAR-SEQ derived - 3*) to put $\texttt{Q}_1$ and $\texttt{Q}_2$ in sequence and after Corollary 2 (*PAR-SEQ derived - 2*) to serialise $\texttt{Q}_1$. Nevertheless, due to the expressiveness of this transformation we prefer to capture it as a new rule.

To solve the base case (2), we need to introduce synchronisation rules, given in Section 4.1.

If the first subprocess, say Q, (of either $\texttt{P}_\texttt{x}$ or $\texttt{P}_\texttt{y}$) to serialise includes more than one communication command it may happen that Corollary 2 (*PAR-SEQ derived - 2*) and Rule 2 (*serialisation - 1*) cannot be applied, due to a violation of the synchronisation restriction expressed in such results. In this case, we need to unnest the first elementary subprocess of Q, in order to apply one result for the base cases. As this elementary subprocess of Q may be inside several levels of SEQ and PAR constructs, we need to exhaustively apply the following results:

- Law 2 (*SEQ assoc*) to transform a pattern of the form $\texttt{SEQ(SEQ(Q}_1\texttt{, Q}_2\texttt{), Q}_3\texttt{)}$ into $\texttt{SEQ(Q}_1\texttt{, SEQ(Q}_2\texttt{, Q}_3\texttt{))}$;
- Rule 3 (*serialisation-2*), given in what follows, to transform a pattern of the form $\texttt{SEQ(PAR(Q}_1\texttt{, Q}_2\texttt{), Q}_3\texttt{)}$ into $\texttt{SEQ(Q}_1\texttt{, SEQ(Q}_2\texttt{, Q}_3\texttt{))}$;
- Corollary 2 (*PAR-SEQ derived - 2*) to transform a pattern of the form $\texttt{PAR(SEQ(Q}_1\texttt{, Q}_2\texttt{), Q}_3\texttt{)}$ into $\texttt{SEQ(Q}_1\texttt{, PAR(Q}_2\texttt{, Q}_3\texttt{))}$;
- Rule 2 (*serialisation-1*) (applied to the PAR construct instead to a PARser construct) to transform a pattern of the form $\texttt{PAR(PAR(Q}_1\texttt{, Q}_2\texttt{), Q}_3\texttt{)}$ into $\texttt{SEQ(Q}_1\texttt{, PAR(Q}_2\texttt{, Q}_3\texttt{))}$.

Rule 3 (*serialisation - 2*)

$$\texttt{SEQ(PAR(Q}_1\texttt{, Q}_2\texttt{), Q}_3\texttt{)} =_{(\texttt{E},\texttt{Q}_1\texttt{Q}_2)} \texttt{SEQ(Q}_1\texttt{, SEQ(Q}_2\texttt{, Q}_3\texttt{))}$$

provided none of the processes involves a PARser construct.

Rule 3 (*serialisation-2*) can be derived from Corollary 3 (*PAR-SEQ derived - 3*) and Law 2 (*SEQ assoc*) and follows similar explanation of these results.

Notice that, for the four patterns mentioned above, the depth in which $\texttt{Q}_1$ is nested decreases by one, after the transformations; this is why the exhaustive application of these rules is able to bring $\texttt{Q}_1$ to the most external level, whenever possible.

The serialisation strategy can not be concluded successfully due to a violation of a synchronisation restriction (see [22] for a more detailed discussion).

This can happen basically in two situations: (1) at least one process to serialise includes an ALT construct interacting in a particular way with the environment and (2) the processes to serialise have an apparent mutual dependency. In the first situation, in the environment, the communication commands which include the channels (guards) of the ALT construct execute in an interleaved way with the communication commands which include the channels of the other process to serialise. In the second situation, the order of execution of some communication commands included in the processes to serialise is reverse to the order of execution of communication commands which include these channels, in the environment. In this case, as we assume that the system is deadlock free, necessarily such channels belong to mutual exclusive processes. When these situations happens, we say that the serialisation strategy *fails* and Law 8 (*PARserFail unit*) is applied to indicate such a failure and to finish the serialisation procedure. In the context of our partitioning approach, the required serialisation will be revised by the designer in a later stage of the partitioning flow.

