

Compositional Verification of Asynchronous Processes via Constraint Solving

Giorgio Delzanno¹ and Maurizio Gabbriellini²

¹ Dip. di Informatica e Scienze dell'Informazione - Università di Genova
via Dodecaneso 35, 16146 Genova, Italy, e-mail: giorgio@disi.unige.it

² Dip. di Scienze dell'Informazione - Università di Bologna
Mura Anteo Zamboni, 7 40127 Bologna, Italy, e-mail: gabbri@cs.unibo.it

Abstract. In this paper we investigate the foundations of a constraint-based compositional verification method for infinite-state systems. We first consider an asynchronous process calculus which is an abstract formalization of several existing languages based on the blackboard model. For this calculus we define a constraint-based symbolic representation of a compositional model based on traces. The constraint system we use combines formulas of integer arithmetics with equalities over uninterpreted function symbols in which satisfiability is decidable. The translation is inductively defined via a CLP program. Execution traces of a process can be compositionally obtained from the solutions of the answer constraints of the CLP encoding. This way, the task of compositional verification can be reduced to constraint computing and solving.

1 Introduction

Compositional verification of infinite state systems introduces several problems along two main axis. On one side, in order to apply verification methods developed for finite state systems (e.g. [19, 7]) we need suitable abstractions to finitely represent infinite sets of states. On the other side, compositionality is usually quite difficult to achieve already in the case of finite state systems, as it requires semantics structures which are often rather complicated. Nevertheless, when considering some important classes of infinite state systems, for example those arising in distributed computing, compositional verification is almost mandatory as the environment cannot be fixed in advance.

In this paper we deal with this problem by proposing a new technique which combines classical compositional semantics based on sequences (or traces) with constraint programming. More specifically, we consider an asynchronous process calculus which is an abstract formalization of several existing languages based on the blackboard model, where processes communicate and synchronize by posting and retrieving messages from a global, common store. For this process calculus we define a symbolical representation, in terms of constraints, of a compositional model based on traces, thus reducing the task of (compositional) verification to constraint solving. Our approach is based on the following steps.

- (1) We define a compositional model based on traces of basic actions on the store which describes correctly the sequences of stores obtained in a finite computation. This follows a standard approach to trace based models of concurrent languages (e.g. see [16, 4, 8]), even though our technical treatment is different.
- (2) We represent the traces arising in the compositional model by using the constraint system C_{E+Z} which combines linear integer arithmetics with uninterpreted function symbols. Arithmetic constraints allow us to describe the addition and deletion of a message to the store at a given time instant. Uninterpreted symbols are used to relate store configurations in different time instants. Indeed, they can be used to represent variables which depend on other variables as in the formula $s_a(i+1) = s_a(i) + 1$ which can be used to represent the addition of a message a to the store at time i . From the results on the combination of decision procedures of [20], we know that the quantifier-free satisfiability problem for constraints in C_{E+Z} is decidable. Solvers like ICS [12] and CVCL [6] implement satisfiability procedures for these constraints.
- (3) In order to deal with infinite processes, the above mentioned constraint system is embedded into a Constraint Logic Programming language. Then we compositionally translate a process of our algebra into a CLP program P and a goal G so as to extract the traces describing the semantics of the process from the constraints computed by the evaluation of G in P .

This approach has several benefits. Firstly, we introduce an implicit, compact representation of traces, which are described by C_{E+Z} constraints and which can be obtained explicitly by taking the solutions of these constraints. Moreover, we can use logical operators to represent the operators on sequences which are the semantic counterpart of the syntactic operators of the language, thus obtaining a compositional construction. Notably, the interleaving of sequences which models the syntactic parallel composition can be simply defined in terms of conjunction of constraints, provided that we select the solutions of the resulting constraint with some care. Secondly, we obtain a natural compositional model by translating processes into CLP programs, where we exploit the and-compositionality of the computed answer constraint of CLP programs [14]. Thirdly, we can combine CLP systems with solvers like ICS and CVCL to define compositional verification procedures based on our symbolic semantics. Indeed, since C_{E+Z} constraints can also be used to express initial and final condition on the store, the verification of a property *Prop* for a process P can be reduced to the existence of a satisfiable answer constraint for the CLP translation of P which is consistent with the constraint defining *Prop*. Some preliminary results have been obtained with a prototype implementation combining SICStus Prolog and CVCL designed for checking properties of bounded computations of Petri Nets in [10]. The goal of this present paper is to define the foundations of a generalization of this approach to process algebra.

