

An Abstract Analysis Framework for Synchronous Concurrent Languages based on source-to-source Transformation

M. Alpuente^{a,5,1} M.M. Gallardo^{b,5,2} E. Pimentel^{b,5,3}
A. Villanueva^{a,5,4}

^a Dept. SIC
Technical University of Valencia
Valencia, Spain

^b Dept. LCC
University of Málaga
Málaga, Spain

Abstract

A pretty wide range of concurrent programming languages have been developed over the years. Coming from different programming traditions, concurrent languages differ in many respects, though all share the common aspect to expose parallelism to programmers. In order to provide language level support to programming with more than one process, a few basic concurrency primitives are often combined to provide the main language constructs, sometimes making different assumptions. In this paper, we analyze the most common primitives and related semantics for the class of synchronous concurrent programming languages, i.e., languages with a *global* mechanism of processes synchronization. Then, we present a generic framework for approximating the semantics of the main constructs which applies to both, declarative as well as imperative concurrent programming languages. We determine the conditions which ensure the correctness of the approximation, so that the resulting abstract semantics safely supports program analysis and verification.

1 Introduction

Concurrent programming languages are programming languages that use language constructs for concurrency. Two main approaches exist to concurrency: the synchronous and the asynchronous models. Asynchronous models are based on the assumption that system components running in parallel proceed at different rates.

¹ Email: alpuente@dsic.upv.es

² Email: gallardo@lcc.uma.es

³ Email: ernesto@lcc.uma.es

⁴ Email: villanue@dsic.upv.es

⁵ This work has been partially supported by the EU (FEDER) and the Spanish MEC, under grants TIN2004-7943-C04-01, TIN2007-68093-C02-02 and TIN2007-67134

Synchronous models differ from asynchronous ones since they assume that all system components share the same clock and are perfectly synchronized. Synchronous languages [2,7] such as Esterel [3], StateCharts [10], and Argos [11] are imperative, whereas the relational languages Signal [9] and `tccp` [14], and the functional language Lustre [5] are declarative. Synchronous languages typically offer primitives to deal with negative information, namely to instantaneously test absence of signals. These languages are also based on the strong synchronous hypothesis, meaning that each reaction of the reactive system is assumed to be instantaneous, and then takes no time. This provides a deterministic semantics of concurrency as well as a formal straightforward interpretation of temporal statements. While the synchronous hypothesis is considered unrealistic when communication time cannot be neglected, it makes sense when programming reactive systems, i.e., systems which continuously interact with the environment: operating systems, real-time control software, client/server applications, and web services typically fall in this category. In this context, to “take no time” is understood in two ways: the environment remains invariant during the reaction, and all sub-processes react instantaneously at the same time.

There are numerous advantages to the synchronous approach. The main one is that the temporal semantics is simplified, thanks to the so-called logical time abstraction: all the parallel processes evolve simultaneously, along a common discrete time scale. This leads to clear temporal constructs and easier time reasoning. On the contrary, asynchronous programs are generally not temporally predictable. Another key advantage is the reduction of state-space explosion, thanks to the discrete logical time abstraction: the system evolves in a sequence of discrete steps, and nothing occurs between two successive steps. A first consequence is that program debugging, testing, and validating is made easier. In particular, formal verification of synchronous programs is possible with techniques like model checking.

Similarly to sequential languages, abstract interpretation is commonly applied to analyzing properties of concurrent languages statically. Since abstraction usually involves adding non-determinism, applying abstract interpretation to synchronous languages raises specific problems which are related to synchronization. The suspension behavior of concurrent programs has been studied in different programming languages and paradigms [6,15]. When we approximate the semantics of concurrent languages by abstract interpretation, it can happen that the original program suspends whereas by using the abstract semantics it does not; hence synchronization in the abstract model might be damaged, as shown in Example 3.2 below. Depending on the concurrent model that we consider, this lack of precision of the approximation does not generally imply incorrectness of the abstract semantics [15]. However, it is always the case when dealing with strongly synchronized processes and partial information (see [1]). This is why both, correctness and accuracy must be considered when defining the abstract semantics of a concurrent language. Trying to achieve a correct abstract semantics has also a payoff related to the precision of the abstract model, since excessively abstracting the original semantics may lead to generating very imprecise abstract models containing execution paths which do not correspond

to any real behavior.

The inspiration for this work comes from [1,8]. With regard to [8], we generalize the abstract semantics of PROMELA to the case of a generic language that we define by combining a set of basic primitives; however, in this work we have only considered the synchronous approach to concurrency, whereas the synchronous as well as the asynchronous nature of PROMELA are considered in [8]. Our abstract framework can be parametrized for different languages, and allows us to analyze how the basic primitives must be abstracted to achieve precision while ensuring 1) the correctness of the approximation, and 2) the possibility to define the abstraction as a source-to-source transformation.

In this paper, we analyze the most common primitives and related semantics for the class of synchronous concurrent languages. Then, we present a generic framework for approximating the semantics of the main constructs which applies to both, declarative and imperative concurrent programming languages. We determine the conditions which ensure the correctness of the approximation so that the resulting abstract semantics safely supports accurate program analysis and verification.

2 The synchronous approach

Semantics for concurrent languages are usually given by means of Plotkin's structural operational semantics (SOS) [12], where processes behavior is described by labeled transitions systems, namely graphs with nodes (states) representing process configurations and labeled arcs representing atomic computation steps. A tuple consisting of two nodes and an arc connecting them is called a transition. A trace is a sequence of transition steps.

In this section, we define some basic notions appearing in the semantics of all concurrent synchronous languages, disregarding the differences linked to particular syntax or combinations of constructs in a particular language, and concretely to the declarative or imperative nature of the language.

2.1 System states

Let us denote with *State* the set of system states. We are general when speaking about states since we intend to consider both, states as valuations of variables (imperative-style), and states as conjunctions of constraints (constraint-based style). Thus, each state $s \in \text{State}$ is either the set of program variables bound to their actual value (when dealing with imperative languages), or a set of constraints (in a constraint-based language). Anyway, we assume that, given a set of variables V , state $\exists_V s$ represents the state s with the information about variables in V removed, called *hiding-variables operation*. In addition, we suppose that *State* contains an element ϵ representing the *empty* state, that is, the state that does not provide any information. Clearly, if V is the set of variables appearing in state s , $\exists_V s = \epsilon$.

We assume that states may be composed by means of $\oplus : \text{State} \times \text{State} \rightarrow \text{State}$. Operator \oplus models the change of state. We assume that \oplus is idempotent,

that is, $\forall s \in State, s \oplus s = s$. The definition of \oplus depends on the execution model considered. For instance, for the constraint-based, declarative synchronous model shown in [4], where maximal parallelism is implemented, composition of states (stores) $s \oplus s'$ is usually defined as $s \sqcup s'$ where \sqcup is the lub operator in the underlying lattice of constraints. Stores increase monotonically during program execution, that is, the information registered is never canceled. However, other declarative languages in the cc paradigm such as the model in [13] differ from this behavior, being the evolution of the store non-monotonic.

In the imperative framework, the \oplus operator is usually defined as a destructive update, i.e., the value of a given variable at one time instant is substituted by a new value at the subsequent time instant. This implies that the order in which updates are executed does matter, since different choices for the ordering might produce different results.

Since in most paradigms composition of states is an idempotent operation, we assume it so in this paper. In addition, we assume that the empty state ϵ works as the neutral element for \oplus , that is, $\forall s \in State, s \oplus \epsilon = \epsilon \oplus s = s$. Finally, in order to properly deal with the hiding-variables operation, we assume that for each set of variables V , and for each states $s, s' \in State, s \oplus \exists_V s = \exists_V s \oplus s = s$, and $\exists_V (s \oplus s') \equiv \exists_V s \oplus \exists_V s'$.

2.2 Basic actions

Basic actions are those atomically executed by the program. In most languages, the set of basic actions mainly boils down to a test (also called *ask*) action and an update (also called *tell*) action. Let us denote *Ask* and *Tell* the sets of test actions and update actions of a language, respectively. Intuitively, the elements of *Tell* represent the language actions that may modify the program states, whereas the actions in *Ask* represent the checking of guards used to implement control instructions and synchronization. We assume that the language supports the *empty test true*: $true \in Ask$.

