Using CafeOBJ to Implement a Reduction Strategy in the Context of Hardware/Software Partitioning

André Luis Silva¹  Manoel Messias Menezes²  Leila Silva³

Departamento de Ciência da Computação e Estatística
Universidade Federal de Sergipe
CEP 49100-000 - São Cristóvão-SE, Brazil

Abstract

The focus of this work is hardware/software partitioning verification. The approach uses occam as specification and reasoning language. The partitioned system is derived from the original description of the system by applying transformation rules, all of them proved from the basic laws of occam. The aim of this work is to show how the rewriting system CafeOBJ can be used to automatically prove the partitioning rules, as well as to implement the reduction strategy that guides the application of these rules. In this way, rewriting systems can be regarded as supporting tools for the construction of partitioning environments, whose emphasis is correctness.

Keywords: Rewriting Systems, Hardware/Software Partitioning, Partitioning Verification.

1 Introduction

Hardware/Software co-design or simply co-design is a design paradigm for the joint specification, design and synthesis of mixed hardware/software systems. The interest in automatic co-design techniques is driven by the increasing diversity and complexity of systems and the need for early prototypes to validate the specification and to provide the customer with feedback during the design process [7].

¹ Email: andre@ufs.br
² Email: manoel@ufs.br
³ Email: leila@ufs.br
In the last years, some tools and methodologies supporting hardware/software co-design for embedded systems have been published (for example, [4,7,9], [16]). A crucial point in co-design is the availability of algorithms to perform the partitioning of the system into hardware and software components. Partitioning is a well know NP-complete problem and in addition to the approaches mentioned above, some heuristics have been defined to guide the partitioning procedure (for example, [1,19,21]). These approaches validate the partitioned system by simulation. Few works [9,10] use formal methods to prove that some properties of the original system are preserved after the partitioning.

Since the applications are increasing in complexity and in diversity, the mere validation of the partitioned system is not enough to the guarantee safety. Thus, the formal verification of the partitioning procedure, which means the proof that the partitioned system preserves the semantics of the original description, is an important task in the co-design flow.

In [25,26] Silva et al propose a partitioning methodology with emphasis in correctness. This approach accepts as input an arbitrary occam description [14] and, from this description, derives the partitioned system by applying transformation rules (also written in occam). The proofs of these rules, given in [25], use the algebraic laws that define the semantics of occam [20]. Furthermore, in [25] the reduction strategy applied to transform the original description into the partitioned description is detailed and proved correct, using structural induction on the constructors of the adopted grammar.

Some case studies have been developed to validate the methodology proposed in [25], among them, an ATM commuter [12], an intravenous infusion system [2] and the convolution problem, used in digital signal processing [11]. To perform these case studies, an implementation of the partitioning strategy has been developed by Iyoda [11,12]. This implementation uses SML [8] and assumes that the manual proofs presented in [25] are correct. In [17] Menezes et al illustrate how rewriting systems can be used to prove the partitioning rules, by detailing the proof of one such rule, using the rewriting system BOBJ [5]. However, [17] does not discuss the mechanisation of the reduction strategy to partitioning presented in [25], in the context of rewriting systems.

The aim of this work is to complement the formal rigour of [25], by using the rewriting system CafeOBJ [18] to prove the partitioning rules, as well as to implement the reduction strategy that guides the application of these rules. To validate the reduction strategy, in the context of this work, a case study has been developed and the result is summarised in the conclusions. In fact, due to space restrictions, this paper focuses on a single phase of the partitioning approach, the splitting phase. However, the procedure here illustrated is generic and can be applied in all partitioning flow. Thus, rewriting sys-
tems can be regarded as supporting tools for the development of partitioning environments, whose emphasis is correctness.

Although this work is a novelty in the context of co-design, the use of rewriting systems to implement algebraic reduction strategies have already been proposed, for example, by Sampaio [22], in the context of compiler verification (using OBJ3 [6]) and by Lira [15], in the context of a reduction strategy for object oriented languages (using Maude [3]). Furthermore, several formalisms for hardware verification (not necessarily restricted to partitioning) have been proposed and are summarised in [13].

This paper is organized as follows. Section 2 presents the formalism used in this work. The approach to partitioning is described in Section 3, where the splitting strategy is detailed. The rewriting system CafeOBJ is briefly introduced in Section 4. Section 5 presents the of the proofs of the partitioning rules, as well as the implementation of the occam laws, of the partitioning rules and of the reduction strategy that guides the application of these rules. Finally, Section 6 gives the conclusions and some directions for future work.

2 The Formalism Used: The occam Language

Silva et al [25] use occam due to two main reasons. The first one is that occam is a powerful model to express concurrency and parallelism, essential to capture the communication among hardware and software components. The second reason is that the semantics of occam is defined by a set of algebraic laws [20]; these laws behave as axioms when proving the partitioning rules.

In fact, [25] adopts a subset of occam, defined in what follows, using the BNF style. For convenience, sometimes the syntax of occam is linearised in this paper. For example, it is possible to write $\text{SEQ}(P_1, P_2, \ldots, P_n)$ instead of the standard vertical style.

