

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

María Alpuente	M. Mar Gallardo	Ernesto Pimentel	Alicia Villanueva
Dept. Sistemas Informát. y Computación	Dept. Lenguajes y Ciencias de la Comput.	Dept. Lenguajes y Ciencias de la Comput.	Dept. Sistemas Informát. y Computación
U. Politécnica de Valencia	U. de Málaga	U. de Málaga	U. Politécnica de Valencia
46022 Valencia	29071 Málaga	29071 Málaga	46022 Valencia
alpuente@dsic.upv.es	gallardo@lcc.uma.es	ernesto@lcc.uma.es	villanue@dsic.upv.es

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. Finally, we ascertain the conditions that make it possible to implement the abstraction by a source to source transformation of the language semantics.

*This work has been partially supported by the EU (FEDER) and the Spanish MEC under grants TIN2004-7943-C04

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. Synchronous models differ from asynchronous ones since they assume that all system components share the same clock and are perfectly synchronized. Synchronous languages [2, 5] such as Esterel [3], StateCharts [8], and Argos [9] are imperative, whereas the relational languages Signal [7] and tccp [11], and the functional language Lustre [4] 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 con-

trol 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. Due to the loss of information caused by the abstraction, the suspension behavior of processes can be disturbed: the suspension of a process in the original model does not generally imply that the process abstractly suspends; hence synchronization in the abstract model might be damaged. Trying to achieve a correct abstract semantics has, at the same time, 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.

In this paper, we analyze the most common primitives and related semantics for the class of synchronous concurrent programming lan-

guages. 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 accurate program analysis and verification. Finally, we ascertain the conditions that make it possible to implement the abstraction by a source to source transformation of the language semantics. Source-to-source transformations are particularly interesting in the context of abstract model checking because they permit reusing model checkers.

2 The synchronous approach

Semantics for concurrent languages are usually given in Plotkin’s structural operational semantics (SOS) style [10], where process behaviors are 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 programming 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 $s \in State$ 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).

We assume that states may be composed by means of $\oplus : State \times State \rightarrow State$. Operator \oplus models the change of state. The definition of \oplus depends on the execution model considered. For the declarative synchronous model (dealing with constraints) 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. Nevertheless, the operator \oplus might perform some satisfiability check before adding new constraints to the store; otherwise, inconsistent states could be reached.

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.

2.2 Basic actions

Basic actions are those actions 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 the set *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 fails¹. Although negation is not uniformly treated within the different programming models, under the considered assumptions the following holds

$$test(a, S) \Leftrightarrow \neg test(\neg a, S)$$

Thus, the 3-uple $\langle State, test, effect \rangle$ (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.

Intuitively, a program is a set of declarations together with an initial action A_0 . A declaration can be seen as a procedure definition $proc(\bar{x})$, 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, end finishes the execution of a thread, $A; B$ 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

¹The $\neg a$ 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. $X > 5$ does not hold, thus we could say that the test of $\neg a$ as defined above holds. Recall that an *opposite* test of $X > 5$ is $X \leq 5$. Note that $\neg X > 5$ does not imply that $X \leq 5$ holds.

Action	::=	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
		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	::=	proc(\bar{x}) {Action}	
		Decl Decl	
Program	::=	Decl[Action]	

Figure 1: Syntax of the synchronous language

the action in the *else* branch. Finally, procedure call $\text{proc}(\bar{v})$ passes values in \bar{v} to the action in the body of the procedure declaration.

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 $\text{nLab} : \rightarrow \mathcal{L}$ 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 $\text{lab}(D) = \{\text{lab}(\text{nLab}(), \text{proc}_i(\bar{x})\{A_i\}) | 1 \leq i \leq n\}$.