Finally, we assume that semantic functions $effect : Tell \times State \rightarrow State$ and $test : Ask \times State \rightarrow \{false, true\}$ define the behavior of the basic actions of the language. Intuitively, $effect$ represents the effect of a specific update action when executed on a particular state, whereas $test$ checks if the condition specified by a basic test action can be derived from a specific state (i.e., the condition is fulfilled). Note that $effect$ depends on the specific implementation of the operator \oplus , which is the operator that makes states evolve.

In order to correctly model control flow instructions, we assume that each test $a \in Ask$ is complemented by another action $\neg a \in Ask$ which represents that a does not hold, i.e., the test fails⁶. Although negation is not uniformly treated within the different programming models, under the considered assumptions, the following

⁶ The $\neg a$ action must not be confused with the action which semantically can be interpreted as the opposite of a . For example, assume a constraint-based system where variable X has the value $X \in [0, 10]$, i.e., the value of variable X can be any of the values in the considered interval. The *opposite* test of $X > 5$ is $X \leq 5$, and none of them follows. However, the fact that $X > 5$ doesn't hold allows us to conclude that $\neg X > 5$ does hold.

holds

$$\text{test}(a, S) \Leftrightarrow \neg \text{test}(\neg a, S)$$

Thus, the 3-uple $\langle \text{State}, \text{test}, \text{effect} \rangle$ (called *semantic context*) defines the semantics of the basic actions of the language.

In the synchronous approach, the execution of basic actions take time, and this is the way time passes. Figure 1 shows the language syntax we choose for the description.

Action	::=	error	
		end	
		$a!$	where $a \in \text{Tell}$ – Update
		Action ; Action	– Sequence
		Action Action	– Concurrency
		$\sum_{i \in I} (a_i? \rightarrow \text{Action}_i)$	– Global Choice
			where $\forall i \in I. a_i \in \text{Ask}$
		if $a?$ then Action else Action	– Conditional
			where $a \in \text{Ask}$
		$\exists_{\bar{x}} \text{Action}$	– Local declaration
		$\text{proc}(\bar{v})$	– Procedure call
		$\sum_{i \in I} (a_i? \rightarrow^* \text{Action}_i)$	– Instantaneous Choice
			where $\forall i \in I. a_i \in \text{Ask}$
Decl	::=	$\text{proc}(\bar{x}) \{ \text{Action} \}$	
		Decl Decl	
Program	::=	Decl[Action]	

Fig. 1. Syntax of the synchronous language

Intuitively, a program is a set of declarations together with an initial action Action. A declaration can be seen as a procedure definition $\text{proc}(\bar{x})$, where parameters can be passed to the body, Action. In the body of a declaration, we can specify the following actions: $a!$ updates the state, $A;B$ executes the action A and, once finished, continues by executing the action B , $A||B$ concurrently executes actions A and B . The *Global Choice* checks whether the current state satisfies any of the guards, and chooses one among the associated actions to be executed in the subsequent time instant. The *Instantaneous Choice* differs from the *Global Choice* since the associated action starts at the same time instant the guard is checked. The *Local Declaration* defines local variables to actions. The *Conditional* executes the

action associated to the *then*-branch provided the condition is satisfied; otherwise executes the action in the *else*-branch. Finally, procedure call $\text{proc}(\bar{v})$ passes values in \bar{v} to the action in its body. Action **error** models an erroneous execution, whereas **end** represents when a thread normally finishes.

As will be apparent later, the following notion of labeling is instrumental to reason about processes synchronization. Actions in program $P = D[A_0]$ can be unequivocally labeled with elements of a set \mathcal{L} of labels. Assume that function $n\text{Lab} : \rightarrow \mathcal{L}$ returns a new fresh label at each invocation. Then, given $D = \{\text{proc}_i(\bar{x})\{A_i\} | 1 \leq i \leq n\}$, function lab defined below constructs the set of labeled declarations $\text{lab}(D) = \{\text{proc}_i(\bar{x})\{\text{lab}(n\text{Lab}(), A_i)\} | 1 \leq i \leq n\}$.

$$\begin{aligned}
\text{lab}(l, \mathbf{error}) &= (l_{\mathbf{error}})\mathbf{error} \\
\text{lab}(l, \mathbf{end}) &= (l_{\mathbf{end}})\mathbf{end} \\
\text{lab}(l, a!) &= (l)a! \\
\text{lab}(l, A; B) &= \text{lab}(l, A); \text{lab}(n\text{Lab}(), B) \\
\text{lab}(l, A || B) &= \text{lab}(l, A) || \text{lab}(n\text{Lab}(), B) \\
\text{lab}(l, \sum_{i \in I} (a_i? \rightarrow A_i)) &= (l) \sum_{i \in I} (a_i? \rightarrow \text{lab}(n\text{Lab}(), A_i)) \\
\text{lab}(l, \text{if } a? \text{ then } A \text{ else } B) &= (l) \text{if } a? \text{ then } \text{lab}(n\text{Lab}(), A) \text{ else } \text{lab}(n\text{Lab}(), B) \\
\text{lab}(l, \exists \bar{x} A) &= \exists \bar{x} \text{lab}(l, A) \\
\text{lab}(l, \text{proc}(\bar{v})) &= (l)\text{proc}(\bar{v}) \\
\text{lab}(l, \sum_{i \in I} (a_i? \rightarrow^* A_i)) &= (l) \sum_{i \in I} (a_i? \rightarrow^* \text{lab}(n\text{Lab}(), A_i))
\end{aligned}$$

2.3 Operational Semantics

Figure 2 defines the operational semantics of the language given by a transition relation between configurations. Each configuration records both the current state and the set of instructions to be executed. The transition relation is parametric w.r.t. $\mathcal{S} = \langle \text{State}, \text{test}, \text{effect} \rangle$. Different states, and/or different definitions for *test* and *effect* determine different transition relations $\rightarrow_{\mathcal{S}}$. In the figure, $a_i (i = 1, \dots, n)$, $a, b \in \text{Ask}$ and $c \in \text{Tell}$. We assume that the parallel and choice actions are commutative, i.e., the order in which the different actions are represented does not affect the final result. Action $A[\bar{v}/\bar{x}]$ represents the action A where the formal parameters \bar{x} have been substituted by the actual parameters \bar{v} .

Definition 2.1 A pair $\Lambda \equiv \langle A, s \rangle$ denotes a program configuration where A is the program to be executed and s the current state.

In the semantics, we assume that each action is *labeled* with the label that points to the program instruction to be subsequently executed. In rule **R-S1**, operator **end** is labeled with the special tag $l_{\mathbf{end}} \in \mathcal{L}$. Similarly, when an inconsistent state is reached, a transition to a state with action **error** is performed. Given a configuration $\langle A, s \rangle$, A being the parallel composition of operators $A_1 || \dots || A_n$, we denote with $p\text{Lab}(A)$ the ordered sequence $l_1 || \dots || l_n$ of labels of operators in A . In rule **R-S6**,