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

In what follows a short description of these commands is given (for more details see, for example, [14]). The SKIP construct has no semantic effect and always terminates successfully. STOP is the canonical process to express
deadlock and can make no further progress. The commands ch?x, ch!e and x:=e are the input, the output and the assignment commands, respectively; the communication in occam is synchronous. The IF and ALT commands select a process to execute based on the evaluation of a boolean condition (IF) or a guard (ALT). The IF choice is deterministic, that is, the lowest index boolean condition that is true activates the corresponding process; more than one boolean condition can be true at the same time. In the ALT command, if more than one guard is satisfied, the choice is non-deterministic. While the IF condition is always a boolean expression, the ALT guard involves typically input commands. The commands SEQ and PAR denote the sequential and parallel composition of processes, respectively. The constructs VAR and CHAN declare local variables and channels, respectively. The approach avoids mentioning a particular type when declaring variables and channels; any type is suitable in these declarations. Although the approach does not consider arbitrary loops, replicators are allowed on the constructors IF, ALT, PAR and SEQ and are useful to deal with arrays of processes. For conciseness reasons, this work does not consider replicators. Thus, they are omitted in the grammar description, as well as in the strategy described in Section 3.1.

Silva et al [25] extend the subset of occam to include new constructors, which can be regarded as annotations, to guide the application of the partitioning rules. These constructors have no semantic effect. The aim of the BOX constructor is to mark a chunk of code to be considered as a whole during the partitioning procedure. The CON constructor indicates that a process is introduced by the partitioning strategy and does not belong to the original description of the system. The PARhw and PARsw constructors are used to annotate a hardware and a software component, respectively. The PARpar and PARser constructors indicate the way (in parallel or in series, respectively) the process should be combined in each component.

As mentioned previously, the semantics of occam is defined by a set of algebraic laws. In this work, we only present the laws necessary to understand the strategy described in Section 3. Each law has a name, that indicates its use, and a number. The operational justification of each law is taken from [20].

The following two laws express the identity of sequential and parallel compositions.

**Law 2.1 (SEQ-SKIP unit)** \(\text{SEQ}(\text{SKIP}, P) = \text{SEQ}(P, \text{SKIP}) = \text{SKIP}\)

**Law 2.2 (PAR-SKIP unit)** \(\text{PAR}(\text{SKIP}, P) = \text{PAR}(P, \text{SKIP}) = \text{SKIP}\)

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 first law. The PAR operator is associative and commutative and these facts are captured by laws 2.4 (PAR assoc) and 2.5 (PAR sym), respectively.

Law 2.3 (SEQ assoc) \( \textsc{SEQ}(P_1, P_2, \ldots, P_n) = \textsc{SEQ}(P_1, \textsc{SEQ}(P_2, P_3, \ldots, P_n)) \).

Law 2.4 (PAR assoc) \( \textsc{PAR}(P_1, P_2, \ldots, P_n) = \textsc{PAR}(P_1, \textsc{PAR}(P_2, P_3, \ldots, P_n)) \).

Law 2.5 (PAR sym) \( \textsc{PAR}(P_1, P_2) = \textsc{PAR}(P_2, P_1) \).

A conditional \( C \) is either a boolean expression followed by a process or an IF constructor. Law 2.6 (IF assoc) is applied to unnest IF constructors, such that all arguments are boolean expressions followed by a process. The ALT constructor obeys a similar law.

Law 2.6 (IF assoc) \( \textsc{IF}(C_1, \textsc{IF}(C_2), C_3) = \textsc{IF}(C_1, C_2, C_3) \).

Law 2.7 (ALT assoc) \( \textsc{ALT}(\textsc{ALT}(G_1), G_2) = \textsc{ALT}(G_1, G_2) \).

The next law allows us to deal with IF constructors when they are nested as processes and not as conditionals. In the rest of this work, the notation \( \textsc{IF}_{k=1}^n b_k P_k \) is used to denote \( \textsc{IF}(b_1 P_1, b_2 P_2, \ldots, b_k P_k) \). Similar notation is also adopted for the constructors VAR and SEQ.

Law 2.8 (\( \wedge \)-IF distrib)
\( \textsc{IF}(C, \textsc{IF}_{k=1}^n b_k P_k) = \textsc{IF}(C, \textsc{IF}_{k=1}^n \textsc{IF}(b \wedge b_k P_k)) \).

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

Law 2.9 (SEQ-IF distrib)
\( \textsc{SEQ}(\textsc{IF}_{k=1}^n c_k P_k, Q) = \textsc{IF}_{k=1}^n \textsc{SEQ}(c_k P_k, Q) \).

Assignment distributes rightward through a conditional, changing occurrences of the assigned variables in the condition.

Law 2.10 (assignment-IF distrib)
\( \textsc{SEQ}(x := e, \textsc{IF}_{k=1}^n c_k P_k) = \textsc{IF}_{k=1}^n c_k [e/x] \textsc{SEQ}(x := e, P_k) \).

The notation \( P[e/x] \) denotes the result of substituting each expression \( e_i \) of \( e \) for each free occurrence of each variable \( x_i \) of \( x \) in \( P \), provided \( e \) and \( x \) have the same length. 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.