The Synchronisation Rules Here we can assume that the first subprocess (T_1) to execute of P_y synchronises with the first subprocess (Q_1) to execute of P_x. The subprocesses of P_x and P_y that execute before the synchronisation of P_x and P_y are already serialised, by applying the intermediate and base rules before presented, otherwise a failure has been detected.

As the description is binary, $\texttt{PARser}(P_x, P_y)$ can have one of the following forms (with possibly changing the roles of P_x and P_y), where SEQ/PAR means either a sequential or a parallel process in this context:

F1: $\texttt{PARser}(Q_1, T_1)$;
F2: $\texttt{PARser}(Q_1, \texttt{SEQ/PAR}(T_1, T_2))$;
F3: $\texttt{PARser}(\texttt{SEQ/PAR}(Q_1, Q_2), \texttt{SEQ/PAR}(T_1, T_2))$.

Forms F1 and F2 are particular cases of Form F3 (due to laws 3 (*PAR-SKIP unit*) and 1 (*SEQ-SKIP unit*)). Thus, here we present only the synchronisation rules for Form F3.

The subprocesses Q_1 and T_1 can be: an output command; an input command; an ALT command; a conditional including an output command; a conditional including an input command; or a conditional including an ALT command. In fact, we need to introduce rules to deal with all possible pairs of these commands. Nevertheless, here we present only the rules involving communication commands inside conditionals and ALT constructs, as the rules for the other possible pair of commands are particular cases of such rules. The full set of synchronisation rules can be found in [22, Chapter 6]. Moreover, we present the rules for the case in which P_x and P_y are sequential processes. The rules for the cases in which one of these process is a parallel processes are similar. In the presentation of such rules we abuse the notation here adopted by expressing the channel, instead of the process, in the synchronisation restriction.

Rule 4 (*sync. IF-in-IF-out-F3*)

$$
\begin{aligned}
& \mathtt{PARser(SEQ(IF(c_1\ ch\ !\ e,\ TRUE\ SKIP),\ Q_2),} \\
& \qquad\quad \mathtt{SEQ(IF(c_2\ ch\ ?\ x,\ TRUE\ SKIP),\ T_2))} \\
=_{\mathtt{DF(E,chQ_2,chT_2)}} \quad & \\
& \mathtt{SEQ(IF\ (c_1\ x\ :=\ e,\ TRUE\ SKIP),\ PARser(Q_2,\ T_2))} \\
& \mathtt{provided}\ \mathtt{c_1} \oplus \mathtt{c_2} = \mathtt{TRUE.}
\end{aligned}
$$

The meaning of this rule is very simple; it expresses the fact that the effect of the synchronisation on the left-hand side of the rule is the assignment of $\mathtt{e}$ to $\mathtt{x}$ on the right-hand side. Observe that if the program is deadlock free and admits only one point of synchronisation the condition $\mathtt{c_1} \oplus \mathtt{c_2} = \mathtt{TRUE}$ (the symbol $\oplus$ stands for exclusive or) is always valid. This also implies that we can chose between $\mathtt{c_1}$ and $\mathtt{c_2}$ to express the condition on the right-hand side of this rule.

Rule 5 (*sync. IF-out-IF-ALT-F3*)

$$
\begin{aligned}
& \mathtt{PARser} \\
& \quad \mathtt{SEQ/PAR(IF(c_1\ ch_1\ !\ e,\ TRUE\ SKIP),\ Q_2)} \\
& \quad \mathtt{SEQ/PAR(IF(c_2\ ALT(b_1\ \&\ ch_1?x_1\ T_1',\ ALT_{k=2}^{n}\ b_k\&g_k\ T_k'),} \\
& \qquad\qquad\qquad \mathtt{TRUE\ SKIP),\ T_2)} \\
=_{\mathtt{DF(E,ch_1Q_2,ch_1T_2,T'_1Q_2)}} \quad & \\
& \mathtt{VAR\ d:} \\
& \mathtt{SEQ(SEQ(d\ :=\ c_1,\ IF\ (d\ SEQ(x\ :=\ e,\ T_1'),\ TRUE\ SKIP)),} \\
& \qquad \mathtt{PARser(Q_2,\ SEQ/PAR(IF(\neg\ d\ IF\ (c_2\ ALT_{k=2}^{n}\ b_k\&g_k\ T_k',} \\
& \qquad\qquad\qquad\qquad \mathtt{TRUE\ SKIP),\ TRUE\ SKIP),\ T_2)))} \\
& \mathtt{provided}\ \mathtt{c_1} \oplus \mathtt{b_1} = \mathtt{TRUE},\ \mathtt{c_1} \oplus \mathtt{b_k} = \mathtt{FALSE},\ \mathtt{for\ each\ k,} \\
& \mathtt{c_1} \Rightarrow \mathtt{c_2},\ \mathtt{d\ is\ a\ fresh\ variable\ and\ Q_2\ and\ T_2\ do\ not} \\
& \mathtt{include\ any\ PARser\ process.}
\end{aligned}
$$