R-S1	$\langle c!, s \rangle \rightarrow_{\mathcal{S}} \langle \text{end}, \text{effect}(c, s) \rangle$	
R-S2a	$\frac{\langle A, s \rangle \rightarrow_{\mathcal{S}} \langle A', s' \rangle}{\langle A; B, s \rangle \rightarrow_{\mathcal{S}} \langle A'; B, s' \rangle}$	if $A' \neq \text{end}$
R-S2b	$\frac{\langle A, s \rangle \rightarrow_{\mathcal{S}} \langle \text{end}, s' \rangle}{\langle A; B, s \rangle \rightarrow_{\mathcal{S}} \langle B, s' \rangle}$	
R-S3a	$\frac{\langle A, s \rangle \rightarrow_{\mathcal{S}} \langle A', s' \rangle, \langle B, s \rangle \rightarrow_{\mathcal{S}} \langle B', s'' \rangle}{\langle A B, s \rangle \rightarrow_{\mathcal{S}} \langle A' B', s' \oplus s'' \rangle}$	
R-S3b	$\frac{\langle A, s \rangle \rightarrow_{\mathcal{S}} \langle A', s' \rangle, \langle B, s \rangle \not\rightarrow_{\mathcal{S}}}{\langle A B, s \rangle \rightarrow_{\mathcal{S}} \langle A' B, s' \oplus s \rangle}$	
R-S4	$\langle \sum_{i=1}^n (a_i? \rightarrow A_i), s \rangle \rightarrow_{\mathcal{S}} \langle A_j, s \rangle$	if $\text{test}(a_j, s)$
R-S5a	$\frac{\langle A, s \rangle \rightarrow_{\mathcal{S}} \langle A', s' \rangle}{\langle \text{if } a? \text{ then } A \text{ else } B, s \rangle \rightarrow_{\mathcal{S}} \langle A', s' \rangle}$	if $\text{test}(a, s)$
R-S5b	$\frac{\langle B, s \rangle \rightarrow_{\mathcal{S}} \langle B', s' \rangle}{\langle \text{if } a? \text{ then } A \text{ else } B, s \rangle \rightarrow_{\mathcal{S}} \langle B', s' \rangle}$	if $\neg \text{test}(a, s)$
R-S6	$\frac{\langle A, \exists_V s \oplus s_L \rangle \rightarrow_{\mathcal{S}} \langle A', s' \rangle}{\langle \exists_V^{s_L} A, s \rangle \rightarrow_{\mathcal{S}} \langle \exists_V^{s'} A', s \oplus \exists_V s' \rangle}$	
R-S7	$\langle p(\bar{v}), s \rangle \rightarrow_{\mathcal{S}} \langle A[\bar{v}/\bar{x}], s \rangle$	if $\text{proc}(\bar{x})\{A\}$
<hr/>		
R-S8a	$\frac{\langle A_j, s \rangle \rightarrow_{\mathcal{S}} \langle A'_j, s' \rangle}{\langle \sum_{i \in I} (a_i? \rightarrow^* A_i), s \rangle \rightarrow_{\mathcal{S}} \langle A'_j, s' \rangle}$	if $\text{test}(a_j, s)$
R-S8b	$\frac{\langle A_j, s \rangle \not\rightarrow_{\mathcal{S}}}{\langle \sum_{i \in I} (a_i? \rightarrow^* A_i), s \rangle \rightarrow_{\mathcal{S}} \langle A_j, s \rangle}$	if $\text{test}(a_j, s)$

Fig. 2. Operational semantics of a synchronous language

s_L is the local store of A ; initially s_L is the state $\epsilon \in \text{State}$, i.e., the neutral element of operator \oplus , and it accumulates the information known by A .

Let us assume that semantic contexts $\mathcal{S} = \langle \text{State}, \text{test}, \text{effect} \rangle$ and $\mathcal{S}' = \langle \text{State}', \text{test}', \text{effect}' \rangle$ give us two different transition relations $\rightarrow_{\mathcal{S}}$ and $\rightarrow_{\mathcal{S}'}$ for a given program $D[A_0]$, and let $\sim \subseteq \text{State} \times \text{State}'$ be a binary relation. Then we reformulate the classic notion of simulation as follows:

Definition 2.2 Transition relation $\rightarrow_{\mathcal{S}'}$ is a \sim -simulation of $\rightarrow_{\mathcal{S}}$ iff for each \mathcal{S} -transition $\langle A, s_1 \rangle \rightarrow_{\mathcal{S}} \langle B, s_2 \rangle$, if $\langle A', s'_1 \rangle$ is a configuration such that $s_1 \sim s'_1$ and

$pLab(A) = pLab(A')$ then there exists a \mathcal{S}' -transition $\langle A', s'_1 \rangle \rightarrow_{\mathcal{S}'} \langle B', s'_2 \rangle$ with $s_2 \sim s'_2$, and $pLab(B) = pLab(B')$.

Assume $\mathcal{S} = \langle State, test, effect \rangle$. Given a synchronous program P of the form $D[A_0]$, and an initial state $s_0 \in State$, the trace based semantics of P determined by the transition relation $\rightarrow_{\mathcal{S}}$, denoted by $\mathcal{T}(\mathcal{S})(D)(\langle A_0, s_0 \rangle)$, is the set of maximal traces of the form $\langle A_0, s_0 \rangle \rightarrow_{\mathcal{S}} \langle A_1, s_1 \rangle \rightarrow_{\mathcal{S}} \dots$ that can be constructed from initial configuration $\langle A_0, s_0 \rangle$.

Consider $\mathcal{S} = \langle State, test, effect \rangle$ and $\mathcal{S}' = \langle State', test', effect' \rangle$, such that $\rightarrow_{\mathcal{S}'}$ is a \sim -simulation of $\rightarrow_{\mathcal{S}}$. It is easy to prove that $\mathcal{T}(\mathcal{S}')(D)(\langle A'_0, s'_0 \rangle)$ is a correct \sim -simulation of $\mathcal{T}(\mathcal{S})(D)(\langle A_0, s_0 \rangle)$, that is, for each trace $\langle A_0, s_0 \rangle \rightarrow_{\mathcal{S}} \langle A_1, s_1 \rangle \rightarrow_{\mathcal{S}} \dots$ in $\mathcal{T}(\mathcal{S})(D)(\langle A_0, s_0 \rangle)$, there exists a trace $\langle A'_0, s'_0 \rangle \rightarrow_{\mathcal{S}'} \langle A'_1, s'_1 \rangle \rightarrow_{\mathcal{S}'} \dots$ in $\mathcal{T}(\mathcal{S}')(D)(\langle A'_0, s'_0 \rangle)$ such that $\forall i \geq 0. s_i \sim s'_i$, and $pLab(A_i) = pLab(A'_i)$.

3 Abstraction of Basic Actions

3.1 Abstracting states

As it is well known, abstract interpretation may be equivalently defined by upper closure operators (*ucos*) and Galois insertions. An *uco* $\rho : \wp(State) \rightarrow \wp(State)$ is a monotonic ($ss_1 \subseteq ss_2 \Rightarrow \rho(ss_1) \subseteq \rho(ss_2)$), idempotent ($\rho(\rho(ss)) = \rho(ss)$) and extensive ($ss \subseteq \rho(ss)$) operator. Given $s \in State$, and an *uco* ρ over $(\wp(State), \subseteq)$, $\rho(\{s\})$ is the best ρ -abstraction of s . Moreover, if $\rho(\{s\}) \subseteq ss$, then ss is a different less precise abstraction of state s .

In order to simulate abstract operator \oplus , we define $\oplus^\rho : State^\rho \times State^\rho \rightarrow State^\rho$ as $ss_1 \oplus^\rho ss_2 = \rho(\{s_1 \oplus s_2 | s_1 \in ss_1, s_2 \in ss_2\})$. Observe that this definition guarantees that $\rho(\{s_1 \oplus s_2\}) \subseteq \rho(\{s_1\}) \oplus^\rho \rho(\{s_2\})$.

Assume that $\iota : \rho(\wp(State)) \rightarrow State^\alpha$ is an isomorphism for a given abstract domain $State^\alpha$; then, $(\wp(State), State^\alpha, \iota\rho, \iota^{-1})$ is a Galois insertion, $\iota\rho$ and ι^{-1} being the abstraction and concretization functions. In the sequel, we denote $\iota\rho$ and ι^{-1} with α and γ .

Now operator \oplus^α may be defined as:

$$s_1^\alpha \oplus^\alpha s_2^\alpha = \iota(\iota^{-1}(s_1^\alpha) \oplus^\rho \iota^{-1}(s_2^\alpha))$$

3.2 Abstracting actions

Once the domain has been abstracted, we have to abstract the basic actions of the language, i.e., updates and tests. Assume that ρ is an *uco* defining an abstraction over states as explained above. We define the abstract test $test^\rho : Ask \times \wp(State) \rightarrow \{false, true\}$ and effect $effect^\rho : Tell \times \wp(State) \rightarrow \wp(State)$ functions as follows:

$$test^\rho(a, ss) = \bigvee_{s \in ss} test(a, s)$$

$$effect^\rho(b, ss) = \rho(\{effect(b, s), s \in ss\})$$

Assuming that α and γ are the abstraction and concretization functions relating $\wp(\text{State})$ and State^α as described above, the preceding definitions may be reformulated as the functions $\text{test}^\alpha : \text{Ask} \times \text{State}^\alpha \rightarrow \{\text{false}, \text{true}\}$ and $\text{effect}^\alpha : \text{Tell} \times \text{State}^\alpha \rightarrow \text{State}^\alpha$:

$$\text{test}^\alpha(a, s^\alpha) = \bigvee_{s \in \gamma(s^\alpha)} \text{test}(a, s)$$

$$\text{effect}^\alpha(b, s^\alpha) = \alpha(\{\text{effect}(b, s), s \in \gamma(s^\alpha)\})$$

Following the terminology used in Section 2.2, we have given an abstract semantic context $\mathcal{S}^\alpha = \langle \text{State}^\alpha, \text{test}^\alpha, \text{effect}^\alpha \rangle$ to the basic actions of the language, which determines the abstract transition relation $\rightarrow_{\mathcal{S}^\alpha}$. Let us define the natural binary relation between concrete and abstract states as

$$s \sim_\alpha s^\alpha \iff \rho(\{s\}) \subseteq \iota^{-1}s^\alpha$$

that is, $s \sim_\alpha s^\alpha$ iff s^α is an abstraction of s or, inversely, iff s is a concretization of s^α . In the rest of the paper, we assume that relation \sim_α is preserved under the hiding-variables operation \exists_V , that is, if $s \sim_\alpha s^\alpha$ then $\exists_V s \sim_\alpha \exists_V s^\alpha$.

