Interactive equivalence prover for distributed system formal sequentialization

Francesc Babot, Miquel Bertran, and August Climent
Grup de Recerca en Sistemes Distribuïts.
Enginyeria i Arquitectura La Salle.
Universitat Ramon Llull.
C/Quatre Camins 2, 08022 Barcelona (Spain).
{fbabot,miqbe,augc}@salle.url.edu

Abstract
The theory of formal sequentialization proofs of distributed system models has been covered in the literature, as is detailed in the introduction. Formal communication elimination is an essential part of these proofs. In spite of the impressive reduction on the upper bound of the number of states obtained in some applications, no interactive tool to aid in the construction of these proofs has been reported in the literature. This is accomplished in this communication. The design of a computer aided verification tool, under development, is outlined. It has simple commands for the application of equivalence laws and more complex ones attempting to apply sequences of them to attain some objective. They are organized in layers. The current version of the tool has been applied in quite complex sequentialization proofs. The formal simplification of a communications protocol is outlined, and serves illustrative purposes.

1 Introduction
The prover works on models expressed in imperative languages with explicit parallelism and communication statements. These provide an intuitive, explicit, and complete framework to express system models and distributed programs with perspective and clarity. This is important in verification. OCCAM [14, 15, 16], the simple programming language SPL of Manna and Pnueli [17, 18], PROMELA of the SPIN model checker [13], and the shared-variable language++, SVL++, in [11] are some representatives.

The only viable verification alternative when the state transition system of distributed programs has infinitely many states or when model checking [13, 19, 10, 21, 20] runs into the state explosion problem is interactive verification. Formal sequentialization, or Distributed program sequentialization (DPS) [5, 7, 8, 4, 6], is an interactive verification approach constructing a formal static proof which attempts to obtain a simplified sequential program equivalent, in a sense to be clarified later, to the original distributed program, inner communication-free, and with less variables. Whereas other simplification methods such as partial order reduction [1] work on the transition system, whose complexity is exponential in the size of the program, DPS works on the program, whose complexity is linear.

Internal synchronous communication elimination is an important step of DPS. It applies laws as reductions. A set of laws for OCCAM was given in [22]. Communication closed layered systems were introduced in [12], and some laws for them were given in [11] in the framework of SVL++. Some laws for SPL were given in [17]. Communication elimination laws
where proposed in [5], proving the requirement of avoiding strong fairness. An equivalence suitable for DPS was presented and studied in [8]. Interface equivalence was introduced since it was the minimal extension that was needed while keeping this work within the Manna-Pnueli framework. Communication elimination laws and a communication elimination algorithm have been reported in [2]. A generalization of automatic communication elimination for a wide class of statements with selection substatements was introduced in [6]. A DPS equivalence proof of a pipelined processor model was covered in [8] as well. The impressive reduction of $2^{-67}$ in the upper bound on the size of the state vector of the model obtained with this DPS proof was reported in [2]. Other DPS proofs were given in [5].

Although a computer aided verification tool adapted to sequentialization proofs is essential to carry them out, the design outline of the tool which is being developed for this purpose, and which has been used in the proofs published so far, has never been presented in the literature before. This communication accomplishes this pending task. The commands of the prover and their layered organization are outlined, together with an illustrative application to a communications protocol.

2 Notation and background notions

Notation syntax In the current version of the tool, programs to be proved are written in an internal ternary tree structured notation, PADD [9]. The translation of other notations to PADD is envisaged. Nevertheless, the current discussion will be carried out in a reduced version of SPL, which has a one to one correspondence to PADD and is general enough to express any practical program. The basic statements are \texttt{skip, nil, stop}, the assignment $u := e$, send $\alpha \leftarrow e$, and receive $\alpha \Rightarrow u$. The two latter will be referred to as \texttt{communications}. The work is limited to synchronous channels $\alpha$, which will be referred to as \texttt{channels}. In them both the sender and the receiver wait for each other before exchanging a value and continuing execution. No intermediate message buffering is involved.