The effect of this rule is also to replace communication with assignment. Observe that as we have assumed that the system is deadlock free originally and has only one point of synchronisation, then necessarily $\mathtt{c_1} \oplus \mathtt{b_1}$ always evaluates to $\mathtt{TRUE}$. Moreover, $\mathtt{c_1} \oplus \mathtt{b_k}$ is necessarily $\mathtt{FALSE}$. This implies that when $\mathtt{c_1}$ is $\mathtt{FALSE}$, another guard of the $\mathtt{ALT}$ construct can be satisfied. Notice also that the case in which $\mathtt{c_1}$ is $\mathtt{TRUE}$ and $\mathtt{b_1}$ is $\mathtt{FALSE}$ is not possible, as it means that the system would be in deadlock originally. In addition, the process $\mathtt{T_1'}$ can modify the value of a variable included in $\mathtt{c_1}$. This is why we need to introduce the new variable $\mathtt{d}$.

To illustrate the proof style of the serialisation rules we develop the proof of Rule 2 (*serialisation-1*). We start with the left-hand side of the equation, and by applying some basic laws and lemmas we obtain the right-hand side.

$$
\begin{array}{rl}
& \mathtt{PARser(PAR(Q_1,\ Q_2),\ T_1)} \\
=_{(\mathtt{E},\mathtt{Q_1Q_2})} & \{Corollary\ 3 (PAR - SEQ\ derived - 3)\} \\
& \mathtt{PARser(SEQ(Q_1,\ Q_2),\ T_1)} \\
= & \{Laws\ 1\ (SEQ - SKIP\ unit)\ and\ 7\ (PARser\ unit)\} \\
& \mathtt{PAR(SEQ(Q_1,\ Q_2),\ SEQ(SKIP,\ T_1)} \\
=_{(\mathtt{E},\mathtt{Q_1T_1})} & \{Lemma\ 1\ (PAR - SEQ\ result)\} \\
& \mathtt{SEQ(PAR(Q_1,\ SKIP),\ PAR(Q_2,\ T_1))} \\
= & \{Laws\ 3\ (PAR - SKIP\ unit)\ and\ 7\ (PARser\ unit)\} \\
& \mathtt{SEQ(Q_1,\ PARser(Q_2,\ T_1))}\ \Box
\end{array}
$$

4.2 The Serialisation Algorithm

In what follows we give the algorithm that guides the application of the rules of Section 4.1, using a pseudo functional language, similar to SML [9]. This algorithm basically applies the rules in the order presented in Section 4.1. Thus, initially Corollary 3 (*PAR-SEQ derived - 3*) is applied. After that, the results for preparing the processes involved in the serialisation procedure are applied. Next, the results for the base and the intermediate cases are exhaustively applied. As the synchronisation rules may introduce variables, the laws of declaration are again applied after the application of the results for the base cases. Finally, if a `PARser` construct remains, Law 8 (*PARserFail unit*) is applied to indicate that the serialisation fails.

All functions accept the global structure `p` that represents the processes to serialise. Each rule and law applied in the serialisation strategy executes only once. If `p` is transformed, the application of the rules and laws returns the transformed process; otherwise, it returns `p` unaltered.