$$\begin{aligned}
\text{lab}(l, \text{end}) &= (l)\text{end} \\
\text{lab}(l, a!) &= (l)a! \\
\text{lab}(l, A; B) &= \text{lab}(l, A); \text{lab}(\text{nLab}(), B) \\
\text{lab}(l, A || B) &= \text{lab}(l, A) || \text{lab}(\text{nLab}(), B) \\
\text{lab}(l, \sum_{i \in I} (a_i? \rightarrow A_i)) &= (l) \sum_{i \in I} (a_i? \rightarrow \text{lab}(\text{nLab}(), A_i)) \\
\text{lab}(l, \text{if } a? \text{ then } A \text{ else } B) &= (l) \text{if } a? \text{ then } \text{lab}(\text{nLab}(), A) \\
&\quad \text{else } \text{lab}(\text{nLab}(), B) \\
\text{lab}(l, \exists_{\bar{x}} A) &= \exists_{\bar{x}} \text{lab}(l, A) \\
\text{lab}(l, \sum_{i \in I} (a_i? \rightarrow^* A_i)) &= (l) \sum_{i \in I} (a_i? \rightarrow^* \text{lab}(\text{nLab}(), A_i))
\end{aligned}$$

2.3 Operational Semantics

Figure 2 defines the operational semantics of the language, given by a transition relation between configurations, where each configuration records both the current state and the set of instructions pending to be executed. The transition relation is parametric w.r.t. a semantic context $\mathcal{S} = \langle \text{State}, \text{test}, \text{effect} \rangle$. Different states, and/or different definitions for *test* and *effect* determine different transition relations $\rightarrow_{\mathcal{S}}$.

Definition 1 (Configuration) 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 Figure 2, we show the relation $\rightarrow_{\mathcal{S}}$ representing the operational semantics of the synchronous language, where $a_i (i = 1, \dots, n), a, b \in \text{Ask}$ 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.

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**, agent end is supposed to be labeled with the special tag $l_{\text{end}} \in \mathcal{L}$. Given a configuration $\langle A, s \rangle$, A being the parallel

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	$\frac{\langle A, s \rangle \rightarrow_{\mathcal{S}} \langle A', s' \rangle, \langle B, s \rangle \not\rightarrow_{\mathcal{S}}}{\langle A B, s \rangle \rightarrow_{\mathcal{S}} \langle A' B, s' \rangle}$	
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	$\frac{\langle A, \exists_V s_G \oplus s_L \rangle \rightarrow_{\mathcal{S}} \langle A', \exists_V s'_G \oplus s'_L \rangle}{\langle \exists_V^{s_L} A, s_G \rangle \rightarrow_{\mathcal{S}} \langle \exists_V^{s'_L} A', s'_G \rangle}$	
R-S7	$\langle p(\bar{v}), s \rangle \rightarrow_{\mathcal{S}} \langle A[\bar{v}/\bar{x}], s \rangle$	if $\text{proc}(\bar{x})\{A\}$
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 $\text{test}(a_j, s)$

Figure 2: Operational semantics of a synchronous language

composition of agents $A_1 || \dots || A_n$, we denote with $\text{pLab}(A)$ the ordered sequence $l_1 || \dots || l_n$ of labels of agents in 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 between both states sets. Then we reformulate the classic notion of simulation as follows:

Definition 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 config-*

uration such that $s_1 \sim s'_1$ and $\text{pLab}(A) = \text{pLab}(A')$ 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 $\text{pLab}(B) = \text{pLab}(B')$.

Assume $\mathcal{S} = \langle \text{State}, \text{test}, \text{effect} \rangle$. Given a synchronous program P of the form $D[A_0]$, and an initial state $s_0 \in \text{State}$, 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 $\mathcal{S} = \langle \text{State}, \text{test}, \text{effect} \rangle$ and $\mathcal{S}' = \langle \text{State}', \text{test}', \text{effect}' \rangle$, 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 $\text{pLab}(A_i) = \text{pLab}(A'_i)$.

In order to discuss the relation among different semantics, we define a notion of observable which is parametric to the specific semantic context $\mathcal{S} = \langle \text{State}, \text{test}, \text{effect} \rangle$. We first define the projection on the second component of a trace \downarrow such that $\epsilon \downarrow = \epsilon$ and $(\langle A_i, s_i \rangle \rightarrow_s s) \downarrow = s_i \cdot s \downarrow$, where s is a trace.