The operator IF can distribute rightward through SEQ, provided some conditions are satisfied.

Law 2.11 (SEQ-IF right distrib)
\( \textsc{SEQ}(P, \textsc{IF}_{k=1}^n c_k Q_k) = \textsc{IF}_{k=1}^n c_k \textsc{SEQ}(P, Q_k) \).
if $c_1 \lor c_2 \lor \ldots \lor c_n \equiv \text{TRUE}$ and no variable in any $c_k$ is altered by $P$.

The empty multiple assignment terminates without changing the state.

**Law 2.12 (SKIP)**

$<> := <> = \text{SKIP}.$

There is no point in assigning to a variable at the very end of its scope, since the value given to it can have no effect. In the same way, if a declared variable is never used, its declaration has no effect. The two next laws capture these facts.

**Law 2.13 (assignment elim)**

$\text{VAR } x: (<x> + y) := (<e> + f) = \text{VAR } x: (y := f).$

**Law 2.14 (VAR elim)**

$\text{VAR } x: P = P$, if $x$ is not free in $P$.

The scope of a bound variable may be increased without effect, provided it does not interfere with another variable having the same name. **Law 2.15 (VAR-IF distrib)** states that if each subprocess of a conditional declares the variable $x$, and this variable is not free in the boolean conditions, then the declaration may be moved outside the constructor. Similar reasoning can be used to increase the scope of declaration of channels, as in the **Law 2.16 (CHAN-PAR)**.

**Law 2.15 (VAR-IF distrib)**

$\text{IF}_{k=1}^{n} c_k \ (\text{VAR } x: P_k) = \text{VAR } x: \text{IF}_{k=1}^{n} c_k P_k$, provided $x$ is free in no $c_k$.

**Law 2.16 (CHAN-PAR)**

$\text{PAR}(\text{CHAN } ch: P, Q) = \text{CHAN } ch: \text{PAR}(P, Q)$.

It does not matter whether the variables are declared in one list or singly.

**Law 2.17 (VAR assoc)**

$\text{VAR } x_1: (\text{VAR } x_2: (\ldots \text{VAR } x_n: P)) \ldots) = \text{VAR } x_1, x_2, \ldots, x_n: P$.

The **VAR** operator can distribute over **SEQ**.

**Law 2.18 (VAR-SEQ 1)**

$\text{SEQ}(\text{VAR } x: P, Q) = \text{VAR } x: \text{SEQ}(P, Q)$, provided $x$ is not free in $Q$.

We can change the name of a bound variable, provided the new name is not already used for a free variable.

**Law 2.19 (VAR rename)**

$\text{VAR } x: P = \text{VAR } y: P[y/x]$, if $y$ is not free in $P$. 
The constructors introduced in [25], to guide the partitioning strategy, have no semantic effect. For the BOX and the PARpar constructors, for example, this fact is captured by the next two laws.

**Law 2.20 (PARpar unit)**

\[ \text{PARpar}(P) = \text{PAR}(P). \]

**Law 2.21 (BOX unit)**

\[ \text{BOX}(P) = P. \]

### 3 The Partitioning Approach

The approach to partitioning proposed in [25] derives the description of the partitioned system, from the original description, by applying transformation rules specific for partitioning and some occam laws. The whole process can be summarised by:

\[
\begin{align*}
\text{Original System} &= \langle \text{partitioning rules, occam laws} \rangle \\
\text{Partitioned System}
\end{align*}
\]

In fact, the partitioning flow is divided into three phases: splitting, definition of components and joining. The aim of the splitting phase is to transform the original description of a system into a description in the splitting normal form.

**Definition 3.1 (Splitting normal form)** A process is in the splitting normal form if it has the following structure:

\[
\text{CHAN } ch_1, ch_2, \ldots, ch_m : \text{PAR}(P_1, P_2, \ldots, P_r)
\]

where each \( P_i, 1 \leq i \leq r \), is simple.

Basically, each simple process executes at most one atomic process, which can be either a primitive process (SKIP, STOP, \( x:=e \), \( ch?x \), \( ch!e \)) or a set of processes encapsulated by a BOX constructor. The detailed forms of simple processes are not relevant in the context of this work and can be found in [24,25,26].

The aim of this transformation is to isolate all relevant process to be analysed in the next phase, where the decision about the components’ composition is taken. Since all processes are in the same level of the description (immediately under the external PAR constructor) and the PAR operator is associative and commutative, the splitting normal form allows full flexibility when grouping processes either in hardware or in software components. Notice that any permutation of processes is possible, without changing the semantics of the
In the phase of definition of components the description is not substantially transformed. In this phase, heuristics are applied to decide which processes will compose the hardware and the software components, as well as the way these processes should be combined, if in series or in parallel. The metrics considered by the partitioning algorithm include the functional similarity, the similarity of the parallelism degree, the concurrent behaviour and the data-dependency among processes. The decision taken in this phase is expressed using the constructors \texttt{PARhw}, \texttt{PARsw}, \texttt{PARser} and \texttt{PARpar} and the associativity and commutativity of the parallel operator. Fig. 1 illustrates a system description after this phase. For example, processes P\textsubscript{9}, P\textsubscript{10}, P\textsubscript{15} and P\textsubscript{18} belong to the same hardware component (annotated by \texttt{PARhw}). Furthermore, it is required that in the partitioned system the processes P\textsubscript{15} and P\textsubscript{18} execute in parallel (\texttt{PARpar}) and the others two in series with them (\texttt{PARser}).