The cooperation or \texttt{parallelism} statement is $n$-ary: $[S_1; \cdots; S_n]$. Its substatements $S_j$ are its \texttt{top parallel statements}. The cooperation is their \texttt{least common ancestor} (LCA). It will be assumed throughout that the $S_j$’s are \texttt{disjoint}, in the sense that they only share read variables, and that they communicate values through synchronous channels only. The \texttt{concatenation statement} is also $n$-ary: $[S_1; \cdots; S_n]$. The iterations are $[\texttt{while } c \texttt{ do } S]$, where $c$ is a Boolean expression, and $[\texttt{loop forever } \texttt{do } S]$. The notion of LCA statement applies to concatenation composition as in the cooperation statement. Two statements are ordered in the \texttt{concatenation ordering} if their LCA is a concatenation statement. This corresponds to the execution order. The \texttt{regular} and the \texttt{communications selection statements} are non-deterministic and have, respectively, the forms $[b_1; S_1; \cdots; or b_n; S_n]$ and $[b_1; c_1; S_1; \cdots; or b_n; c_n; S_n]$, where the $b_i$’s are Boolean expressions referred to as \texttt{Boolean guards}, and the $c_i$’s are communications referred to as \texttt{communications guards}. The $S_i$’s are the \texttt{proper alternatives}.

Example Illustrating the notation usage, the following procedures model the transmission of one message of a stop and wait communications protocol

$$
\begin{align*}
(a) & := \texttt{SimpleStopWait} (b) :: \\
& [\texttt{local } a \leftarrow \texttt{boolean} \\
& b \Rightarrow m; \quad a \leftarrow F; \\
& \texttt{while } \neg a \Rightarrow \texttt{do } [(a, a) \leftarrow \texttt{Step} (m)]]
\end{align*}
$$

where, a step of the protocol is modeled as

$$
\begin{align*}
(a, a) & := \texttt{Step} (m) :: \\
& [\texttt{local } \delta, \eta; \texttt{channel of message} \\
& \texttt{local } \varepsilon; \texttt{channel of nil} \\
& \texttt{Emitter} :: [\eta \Leftarrow m; \delta \Rightarrow \texttt{ack}] \\
& || \\
& \texttt{DataChannel} := [\texttt{local } d; \texttt{message} \\
& \eta \Rightarrow d; \quad [\texttt{true}, \delta \Leftarrow d; \texttt{nil}] \\
& \texttt{or} \quad [\texttt{true}, \varepsilon \Leftarrow \texttt{nil}] \\
& || \\
& \texttt{Receiver} :: [\texttt{true}, \delta \Rightarrow r; \alpha \Leftarrow r; \gamma \Rightarrow T] \\
& \texttt{or} \quad [\texttt{true}, \varepsilon \Rightarrow ;\gamma \Rightarrow F]
\end{align*}
$$
Some laws Some intuitive congruences, used here as auxiliary laws, have been proved sound in [5], many of them do not hold when strong fairness is assumed. They are needed to transform a statement to a form where a proper communication elimination law can be applied. Some of them are the congruences nil; S ≈ S, S||nil || S, S; skip ≈ S, S||skip || S, and the iteration-fold-unfold [loop forever do S] ≈ [S; loop forever do S]. In addition, both sequential and parallel composition are associative. The latter is also commutative. The two selection statements and parallelism are permutative, so that statements with different orders of their alternatives or top parallel statements are congruent. Only the regular selection has laws of the form

\[ b_1, S_1 \text{ or } b_2, (b_21, S_21 \text{ or } b_22, S_22) \]

≈

\[ b_1, S_1 \text{ or } b_2 \land b_21, S_21 \text{ or } b_2 \land b_22, S_22 \]

Sequential composition distributes from the right over selection composition; as in

\[ (S_1 \text{ or } S_2); S_3 \approx (S_1; S_2) \text{ or } (S_2; S_3) \]

Equivalence Proper communication elimination laws require an equivalence weaker than congruence. Statements S1 and S2 are interface equivalent with respect to interface set O, written S1 =_O S2, when any interface behavior of any of them is equivalent to an interface behavior of the other. Two behaviors are equivalent when they share the same interface set, and for all their pairs of shared variables their two components, one in each behaviour, correspond to the same list of values, whose repeated values are deleted with the exception of the last one. The relative order of value changes among different variable components is thus lost. Interface equivalence will be referred to in the sequel as equivalence.