Definition 3 (Observable) *Given a semantics context defined by the 3-uple $\mathcal{S} = \langle \text{State}, \text{test}, \text{effect} \rangle$, a program P of the form $D[A_0]$ and an initial configuration $\langle A_0, s_0 \rangle$, we define the set of observable traces $\mathcal{O}_{\mathcal{S}}(P) = \{s \downarrow \mid s \in \mathcal{T}(\mathcal{S})(P)(\langle A_0, s_0 \rangle)\}$*

Now we are ready to formalize the abstract semantics.

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* $\rho : \wp(\text{State}) \rightarrow \wp(\text{State})$ 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 $s \in \text{State}$, and an *uco* ρ over $(\wp(\text{State}), \subseteq)$, $\rho(\{s\})$ 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 $\oplus^\rho : \text{State}^\rho \times \text{State}^\rho \rightarrow \text{State}^\rho$ as $ss_1 \oplus^\rho ss_2 = \rho(\{s_1 \oplus s_2 \mid s_1 \in ss_1, s_2 \in ss_2\})$. Observe that this definition guarantees that $\rho(\{s_1 \oplus s_2\}) \subseteq \rho(\{s_1\}) \oplus^\rho \rho(\{s_2\})$.

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

In the sequel, we denote $\iota\rho$ and ι^{-1} with α and γ , respectively.

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 $\text{test}^\rho : \text{Ask} \times \wp(\text{State}) \rightarrow \{\text{false}, \text{true}\}$ and effect $\text{effect}^\rho : \text{Tell} \times \wp(\text{State}) \rightarrow \wp(\text{State})$ functions as follows:

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

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

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:

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

Now we can easily prove the following result:

Proposition 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 $\rho(\{s_1 \oplus s_2\}) \subseteq \rho(\{s_1\}) \oplus^\rho \rho(\{s_2\})$. Since, by hypothesis, $\rho(\{s_1\}) \subseteq \iota^{-1}(s_1^\alpha)$ and $\rho(\{s_2\}) \subseteq \iota^{-1}(s_2^\alpha)$, we may deduce that $\rho(\{s_1\}) \oplus^\rho$

$\rho(\{s_2\}) \subseteq \iota^{-1}(s_1^\alpha) \oplus^\rho \iota^{-1}(s_2^\alpha)$. Since, by definition of \oplus^α , we have that $\iota^{-1}(s_1^\alpha) \oplus^\rho \iota^{-1}(s_2^\alpha) = \iota^{-1}(s_1^\alpha \oplus^\alpha s_2^\alpha)$, we obtain that $\rho(\{s_1 \oplus s_2\}) \subseteq \iota^{-1}(s_1^\alpha \oplus^\alpha s_2^\alpha)$, as desired. ■

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

Example 1 Consider the set of states $State = \{X = n \mid n \in \mathbb{N}\}$, where X is a variable ranging over the natural numbers. Let $\alpha : State \rightarrow State^\alpha$ be the usual even/odd abstraction, where $State^\alpha = \{X \bmod 2 = 0, X \bmod 2 \neq 0\}$.

Clearly, given $\mathcal{S} = \langle State, test, effect \rangle$, 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 $\mathcal{S}^\alpha = \langle State^\alpha, test^\alpha, effect^\alpha \rangle$, where $test^\alpha$ is defined as specified above, we have that action $\langle \text{if } (X = 4)? \text{ then } P \text{ else } Q \parallel A \rangle$ cannot evolve by means of $\rightarrow_{\mathcal{S}^\alpha}$ to any configuration of the form $\langle Q \parallel A, s \rangle$. In fact, 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, $\text{pLab}(Q \parallel A) \neq \text{pLab}(P \parallel A)$. 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 agents (as illustrated in the example), and the concrete and abstract semantics produce completely different traces. Thus, the abstraction is not correct since the abstracted program cannot produce a trace that simulates the concrete one.

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 $s_0 \in State$, $\mathcal{T}(\mathcal{S})(\alpha(D))(\langle \alpha(A_0), \alpha(s) \rangle)$ 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 $State$.

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 function² for tests. For each $a \in Ask$, we need to find an action $\alpha(a) \in Ask$ such that, for all $s^\alpha \in State^\alpha$:

$$test^\alpha(a, s^\alpha) \iff test(\alpha(a), s^\alpha)$$

Recall that it is possible to apply the function $test$ to abstract states because $State^\alpha \subseteq State$.