Fig. 1. An example of a description generated after the phase of definition of components.

The final description of the partitioned system is in fact achieved in the joining phase, by applying transformations rules, like in the splitting phase. The aim of this phase is to transform the description generated in the splitting phase (and annotated in the phase of definition of components) into a description in the joining normal form, which expresses the partition of the system into hardware and software components.

\textbf{Definition 3.2} (Joining Normal Form) A description is in the joining normal form if it has the following structure:

\[
\text{CHAN } \mathit{ch}_1, \mathit{ch}_2, \ldots, \mathit{ch}_m; \text{ PAR(Q}_1, Q_2, \ldots, Q_t)\]

where, by comparing with Definition 3.1, \(s \leq m, t \leq r\) and each \(P_i, 1 \leq i \leq r\) belongs to exactly one \(Q_j, 1 \leq j \leq t\). Each \(Q_k, 1 \leq k \leq t-1\) represents either a software or a hardware component and \(Q_t\) represents the interface process between hardware and software.

In total, ninety rules for partitioning have been proposed and proved manually in [25]. To show how rewriting systems (in particular, CafeOBJ) could
be used in the mechanisation of the proofs of such rules and in the implementa-
tion of the reduction strategy to partitioning, this paper focuses on the split-
ing phase. However, the procedure here illustrated is generic and can be
extended to the whole partitioning flow. In the next section, the reduction
strategy to the splitting normal form is detailed.

3.1 The Splitting Strategy

The aim of the splitting phase is to transform the original description of the
system into a description in the normal form given by Definition 3.1.

To achieve the normal form, a reduction strategy is applied, based on the
application of algebraic rules. This strategy comprises two main steps. In the
first step, the IF and ALT commands are reduced to the simple form. In the
second step, the processes of the description generated in the first step are put
in parallel. To transform an arbitrary program (described using the grammar
of Section 2) into a description in the splitting normal form, eight rules are
necessary and sufficient.

Due to conciseness reasons, in this paper we detail some of such rules. The
complete description of the strategy can be found in [24,25]. Rule 3.1 (IF
fragmentation) transforms an arbitrary conditional into a sequence of binary
conditionals, one for each branch of the original conditional. The application
of this rule allows a flexible analysis of each subprocess of a conditional, by
the phase of definition of components.

\textbf{Rule 3.1 (IF fragmentation)}

\begin{align*}
\text{IF}_{k=1}^{n} b_k & P_k \\
= & \text{VAR}_{k=1}^{n} c_k \\
\text{SEQ} \\
\text{BOX(SEQ}_{k=1}^{n} c_k := \text{FALSE})} \\
\text{IF}_{k=1}^{n} b_k & c_k := \text{TRUE} \\
\text{SEQ}^{n}_{k=1} \text{IF}(c_k P_k, \text{TRUE SKIP}) \\
\text{provided each } c_k \text{ is a fresh variable (occurring only where }
\text{explicitly shown).}
\end{align*}

Notice that the role of the first IF operator on the right-hand side of Rule
3.1 (IF fragmentation) is to make the choice and to allow the subsequent
conditionals to be carried out in the sequence. This is why the fresh variables
c_1, c_2, \ldots, c_n are introduced. Otherwise, the execution of one conditional
could interfere in the condition of a subsequent one in the sequence. Note that both
sides of the rule have the same behaviour. On the left-hand side, the first
b₁ to be true activates the corresponding process P₁. On the right-hand side, when b₁ is true, c₁ is true and the others cₖ, k≠₁, are false. Thus, only P₁ executes.

**Rule 3.2 (ALT fragmentation)**

\[
\text{ALT}_{k=1}^{n}(b_k \& g_k \ P_k) = \\
\text{VAR}_{k=1}^{n} c_k \\
\text{SEQ} \\
\text{BOX}(\text{SEQ}_{k=1}^{n} c_k := \text{FALSE}) \\
\text{ALT}_{k=1}^{n} (b_k \& g_k \ c_k := \text{TRUE}) \\
\text{SEQ}_{k=1}^{n} (\text{IF}(c_k P_k, \text{TRUE SKIP}))
\]

provided each cₖ is a fresh variable (occurring only where explicitly shown).