To apply exhaustively one function (which may be the implementation of one rule or law) the function `fix` is used. This function is implemented as follows:

```
fun fix name_of_function arg =
  let val arg' = name_of_function arg
  in if arg = arg' then arg else fix name_of_function arg'
  end;
```

where `name_of_function` is the name of the function to apply and `arg` is the argument of that function. For legibility, we use abbreviations to exhaustively apply each function. For example, to apply exhaustively Rule 2 (*serialisation-1*), function `apply_serialisation_1` is introduced.

```
fun apply_serialisation_1 p = fix seq_1 p;
```

Function `apply_exhaustively` receives a list of functions and an argument. These functions are executed in the order given. If the argument is transformed by the application of one of these functions, `apply_exhaustively` is called recursively; otherwise the argument is returned unaltered. For example,

```
apply_exhaustively [apply_SEQ_assoc, apply_PAR_assoc] p
```

has the effect of applying Law 2 (*SEQ assoc*) exhaustively and, after that, Law 4 (*PAR assoc*) is also applied exhaustively. If `p` is transformed during the application of these laws, `apply_exhaustively` is called recursively and, as consequence, these laws are again exhaustively applied, in this order; otherwise `apply_exhaustively` returns `p` unaltered.

The symbol $\circ$ stands for function composition. For example, the function `serialisation`, at the beginning of the description of the algorithm, implements the serialisation strategy and is defined as a composition of functions `easy_case`, `binary_description`, `gradual_serialisation` and `failure`, in this order. This means that firstly the argument `p` is applied to function `easy_case`. The result of the application of this function, say `p'`, serves as argument to the application of function `binary_description`, and so on.

Algorithm 1 (Serialisation Strategy)

```
fun serialisation_strategy p =
  (failure ∘ gradual_serialisation ∘ binary_description ∘ easy_case) p;

fun ease_case p = (PAR_SEQ_derived_3 ∘ PARser_sym ∘ PAR_SEQ_derived_3) p;

fun binary_description p =
  if p is still a PARser process then
    apply_exaustively[apply_SEQ_assoc, apply_SEQ_SKIP_unit,
                      apply_PAR_assoc, apply_PAR_SKIP_unit,
                      apply_laws_of_declaration,
                      apply_internal_serialisation] p
  else p;

fun gradual_serialisation p=
  apply_exhaustively[apply_base_cases012, apply_laws_of_declaration
                     apply_intermediate_rules] p;

(* the function PAR_SEQ_der_2 applies Corollary PAR-SEQ derived - 2, in
   conjunction with Law PAR sym, as in function ease_case. The same
   comment is valid for the other rules. *)

fun base_cases012 p =
  apply_exhaustively[apply_ease_case, apply_PAR_SEQ_der_2,
                     apply_Serialisation_1,
                     apply_synchronisation_rules] p;
```

```
fun intermediate p =
  apply_exhaustively[apply_SEQ_assoc, apply_Serialisation_2,
                     apply_PAR_SEQ_der_2, apply_Serialisation_1] p;

fun failure p = apply_PARserFail_unit p;
```

Complexity This algorithm costs $O((m/2)^2.c\_r)$ (see [22]), where m is the number of elementary subprocesses of $\mathtt{P_x}$ and $\mathtt{P_y}$, and c_r is the cost of the implementation of one rule or law.

5 Implementation Features of the Serialisation Strategy

The aim of this section is to briefly discuss some implementation issues related to the applicability of the serialisation strategy (see [22] for a detailed discussion).

The proposed serialisation rules are applied only if the environment can satisfy the restriction imposed on the order of synchronisation of events. Thus, when implementing the rules, we need to introduce an automatic mechanism to identify such order. A simple possibility is to derive a graph from the environment description, called here *sequence graph*, defined in the sequel.

Definition 8 (*Sequence Graph***)** *Let* `P` *be an occam process deadlock free and allowing only one point of synchronisation. The* sequence graph *of* `P`*, denoted by* $GS(P)$ *is a directed graph whose set of vertices is the set of channel variables of* `P`*. Two vertices* $\mathtt{ch_1}$ *and* $\mathtt{ch_2}$ *in* $GS(P)$ *are connected by a directed path from* $\mathtt{ch_1}$ *to* $\mathtt{ch_2}$ *if and only if* $\mathtt{ch_1}$ *must synchronise before* $\mathtt{ch_2}$ *in* `P`.