Similarly, let $\alpha : Tell \rightarrow Tell$ be the abstraction function for tell actions. For each $b \in Tell$,

²By abusing notation, we use the same name for the abstract function applied to the different elements of the language: states, actions, etc.

we need to find a tell action $\alpha(b) \in \text{Tell}$ 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.

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: how to abstract the domain and the basic actions. In the following section we discuss how the rest of actions can be generally abstracted. The accuracy of these abstractions directly determine 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 in 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 agent must be handled with particular care. Actually, the conditional agent cannot be abstracted preserving the notion of time within the same source language if the language being abstracted does not provide an instantaneous choice agent. 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 own 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.

end The abstraction of the end operator is

straightforward:

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

c! The abstraction of the $c!$ operator is straightforward:

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

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 agent by using the conditional agent.

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

Let us justify the structure of the abstraction of the global choice agent 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 set of possible executions of the concrete program. Note that, whenever the condition $\alpha(\neg a_1)$ is not satisfied, we know that there exist no concretization which satisfies $\neg a_1$. In the concrete model, this means that the condition is satisfied thus the choice agent won't suspend. However, we cannot ensure anything whenever the above condition is satisfied. Therefore, for the case when the second condition $\alpha(\neg a_2)$ is satisfied, we must consider both possibilities: suspension and evolution, and thus proceed.

Instantaneous choice The abstraction of this agent is similar to that of global choice.

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

conditional For the abstraction of the conditional agent, and in order to preserve the execution time of the program, we need that the language has an operation that models the instantaneous guarded choice, i.e., a global choice agent performing the test of guards instantaneously (see the operational semantics of this agent 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 that the execution of the abstract program to take more time than the intended one.

The abstraction of the conditional agent is given by

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

The intuition behind this encoding is as follows. If we are sure that the condition a is satisfied by the current state, i.e., no concretization of $\neg a$ is true, then we can simply execute A , which corresponds to the else branch of the original action. If there exists the possibility that a was not satisfied in the concrete model, then we model two traces by means of a choice

agent. 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 then the B action which corresponds to the then branch of the original action.

The following result ensures that the abstract version of the program is correct w.r.t. the concrete program. The proof can be seen in Appendix A.

Theorem 1 *Let $P = Decl[A_0]$ be a synchronous program and assume that $S = \langle State, test, effect \rangle$ is a semantic context. Let $State^\alpha$ be an abstract set of states, $\alpha : State \rightarrow State^\alpha$ being an abstraction function which transforms the original states into abstract ones. Consider now the abstract semantic context $S^\alpha = \langle State^\alpha, test^\alpha, effect^\alpha \rangle$ defined as described in Section 3.2. Then the trace-based semantic $T(S^\alpha)(\alpha(Decl))(\langle \alpha(A_0), \alpha(s_0) \rangle)$ is a correct \sim_α -simulation of $T(S)(Decl)(\langle A_0, s_0 \rangle)$.*

5 Conditions to the source-to-source transformation

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

In the framework presented so far, we are ready to analyze the conditions that ensure that the considered language can be abstracted by employing source-to-source transformations.

Actually, our claim mainly boils down to say that, if the programming language does not provide an instantaneous choice action, then no source-to-source transformation can be given which does preserve the timing of programs. This was one of the main problems found in [1], where the execution of the abstract program generally needs more time than the concrete one. The framework in this

paper allows one to overcome this drawback. Currently, we are working on the proof of this result by defining a collecting, fully abstract denotational semantics which disregards the instantaneous choice action, and then showing that the abstract program does not mimic the concrete one unless stuttering is introduced.

6 Concluding remarks

Abstract model checking is becoming one of the most promising approaches to improve the automatic verification of large systems. Applying abstract model checking involves the abstraction of both the model to be analyzed and the properties to be checked within the model. When abstracting the model, both primitives and language constructs change their intended semantics with respect to the original language. On the other hand, the resulting approximation should satisfy some correctness properties in order to safely support accurate program analysis and verification. In this paper, we have defined a generic framework for applying abstract model checking techniques to concurrent languages with maximum parallelism and a synchronous semantics for communication.