The ALT constructor obeys a similar rule, Rule 3.2 (ALT fragmentation). The difference between IF and ALT rules resides on the substitution of the conditional process with multiple branches by an equivalent ALT process (on both sides of the rule). Moreover, observe that, as the BOX process is atomic, the first process on the right-hand side of these rules is in a simple form. In addition, the commands IF and ALT with multiple branches execute only a primitive process (an assignment to a boolean variable) and thus, are also simple. In this way, the application of these rules unifies the treatment of ALT and IF commands, since only binary conditionals may not already be simple, as Pₖ may not be atomic. In this case, rules are necessary to distribute a conditional over SEQ, PAR and IF constructors; in the rest of this work these rules are referred as rules of IF distribution. The aim of these rules is to move all conditionals to the most internal level of the description. Rule 3.3 (IF-SEQ distrib), for example, is used to distribute the IF constructor over the SEQ constructor.

**Rule 3.3 (IF-SEQ distrib)**

\[
\text{IF}(b \ \text{VAR} \ x : \ \text{SEQ}_{k=1}^{n} P_k, \ \text{TRUE SKIP}) = \\
\text{VAR} \ c : \ \text{SEQ}(c := b, \ \text{VAR} \ x : \ \text{SEQ}_{k=1}^{n} \ \text{IF}(c P_k, \ \text{TRUE SKIP}))
\]

provided c is a fresh variable.

After reducing the IF and the ALT commands, the description of the system is a hierarchy of parallel and sequential constructors, and in the second step, these constructors are reduced with the aim of achieving the splitting normal form. The second step begins by applying laws 2.1 (SEQ-SKIP unit), 2.2
(PAR-SKIP unit), 2.3 (SEQ assoc) and 2.4 (PAR assoc) to transform the
description generated in the first step into a description in a binary form.
Law 2.19 (VAR rename) can also be applied to guarantee that the names of
local variables are disjoint from the names of global variables. After that, the
commands SEQ and PAR are finally reduced.

Rule 3.4 (SEQ splitting) is used to put in parallel two processes originally
in sequence. Since sequential processes can have data dependency and as oc-
cam parallel processes do not share variables, to execute this transformation
communication is introduced. In this rule, each original process is closed, in
the sense that, all variables used and assigned are local. The original processes
interact with the environment through a controlling process (annotated with
the CON operator), except, of course, when the original process includes input
or output commands. The controlling process acts as an interface between
the original processes, and between them and the environment.

**Rule 3.4 (SEQ splitting)**

\[
\begin{align*}
\text{VAR } z & : \text{ SEQ}(P_1, P_2) \\
= & \text{ CHAN } ch_1, ch_2, ch_3, ch_4 : \\
\text{PAR} & \\
\text{VAR } x_1, z_1 : \text{ SEQ}(ch_1 ? x_1, z_1, P_1, ch_2 ! x'_1, z'_1) \\
\text{VAR } x_2, z_2 : \text{ SEQ}(ch_3 ? x_2, z_2, P_2, ch_4 ! x'_2, z'_2) \\
\text{CON}(\text{VAR } z : \text{ SEQ}(\text{SEQ}(ch_1 ! x_1, z_1, \text{ch}_2 ? x'_1, z'_1), \\
\text{SEQ}(\text{ch}_3 ! x_2, z_2, \text{ch}_4 ! x'_2, z'_2))) \\
\text{provided free}(P_1) = x_1 \cup z_1, \text{ free}(P_2) = x_2 \cup z_2, x := x_1 \cup x_2, \\
z = z_1 \cup z_2, x_1' \cup z_1' = \text{ass}(P_1), i = 1, 2, \text{ch}_1, \text{ch}_2, \text{ch}_3 \text{ and ch}_4 \\
\text{occur only where explicitly shown.}
\end{align*}
\]

The term \( z \) is a list of local variables of \( \text{SEQ}(P_1, P_2) \), \( \text{free}(P) \) stands
for free variables of \( P \) and \( \text{ass}(P) \) stands for free variables assigned within \( P \).
Observe that, although all processes are in parallel on the right-hand side of
Rule 3.4 (SEQ splitting), the controlling process guarantees that \( P_2 \) executes in
sequence with \( P_1 \) (\( P_2 \) only executes after the synchronisation through channel
\( ch_2 \)). Thus, both sides of the rule have the same behaviour.

To reduce the PAR constructor the strategy includes a rule similar to Rule
3.4 (SEQ Splitting). Furthermore, an auxiliary rule to distribute communication
is also necessary. These three rules are referred as rules of the second
step in the rest of this work. After this step, laws 2.4 (PAR assoc) and 2.16
(CHAN- PAR) are applied, to achieve the splitting normal form.

Each of these rules is proved in [25], in the style of the proof of the binary case of Rule 3.3 (IF-SEQ distrib), shown in what follows. The proof begins with the right-hand side of the equation and by applying some occam laws, the left-hand side is derived. This proof uses the Lemma 3.3 (IF branch elim) proved in [25] and given in what follows.

**Lemma 3.3 (IF branch elim)**

\[
\text{IF}(b_1 \ P_1, \ b_2 \ P_2, \ ..., \ b_i \ P_i, \ ..., \ b_j \ P_j, \ ..., \ b_n \ P_n) = \\
\text{IF}(b_1 \ P_1, \ b_2 \ P_2, \ ..., \ b_i \ P_i, \ ..., \ b_n \ P_n)
\]

provided \( b_j \Rightarrow b_1 \).