Now we can easily prove the following result:

Proposition 3.1 *If $s_1 \sim_\alpha s_1^\alpha$ and $s_2 \sim_\alpha s_2^\alpha$ then $s_1 \oplus s_2 \sim_\alpha s_1^\alpha \oplus^\alpha s_2^\alpha$.*

Proof. By definition of \oplus^ρ , we have that $\rho(\{s_1 \oplus s_2\}) \subseteq \rho(\{s_1\}) \oplus^\rho \rho(\{s_2\})$. By hypothesis, $\rho(\{s_1\}) \subseteq \iota^{-1}(s_1^\alpha)$ and $\rho(\{s_2\}) \subseteq \iota^{-1}(s_2^\alpha)$, hence we deduce that $\rho(\{s_1\}) \oplus^\rho \rho(\{s_2\}) \subseteq \iota^{-1}(s_1^\alpha) \oplus^\rho \iota^{-1}(s_2^\alpha)$. Now, by definition of \oplus^α , we have that $\iota^{-1}(s_1^\alpha) \oplus^\rho \iota^{-1}(s_2^\alpha) = \iota^{-1}(s_1^\alpha \oplus^\alpha s_2^\alpha)$. Therefore, we obtain that $\rho(\{s_1 \oplus s_2\}) \subseteq \iota^{-1}(s_1^\alpha \oplus^\alpha s_2^\alpha)$, as desired. \square

Although the function test^α above aimed at preserving the concrete behavior, we have that, in general, $\rightarrow_{\mathcal{S}^\alpha}$ is not a \sim_α -simulation of $\rightarrow_{\mathcal{S}}$ as shown in the following example.

Example 3.2 Consider the set of states $\text{State} = \{X = n | n \in \mathbb{N}\}$, where X is a variable ranging over the natural numbers. Let $\alpha : \text{State} \rightarrow \text{State}^\alpha$ be the usual even/odd abstraction, where $\text{State}^\alpha = \{X \bmod 2 = 0, X \bmod 2 \neq 0\}$.

Clearly, given $\mathcal{S} = \langle \text{State}, \text{test}, \text{effect} \rangle$, where function test is the ordinary test for variable values, we have that

$$\langle \text{if } (X = 4)? \text{ then } P \text{ else } Q \parallel A, X = 2 \rangle \rightarrow_{\mathcal{S}} \langle Q' \parallel A', X = 2 \rangle$$

However, considering the abstract semantic context $\mathcal{S}^\alpha = \langle \text{State}^\alpha, \text{test}^\alpha, \text{effect}^\alpha \rangle$, where test^α is defined as specified above, we have that action $\langle \text{if } (X = 4)? \text{ then } P \text{ else } Q \parallel A, X = 2 \rangle$ cannot evolve by means of $\rightarrow_{\mathcal{S}^\alpha}$ to any configuration of the form $\langle Q' \parallel A', s \rangle$. In fact, since $\text{test}^\alpha(X = 4, X \bmod 2 = 0)$ holds, the only possible transition is

$$\langle \text{if } (X = 4)? \text{ then } P \text{ else } Q \parallel A, X = 2 \rangle \rightarrow_{\mathcal{S}^\alpha} \langle P' \parallel A', X = 2 \rangle$$

and, clearly, $P \neq Q$ implies $pLab(Q \parallel A) \neq pLab(P \parallel A)$.

The problem is that function $test^\alpha$ is an over-approximation of $test$ and may return *true* even if the concrete version returns *false*. This is critical in synchronous languages, since it may lead to losing the synchronization between the parallel operators (as illustrated in the example), and the concrete and abstract semantics produce completely different traces. Since the abstract program cannot produce a trace that simulates the concrete one, the abstract relation $\rightarrow_{\mathcal{S}^\alpha}$ is incorrect.

4 Correct Abstraction of Actions

Let $P = D[A_0]$ be a program. Consider a semantic context $\mathcal{S} = \langle State, test, effect \rangle$ together with an abstraction function $\alpha : State \rightarrow State^\alpha$, which determines an abstract semantic context $\mathcal{S}^\alpha = \langle State^\alpha, test^\alpha, effect^\alpha \rangle$. In this section, we show how to construct an abstract program $P^\alpha = \alpha(D)[\alpha(A_0)]$ such that, for each initial state $s_0 \in State$, $\mathcal{T}(\mathcal{S}^\alpha)(\alpha(D))(\langle \alpha(A_0), \alpha(s) \rangle)$ is a correct \sim_{α} -simulation of $\mathcal{T}(\mathcal{S})(D)(\langle A_0, s_0 \rangle)$. The key point of this transformation is to guarantee that the new abstract program $\alpha(D)[\alpha(A_0)]$ may be executed under the same semantic context \mathcal{S} as the original program $D[A_0]$. Recall that our main interest is to define a source-to-source transformation, which allows us to exploit the maturity, generality and sophistication of other eventually existing processing techniques and tools for the source language.

In order to effectively define a source-to-source transformation from concrete programs into abstract ones, we need that each abstract element in $State^\alpha$ be in fact an element in $State$, that is, $State^\alpha$ must be a proper subset of $State$. In this way, each abstract execution consists of a sequence of elements in $State$. For instance, in Example 3.2, the set of abstract states $State^\alpha$ could be defined as the subset $\{X = 0, X = 1\}$ of $State$, $X = 0$ and $X = 1$ representing the even and odd values for variable X , respectively.

4.1 Implementing the abstract basic actions

The first step to implement abstraction by source-to-source transformation consists in constructing abstract versions of the basic actions by reusing the concrete $test$ and $effect$ functions given by the original semantics of the language. For this purpose, we have to combine standard actions so that they behave like the abstract ones, provided that the corresponding correctness conditions are satisfied.

Let $\alpha : Ask \rightarrow Ask$ be the abstraction function⁷ for tests. For each $a \in Ask$, we need to find an action $\alpha(a) \in Ask$ such that, for all $s^\alpha \in State$, $test^\alpha(a, s^\alpha) \iff test(\alpha(a), s^\alpha)$. Recall that it is possible to apply the function $test$ to abstract states since $State^\alpha \subseteq State$. For instance, considering the simple even/odd example given above, test $X \bmod 2 = 0$ can be easily abstracted into test $X = 0$.

⁷ By abusing notation, we use the same name for the abstract function applied to the different elements of the language: states, actions, etc.

Similarly, let $\alpha : \text{Tell} \rightarrow \text{Tell}$ be the abstraction function for tell actions. For each $b \in \text{Tell}$, we need to find $\alpha(b) \in \text{Tell}$ such that, for all $s^\alpha \in \text{State}$, $\text{effect}^\alpha(b, s^\alpha) = \text{effect}(\alpha(b), s^\alpha)$. This means that the abstract action has the same effect than abstracting the result of the concrete action. Following the previous examples, concrete tell action $X = X + 1$ may be transformed into the abstract tell action $X = (X + 1) \bmod 2$.

In order to obtain an effective approximation of the semantics based on abstract interpretation, some sensible decisions would be made at this level which depend on the specific language and are not considered here. These decisions regard in particular how to abstract both, the domain as well as the basic actions. In the following section we discuss how the operators can be generally abstracted. The accuracy of these abstractions directly determines the amount of *non real* traces added to the abstract model.

4.2 Abstracting operators

When abstracting a synchronous language, the main problem for the correctness of the abstract model is the preservation of the suspension behavior of the program; in the asynchronous model this problem does not show up.