Model checkers usually offer powerful tools for automated analysis and verification of properties. As a way to get the benefits provided by these tools when abstract interpretation techniques are applied, a very interesting issue is the possibility of implementing the model abstraction as a source-to-source transformation, which would allow us to reuse existing model checkers. We have also dealt with this problem, and a translation scheme has been proposed to interpret abstract actions into the original language. Moreover, a characterization of the conditions allowing abstractions by source-to-source transformations was provided.

As particular cases of the proposed approach, we have already applied these kinds of transformations both in an imperative and a declarative context. In fact, in [6] we defined a generalized semantics of PROMELA for abstract model checking, whereas in [1] an

abstract semantics was defined for the timed concurrent constraint programming language tccp.

As future work we plan to extend the achieved results 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 tccp programs. *Theor. Comput. Sci.*, 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. *Proc. of the IEEE*, 91:64–83, 2003.
- [3] G. Berry and G. Gonthier. The estereel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
- [4] P. Caspi, D. Pilaud, N. Halbwachs, and J. Place. Lustre: a declarative language for programming synchronous systems. In *ACM Symp. on Princ. of Prog. Langs. (POPL '87)*, 1987.
- [5] S. A. Edwards, N. Halbwachs, R. v. Hanxleden, and T. Stauner, editors. Number 04491 in Dagstuhl Seminar Proc. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.
- [6] M. M. Gallardo, P. Merino, and E. Pimentel. A generalized semantics of promela for abstract model checking. *Formal Asp. Comput.*, 16(3):166–193, 2004.
- [7] P. L. Guernic, T. Gautier, M. L. Borgne, , and C. de Marie. Programming real-time applications with SIGNAL. *Proc. of the IEEE*, 79(9):1321–1335, September 1991.

- [8] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [9] F. Maraninchi. The argos language: Graphical representation of automata and description of reactive systems, 1991.
- [10] G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
- [11] 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.

A Proof of Theorem 1

Proof To prove the theorem, we must prove that each transition step in the concrete semantics is mimicked by a corresponding abstract transition step. Note that the set of labels in the abstract program is the same than in the concrete one. We proceed by induction. The base case is defined for the first execution step, i.e., we consider each possible kind of action for A_0 .

$A_0 \equiv \text{end}$. In both cases (concrete and abstract programs) there is no rule defined for end, thus there is no transition. The two initial configurations are $\langle (l)\text{end}, s_0 \rangle$ which is simulated by $\langle \alpha((l)\text{end}), \alpha(s_0) \rangle \equiv \langle (l)\text{end}, \alpha(s_0) \rangle$ since $s_0 \sim_\alpha \alpha(s_0)$.

$A_0 \equiv c!$. $\langle (l)c!, s_0 \rangle \rightarrow_S \langle (l_{\text{end}})\text{end}, s_1 \rangle$ and $s_1 = \text{effect}(c, s_0)$. On the other hand, $\langle \alpha((l)c!), \alpha(s_0) \rangle \rightarrow_{S^\alpha} \langle (l_{\text{end}})\text{end}, s'_1 \rangle$ where $s'_1 = \text{effect}(\alpha(c), \alpha(s_0))$. Note that $\alpha((l)c!) = (l)\alpha(c)!$. Labels clearly coincide and $s_1 \sim_\alpha s'_1$ since, by definition of α and S^α , $\text{effect}(c, s_0) \sim_\alpha \text{effect}(\alpha(c), \alpha(s_0))$.

$A_0 \equiv A_1; A_2$. There are two possibilities: (1) $\langle (l_1)A_1; (l_2)A_2, s_0 \rangle \rightarrow_S \langle (l)\text{end}, s_1 \rangle$

where s_1 is the arrival state in the transition $\langle (l_1)A_1, s_0 \rangle \rightarrow_S \langle (l)\text{end}, s_1 \rangle$. In the abstract version, $\langle \alpha((l_1)A_1; (l_2)A_2), \alpha(s_0) \rangle \equiv \langle (l_1)\alpha(A_1); (l_2)\alpha(A_2) \rangle \rightarrow_{S^\alpha} \langle (l')\text{end}, s'_1 \rangle$ is the result of the transition $\langle (l_1)\alpha(A), \alpha(s_0) \rangle \rightarrow_{S^\alpha} \langle (l')\text{end}, s'_1 \rangle$.