**Proof.**

\[
\text{VAR} \ c : \ \text{SEQ}(c := b, \ \text{VAR} \ x : \ \text{SEQ}(\text{IF}(c \ P_1, \ \text{TRUE} \ \text{SKIP}), \ \text{IF}(c \ P_2, \ \text{TRUE} \ \text{SKIP})))
\]

\[
= \{ \text{Law (2.9) < SEQ - IF distrib >} \}
\]

\[
\text{VAR} \ c : \ \text{SEQ}(c := b, \ \text{VAR} \ x : \ \text{IF}(c \ \text{SEQ}(P_1, \ \text{IF}(c \ P_2, \ \text{TRUE} \ \text{SKIP}))) \ \text{TRUE} \ \text{SEQ}(\text{SKIP}, \ \text{IF}(c \ P_2, \ \text{TRUE} \ \text{SKIP})))
\]

\[
= \{ c \notin \text{free}(P_1), \text{laws (2.11) < SEQ - IF right distrib > and (2.1) < SEQ - SKIP unit >} \}
\]

\[
\text{VAR} \ c : \ \text{SEQ}(c := b, \ \text{VAR} \ x : \ \text{IF}(c \ \text{IF}(c \ \text{SEQ}(P_1, \ P_2), \ \text{TRUE} \ P_1) \ \text{TRUE} \ \text{IF}(c \ P_2, \ \text{TRUE} \ \text{SKIP})))
\]

\[
= \{ \text{Law (2.8) < ∧ - IF distrib > and Boolean algebra} \}
\]

\[
\text{VAR} \ c : \ \text{SEQ}(c := b, \ \text{VAR} \ x : \ \text{IF}(c \ \text{SEQ}(P_1, \ P_2), \ c \ P_1, \ c \ P_2, \ \text{TRUE} \ \text{SKIP}))
\]

\[
= \{ \text{Lemma (3.3) < IF branch elim >, c \notin x, law (2.15) < VAR - IF distrib >} \}
\]

\[
\text{VAR} \ c : \ \text{SEQ}(c := b, \ \text{IF}(c \ \text{VAR} \ x : \ \text{SEQ}(P_1, \ P_2), \ \text{TRUE} \ \text{VAR} \ x : \ \text{SKIP}))
\]

\[
= \{ \text{Laws (2.14) < VAR elim > and (2.10) < assignment - IF distrib >} \}
\]

\[
\text{VAR} \ c : \ \text{IF}(b \ \text{SEQ}(c := b, \ \text{VAR} \ x : \ \text{SEQ}(P_1, \ P_2)), \ \text{TRUE} \ \text{SEQ}(c := b, \ \text{SKIP}))
\]

\[
= \{ c \notin \text{free}(P_1) \cap \text{free}(P_2), \text{laws (2.15) < VAR - IF distrib >, (2.13) < assignment elim > and (2.14) < VAR elim >} \}
\]

\[
\text{IF}(b \ \text{VAR} \ x : \ \text{SEQ}(P_1, \ P_2), \ \text{TRUE} \ \text{SKIP})
\]

\( \square \)

The splitting strategy can be summarised by the algorithm described below:
Algorithm 1 (Splitting Strategy)

Step 1: Apply exhaustively laws 2.6 (IF assoc), 2.7 (ALT assoc) and 2.17 (VAR assoc).
Step 2: Apply exhaustively rules 3.1 (IF fragmentation), 3.2 (ALT fragmentation) and the rules of IF distribution.
Step 3: Apply exhaustively laws 2.1 (SEQ-SKIP unit), 2.3 (SEQ assoc), 2.2 (PAR-SKIP unit) and 2.4 (PAR assoc).
Step 4: Apply Law 2.19 (VAR rename), if necessary.
Step 5: Apply exhaustively the rules of the second step.
Step 6: Apply exhaustively laws 2.4 (PAR assoc) and 2.16 (CHAN-PAR).

4 CafeOBJ

CafeOBJ [18] is a new generation algebraic specification and programming language. As a successor of the OBJ family (OBJ1, OBJ2, OBJ3) [6], it inherits features such as: powerful typing system with sub-types; sophisticated module composition system, featuring several kinds of imports; parameterised modules; views for instantiating parameters and the module expressions, among other issues. CafeOBJ implements new paradigms, such as rewriting logic and hidden algebra, as well as their combinations. It is mainly used for system specification, formal verification of specifications, rapid prototyping, programming and automatic theorem proving.

CafeOBJ is chosen due to some characteristics, among them, the availability of documentation, the facility in the use of the reduction mechanism, the possibility of applying the rules in two ways and in subterms of the term to be reduced.

A module in CafeOBJ has the syntax defined by module <mod_id> mod_elem, where <mod_id> is the name of the module and mod_elem is an element of the module. A module element is either an import declaration, a sort declaration, an operator declaration, a record declaration, a variable declaration, an equation declaration or a transition declaration. These elements are structured into three main parts. The first part, imports, specifies which modules should be imported, that is, inherited. There are three forms of importing modules: protecting (the imported module can not be changed), extending (the imported module can be extended, but the original description remains unchanged) and using (the imported module can be extended or can change the original description). The second part, signature, declares sorts, operators, records and subsorts used by the module. Finally, axioms includes declaration of variables, equations and transitions and expresses the behaviour of the module.