In the following, we focus on the abstraction of those actions that are more critical in the sense that their execution can suspend. This includes the global choice action. As we are going to see, also the conditional operator must be handled with particular care. Actually, we are able to abstract the conditional operator without altering the notion of time of the source language only if the language being abstracted does provide an instantaneous choice operator. Below we clarify this point.

Now we are ready to define the abstract versions of the different actions of the language given in Figure 2 in terms of actions of the very same language, i.e., without introducing any new operator that didn't exist in the original language. Observe that the abstraction function α takes into account the labels of the program to be transformed. This is necessary to ensure the correctness of the transformation. It is worth noting that, in the following transformation scheme, we simulate suspensions that occur in the concrete set by replicating configurations in the abstract model. That is, our transformation guarantees that, whenever a concrete configuration $\langle A, s \rangle$ suspends, the corresponding abstract configuration $\langle \alpha(A), s^\alpha \rangle$ is replicated in the subsequent time instant.

Error The abstraction of the error operator is straightforward:

$$\alpha((l_{error})\text{error}) = (l_{error})\text{error}$$

End The abstraction of the end operator is also immediate:

$$\alpha((l_{end})\text{end}) = (l_{end})\text{end}$$

c! The abstraction of the c! operator is:

$$\alpha((l)c!) = (l)\alpha(c)!$$

Sequence The abstraction of the sequentialization is defined as follows:

$$\alpha((l_a)A; (l_b)B) = (l_a)\alpha(A); (l_b)\alpha(B)$$

Parallel The abstraction of the parallel operation is given by:

$$\alpha((l_a)A || (l_b)B) = (l_a)\alpha(A) || (l_b)\alpha(B)$$

Global choice We abstract the behavior of this operator by using the conditional operator. Let $A \equiv (l)\sum_{i=1}^n a_i? \rightarrow (l_i)A_i$, then

$$\begin{aligned} \alpha(A) = & (l)\text{if } \alpha(\neg a_1)? \text{ then} \\ & \dots \\ & \text{if } \alpha(\neg a_n)? \text{ then} \\ & \quad \text{true?} \rightarrow^* (l)(\sum_{i=1}^n \alpha(a_i)? \rightarrow (l_i)\alpha(A_i)) \\ & \quad + \\ & \quad \text{true?} \rightarrow^* p \\ & \quad \text{else } (l)(\sum_{i=1}^n \alpha(a_i)? \rightarrow (l_i)\alpha(A_i)) \\ & \quad \dots \\ & \text{else } (l)(\sum_{i=1}^n \alpha(a_i)? \rightarrow (l_i)\alpha(A_i)) \end{aligned}$$

where the procedure p is defined as $p\{\alpha(A)\}$. Let us justify the structure of the abstraction of the global choice operator in order to ensure correctness w.r.t. the suspension behavior. Note that, in order to achieve correctness, we must ensure that whenever the concrete execution of the program suspends, there exists at least one abstract execution that also suspends; otherwise, the abstract program wouldn't include the whole set of possible executions of the concrete program. Note also that, whenever the condition $\alpha(\neg a_1)$ is not satisfied by an abstract state s^α , we know that there exist no concretization of s^α which satisfies $\neg a_1$. In the concrete model, this means that condition a_1 is satisfied, and thus the choice operator won't suspend. However, in case the above condition $\alpha(\neg a_1)$ is satisfied by s^α , we cannot ensure anything. Therefore, for the case when all conditions $\alpha(\neg a_i)$ are satisfied, we must consider both possibilities: suspension and evolution.

Instantaneous choice The abstraction of this operator is similar to that of the global choice. Let $A \equiv (l)\sum_{i=1}^n a_i? \rightarrow^* (l_i)A_i$, then

$$\begin{aligned} \alpha(A) \equiv & (l)\text{if } \alpha(\neg a_1)? \text{ then} \\ & \dots \\ & \text{if } \alpha(\neg a_n)? \text{ then} \\ & \quad \text{true?} \rightarrow^* (l)(\sum_{i=1}^n \alpha(a_i)? \rightarrow^* (l_i)\alpha(A_i)) \\ & \quad + \\ & \quad \text{true?} \rightarrow^* p \\ & \quad \text{else } (l)(\sum_{i=1}^n \alpha(a_i)? \rightarrow^* (l_i)\alpha(A_i)) \\ & \quad \dots \\ & \text{else } (l)(\sum_{i=1}^n \alpha(a_i)? \rightarrow^* (l_i)\alpha(A_i)) \end{aligned}$$

where procedure p is defined as $p\{\alpha(A)\}$.

Conditional For the abstraction of the conditional operator, and in order to preserve the execution time of the program, the language is required to have an operation that models the instantaneous guarded choice (like \rightarrow^*), i.e., a global choice operator performing the test of guards instantaneously (see the operational semantics of this operator in Figure 2). For a language which would not have such an operator, we could try to use the non-instantaneous version of the global choice; unfortunately, in order to achieve a source-to-source transformation we would need to apply a kind of *time expansion* similar to the one in [1] which will cause the execution of the abstract program to take more time than the intended one. The abstraction of the conditional operator is given by

$$\begin{aligned} \alpha((l)\text{if } a? \text{ then } (l_a)A \text{ else } (l_b)B) &= (l)\text{if } \alpha(\neg a)? \text{ then} \\ &\quad \alpha(a)? \rightarrow^* (l_a)\alpha(A) \\ &\quad + \\ &\quad \text{true?} \rightarrow^* (l_b)\alpha(B) \\ &\quad \text{else } (l_a)\alpha(A) \end{aligned}$$

The intuition behind this encoding is as follows. If we are sure that the condition a is satisfied by any concretization of the current abstract state, i.e., no concretization satisfies $\neg a$ (condition $\alpha(\neg a)$ does not hold), then we can simply execute $\alpha(A)$, which corresponds to the *then*-branch of the original action (labeled l_a). However, if there exists the possibility that a was not satisfied in the concrete model, then we produce two traces by means of a choice operator. The first one can only be executed provided there exists at least the possibility that a was true in the concrete model. If this possibility does not exist, then only the second branch can be followed, executing the $\alpha(B)$ action which corresponds to the *else*-branch of the original conditional action (labeled l_b).

Local declaration The abstraction of the local declaration is straightforward:

$$\alpha((l)\exists_{\bar{x}}A) = (l)\exists_{\bar{x}}\alpha(A)$$

Procedure call The abstraction of the procedure call operator is also immediate:

$$\alpha((l)\text{proc}(\bar{v})) = (l)\text{proc}(\bar{v})$$

Finally, to complete the method, we define the abstraction of programs as follows. Given a program P of the form $D[A_0]$ and let $D = \{\text{proc}_i(\bar{x})\{A_i\} \mid 1 \leq i \leq n\}$ with n declarations, then the abstract version of D is defined as $\alpha(D) = \{\text{proc}_i(\bar{x})\{\alpha(A_i)\} \mid 1 \leq i \leq n\}$ whereas the abstract version of the program P is $\alpha(P) = \alpha(D)[\alpha(A_0)]$.

We aim to prove that the abstract versions of the actions given above produce abstract programs that correctly simulate the corresponding concrete ones. We need the following proposition that ensures that, whenever an operator cannot proceed in the concrete framework, the abstract version may proceed to a configuration where the stored information does not change.

Proposition 4.1 *Let $\mathcal{S} = \langle \text{State}, \text{test}, \text{effect} \rangle$ be a concrete semantic context. Given*

an abstraction function $\alpha : State \rightarrow State^\alpha$, let $State^\alpha$ be an abstract set of states, and consider the abstract semantic context $\mathcal{S}^\alpha = \langle State^\alpha, test^\alpha, effect^\alpha \rangle$. Given an operator A and a state $s \in State$, if $\langle A, s \rangle \not\vdash_{\mathcal{S}}$, then $\forall s^\alpha \in State^\alpha$ such that $s \sim_\alpha s^\alpha$, there exists $s'^\alpha \in State$ s.t. $\langle \alpha(A), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle \alpha(A), s'^\alpha \rangle$ and $s \sim_\alpha s'^\alpha$.

Proof. We proceed by induction on the structure of operator A . We only consider operators that may suspend. In order to simplify the presentation, we have dropped labels from the operators in configurations.