Thus, to identify if restriction $\mathtt{P_xP_y}$ is satisfied by the environment `E`, it is enough to generate $GS(E)$ and to check the following condition:

$$\forall\ \mathtt{ch_i}\in\mathtt{P_x},\ \mathtt{ch_j}\in\mathtt{P_y}\ .\ \mathtt{not(path(ch_j,\ ch_i,}\ GS(E)\mathtt{))}.$$

where `path` is a function which returns `TRUE` if there exists a path from $\mathtt{ch_j}$ to $\mathtt{ch_i}$ in *GS(E)*; otherwise, it returns `FALSE`. This function may be implemented by using the classic depth-first search algorithm, by taking $\mathtt{ch_j}$ as the root of the search.

The algorithm to derive GS of an environment `E` is given in [22], as well as a proof of its correctness. Here it is enough to mention that there are only two constructs in `E` that generate edges (mandatory sequences of synchronisation) in GS:

1. the sequential operator (`SEQ`), which by definition imposes order on the execution of processes included by it;
2. the `ALT` operator, where the channel belonging to a given guard executes in sequence with the channels included in the process activated when that guard is satisfied.

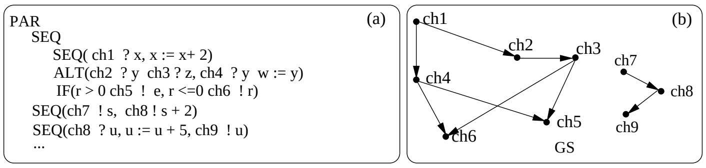

Fig. 3. An example of a program and its associated sequence graph.

Figure 3(a) and (b) show an example of an environment and the associated sequence graph, respectively. Notice that, for example, as the channels $\mathtt{ch}_1$ and $\mathtt{ch}_7$ are structured in parallel (and thus have no mandatory sequence of execution), there is no path connecting $\mathtt{ch}_1$ and $\mathtt{ch}_7$ in GS. The same reasoning is applied to channels $\mathtt{ch}_5$ and $\mathtt{ch}_6$, inside the conditional command, and channels $\mathtt{ch}_2$ and $\mathtt{ch}_4$, inside distinct branches of the `ALT` construct. Nevertheless, notice that channels $\mathtt{ch}_2$ and $\mathtt{ch}_3$ are linked by an edge, as the execution of channel $\mathtt{ch}_3$ depends on the execution of channel $\mathtt{ch}_2$. A similar situation happens, for example, to channels $\mathtt{ch}_1$ and $\mathtt{ch}_2$; the channel $\mathtt{ch}_1$ must execute before the channel $\mathtt{ch}_2$, due to the sequential operator. In fact, the conditional and the `ALT` constructs of this example are not in a simple form. In the context of our approach, processes for which the serialisation is not required (processes belonging to the environment) can include conditionals and `ALT` constructs in a more flexible form. These commands are encapsulated by a new constructor, with no semmantic effect, not given in Section 2. For conciseness reasons, we have omited the discussion about the form of all simple processes considered here.

Although edges in GS could only be generated by the `SEQ` and the `ALT` operators, a path in GS can also be generated by the parallel operator, if two parallel processes synchronise by a given channel. This is the case of the path from $\mathtt{ch}_7$ to $\mathtt{ch}_9$ in Figure 3(b): the synchronisation through channel $\mathtt{ch}_8$ generates the referred path.

6 A simple example

Consider Figure 4, which shows the application of the serialisation strategy to processes `P` and `Q`. Process `E` represents the environment. In (a) the description of the processes is given and the sequence graph of the environment, $GS(E)$, is depicted in (b). Frames (c) to (f) shows the transformations of the original `PARser` construct during the several steps of the strategy. Observe that the conditional of `P` and the `ALT` construct of `Q` are in a reduced form. Thus, the rules of the first step of the strategy are not applied. Then, the description of `P` and `Q` is made binary by applying Law 2 (*SEQ assoc*). Next, as $\mathtt{ch}_1$ is the source of all paths in $GS(E)$, Corollary 2 (*PAR-SEQ derived - 2*) is applied to serialise the sequential process of `P` which includes $\mathtt{ch}_1$, as shown in (c). Now, observe that channel $\mathtt{ch}_2$ is common to the conditional and the `ALT` constructs. Thus, a particular case of Rule 5 (*sync. IF-out-IF-ALT-F3*) is applied, as shown in (d). After that, Corollary 2 (*PAR-SEQ derived - 2*) is applied to serialise the process which includes $\texttt{ch}_3$ (see (e)) and Corollary 3 (*PAR-SEQ derived - 3*) is finally applied to serialise the processes which include $\texttt{ch}_4$ and $\texttt{ch}_5$ (see (f)).