Related Work The use of sequences (or traces) in the semantics of concurrent languages is not new: compositional models based on traces have been defined for a variety of concurrent languages, ranging from dataflow languages [16] and imperative ones [4, 8] to (timed) concurrent constraint programming [3] and

Linda-like languages [2]. However, all the existing approaches consider sequences explicitly and do not take into account the issue of defining a suitable language for expressing and manipulating them implicitly. As a consequence, the resulting models cannot be used directly to define automatic verification tools. On the other hand, our approach introduces a constraint system for expressing and manipulating sequences, where the semantic parallel composition operator can be simply seen in terms of conjunction of constraints. This allows us to express symbolically the semantic of a process and to reason on it without the need to generate explicitly all the sequences, thus reducing the task of property verification to constraint solving. Furthermore, in order to reason on infinite processes, we use CLP programs to express the semantics of a process. This allows us to use directly CLP engines for program verification. In other CLP-based verification methods like [11, 13] arithmetic constraints are used to represent infinite sets of states and (constraint) logic programs are used to implement a transition system which specifies the operational semantics. In our approach we translate directly processes into CLP programs and we use richer constraints to represent directly set of traces and to reason compositionally about the original processes.

2 An Asynchronous Process Algebra

The calculus that we consider in this paper is an abstract formalization of several existing concurrent languages based on the blackboard model, e.g., [5, 15, 9, 21]. The calculus is equipped with two basic operations $out(a)$ and $in(a)$ for adding and removing a message from a common global store. Then we have the usual parallel composition and (internal) choice operators. We also have a construct which allows to test for presence of a message, allowing different continuations depending on the result of the test. This construct, which cannot be simulated by the basic operations, is needed in order to obtain Turing completeness. Infinite behaviors are expressed by allowing the recursive definition of process constants. Specifically, given a finite set of messages Msg , with typical elements a, b, \dots , and a set of process constants K with typical elements p, q, \dots , the syntax of processes is given by the following grammar:

$$\begin{aligned}
 P, Q &::= \alpha.P \mid \beta?P : Q \mid P \parallel Q \mid P + Q \mid p \mid halt \\
 \alpha &::= out(a) \mid in(a) \\
 \beta &::= inp(a) \mid rdp(a)
 \end{aligned}$$

where we assume that for each process constant p there exists a single definition $p =_{def} B$. A program is a pair $\langle P, D \rangle$ where P is a process and D is the set of definitions for all the process constants in P . Since the out operation is non blocking the communication is asynchronous: a process that wants to communicate with another one simply adds a message to the global store and then proceeds in its computation. The process that wants to receive a message can retrieve it from the global store by performing an in operation, which is blocking: if (an occurrence of) the required message is not present in the store then the computation suspends, possibly being resumed later when the message is

R1	$\langle out(a).P, \mathcal{M} \rangle \rightarrow \langle P, \mathcal{M} \oplus \{a\} \rangle$	
R2	$\langle in(a).P, \mathcal{M} \rangle \rightarrow \langle P, \mathcal{M} \ominus \{a\} \rangle$	provided $a \in \mathcal{M}$
R3	$\langle rdp(a)?P : Q, \mathcal{M} \rangle \rightarrow \langle P, \mathcal{M} \rangle$	provided $a \in \mathcal{M}$
	$\langle rdp(a)?P : Q, \mathcal{M} \rangle \rightarrow \langle Q, \mathcal{M} \rangle$	provided $a \notin \mathcal{M}$
R4	$\langle inp(a)?P : Q, \mathcal{M} \rangle \rightarrow \langle P, \mathcal{M} \ominus \{a\} \rangle$	provided $a \in \mathcal{M}$
	$\langle inp(a)?P : Q, \mathcal{M} \rangle \rightarrow \langle Q, \mathcal{M} \rangle$	provided $a \notin \mathcal{M}$
R5	$\langle P + Q, \mathcal{M} \rangle \rightarrow \langle P, \mathcal{M} \rangle$	
R6	$\frac{\langle P, \mathcal{M} \rangle \rightarrow \langle P', \mathcal{M}' \rangle}{\langle P \parallel Q, \mathcal{M} \rangle \rightarrow \langle P' \parallel Q, \mathcal{M}' \rangle}$	
R7	$\langle p, \mathcal{M} \rangle \longrightarrow \langle P, \mathcal{M} \rangle$	provided $(p =_{def} P) \in D$