- $A \equiv A_1; A_2$. By rule **R-S2a**, if $\langle A, s \rangle \not\vdash_{\mathcal{S}}$ then $\langle A_1, s \rangle \not\vdash_{\mathcal{S}}$. By induction hypothesis, we know that $\langle \alpha(A_1), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle \alpha(A_1), s'^\alpha \rangle$ and $s \sim_\alpha s'^\alpha$. Now, applying rule **R-S2a**, we have that $\langle \alpha(A_1); \alpha(A_2), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle \alpha(A_1); \alpha(A_2), s'^\alpha \rangle$.
- $A \equiv A_1 || A_2$. By rules **R-S3a** and **R-S3b**, if $\langle A, s \rangle \not\vdash_{\mathcal{S}}$ then $\langle A_1, s \rangle \not\vdash_{\mathcal{S}}$ and $\langle A_2, s \rangle \not\vdash_{\mathcal{S}}$. By induction hypothesis, we know that $\langle \alpha(A_1), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle \alpha(A_1), s'^\alpha \rangle$, $\langle \alpha(A_2), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle \alpha(A_2), s''^\alpha \rangle$, and $s \sim_\alpha s'^\alpha, s \sim_\alpha s''^\alpha$. Now, by applying rule **R-S3a**, we have that $\langle \alpha(A_1) || \alpha(A_2), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle \alpha(A_1) || \alpha(A_2), s'^\alpha \oplus^\alpha s''^\alpha \rangle$. Finally, since \oplus is idempotent, by Proposition 3.1 we deduce that $s \sim_\alpha s'^\alpha \oplus^\alpha s''^\alpha$.
- $A \equiv \sum_{i=1}^n (a_i? \rightarrow A_i)$. Using the semantic rules, if $\langle A, s \rangle \not\vdash_{\mathcal{S}}$ then we know that there exist no index j , $1 \leq j \leq n$, such that $test(a_j, s)$ holds, or equivalently, $\forall 1 \leq j \leq n$, $test(\neg a_j, s)$ holds. Let us consider the construction of $\alpha(a_j)$ given in Section 4.1. Since $s \sim_\alpha s^\alpha$, we deduce that $\forall j$, $test(\alpha(\neg a_j), s^\alpha)$ holds. Now, by definition we have

$$\begin{aligned} \alpha(A) &= (l) \text{if } \alpha(\neg a_1)? \text{ then} \\ &\quad \dots \\ &\quad \text{if } \alpha(\neg a_n)? \text{ then} \\ &\quad \quad true? \rightarrow^* (l)(\sum_{i=1}^n \alpha(a_i)? \rightarrow (l_i)\alpha(A_i)) \\ &\quad \quad + \\ &\quad \quad true? \rightarrow^* p \\ &\quad \quad \text{else } (l)(\sum_{i=1}^n \alpha(a_i)? \rightarrow (l_i)\alpha(A_i)) \\ &\quad \quad \dots \\ &\quad \text{else } (l)(\sum_{i=1}^n \alpha(a_i)? \rightarrow (l_i)\alpha(A_i)) \end{aligned}$$

By applying rule **R-S5a** (n times) and rules **R-S8a**, and **R-S7**, we deduce that $\langle \alpha(A), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle \alpha(A), s^\alpha \rangle$.

- $A \equiv \sum_{i=1}^n (a_i? \rightarrow^* A_i)$. This case is similar to the previous one.
- $A \equiv \exists_V^{s_L} A$. To prove this case, we first note that s_L is the local state accumulated during the computation of A , and that V is the set of local variables in A . Thus, by definition, we have that $\exists_V s_L = \epsilon$. Applying rule **R-S6**, if $\langle \exists_V^{s_L} A, s \rangle \not\vdash_{\mathcal{S}}$ then $\langle A, \exists_V s \oplus s_L \rangle \not\vdash_{\mathcal{S}}$. Let $s_L^\alpha, s^\alpha \in State^\alpha$ such that $s_L \sim_\alpha s_L^\alpha$ and $s \sim_\alpha s^\alpha$. Then, by Proposition 3.1, $\exists_V s \oplus s_L \sim_\alpha \exists_V s^\alpha \oplus s_L^\alpha$. Now, by applying the induction hypothesis, we have that $\langle \alpha(A), \exists_V s^\alpha \oplus s_L^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle \alpha(A), s'^\alpha \rangle$, and $\exists_V s \oplus s_L \sim_\alpha s'^\alpha$. Using again rule **R-S6** in the abstract set, we deduce that $\langle \exists_V^{s_L^\alpha} \alpha(A), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle \exists_V^{s'^\alpha} \alpha(A), s^\alpha \oplus \exists_V s'^\alpha \rangle$. To finish this proof, we have to demonstrate that $s \sim_\alpha s^\alpha \oplus \exists_V s'^\alpha$. Since $\exists_V s \oplus s_L \sim_\alpha s'^\alpha$, then we have that $\exists_V s \oplus \exists_V s_L \sim_\alpha \exists_V s'^\alpha$. As $\exists_V s_L \equiv \epsilon$, we have that $\exists_V s \sim_\alpha \exists_V s'^\alpha$. Finally, since $s \equiv \exists_V s \oplus s$ and, by

hypothesis, $s \sim_\alpha s^\alpha$, by Proposition 3.1 we deduce that $s \sim_\alpha s^\alpha \oplus \exists_V s'^\alpha$, as desired. \square

Now we are ready to prove the following result, which ensures that the abstract versions of actions given above produce abstract programs that correctly simulate the corresponding concrete ones.

Theorem 4.2 *Let $P = Decl[A_0]$ be a synchronous program and assume that $\mathcal{S} = \langle State, test, effect \rangle$ is a semantic context. Given an abstraction function $\alpha : State \rightarrow State^\alpha$, let $State^\alpha$ be an abstract set of states, and consider the abstract semantic context $\mathcal{S}^\alpha = \langle State^\alpha, test^\alpha, effect^\alpha \rangle$. Then the trace-based semantic $\mathcal{T}(\mathcal{S}^\alpha)(\alpha(Decl))(\langle \alpha(A_0), \alpha(s_0) \rangle)$ is a correct \sim_α -simulation of $\mathcal{T}(\mathcal{S})(Decl)(\langle A_0, s_0 \rangle)$.*

Proof. We must prove that each transition step in the concrete semantics is mimicked by a corresponding abstract transition step. More specifically, given two labeled operators $(l)A$ and $(l')B$ and two states $s_1, s_2 \in State$ such that $\langle (l)A, s_1 \rangle \rightarrow_{\mathcal{S}} \langle (l')B, s_2 \rangle$ if $s_1^\alpha \in State^\alpha$ is an abstraction of s_1 , that is, $s_1 \sim_\alpha s_1^\alpha$, then there exists $s_2^\alpha \in State^\alpha$ such that $s_2 \sim_\alpha s_2^\alpha$ and $\langle (l)\alpha(A), s_1^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle (l')\alpha(B), s_2^\alpha \rangle$. Note that we have defined the abstraction of actions in such a way that the set of labels in the abstract program is the same as in the concrete one. We proceed by induction on the structure of operator A .