Basic notions for communication elimination A statement S is said to be of bounded communication (BC) if: (a) all its parallel substatements are disjoint, and (b) any internal communication is outside iteration bodies. Consequently, execution of a BC statement (BCS) generates a finite number of internal communication events. The communication front of S, written ComFront(1,S), is the subset of minimal elements of the set of communication statements in its concatenation ordering. Two internal communications of S, l and r, are said to form a matching communication pair, p, if they are parallel, one is an output and the other an input over the same channel. The set of competing pairs of S, written CompPairs(I,S), is, by definition, the set of matching pairs p : (l, r) which can be formed with the communications in ComFront(1,S). Two matching pairs are disjoint if they share no communication statement.

Example For the Step procedure of the above illustration, I = {δ, η, ε, γ} and at this initial stage ComFront(1,Step) = {η ⊆ m, η ⇒ d, δ ⇒ r, ε ⇒} and ComPairs(1,Step) = {η ⊆ m, η ⇒ d}).

Proper elimination laws The simplest law for the elimination of a single matching communication pair corresponds to the congruence

\[ \alpha \leftarrow \epsilon \parallel \alpha \Rightarrow u \approx \{ u := \epsilon \} \]

which is identified with \[ G_0' || G_0'' \approx G_0 \]. The recursive law for the elimination of all the internal communications of a selection-free BC statement S is defined for an arbitrary k ≥ 0 as

\[
\begin{align*}
\text{CompPairs}(\delta, S) & = \{ \eta : (\eta \leftarrow m, \eta \Rightarrow d, \delta \Rightarrow r, \epsilon \Rightarrow) \} \\
\text{CompPairs}(1, S) & = \{ \{ \eta \leftarrow m, \eta \Rightarrow d \} \} \\
\text{ComFront}(1, S) & = \{ \{ \eta \leftarrow m, \eta \Rightarrow d \} \} \\
\end{align*}
\]

\[
\begin{align*}
\text{CompPairs}(\delta, S) & = \{ \eta : (\eta \leftarrow m, \eta \Rightarrow d, \delta \Rightarrow r, \epsilon \Rightarrow) \} \\
\text{CompPairs}(1, S) & = \{ \{ \eta \leftarrow m, \eta \Rightarrow d \} \} \\
\text{ComFront}(1, S) & = \{ \{ \eta \leftarrow m, \eta \Rightarrow d \} \} \\
\end{align*}
\]

\[
\begin{align*}
\alpha \leftarrow \epsilon \parallel \alpha \Rightarrow u & \approx \{ u := \epsilon \} \\
\end{align*}
\]

Greek letters denote channels. The message to be sent is input via channel β and passed to Step in variable m of global memory within Emitter. The message is output by Receiver via α. A transmission error is simulated, nondeterministically, by communicating through internal channel ε, as an alternative of a communications selection. An erroneous reception causes a false ack as a result. Natural interface sets for Step and SimpleStop&Wait would be O STEP : {α, ack, m} and O SimpleStop&Wait : {α, β}.

Example For the Step procedure of the above illustration, I = {δ, η, ε, γ} and at this initial stage ComFront(1,Step) = {η ⊆ m, η ⇒ d, δ ⇒ r, ε ⇒} and CompPairs(1,Step) = {η ⊆ m, η ⇒ d}).