By induction hypothesis, we know that $s_1 \sim_\alpha s'_1$, and $l = l'$. The second possibility (2) is when $\langle (l_1)A_1; (l_2)A_2, s_0 \rangle \rightarrow_S \langle (l_3)A_3; (l_2)A_2, s_1 \rangle$ where s_1 is the arrival state in the transition $\langle (l_1)A_1, s_0 \rangle \rightarrow_S \langle (l_3)A_3, s_1 \rangle$. In the abstract version, $\langle \alpha((l_1)A_1; (l_2)A_2), \alpha(s_0) \rangle \equiv \langle (l_1)\alpha(A_1); (l_2)\alpha(A_2), \alpha(s_0) \rangle \rightarrow_{S^\alpha} \langle (l'_3)(\alpha(A'_3)); (l_2)\alpha(A_2), s'_1 \rangle$ where $\langle (l_1)\alpha(A_1), \alpha(s_0) \rangle \rightarrow_{S^\alpha} \langle (l'_3)(\alpha(A'_3)), s'_1 \rangle$. By induction hypothesis,

$\langle (l_1)\alpha(A_1), \alpha(s_0) \rangle \rightarrow_{S^\alpha} \langle (l'_3)\alpha(A'_3), s'_1 \rangle$ and $\langle (l_1)A_1, s_0 \rangle \rightarrow \langle (l_3)A_3, s_1 \rangle$ and $l_3 = l'_3$, $s_1 \sim_\alpha s'_1$ which proves the assertion.

$A_0 \equiv A_1 || A_2$. We consider again two cases separately: (1) If $\langle (l_1)A_1, s_0 \rangle \rightarrow_S \langle (l_3)A_3, s_{1a} \rangle$ and $\langle (l_2)A_2, s_0 \rangle \rightarrow_S \langle (l_4)A_4, s_{1b} \rangle$, then $\langle (l_1)A_1 || (l_2)A_2, s_0 \rangle \rightarrow_S \langle (l_3)A_3 || (l_4)A_4, s_1 \rangle$ with $s_1 = s_{1a} \oplus s_{1b}$. In this case, $\langle \alpha((l_1)A_1 || (l_2)A_2), \alpha(s_0) \rangle \equiv \langle (l_1)\alpha(A_1) || (l_2)\alpha(A_2), \alpha(s_0) \rangle \rightarrow_{S^\alpha} \langle (l'_3)\alpha(A'_3) || (l'_4)\alpha(A'_4), s'_1 \rangle$ since $\langle (l_1)\alpha(A_1), \alpha(s_0) \rangle \rightarrow_{S^\alpha} \langle (l'_3)\alpha(A'_3), s'_{1a} \rangle$ and $\langle (l_2)\alpha(A_2), \alpha(s_0) \rangle \rightarrow_{S^\alpha} \langle (l'_4)\alpha(A'_4), s'_{1b} \rangle$ and $s'_1 = s'_{1a} \oplus s'_{1b}$. By induction hypothesis, (i) $\langle (l_1)\alpha(A_1), \alpha(s_0) \rangle \rightarrow_{S^\alpha} \langle (l'_3)\alpha(A'_3), s'_{1a} \rangle$ and $\langle (l_1)A_1, s_0 \rangle \rightarrow_S \langle (l_3)A_3, s_{1a} \rangle$, with $s_{1a} \sim_\alpha s'_{1a}$, and $\text{pLab}(A_3) = \text{pLab}(A'_3)$; (ii) $\langle (l_2)\alpha(A_2), \alpha(s_0) \rangle \rightarrow_{S^\alpha} \langle (l'_4)\alpha(A'_4), s'_{1b} \rangle$ and $\langle (l_2)A_2, s_0 \rangle \rightarrow_S \langle (l_4)A_4, s_{1b} \rangle$, thus $s_{1b} \sim_\alpha s'_{1b}$. Also, $\text{pLab}(A_4) = \text{pLab}(A'_4)$. Therefore, $\langle (l_1)\alpha(A_1) || (l_2)\alpha(A_2), \alpha(s_0) \rangle \rightarrow_{S^\alpha} \langle (l'_3)\alpha(A'_3) || (l'_4)\alpha(A'_4), s'_1 \rangle$ and $\langle (l_1)A_1 || (l_2)A_2, s_0 \rangle \rightarrow_S \langle (l_3)A_3 || (l_4)A_4, s_1 \rangle$.