- $(l)A \equiv \mathbf{end}, (l)A \equiv \mathbf{error}$. Both cases hold trivially.
- $(l)A \equiv (l)c!$. By definition of \sim_α , to prove this case it suffices to observe that, if $s \sim_\alpha s^\alpha$, then $\forall c \in Tell, effect(c, s) \sim_\alpha effect^\alpha(c, s^\alpha)$.
- $(l)A \equiv (l)A_1; (l_2)A_2$. First observe that, by definition, $(l)\alpha(A) \equiv (l)\alpha(A_1); (l_2)\alpha(A_2)$. Now, there are two possibilities: (1) If $\langle (l)A_1, s \rangle \rightarrow_{\mathcal{S}} \langle (l_{end})\mathbf{end}, s_1 \rangle$ then by rule **R-S2b** we have that $\langle (l)A_1; (l_2)A_2, s \rangle \rightarrow_{\mathcal{S}} \langle (l_2)A_2, s_1 \rangle$. By induction hypothesis, we know that if $s \sim_\alpha s^\alpha$, then $\langle (l)\alpha(A_1), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle (l_{end})\mathbf{end}, s_1^\alpha \rangle$ and $s_1 \sim_\alpha s_1^\alpha$. Applying rule **R-S2b** again, we obtain that $\langle (l)\alpha(A_1); (l_2)\alpha(A_2), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle (l_2)\alpha(A_2), s_1^\alpha \rangle$, as desired.
The second possibility (2) is when $\langle (l)A_1; (l_2)A_2, s \rangle \rightarrow_{\mathcal{S}} \langle (l_3)A_3; (l_2)A_2, s' \rangle$ where s' is the arrival state in the transition $\langle (l)A_1, s \rangle \rightarrow_{\mathcal{S}} \langle (l_3)A_3, s' \rangle$. The expected result is proved as in (1), but now using rule **R-S2a** to proceed with the concrete and abstract configurations.
- $(l)A \equiv (l)A_1 || (l_2)A_2$. Observe again that, similarly to the previous case, by definition $(l)\alpha(A) \equiv (l)\alpha(A_1) || (l_2)\alpha(A_2)$. We consider again two cases: (1) If $\langle (l)A_1, s \rangle \rightarrow_{\mathcal{S}} \langle (l_3)A_3, s_{1a} \rangle$ and $\langle (l_2)A_2, s \rangle \rightarrow_{\mathcal{S}} \langle (l_4)A_4, s_{1b} \rangle$, then by rule **R-S3a** $\langle (l)A_1 || (l_2)A_2, s \rangle \rightarrow_{\mathcal{S}} \langle (l_3)A_3 || (l_4)A_4, s_1 \rangle$ with $s_1 = s_{1a} \oplus s_{1b}$. By induction hypothesis we know that if $s \sim_\alpha s^\alpha$, then (i) $\langle (l)\alpha(A_1), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle (l_3)\alpha(A_3), s_{1a}^\alpha \rangle$ and $s_{1a} \sim_\alpha s_{1a}^\alpha$, and (ii) $\langle (l_2)\alpha(A_2), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle (l_4)\alpha(A_4), s_{1b}^\alpha \rangle$ and $s_{1b} \sim_\alpha s_{1b}^\alpha$. By applying **R-S3a** we obtain that $\langle (l)\alpha(A_1) || (l_2)\alpha(A_2), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle (l_3)\alpha(A_3) || (l_4)\alpha(A_4), s_{1a}^\alpha \oplus s_{1b}^\alpha \rangle$ as desired since, by Proposition 3.1, $s_{1a} \oplus s_{1b} \sim_\alpha s_{1a}^\alpha \oplus s_{1b}^\alpha$.

The second possibility (2) is presented when one of the parallel operators can-

not proceed. That is, by rule **R-S3b**, $\langle (l)A_1 || (l_2)A_2, s \rangle \rightarrow_{\mathcal{S}} \langle (l_3)A_3 || (l_2)A_2, s' \oplus s \rangle$ with $\langle (l)A_1, s \rangle \rightarrow_{\mathcal{S}} \langle (l_3)A_3, s' \rangle$ and $\langle (l_2)A_2, s \rangle \not\rightarrow_{\mathcal{S}}$. By applying Proposition 4.1, we have that $\langle (l_2)\alpha(A_2), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle (l_2)\alpha(A_2), s''^\alpha \rangle$, and $s \sim_\alpha s''^\alpha$. By induction hypothesis, if $s \sim_\alpha s^\alpha$, then there exists an abstract state s'^α , such that $\langle (l)\alpha(A_1), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle (l_3)\alpha(A_3), s'^\alpha \rangle$. Now, using the rule **R-S3a** to proceed in the abstract configuration, we obtain that $\langle (l)\alpha(A_1) || (l_2)\alpha(A_2), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle (l_3)\alpha(A_3) || (l_2)\alpha(A_2), s'^\alpha \oplus s''^\alpha \rangle$. Finally, by Proposition 3.1, $s' \oplus s \sim_\alpha s'^\alpha \oplus s''^\alpha$, and the desired result follows.

- $(l)A \equiv (l) \sum_{i=1}^n (a_i? \rightarrow (l_i)A_i)$. Observe that, by definition,

$$\begin{aligned} (l)\alpha(A) &= (l)\text{if } \alpha(\neg a_1)? \text{ then} \\ &\quad \dots \\ &\quad \text{if } \alpha(\neg a_n)? \text{ then} \\ &\quad \quad \text{true?} \rightarrow^* (l)(\sum_{i=1}^n \alpha(a_i)? \rightarrow (l_i)\alpha(A_i)) \\ &\quad \quad + \\ &\quad \quad \text{true?} \rightarrow^* p \\ &\quad \quad \text{else } (l)(\sum_{i=1}^n \alpha(a_i)? \rightarrow (l_i)\alpha(A_i)) \\ &\quad \quad \dots \\ &\quad \text{else } (l)(\sum_{i=1}^n \alpha(a_i)? \rightarrow (l_i)\alpha(A_i)) \end{aligned}$$

By hypothesis, we can safely assume that operator A does not suspend. Then, by rule **R-S4**, we know that there exist a $\text{test}(a_j, s)$ that holds for some index j . Given an abstract state s^α such that $s \sim_\alpha s^\alpha$, we can deduce the two following assertions.

- (a) By rule **R-S4**, $\langle (l) \sum_{i=1}^n (a_i? \rightarrow (l_i)A_i), s \rangle \rightarrow \langle (l_j)A_j, s \rangle$.
- (b) Consider the construction of $\alpha(a)$ given in Section 4.1. Since $s \sim_\alpha s^\alpha$, then $\text{test}(a_j, s)$ implies $\text{test}(\alpha(a_j), s^\alpha)$.

Now consider the configuration $C \equiv \langle (l)\alpha(A), s^\alpha \rangle$. We may consider two cases. First (1), when $\text{test}(\alpha(\neg a_k), s^\alpha)$ holds for all $k > 0$. In this case, by rules **R-S5a** (applied n times), **R-S8a** and **R-S4**, and using assertion (b), we deduce that $C \rightarrow_{\mathcal{S}^\alpha} \langle (l_j)\alpha(A_j), s^\alpha \rangle$. In the second case, assume that $\text{test}(\alpha(\neg a_k), s^\alpha)$ does not hold for some $k \geq 1$, and that $\text{test}(\alpha(\neg a_i), s^\alpha)$ holds for all $i < k$. Then, by rules **R-S5a** (applied $k - 1$ times), **R-S5b** and **R-S4** we deduce $C \rightarrow_{\mathcal{S}^\alpha} \langle (l_j)A_j, s^\alpha \rangle$ as desired.

- $(l)A \equiv (l) \sum_{i=1}^n (a_i? \rightarrow^* (l_i)A_i)$. This proof is similar to the previous one.
- $(l)A \equiv (l)\text{if } a? \text{ then } (l_1)A_1 \text{ else } (l_2)A_2$. Observe that, by definition, $(l)\alpha(A) \equiv (l)\text{if } \alpha(\neg a)? \text{ then } (\alpha(a)? \rightarrow^* (l_1)\alpha(A_1) + \text{true?} \rightarrow^* (l_2)\alpha(A_2)) \text{ else } (l_1)\alpha(A_1)$.

We consider two cases: the case when the condition a holds, and the case when it doesn't. In the first case (1), assuming that $\text{test}(a, s)$ holds and that $\langle (l_1)A_1, s \rangle \rightarrow_{\mathcal{S}} \langle (l_3)A_3, s' \rangle$, given an abstract state s^α such that $s \sim_\alpha s^\alpha$, we may deduce the following assertions:

- (a) By rule **R-S5a**, $\langle (l)\text{if } a? \text{ then } (l_1)A_1 \text{ else } (l_2)A_2, s \rangle \rightarrow_{\mathcal{S}} \langle (l_3)A_3, s' \rangle$
- (b) By induction hypothesis, since $s \sim_\alpha s^\alpha$, then $\langle (l_1)\alpha(A_1), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle (l_3)\alpha(A_3), s'^\alpha \rangle$ and $s' \sim_\alpha s'^\alpha$.
- (c) Consider the construction of $\alpha(a)$ given in Section 4.1. Since $s \sim_\alpha s^\alpha$, we have

that $test^\alpha(a, s)$ implies $test(\alpha(a), s^\alpha)$.