To illustrate a module description in CafeOBJ, considers the following example. The module SQRINT inherits the sorts and the operators defined in
modules INT and NAT. Section signature declares two sorts Nat and Int. The symbol \( \lt \) means that Nat is a subsort of Int. The \texttt{sqr} operator, introduced by \texttt{op}, receives an integer argument and returns its square, which is a natural number. The behaviour of \texttt{sqr} is captured by an expression, introduced by \texttt{eq}.

\% module SQRINT{ 
  imports {
    protecting (NAT)
    protecting (INT)
  }
  signature {
    [Nat < Int]
    op sqr : Int -> Nat
  }
  axioms {
    var I : Int 
    eq sqr(I) = (I * I) .
  }
}

5 The Mechanisation of the Splitting Strategy using CafeOBJ

The mechanisation of the splitting strategy, using CafeOBJ, has three main stages: the implementation of the occam laws and of the splitting rules (Section 5.1), the proof of such rules (Section 5.2) and the implementation of the reduction strategy (Section 5.3). In what follows, we detail these stages.

5.1 The Implementation of the occam Laws and of the Splitting Rules

Before implementing the occam laws and the splitting rules a module BASE is defined. This module includes the declaration of sorts and operators that are used by other modules. For example, the module BASE includes the sorts Process, Expression, List_Processes and List_Expressions, used to declare a process, an expression, a list of processes and a list of expressions, respectively. The symbol "," is an example of an operator of this module, suitable for constructing lists of processes, channels and variables.

The occam laws are implemented in the module OCCAM-LAWS. In what follows, a fragment of this module is shown to illustrate the implementation of laws 2.1 (SEQ-SKIP unit), 2.9 (SEQ-IF distrib) and 2.11 (SEQ-IF right distrib), used in the two initial steps of the proof of Rule 3.3 (IF-SEQ distrib), presented in Section 3.1. In the remainder of this paper, the lines of the code are numbered for didactic reasons.

% Lines 5 to 10 describe the signatures of the operators necessary to the implementation of the occam laws. Lines 18 and 19 express the implementation of Law 2.1 (SEQ-SKIP unit) and capture the fact that the sequential composition of a process P with SKIP has no effect. Line 30 implements Law 2.9 (SEQ-IF distrib). This implementation uses auxiliary functions, conditions and processes, to collect the conditions and the processes of the list C of conditional processes, respectively. The implementation of these functions are in lines 20 to 27. The distribution of the SEQ operator over the IF operator is performed using the ":::" operator (lines 28 and 29). This operator receives two lists of the same length (a list of boolean expressions (G, E) and a list of processes (P, L)) and a process Q. It generates a list of conditional processes, in the form of c_i SEQ(P_i, Q), where c_i (G, E) and P_i ∈ (P, L), 1 ≤ i ≤ n, n = |G, E| = |P, L|. Analogously, the implementation of Law 2.11 (SEQ-IF right distrib) uses the same auxiliary operators and functions.

The splitting rules are implemented in the module RULES. Due to space restrictions, in this paper we only show the implementation of the main equation of Rule 3.3 (IF-SEQ distrib). Several auxiliary operators and functions used in the implementation of this rule are not detailed. The complete description of the module RULES is available in [23].

% 1 module RULES { 2 imports { protecting(OCCAM-LAWS) } 3 signature { ... } 4 axioms { ... 5 eq gerIf(c (P,LP)) = IF(c P,true SKIP), gerIf(c LP) . 6 eq gerIf(c P) = IF(c P, true SKIP) . 7 eq [IF-SEQ-distrib]: IF(b (VAR LV : SEQ(LP)),true
Line 7 implements the main equation of Rule 3.3 \((IF-SEQ\ distrib)\). The \texttt{getV}\ operator is used to create new variables (variable \(c\) in the description of Rule 3.3 \((IF-SEQ\ distrib)\)). To perform this task, the operator receives as a parameter the amount of variables to be created (in this case, a single one) and a list of the processes associated with these new variables. The purpose of passing this list is to identify which variable names have already been used, to avoid name conflicts. The \texttt{gerIF}\ operator is responsible for generating the conditional processes that appear on the right-hand side of Rule 3.3 \((IF-SEQ\ distrib)\), in the form \(IF(c\ P_k, TRUE\ SKIP)\). To do this, the operator receives the new variable (generated by \texttt{getV}) and the list of processes \(P_1, P_2, \ldots, P_k\). The implementation of \texttt{gerIf}\ is given in lines 5 and 6.