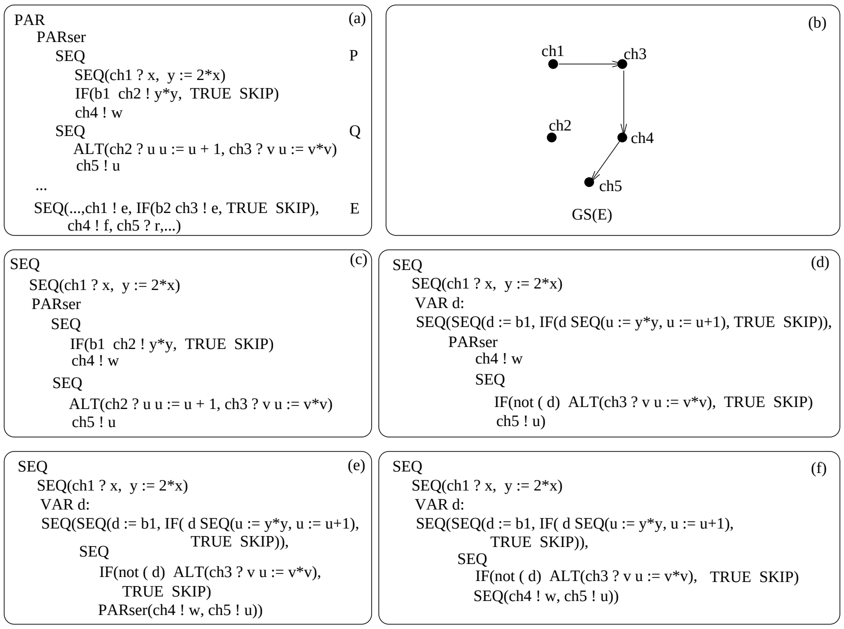

Fig. 4. A simple example of the application of the serialisation strategy.

7 Conclusions

The main contribution of this paper is to present a formal strategy for serialising parallel processes in the context of hardware/software co-design. This strategy is the closing step of the works [20] and [21], which presents an innovative approach to partitioning verification. The formalism used is occam and the algebraic laws which define its semantics. The serialisation of parallel processes is performed by applying algebraic rules, all of them proved from the basic laws of occam, as illustrate here and shown in more detail in [22].

Although this work is inserted into the PISH co-design project, the proposed approach to partitioning verification (which includes the serialisation strategy presented here) is completely orthogonal to the decision about which parts of the system will be implemented in hardware or in software, as well as the way the processes should be combined. This means that the splitting [20], the joining

[21] and the serialisation strategy can be used in conjunction with any algorithm for partitioning, provided the internal reasoning mechanism is occam. Moreover, the approach is independent of a possible evolution of the target architecture in which the system will be implemented and also independent of the improvements of the algorithms for partitioning, which are imminent, as research in co-design is in an initial stage.

Another point is that the occam results here presented do not incorporate any co-design particularity. Thus, we believe that they can be used in other areas in which the serialisation of processes is required.

Although this work does not include iteration, the serialisation strategy here presented was extended (see [22]) to consider replicated constructs (useful to deal with array of processes) in all constructs of the grammar adopted, enlarging, in this way, the expressiveness of the subset of occam considered here. To deal with procedures, functions and arbitrary loops is our immediate task of future research.

To carry out the partitioning verification automatically, Iyoda [12, 13] has developed an environment, the Partitioning Transformation System (ParTS). This tool has been developed as an extension of the occam Transformation System (OTS) [7], which performs general transformations of occam programs. While the basic laws of occam [17] implemented in OTS are useful for program transformation in general, they are not suitable to capture the partitioning problem. ParTS extends OTS with the implementation of the splitting, joining and serialisation rules, as well as the strategies which guide the application of these rules. Each transformation rule is captured in ParTS by a function in the SML [9] language. The strategies are also coded as functions, taking advantage of the pattern matching facilities of SML.

In the context of the PISH system, some small case studies have been developed, among them, the convolution program [22], an intravenous infusion control system [4] and an ATM commuter [13]. To develop medium and large case studies is the immediate task of the PISH team.