Proper elimination laws The simplest law for the elimination of a single matching communication pair corresponds to the congruence

\[ \alpha \leftarrow \epsilon \parallel \alpha \Rightarrow u \approx \{ u := \epsilon \} \]

which is identified with \[ G_0' || G_0'' \approx G_0 \]. The recursive law for the elimination of all the internal communications of a selection-free BC statement S is defined for an arbitrary k ≥ 0 as

\[
\begin{align*}
\text{CompPairs}(\delta, S) & = \{ \eta : (\eta \leftarrow m, \eta \Rightarrow d, \delta \Rightarrow r, \epsilon \Rightarrow) \} \\
\text{CompPairs}(1, S) & = \{ \{ \eta \leftarrow m, \eta \Rightarrow d \} \} \\
\text{ComFront}(1, S) & = \{ \{ \eta \leftarrow m, \eta \Rightarrow d \} \} \\
\end{align*}
\]

\[
\begin{align*}
\alpha \leftarrow \epsilon \parallel \alpha \Rightarrow u & \approx \{ u := \epsilon \} \\
\end{align*}
\]

\[
\begin{align*}
\text{CompPairs}(\delta, S) & = \{ \eta : (\eta \leftarrow m, \eta \Rightarrow d, \delta \Rightarrow r, \epsilon \Rightarrow) \} \\
\text{CompPairs}(1, S) & = \{ \{ \eta \leftarrow m, \eta \Rightarrow d \} \} \\
\text{ComFront}(1, S) & = \{ \{ \eta \leftarrow m, \eta \Rightarrow d \} \} \\
\end{align*}
\]

\[
\begin{align*}
\alpha \leftarrow \epsilon \parallel \alpha \Rightarrow u & \approx \{ u := \epsilon \} \\
\end{align*}
\]
where the $H$ statements have no inner communication. The set $O$ can not contain channel variables for channels in $I$. When this equivalence is identified with \([ G^k_{k+1} \parallel G^k_{k+1} ] = 0\) $G_{k+1}$, a recursive definition of $G_k$, $G_k$, and $G_k$ is obtained. For a given value of $k = k_0$, the corresponding law would be constructed recursively, applying the same equivalence to the inner $G_k$, which stands for \([ G^k_{k} \parallel G^k_{k} ]\), for $k = k_0, k_0 - 1, \cdots, 1, 0$. Finally, the last inner parallelism \([ G^0_{0} \parallel G^0_{0} ]\) would be replaced by the corresponding right hand side $G_0$ of the basic congruence given earlier, and the law for $k = k_0$ would thus be obtained. There is a law for any finite integer $k = 0, 1, \cdots$ which may be applied as a reduction from left to right in order to eliminate a single communication pair.

Observe that some substatements, like $T^k_{k}$ and $P^k_{k}$, are parallel in one side but not in the other. This disordering may introduce deadlock in some very specific situation. Nevertheless, most situations correspond to communication closed layer systems, for which deadlock is not introduced. These systems, together with their laws, are treated in [11], with a semantics different to the one used here, but the laws also hold here.

3 Interactive prover overview

Being developed for mechanizing and partially automating DPS proofs, it works from an input program or statement which is transformed guaranteeing that only laws and transformation procedures are applied. A transformation procedure (TP) guarantees that the output or resulting program can be obtained from its input program by a sequence of law applications, as reductions.

Having in mind a final goal form, the user has to guide the prover, via commands, in order to obtain the goal. An interface command selects a law or a transformation procedure, from two corresponding repositories, for application. In general, a command specifies the application point within the current program. The current program, or statement, is the result of a sequence of commands applied to the input and to its transformation successors up to the present.

The user interface has a top level window showing the current program (at the right). Commands are selected from its top bar. A navigation tree window, at the left of the screen, shows the sequence of proof commands, as a proof navigation tree, for browsing proofs. Each node of the navigation tree may be unfolded in order to display information about the corresponding command. The design contemplates branching points at the navigation trees to represent proofs sharing an initial segment.

Internal notation The input program, a set of procedures, is expressed in PADD, a ternary-tree like notation. It is easily translated from the SPL syntax used in this communication, and is checked for syntactic correctness as a first step. The leaves of the tree correspond to the basic statements and procedure references. A motivation for using PADD is the existence of a distributed software development environment, RALE [9], for this notation which we have been using extensively for the development of industrial projects via modeling and simulation. Thus, RALE is evolving now into a simulation and verification environment. The leaves of the PADD tree are further expanded, as part of the initial processing, into the same ternary tree pattern, called expanded tree. For instance, arithmetic expressions and assignments are represented as trees. This facilitates the structure matching required for the application of laws. Laws are represented in the same tree form with right and left sides, which may have their corresponding applicability conditions. There is a notation to express these conditions in terms of sets of variables, channels, and other program items. It has Boolean, and set operators as well.