5.2 Proving the Splitting Rules

To guarantee the correctness of the reduction strategy, it is necessary to prove the splitting rules. The mechanisation of these proofs follows a style similar to the one illustrated in Section 3.1. Thus, to prove Rule 3.3 \((IF-SEQ\ distrib)\), we start with the right-hand side of the rule and, by successive application of the occam laws already implemented, we derive the left-hand side of the rule. In what follows, we illustrate the reductions performed in the two initial steps of the proof described in Section 3.1, for conciseness reasons. Furthermore, we detail only the reductions concerning the application of the occam laws. Auxiliary procedures necessary to complete these steps of the proof are not detailed.
Initially, the modules OCCAM-LAWS and BASE are loaded and the constants used in the conduction of the proof are declared. The proof starts with the description of the right-hand side of Rule 3.3 (IF-SEQ distrib), by applying the start command (Line 1). Likewise the steps described in Section 3.1, Law 2.9 (SEQ-IF distrib) is applied in Line 2. The parameters (2 1 2 2) are used to capture the term in which the transformation is applied, in this case, the most internal SEQ constructor. This constructor belongs to the second argument of the most external VAR operator (the first one is the variable c). This argument, which is also a SEQ constructor, has itself a single argument: the list of processes expressed using the "," operator. The term to be reduced belongs to the second process of this list and is the second argument of the most internal VAR operator. After the application of Law 2.9 (SEQ-IF distrib) and the reduction of the auxiliary operators and functions detailed in Section 5.1, we obtain a description similar to the one achieved after the first step of the manual proof of Rule 3.3 (IF-SEQ distrib), described in Section 3.1 (see Line 5).

After that, Law 2.11 (SEQ-IF right distrib) is applied twice (lines 6 and 9) and after performing auxiliary reductions, we achieve a description in which the IF operators are distributed rightward over the SEQ operator (Line 11). In the next step, Law 2.1 (SEQ-SKIP unit) is applied three times (lines 12, 14 and 16) to eliminate the SKIP processes inside the sequential constructors. The result achieved in Line 17 is similar to the one presented after the second step of the manual proof of Rule 3.3 (IF-SEQ distrib), in Section 3.1. The proof proceeds analogously. The remainder of the splitting rules is proved similarly.

5.3 The Implementation of the Splitting Strategy

To implement the strategy, using CafeOBJ, the modules OCCAM-LAWS and RULES are divided in six modules, named STEP-i, 1 ≤ i ≤ 6. Module STEP-i implements the set of rules and laws used in the i-th step of Algorithm 1 (Splitting Strategy). The convergence of this algorithm is proved in [25]. All modules inherit the general sorts and operators implemented in module BASE, as mentioned previously. Fig. 2 depicts the global architecture of the reduction environment, where the dashed arrows represent the dependency between modules and the continuous arrows represent the flow of execution of Algorithm 1 (Splitting Strategy).

The implementation of the strategy in CafeOBJ is given below. The open
command allocates temporally in memory the current module, allowing the reduction of a term of this module. Initially, an arbitrary occam description is passed as an argument of the start command. The laws and rules implemented in module \texttt{STEP-1} are then automatically applied, by using the apply command (Line 4). Due to limitations of CafeOBJ, the results obtained after these reductions must be manually copied to act as input for the second step. The remainder of the strategy proceeds analogously.

1 in \texttt{STEP-1} 2 open \texttt{STEP-1} 3 start \texttt{<original description> <4 
apply red at term >} 4
6 close 7 in \texttt{STEP-2} 8 open \texttt{STEP-2} 9 start \texttt{<results obtained in the \texttt{STEP-1} >
apply red at term >} 11 \texttt{<copy of the results obtained in \texttt{STEP-2}>}
12 close 13 ... 14 in \texttt{STEP-6} 15 open \texttt{STEP-6}
16 start \texttt{<results obtained in the \texttt{STEP-6} >
apply red at term >} 18 close

6 Conclusions

Formal verification in co-design is in an incipient stage and few approaches deal with this issue. The contribution of this work is to complement the formal rigour of the work proposed by Silva \textit{et al} [25,26], in the sense that our work mechanises the proofs of the rules used in one of the partitioning phases described in [25,26], as well as implements the reduction strategy that guides the application of such rules.

The use of CafeOBJ was of great value to this work. The specification of the subset of occam and the implementation of the occam laws and of the partitioning rules impose no problems. Moreover, the implementation of the reduction strategy was also developed naturally. Thus, this work could also be regarded as a contribution to the CafeOBJ community, as it could be considered as a programming transformation case study, in the context of this rewriting system. On the other hand, for the co-design community, this work

\textbf{Fig. 2.} Global architecture of the reduction environment.
suggests that rewriting systems could be considered as supporting tools for methodologies whose emphasis is correctness.

We have detected three omissions of basic occam laws in the manual proofs of the splitting rules, presented in [25]. This fact corroborates the importance of the use of rewriting systems to perform automatic proofs.

To validate the strategy presented, we have developed a case study, the convolution problem, also described in [26]. In this case study, 117,350 rewritings are necessary, consuming 76.35 s.

Although this paper is restricted to the splitting phase, the procedure used in the mechanisation of the proofs and in the implementation of the reduction strategy is generic and can be extended to the overall partitioning flow. The next step is the mechanisation of the proofs of the joining rules, as well as the implementation of the reduction strategy that guides the application of these rules. As a consequence, a partitioning environment, using CafeOBJ as supporting tool, will be constructed. This environment will be used in the development of larger case studies.

References