Table 1. The transition system (symmetric rules omitted).

available. The process $inp(a)?P : Q$ tests for the presence of (an occurrence of) message a in the current store: if present, (an occurrence of) a is removed and the process continues as P , otherwise the process continues as Q . The process $rdp(a)?P : Q$ behaves similarly with the exception that rdp only tests for the presence of a without removing it. The parallel operator is modeled in terms of interleaving, as usual. As for the choice, we consider here the internal (or local) choice, where the environment cannot influence the choice. External (or global) choice could be considered at the price of some technical complications.

The operational semantics is formally described by a transition system $T = (Conf, \longrightarrow)$. Configurations (in $Conf$) are pairs consisting of a process and a *multiset* of messages \mathcal{M} representing the common *store*. We will use \ominus and \oplus to denote multiset union and difference, respectively. The transition relation \longrightarrow is the least relation satisfying the rules R1-R7 in Table 1, where we omit definitions since these do not change during the computation. So $\langle P, \mathcal{M} \rangle \longrightarrow \langle Q, \mathcal{M}' \rangle$ means that the process P with the store \mathcal{M} and a given set of declarations D can perform a transition step yielding the process Q and the store \mathcal{M}' . An execution run is a sequence of configurations $C_1 C_2 \dots C_i C_{i+1}$ such that $C_i \longrightarrow C_{i+1}$. In the following $Store$ denotes the set of possible stores, i.e. multiset over Msg , while $Store^*$ indicates the set of finite sequences over $Store$. A sequence $\mathcal{M}_1 \dots \mathcal{M}_k$ denotes a successful execution of a process. The observables $\mathcal{O}(\langle P, D \rangle)$ are obtained by projecting runs over sequences in $Store^*$. Formally, for any program $\langle P, D \rangle$,

$$\mathcal{O}(\langle P, D \rangle) = \{\mathcal{M}_1 \dots \mathcal{M}_n \mid \langle P_1, \mathcal{M}_1 \rangle \rightarrow \dots \rightarrow \langle P_n, \mathcal{M}_n \rangle \not\rightarrow, P_1 = P\}$$

Furthermore, we define the observable of *successful runs* as

$$\mathcal{O}_{halt}(\langle P, D \rangle) = \{\mathcal{M}_1 \dots \mathcal{M}_n \mid \langle P_1, \mathcal{M}_1 \rangle \rightarrow \dots \rightarrow \langle P_n, \mathcal{M}_n \rangle, P_1 = P, P_n \equiv halt\}$$

where, as in the following, we assume that $halt \equiv halt \parallel halt$.

E1	$\mathcal{D}[\mathit{halt}] = \{\epsilon\}$
E2	$\mathcal{D}[\mathit{out}(a).P] = \{\mathit{out}(a) \cdot s \mid s \in \mathcal{D}[P]\}$
E3	$\mathcal{D}[\mathit{in}(a).P] = \{\mathit{in}(a) \cdot s \mid s \in \mathcal{D}[P]\}$
E4	$\mathcal{D}[\mathit{rdp}(a)?P : Q] = \{\mathit{rdp}(a) \cdot s \mid s \in \mathcal{D}[P]\} \cup \{\overline{\mathit{rdp}}(a) \cdot s \mid s \in \mathcal{D}[Q]\}$
E5	$\mathcal{D}[\mathit{inp}(a)?P : Q] = \{\mathit{inp}(a) \cdot s \mid s \in \mathcal{D}[P]\} \cup \{\overline{\mathit{inp}}(a) \cdot s \mid s \in \mathcal{D}[Q]\}$
E6	$\mathcal{D}[P \parallel Q] = \mathcal{D}[P] \tilde{\parallel} \mathcal{D}[Q]$
E7	$\mathcal{D}[P + Q] = \mathcal{D}[P] \cup \mathcal{D}[Q]$.
E8	$\mathcal{D}[p] = \mathcal{D}[B] \quad \text{if } p = B \in D$