Laws are represented in tree-like PADD notation. The variables for matching are declared under the scope of the keyword var, as the following example of a law shows. These variables are always of type statement. The left and right hand sides of the law, which correspond to an equivalence, are located under lhs and rhs respectively.
The left hand side corresponds to a parallelism of two binary concatenations. The right hand side corresponds to a concatenation of two binary parallel compositions. Since the law can be applied from left to right or vice-versa, the applicability conditions are located under the scope of each cnd keyword, vertically underneath keywords lhs and rhs. Laws can contain several conditions, one below the other, in sequence. This means that they are linked implicitly with a Boolean AND operator, thus all of them must be fulfilled before applying the law.

The above law corresponds to the one detailed later in Section 6. The expression \( X \text{ disjoint } Y \) takes the value true if \( X \) and \( Y \) are disjoint statements, and \( X \text{ comm } Y \) takes the value true if \( X \) communicates via some synchronous channel with \( Y \).

**Structuring of transformation procedures** New TPs can be defined in PADD. They may invoke the already existing ones, like the procedure which carries out a transformation corresponding to a single law, or other basic TPs to be defined later. TPs are organized in layers. One of the basic layers contains TPs that apply a short sequence of laws. More complex TPs, to be treated later, are Comelim and DPElim, for iterative elimination of inner communications from subclasses of selection-free and non selection-free statements respectively. A static tool could check that a TP transforms the input program by applying only laws or, in addition, by invocations to already checked TPs. The prover is open to the incorporation of new TPs, but their addition requires special privileges.

## 4 The three basic layers of transformation procedures

**The ground layer** The TPs in this layer allow simple operations: the application of a single law and the manipulation of procedures; substitution of references to equivalent procedures and encapsulation of a statement into a new procedure.

**apply** The simplest TP layer has only this fundamental prover command. It applies a law, involving several steps, which are encapsulated into procedure \( (\text{failure},S')::=\text{apply}(d,l,p,S) \). This may be invoked either from the interface as a command or from within a TP. This procedure tries to apply law \( l \) at a specific point \( p \) of statement \( S \), to obtain an equivalent statement \( S' \). The application is from left to right or vice-versa depending on parameter \( d \). First, it checks whether the statement at location \( p \) of \( S \) matches side \( d \) of law \( l \). After this succeeds, the applicability conditions are checked. If these are fulfilled the transformation is carried out. If some check fails, the transformation is not done and a true value of failure is returned.

**ProcRefSubstitute** Substitutes a procedure reference statement by the reference to another interface equivalent procedure. The validity of the substitution has to be checked by a deadlock-freeness proof.

**ProcEncapsulate** Encapsulates a statement as a procedure, replacing the statements by a reference to the new procedure.

**SeqPermute** Carries out a specified permutation of the statements in a sequence. It is
only applicable when the concatenated sub-statements are disjoint.

**SeqAssociation** Carries out the association of a specified subsequence within a sequence of concatenated substatements.

**SeqFlatten** Obtains a unique concatenation from one with associations.

**ParPermute** Obtains a specified permutation of the top substatements of a cooperation. Obtainable from associative and commutative laws.

**ParAssociation** Carries out the association of a specified subset of the top substatements of a parallel composition. **BinParAssociation** is a frequently used special case, where only two substatements have to be associated.

**ParFlatten** Obtains a unique cooperation from one with associations.

**SelPermute** Obtains a given permutation of the alternatives of a selection.

**SelFlatten** When applicable, obtains a unique selection from one with associations.

**SelSeqRgtAsso** Applies the selection right association law, concatenating to the alternatives of a given selection the identified state-ment following the selection in sequence.

**IfSimplify** Simplifies an if statement in the obvious way when its condition static evaluation gives true or false.

**ParToSeq** Transforms a cooperation of disjoint sub-statements without communications, into a concatenation in a specified substate-ment order.