Consider now configuration $C \equiv \langle (l) \text{if } \alpha(\neg a)? \text{ then } (\alpha(a)? \rightarrow^* (l_1)\alpha(A_1) + \text{true}? \rightarrow^* (l_2)\alpha(A_2)) \text{ else } (l_1)\alpha(A_1), s^\alpha \rangle$. We can proceed in two ways. First, if $test^\alpha(\alpha(\neg a), s^\alpha)$ holds then, using points (b) and (c) above, and rules **R-S5a** and **R-S8a**, we may deduce that $C \rightarrow_{\mathcal{S}^\alpha} \langle (l_3)\alpha(A_3), s'^\alpha \rangle$, as desired. Second, if $test^\alpha(\alpha(\neg a), s^\alpha)$ does not hold then, using point (b) and rule **R-S5b**, we obtain again that $C \rightarrow_{\mathcal{S}^\alpha} \langle (l_3)\alpha(A_3), s'^\alpha \rangle$.

For the second case (2), assume that $test(a, s)$ does not hold and that $\langle (l_1)A_2, s \rangle \rightarrow_{\mathcal{S}} \langle (l_3)A_3, s' \rangle$. Given an abstract state s^α such that $s \sim_\alpha s^\alpha$, we deduce the following assertions:

- (a) By rule **R-S5b**, $\langle (l) \text{if } a? \text{ then } (l_1)A_1 \text{ else } (l_2)A_2, s \rangle \rightarrow_{\mathcal{S}} \langle (l_3)A_3, s' \rangle$
- (b) By induction hypothesis, since $s \sim_\alpha s^\alpha$, then $\langle (l_1)\alpha(A_2), s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle (l_3)\alpha(A_3), s'^\alpha \rangle$ and $s' \sim_\alpha s'^\alpha$.
- (c) By definition of $test$ given in Section 2.2, $\neg test(a, s)$ is equivalent to $test(\neg a, s)$.

Now, consider the construction of $\alpha(\neg a)$ given in Section 4.1. Since $s \sim_\alpha s^\alpha$, we have that $test(\neg a, s)$ implies $test(\alpha(\neg a), s^\alpha)$.

Finally, if C is the configuration defined above, using rules **R-S5a** and **R-S8a** and points (b) and (c), we have $C \rightarrow_{\mathcal{S}^\alpha} \langle (l_3)\alpha(A_3), s'^\alpha \rangle$.

- $(l)A \equiv \exists_V(l)A_1$. Observe that, by definition, $(l)\alpha(A) \equiv \exists_V(l)\alpha(A_1)$. By rule **R-S6** we have that $\langle \exists_V^{s_L} A_1, s \rangle \rightarrow_{\mathcal{S}} \langle \exists_V^{s'} A'_1, s \oplus \exists_V s' \rangle$ provided $\langle A_1, \exists_V s \oplus s_L \rangle \rightarrow_{\mathcal{S}} \langle A'_1, s' \rangle$, where s_L is the local store; initially s_L is state $\epsilon \in State$, the neutral element of operator \oplus . Given s^α and s_L^α such that $s \sim_\alpha s^\alpha$ and $s_L \sim_\alpha s_L^\alpha$, by Proposition 3.1 we know that $s \oplus s_L \sim_\alpha s^\alpha \oplus s_L^\alpha$. Moreover, by definition, state $\exists_V s$ represents the state s with the information about variables V removed, and if $s \sim_\alpha s^\alpha$, then $\exists_V s \sim_\alpha \exists_V s^\alpha$. By induction hypothesis, we have that $\langle \alpha(A_1), \exists_V s^\alpha \oplus s_L^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle \alpha(A'_1), s'^\alpha \rangle$ and $s' \sim_\alpha s'^\alpha$. Now, by rule **R-S6** again, we have that $\langle \exists_V^{s_L^\alpha} A_1, s^\alpha \rangle \rightarrow_{\mathcal{S}^\alpha} \langle \exists_V^{s'^\alpha} \alpha(A_1), s^\alpha \oplus \exists_V s'^\alpha \rangle$. Then, by applying again Proposition 3.1 we have that $s \oplus \exists_V s' \sim_\alpha s^\alpha \oplus \exists_V s'^\alpha$ as desired.
- $A \equiv \text{proc}(\bar{v})$. The proof of this case is trivial. □

5 Conditions to the source-to-source transformation

A popular approach to implement abstraction is based on using source-to-source transformations. This approach is very convenient since it makes possible to reuse existing analysis and verification tools and techniques for the programming language at hand.

An important advantage of the framework presented so far is that it allows us to analyze the conditions which ensure that a given language can be abstracted by means of a source-to-source transformation. Actually, if the programming language does not provide an instantaneous choice action, then no source-to-source transformation can be given which preserves the timing of programs. This allows us to overcome one of the main problems found in [1], where the execution of the abstract program generally needs more time than the concrete one.

Program abstraction implies a lack of precision. In particular, when a condition is checked in the concrete program, it may occur that the abstract version answers both positively and negatively. Therefore, since we are not sure about the condition satisfaction, both behaviors (when the condition holds and when it does not) must be considered. To this end, the non-determinism is used: a choice between the two runs is defined. In general this means that, in the presence of the conditional operator `or`, whenever some kind of check is done in the original source language, a suitable non-deterministic operator must be used in the abstract version.

Moreover, since the check action and the body of the conditionals run in the same time instant, the non-deterministic operator must be able to execute the check and the body of the conditional operator also at the same time instant. This is why we need an instantaneous choice operator to abstract programming languages which have a conditional operator.

Note that, even if the source-to-source transformation was not possible, it may still be interesting to abstract a particular programming language to perform some kind of analysis or verification. The only drawback is that we won't be able to reuse the analysis or verification tools that may exist for the original language.

6 Concluding remarks

Abstract model checking is becoming one of the most promising approaches to improve the automatic verification of large systems. In this paper, we have defined a generic framework for applying abstract verification and analysis methods including model checking techniques to concurrent languages with maximum parallelism and a synchronous semantics for communication. When abstracting the language, the (abstract) primitives as well as the language constructs have a different intended semantics with respect to the original language. As a consequence, the resulting approximation is required to satisfy some correctness properties in order to safely support accurate program analysis and verification.

Powerful tools for automated analysis and verification of properties, for instance model-checking tools, have been defined for different concurrent languages. As a way to get the benefits provided by these tools when abstract interpretation techniques are applied, an interesting possibility is to implement the model abstraction as a source-to-source transformation, which would allow us to reuse existing these tools. Following this approach, a translation scheme has been proposed that interprets the abstract actions into the original language. Moreover, a characterization of the conditions on the source language ensuring that the abstraction can be defined as a source-to-source transformation has also been provided.

We have applied the transformation methodology both in an imperative and a declarative context: a generalized semantics of PROMELA for abstract model checking was defined in [8], whereas in [1] an abstract semantics was defined for the timed concurrent constraint programming language `tccp`.

As future work we plan to extend our abstraction technique to the asynchronous model of concurrency.

References

- [1] M. Alpuente, M. M. Gallardo, E. Pimentel, and A. Villanueva. A semantic framework for the abstract model checking of tcp programs. *Theoretical Computer Science*, 346(1):58–95, 2005.
- [2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91:64–83, 2003.
- [3] G. Berry and G. Gonthier. The esternel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
- [4] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161:45–83, 2000.
- [5] P. Caspi, D. Pilaud, N. Halbwachs, and J. Place. Lustre: a declarative language for programming synchronous systems. In *ACM Symposium on Principles of Programming Languages (POPL '87)*, 1987.
- [6] M. Codish, M. Falaschi, K. Marriot, and W. Winsborough. A Confluent Semantic Basis for the Analysis of Concurrent Constraint Logic Programs. *Journal of Logic Programming*, 30(1):53–81, 1997.
- [7] S. A. Edwards, N. Halbwachs, R. v. Hanxleden, and T. Stauner, editors. *Synchronous Programming - SYNCHRON'04*. Number 04491 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.
- [8] M. M. Gallardo, P. Merino, and E. Pimentel. A generalized semantics of promela for abstract model checking. *Formal Aspects of Computing*, 16(3):166–193, 2004.
- [9] P. L. Guernic, T. Gautier, M. L. Borgne, , and C. de Marie. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1335, September 1991.
- [10] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [11] F. Maraninchi. The argos language: Graphical representation of automata and description of reactive systems, 1991.
- [12] G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
- [13] V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming, 1994.
- [14] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of Timed Concurrent Constraint Programming. In *Proc. 9th Annual IEEE Symposium on Logic in Computer Science*, pages 71–80, New York, 1994. IEEE.
- [15] E. Zaffanella, R. Giacobazzi, and G. Levi. Abstracting Synchronization in Concurrent Constraint Programming. *Journal of Functional and Logic Programming*, 1997(6), November 1997.