Table 2. Denotational semantics

For instance, let $P = \mathit{out}(a).\mathit{halt}$, $Q = \mathit{out}(b).\mathit{in}(a).\mathit{halt}$ and let \mathcal{M} be the initial store. Then, $\mathcal{M} \cdot \mathcal{M} \oplus \{a\}$ is in $\mathcal{O}_{\mathit{halt}}(P)$, $\mathcal{M} \cdot (\mathcal{M} \oplus \{b\}) \cdot ((\mathcal{M} \oplus \{b\}) \ominus \{a\})$ is in $\mathcal{O}(Q)_{\mathit{halt}}$ only if $a \in \mathcal{M}$, whereas $\mathcal{M} \cdot (\mathcal{M} \oplus \{b\})$ is in $\mathcal{O}(Q)$ but not in $\mathcal{O}_{\mathit{halt}}(Q)$ and $\mathcal{M} \cdot (\mathcal{M} \oplus \{a\}) \cdot (\mathcal{M} \oplus \{a, b\}) \cdot (\mathcal{M} \oplus \{b\})$ is in $\mathcal{O}_{\mathit{halt}}(P \parallel Q)$.

Denotational Semantics We can compositionally characterize the observables using sequences in \mathcal{A}^* defined on the set of actions

$$\mathcal{A} = \{\mathit{in}(a), \mathit{out}(a), \mathit{inp}(a), \mathit{rdp}(a), \overline{\mathit{inp}}(a), \overline{\mathit{rdp}}(a) \mid a \in \mathit{Msg}\}$$

Furthermore, to define the semantics of parallel, we define an interleaving operator $\tilde{\parallel}$ over \mathcal{A}^* . (The operator $\tilde{\parallel}$ associates less than concatenation.) Intuitively, $s \tilde{\parallel} s'$ merges in all possible ways s and s' maintaining however the internal order of s and s' . Given non empty $s, t, u \in \mathcal{A}^*$ and $x, y \in \mathcal{A}$, $\tilde{\parallel}$ is inductively defined as follows (we assume that $\tilde{\parallel}$ associates less than the concatenation \cdot):

$$\begin{aligned} (x \cdot s) \tilde{\parallel} y &= y \tilde{\parallel} (x \cdot s) = \{(x \cdot t) \mid t \in s \tilde{\parallel} y\} \cup \{y \cdot x \cdot s\} \\ (x \cdot s) \tilde{\parallel} (y \cdot t) &= (y \cdot t) \tilde{\parallel} (x \cdot s) = \{(x \cdot u) \mid u \in s \tilde{\parallel} (y \cdot t)\} \cup \{(y \cdot u) \mid u \in (x \cdot s) \tilde{\parallel} t\} \end{aligned}$$

The operator is extended to sets of sequences in the natural way.

Definition 1. *The denotational semantics is the least function \mathcal{D} from programs to $\wp(\mathcal{A}^*)$ which satisfies the equations in Table 2 (we omit definitions).*

As an example, consider the processes $p = \mathit{inp}(a)?q : r$, $q = \mathit{out}(b).\mathit{out}(c).\mathit{halt}$ and $r = \mathit{out}(c).\mathit{out}(b).\mathit{halt}$. Then, the denotational semantics $\mathcal{D}[p]$ of p contains the sequences $\mathit{inp}(a) \cdot \mathit{out}(b) \cdot \mathit{out}(c)$ and $\overline{\mathit{inp}}(a) \cdot \mathit{out}(c) \cdot \mathit{out}(b)$.

From the denotational semantics we can reconstruct the observables by resorting

to the partial map $eval \in \mathcal{A}^* \times Store \rightarrow Store^*$ defined inductively as follows:

$$\begin{aligned}
eval(\epsilon, \mathcal{M}) &= \mathcal{M} \\
eval(out(a) \cdot s, \mathcal{M}) &= \mathcal{M} \cdot eval(s, \mathcal{M} \oplus \{a\}) \\
eval(in(a) \cdot s, \mathcal{M}) &= \begin{cases} \mathcal{M} \cdot eval(s, \mathcal{M} \ominus \{a\}) & \text{if } a \in \mathcal{M} \\ \mathcal{M} & \text{otherwise} \end{cases} \\
eval(inp(a) \cdot s, \mathcal{M}) &= \mathcal{M} \cdot eval(s, \mathcal{M} \ominus \{a\}) \quad \text{if } a \in \mathcal{M} \\
eval(rdp(a) \cdot s, \mathcal{M}) &= \mathcal{M} \cdot eval(s, \mathcal{M}) \quad \text{if } a \in \mathcal{M} \\
eval(\overline{rdp}(a) \cdot s, \mathcal{M}) &= eval(\overline{inp}(a) \cdot s, \mathcal{M}) = \mathcal{M} \cdot eval(s, \mathcal{M}) \quad \text{if } a \notin \mathcal{M}
\end{aligned}$$