**WhileUnfold** It applies to a while statement whose iteration condition static evaluation gives the value true. Obtains a concatenation whose last statement is the while loop, preceded by an unfolded body.

**SimplExp** Simplifies a given arithmetic or Boolean expression. As a result of the simplification some if statements may be removed and a more compact sequentialization is obtained.

**IteLoopUnfold** It applies to an indefinite loop statement only. Obtains a concatenation whose last statement is the indefinite loop, preceded by a number, indicated by the user, of unfoldings of its body. Applies the iteration fold-unfold law.

**IteIfSimply** Simplifies all redundant if statements found within the given statement. Applies iteratively **IfSimply**.

**IteSimplExp** Simplifies and eliminates all arithmetic and Boolean expressions found within the given statement. Applies iteratively **SimplExp**.

**IteRedVarElim**Eliminates all redundant variables found within a given statement, invoking **RedVarElim**, covered in Section 6.

## 5 Communication elimination TPs

The two TPs which follow apply only to selection-free BCSs, all of whose internal communications are outside the scope of both selections and communication selections.

**PairElim** Shortened as **PElim**. It carries out the elimination of a communication pair applying the recursive law given in Section 2. When applicability conditions do not hold a Boolean result is returned with a false value, and its algorithm terminates unsuccessfully.

**ComElim** Attempts to eliminate all internal communications. Assuming disjointness of all possible pairs, a communication elimination algorithm proposed and studied in [2] is included in this TP. It applies iteratively procedure **PElim**. When the loop of **PElim** invocations terminates without failure, and there is still some communication left in **ComFront**((I, S)), this means that the original program has deadlock possibility. This proof construction algorithm simulates an execution where the elimination of a pair of matching communication statements in the proof corresponds to a communication event in an execution.

The following definitions specify the subclass of selection-free BCSs, which can be handled by the rest of TPs in this section. The term basic statement includes procedure references. The notions of initial and terminal basic statement are intuitive.

**Definition 1 (Concatenation chain of a statement)** A list of some of its basic sub-
statements, in consecutive ascending concatenation order, with parallelism and selection symbols. Its first element is an initial substatement or a symbol of selection or parallelism. Its last element is a terminal or an exit substatement. Each occurrence of the two symbols represents the LCA selection or parallelism statement of its immediate successor.

**Definition 2 (Single selection embedding BCS)** A BCS is single selection embedding BC when all its concatenation chains contain at most one selection symbol in between any pair of internal communications, or in between the starting point and the first internal communication.

**GenPairElim** Shortened to GPElim. The laws for the elimination under selection given in [4, 6] are encoded in this TP for the elimination of a single communication pair \( p : (c_1, c_2) \in \text{CompPairs}(I, S) \) from a single selection embedding BCS \( S \). At the end of the procedure, selection flattening is applied to combine the newly generated selection with the possibly already existing top selection. Variable elimination and other laws may be necessary for simplification as well. If this is the case, they are applied automatically. The procedure reference statement is

\[
\text{(failure, Sr)} := \text{GPElim}(c_1, c_2, \text{sel1}, \text{sel2}, S)
\]

Variables sel1 and sel2 are assumed to contain information about the specific selection embedding communications \( c_1 \) and \( c_2 \) respectively. It can be proved that the disjoint pair \( (c_1, c_2) \) has been eliminated in \( Sr \) resulting from GPElim when failure = false. Then \( Sr = O S \), for any \( O \) not containing the inner channel shared by \( c_1 \) and \( c_2 \).

This procedure needs the backward propagation \( bp(b, H) \) of Boolean expression \( b \) within \( H \), all the concatenated substatements preceding \( b \). In general, the automatic computation of \( bp \) is not possible, since it may involve automatic invariant generation. In these situations the algorithm does not apply. Nevertheless, when \( H \) involves no loops the backward propagation can be computed.