Finally, we are not aware of any other work which presents a formalisation of the partitioning problem as done here and in [20] and [21]. Nevertheless, it is worth mentioning that the kind of algebraic framework used here has been used previously to characterise and reason about a number of other applications [14, 17, 19]. All these works can be regarded as applications of refinement algebra.

References

1. E. Barros. *Hardware/Software Partitioning Using UNITY*. PhD thesis, Tübingen University, 1993.
2. E. Barros and A. Sampaio. Towards Probably Correct Hardware/Software Partitioning Using Occam. In *Proceedings of the Third International Workshop on Hardware/Software Co-design CODES/CASHE94*, 1994, 210-217, IEEE Press.
3. E. Barros, A. Sampaio, M. E. Lima, L. Silva, R. Jacobi and N. Calazans. The PISH Methodology for Hardware/Software Co-design. In *Workshop of ProTem-CC, CNPq*, 1998, 65–98.

4. E. Barros and M. V. D. Santos. A Safe, Accurate Intravenous Infusion Control System. In *IEEE Micro Magazine*, Sept/Oct 1998.
5. M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno and A. Sangiovanni-Vicentelli. A Formal Methodology for Hardware/Software Co-design of Embedded Systems. In *IEEE Micro*, August 1994.
6. R. Ernst and J. Henkel. Hardware-Software Co-design of Embedded Controllers Based on Hardware Extraction. In *Handouts of the International Workshop on Hardware-Software Co-Design*, October 1992.
7. M. Goldsmith. *The oxford* **occam** *transformation system.* Technical report, Oxford University Computing Laboratory, January 1988.
8. R. Gupta and G. De Micheli. System-level Synthesis Using Re-programmable Components. In *Proceedings of EDAC*, 1992, 2–7, IEEE Press.
9. R. Harper, D. MacQueen and R. Milner. *Standard ML.* Edinburgh University LFCS report series ECS-LFCS-86-2, 1986.
10. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice-Hall, 1998.
11. R. B. Hughes and G. Musgrave. The Lambda Approach to System Verification. In *Hardware/Software Coo-design*, Kluwer Academic Publisher, 1996, 427–451.
12. J. Iyoda, A. Sampaio and L. Silva. ParTS: A Partitioning Transformation System. In *Proceedings of FM99 (World Congress on Formal Methods)*, France, 1999.
13. J. Iyoda. *ParTS: A Support Tool to Hardware/Software Partitioning.* Master thesis, Federal University of Pernambuco, Brazil, May 2000 (in Portuguese).
14. H. Jifeng, I. Page, and J. Bowen. A Provable Hardware Implementation of occam. In *Correct Hardware Design and Verification Methods (Advanced Research Working Conference, CHARME' 93), Lecture Notes in Computer Science*, Springer Verlag, 1993, 693:214–225.
15. G. Jones and M. Goldsmith. *Programming in occam 2.* Prentice-Hall, 1988.
16. 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.
17. A. W. Roscoe and C. A. R. Hoare. The laws of **occam** programming. In *Theoretical Computer Science*, 1988, 60:177–229.
18. D. Saha, R. S. Mitra and A. Basu. Hardware/Software Partitioning Using Genetic Algorithm. In *Proc. of 10th International Conference on VLSI Design*, India, 1997, 155–160.
19. A. Sampaio. *An Algebraic Approach to Compiler Design.* Volume 4 of Algebraic Methodology and Software Technology (AMAST) Series in Computing, World Scientific, 1997.
20. 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*, September 1997, 1313:624–643.
21. L. Silva, A. Sampaio, E. Barros and J. Iyoda. An Algebraic Approach to Combining Processes in a Hardware/Software Partitioning Environment. *7th International Conference on Algebraic Methodology and Software Technology (AMAST) 98, Lecture Notes in Computer Science*, January 1999, 1548:308–324.
22. L. Silva. *An Algebraic Approach Hardware/Software Partitioning.* PhD. Thesis, Federal University of Pernambuco, Recife, Brazil, July 2000.
23. P. Welch and G. R. R. Justo. On the serialisation of parallel programs. *Occam and the Transputer - Current Developments*, 159-180, IOS Press, Amsterdam, 1991.