Going back to the previous example, $eval(inp(a) \cdot out(b) \cdot out(c), \emptyset)$ is undefined, since the evaluation of $inp(a)$ in the empty store does not succeed, whereas $eval(\overline{inp}(a) \cdot out(c) \cdot out(b), \emptyset) = \emptyset \cdot \{c\} \cdot \{b, c\}$. Notice that in the definition of the $eval$ map the treatment of $in(a)$ and $inp(a)$ is different: in fact, in case the message a is not present in the store the evaluation of $in(a)$ suspends, hence the result of the $eval$ is simply the store \mathcal{M} , without any further continuation. On the other hand, if the message a is not present in the store the evaluation of $inp(a)$ does not suspend and follows the alternative branch which, in our sequences, is indicated by the $\overline{inp}(a)$ construct (see equation E5 in table 2). Hence, in this case the sequence we are considering does not correspond to any computation.

The following result states the correctness of the denotational semantics with respect to the notion of observables that we consider.

Theorem 1. *For any P , $\mathcal{O}(P) = \{eval(s, \mathcal{M}) \mid s \in \mathcal{D}[[P]], \mathcal{M} \in Store\}$.*

3 The Logic Language $CLP(C_{E+Z})$

In this section we introduce the $CLP(C_{E+Z})$ language, obtained as a specific instance of the CLP scheme [17] by considering constraints defined over the combination of the first order theories of equality over uninterpreted functions and of integer arithmetics. The combined constraint system C_{E+Z} is defined as follows (see Appendix A.1 for preliminary notions on theories). C_{E+Z} constraints are quantifier free formulas built over the signature $\Sigma_E \cup \Sigma_Z$, where Σ_E is a set of uninterpreted functions symbols, and Σ_Z is the usual signature of integer arithmetics containing the symbols $0, 1, \dots, +, -, <, \leq, \dots$. An example of constraint in this language is

$$x \leq y \wedge f(x) = g(y) + 1 \wedge f(g(x) + 1) \leq 2$$

Thus, with these constraints we can represent integer arithmetics over variables that depend on other variables. The theory of equality \mathcal{T}_E is the Σ_E -theory with no axioms, whereas the theory of arithmetics \mathcal{T}_Z is defined as the set of Σ_Z -sentences that are true in the standard interpretation of constants, function and predicate symbols in Σ_Z . Since quantifier-free satisfiability is decidable in \mathcal{T}_E [1] and \mathcal{T}_Z and they are both stably infinite (see [20]), from the general results in of Nelson and Oppen [20] it follows that quantifier-free satisfiability is decidable

$$\begin{aligned}
out(a_i, x) &\Leftarrow s_i(x+1) = s_i(x) + 1 \wedge \bigwedge_{i \neq j} s_j(x+1) = s_j(x) \\
in(a_i, x) &\Leftarrow s_i(x) \geq 1 \wedge s_i(x+1) = s_i(x) - 1 \wedge \bigwedge_{i \neq j, j=1}^k s_j(x+1) = s_j(x) \\
inp(a_i, x) &\Leftarrow s_i(x) \geq 1 \wedge s_i(x+1) = s_i(x) - 1 \wedge \bigwedge_{i \neq j, j=1}^k s_j(x+1) = s_j(x) \\
\overline{inp}(a_i, x) &\Leftarrow s_i(x) = 0 \wedge \bigwedge_{j=1}^k s_j(x+1) = s_j(x) \\
rdp(a_i, x) &\Leftarrow s_i(x) \geq 1 \wedge \bigwedge_{j=1}^k s_j(x+1) = s_j(x) \\
\overline{rdp}(a_i, x) &\Leftarrow s_i(x) = 0 \wedge \bigwedge_{j=1}^k s_j(x+1) = s_j(x)
\end{aligned}$$

Fig. 1. Definition of store atoms: a_i is a message in Msg .

in the combined theory \mathcal{T}_{E+Z} . The Nelson-Oppen combination method can also be used to define a decision procedure for C_{E+Z} constraints combining the decision procedures for \mathcal{T}_E and \mathcal{T}_Z . Solvers like ICS [12] and CVCL [6] implement decision procedures to check satisfiability of these constraints.