**DisjPairsElim** Shortened to DPElim. For the elimination from single selection embedding BCSs. At any point of an elimination proof, all the pairs in CompPairs(I, S) have to be disjoint, sharing no communication. The invocation to this TP is (failure, deadlock, \( Sr := \text{DisjPairsElim}(I, S) \). It uses procedure GPElim, overviewed above, and ObtainCompPair which obtains a communication pair \( (c_1, c_2) \) from the current communication front, as well as information, within sel1 and sel2, on the selections embedding the two communications. The search for a pair within this procedure is done over a new communication front in each invocation of ObtainCompPair, since the previous pair has been eliminated in the loop, and this may uncover new communications.

It can be proved that the \( Sr \) resulting from DisjPairsElim when failure=deadlock=false is such that \( Sr = O S, S \) is deadlock-free, and \( Sr \) has no inner communication statements. When deadlock=true, \( S \) is not deadlock-free.

### 6 Proper sequentialization and variable elimination TPs

Distributed program sequentialization (DPS) is a proof with three types of step: (a) elimination of internal communication pairs, (b) parallelism to concatenation formal transformation, and (c) redundant variable elimination. Usually, the two latter steps are interlaced. DPS for commonly encountered non-BCSs has been treated in [8]. The TPs in this section are intended for steps (b) and (c) of DPS proofs. They are applied after the communication elimination proof, which results in disjoint parallel substatements.

Redundant variables appear in the program resulting from communication elimination. Usually, they correspond to buffers which are no longer needed. They appear within assignment statements which are also redundant. The reduction of the upper bound of the number of states of the semantic model is directly linked to the elimination of redundant variables. More specifically, the elimination of a variable of \( n \) bits reduces the up-
per bound on the number of states by a factor of $2^{-m}$. The sequentialization proof of a pipeline processor model reported in [3, 7, 8] provides an example where the reduction factor is $2^{-671}$.

These are TPs making use of laws of the following types: Parallelism to concatenation transformation and redundant variable elimination. The former is carried out applying permutation laws for transforming the parallel compositions of disjoint processes to equivalent sequential forms. For instance, since $S_1$ and $S_2$ are disjoint $S_1 \parallel S_2 = O S_1; S_2$. Another class of these laws was treated in [11]. As an example, let the statement pairs $(A_1, B_1)$ and $(A_2, B_1)$ be non-communicating, and $[B_1; A_1]$ be disjoint with $[B_2; A_2]$. Then
\[ [B_1; A_1][[B_2; A_2]] = O \ [B_1][B_2]; [A_1][A_2] \]
and either both sides are deadlock-free or none of them is. The expression of this law within the prover has been illustrated in Section 3.

Redundant variable elimination is done with one of the various generalizations of the law given below. It can be generalized to multiple variables.

Let $e$ be an expression having no reference to variable $v$, such that $v \notin O$. Let $S_1(v)$ have only read references to $v$. $S_2$ have no read reference to $v$, and be either the last statement within the scope of $v$ or located just before a new assignment to $v$, with respect to the concatenation order of the program. Then
\[ v \equiv e; S_1(v); S_2 \rightarrow O \ [S_1(e); S_2] \]

VarElim Eliminates a given variable and assignment by the above law, when applicable. It has generalizations for lists of variables appearing within multiple assignment statements.

GenSeqs Taking into account all the applicable restrictions, generates all the possible sequentializations of a statement having parallel and sequential compositions, and whose parallel processes are disjoint.

SeqVarElims Generates all possible sequentializations of a given substatement. Attempting to eliminate a set of variables indicated by the user. Invokes GenSeqs, IterRedVarElim, IterIfSimply and IterSimplExp.