On the other hand, for the second case (2) $\langle (l_1)A_1 || (l_2)A_2, s_0 \rangle \rightarrow_S \langle (l_3)A_3 || (l_2)A_2, s_1 \rangle$ since $\langle (l_1)A_1, s_0 \rangle \rightarrow_S$

$\langle (l_3)A_3, s_1 \rangle$ and $\langle (l_2)A_2, s_0 \rangle \not\rightarrow_{\mathcal{S}}$. In this case, $\langle \alpha((l_1)A_1 || (l_2)A_2), \alpha(s_0) \rangle \equiv \langle (l_1)\alpha(A_1) || (l_2)\alpha(A_2), \alpha(s_0) \rangle \xrightarrow{\mathcal{S}^\alpha} \langle (l'_3)A'_3 || (l_2)\alpha(A_2), s'_1 \rangle$ since $\langle (l_1)\alpha(A_1), \alpha(s_0) \rangle \xrightarrow{\mathcal{S}^\alpha} \langle (l'_3)\alpha(A'_3), s'_1 \rangle$ and $\langle (l_2)\alpha(A_2), \alpha(s_0) \rangle \not\rightarrow_{\mathcal{S}^\alpha}$. By induction hypothesis, these two simpler steps simulate their corresponding concrete versions. This means that the sequence of labels associated to A'_3 and A_3 coincide and that $s_1 \sim_\alpha s'_1$. Therefore, we can say that $\langle (l_1)\alpha(A_1) || (l_2)\alpha(A_2), \alpha(s_0) \rangle \xrightarrow{\mathcal{S}^\alpha} \langle (l'_3)A'_3 || (l_2)\alpha(A_2), s'_1 \rangle$ and $\langle (l_1)A_1 || (l_2)A_2, s_0 \rangle \xrightarrow{\mathcal{S}^\alpha} \langle (l_3)A_3 || (l_2)A_2, s_1 \rangle$.

In both cases, the correspondence between the abstract and concrete states is proved by using the relation between \sim_α and \oplus^α given in Section 3.2.

$A_0 \equiv \mathbf{if } a? \mathbf{ then } A_1 \mathbf{ else } A_2$. If $test(a, s_0)$ holds,

then $\langle (l)\mathbf{if } a? \mathbf{ then } (l_a)A \mathbf{ else } (l_b)B, s_0 \rangle \rightarrow_{\mathcal{S}} \langle (l_c)A_c, s_1 \rangle$ provided that $\langle (l_a)A, s_0 \rangle \rightarrow_{\mathcal{S}} \langle (l_c)A_c, s_1 \rangle$.

In the abstract version, we have that $\langle \alpha((l)\mathbf{if } a? \mathbf{ then } (l_a)A \mathbf{ else } (l_b)B), \alpha(s_0) \rangle \equiv \langle (l)\mathbf{if } \alpha(\neg a)? \mathbf{ then } (\alpha(a)? (l_a)\alpha(A) + true? (l_b)\alpha(B)) \mathbf{ else } (l_a)\alpha(A), \alpha(s_0) \rangle \xrightarrow{\mathcal{S}^\alpha} \langle (l'_c)\alpha(A_c)', s'_1 \rangle$ provided $\langle (l_a)\alpha(A), \alpha(s_0) \rangle \xrightarrow{\mathcal{S}^\alpha} \langle (l'_c)\alpha(A_c)', s'_1 \rangle$. By induction hypothesis, $\langle (l_a)\alpha(A), \alpha(s_0) \rangle \xrightarrow{\mathcal{S}^\alpha} \langle (l'_c)\alpha(A_c)', s'_1 \rangle$ and $\langle (l_a)A, s_0 \rangle \rightarrow_{\mathcal{S}} \langle (l_c)A_c, s_1 \rangle$, thus the sequence of labels associated to the programs are the same and the condition holds. The other case is solved similarly.

The proof of the remaining cases is analogous.