7 Illustration

Continuing with the above communications protocol example, the interface set of Step would be $O : \{\alpha, ack, m\}$. Its elements being the input and output variables $m$ and $ack$, and the output channel variable $\alpha$. The equivalent program resulting from a DPS proof would be the following
\[(\alpha, \text{ack}) ::= \text{SimpleStep} \ (m) :: \]
\[ \left[ \left[ \text{true, } \alpha = m; \text{ack} = T \right] \right] \]
\[ \text{or} \]
\[ \left[ \text{true, } \text{ack} = F \right] \]
Thus Step $\rightarrow (\alpha, \text{ack}, m) \text{ SimpleStep}$. It captures the essential behavior as seen from its interface. The proof is generated by DPElim TP. Some intermediate forms of the transformed body of Step are given as an illustration. The first records the state just ready to eliminate the pair over $\delta$, within the first alternative of the section.
\[ \left[ \text{true, } \alpha = m; \text{ack} = T \right] \]
\[ \text{or} \]
\[ \left[ \text{true, } \text{ack} = F \right] \]
Any selection is assumed to have an empty alternative $\emptyset$, for uniformity of processing. The following is an intermediate stage when the pair of channel $\delta$ has already been eliminated and the pairs of $\epsilon$ and $\gamma$ have still to be eliminated.
\[ \left[ \text{true, } r = m; \alpha = r; \text{ack} = T \right] \]
\[ \text{or} \]
\[ \left[ \text{true, } \epsilon = \gamma = F \right] \]
Observe that redundant variable $r$ and its assignment remain to be eliminated applying RedVarElim of last section, which is carried out automatically by DPElim.

After substitution in SimpleStep\&Wait of Section 2, the following results:
\[ \beta \Rightarrow m; \text{ack} = F; \]
\[ \text{while } \neg \text{ack do} \]
\[ \left[ \left[ \text{true, } \alpha = m; \text{ack} = T \right] \right] \]
\[ \text{or} \]
\[ \left[ \text{true, } \text{ack} = F \right] \]
After loop unfolding via $\text{WhileUnfold}$ of Section 4, one obtains:

$$
\begin{align*}
\beta & \Rightarrow m; \quad \text{ack} := F; \\
\text{or} & \begin{cases}
\text{true, [ack := T]}
\end{cases}; \\
\text{while} & -\text{ack do} \\
\text{or} & \begin{cases}
\text{true, [ack := F]}
\end{cases}
\end{align*}
$$

After sequence right and left association over selection $\text{SelSeqRgtAsso}$, and elimination of while false do in the first alternative:

$$
\begin{align*}
\beta & \Rightarrow m; \\
\text{or} & \begin{cases}
\text{true, [ack := F; \alpha := m; ack := T]}
\end{cases}; \\
\text{while} & -\text{ack do} \\
\text{or} & \begin{cases}
\text{true, [ack := F;\alpha := m; ack := T]}
\end{cases}
\end{align*}
$$

Now, the same while preceded by ack:= F appears again in the second alternative of the or statement, as in (1). This recursive form is equivalent to

$$
\begin{align*}
\beta & \Rightarrow m; \\
\text{loop} & 1.. [\text{ack} := F]; \quad \alpha \Leftarrow m; \quad \text{ack} := T
\end{align*}
$$

where loop 1..[] means at least one repetition of the iteration body [1].

Finally, since ack is not in the interface set of $\text{SimpleStop\&Wait}$, the above form is equivalent to $\beta \Rightarrow m; \quad \alpha \Leftarrow m; \quad \text{ack} := T$, which shows very simply the essential function of the initial protocol.

8 Conclusions and further work

The design of an interactive prover to help in the construction of sequentialization proofs has been outlined. It includes the following layers of transformation procedures (TPs), also activated as interface commands, of increasing complexities: apply for the application of laws as reductions and the substitution of procedure references, TPs carrying out simple transformations such as parallelism permutation, a third layer of iterative application of TPs and more complex TPs such as $	ext{Come-lim}$ for the automatic elimination of communication pairs. Automatic elimination under selections is not yet implemented. Some of the proofs carried out with the current version of the prover have resulted in a reduction of $2^{-671}$ of the upper bound on the number of states of the state transition system associated with the initial program. The prover has an expandable design, admitting more TPs. Further work on the automatic construction of more parts of DPS proofs, such as TPs combining parallelism to concatenation transformations with redundant variable elimination should be carried out. The extension of the class of non-selection-free BCSs which is currently handled, and many other pending topics, should be undertaken also.

Acknowledgements

We thank the encouragement received during the last years from Zohar Manna, Bernd Finkbeiner, and Tomas Uribe.

References