The language $CLP(C_{E+Z})$ is defined as follows. Let Π be a finite set of predicate symbols (program predicates), disjoint from Σ_{E+Z} and let \mathcal{V} be a denumerable set of variables. Then, *clauses* are implicitly universally quantified formulas of the form $A_0 \Leftarrow \varphi \wedge A_1 \wedge \dots \wedge A_n$, and *goals* are implicitly existentially quantified formulas of the form $\varphi \wedge A_1 \wedge \dots \wedge A_n$, where φ is a C_{E+Z} constraint over \mathcal{V} , $A_i = p_i(x_1^i, \dots, x_{n_i}^i)$ is an atomic formula with $p_i \in \Pi$ for $i = 0, \dots, n$ and $x_j^k \in \mathcal{V}$ for any $j, k \geq 0$. A $CLP(C_{E+Z})$ program is a set of clauses.

We use \Rightarrow_P to denote the usual notion of derivation relation for goals and (variants of) clauses taken from a CLP program P while \Rightarrow_P^* denotes its reflexive and transitive closure. Intuitively, a derivation step replaces an atomic formula A in a goal G with the body B of a clause $A' \Leftarrow B$ whose head A' can be unified with A . The constraint resulting from unification in conjunction with those in B and in G must be satisfiable. The formal definition is in Appendix A.2 where we also discuss the slight differences of our approach from the CLP scheme.

CLP programs allow to compute *answer constraints*: intuitively, an answer constraint is the conjunction of constraints which are left when all the program predicate have been rewritten. It is possible to give a simple (fixpoint) characterization of the semantics which associate the set of answer constraints to a goal in a given program [14]. The following is a slight modification of such a semantics where we consider a specific goal in the definition.

Definition 2. *The answer constraint semantics of the CLP program P and goal G is defined as*

$$\mathcal{A}(P, G) = \{\varphi \mid G \Rightarrow_P^* \varphi, \varphi \text{ is a } C_{E+Z} \text{ constraint}\}$$

4 Encoding Processes in $CLP(C_{E+Z})$

In the rest of the paper we will focus our attention on the subclass of C_{E+Z} constraints defined below. Suppose that the finite set of messages used in our process language is $Msg = \{a_1, \dots, a_k\}$. Starting from Msg , we build a signature

Σ_E (contained in Σ_{E+Z} by definition) that contains k unary function symbols s_1, \dots, s_k which represent functions from integers to integers. Intuitively, the term $s_i(t)$ represents the number of occurrences of a_i in the store at time t .

The effect of an action $\alpha \in \{out, in, inp, rdp, \overline{inp}, \overline{rdp}\}$ executed at time x can then be described via a *store atom* $\alpha(a_i, x)$, defined via the CLP clauses in Fig. 1. As an example the formula that defines $out(a_i, x)$ in Fig. 1 describes the addition of a new occurrence of message a_i to the store. *Store constraints* combine store atoms and arithmetic formulas.

In order to relate store constraints to the denotational semantics of our process language, we need different notions of *solutions* (see prel. in Appendix A.1). Let φ be a store constraint and V be its set of free variables. A *partial solution* ν of φ is a map from V to \mathbb{Z} such that φ evaluates to true under some \mathcal{T}_{E+Z} -interpretation which extends ν in the natural way. A *solution* extends a partial solution with maps from the function symbols s_i to the functions $\overline{s}_i : \mathbb{Z} \rightarrow \mathbb{Z}$ for $i : 1, \dots, k$. A (partial) solution ν is *injective* for $U \subseteq V$ if $\nu(x) \neq \nu(y)$ for every $x \neq y \in U$. Given a subset $U \subseteq V$ with cardinality k , a (partial) solution ν is *closed* for U if $\nu(x) \in [1, k]$ for every $x \in U$. An C_{E+Z} constraint is satisfiable if it has a solution. The *store function* of a solution ν is defined as $s(t) = \langle \overline{s}_1(t), \dots, \overline{s}_k(t) \rangle$. Notice that $s(t)$ represents the content of the store at time t , i.e. the multiset in which a_i has $s_i(t)$ occurrences for $i : 1, \dots, k$.

An injective solution associates distinct values to updates represented in a store constraint. A closed solution associates values from a closed interval. With a closed injective solution each variable in $U = \{x_1, \dots, x_k\}$ is assigned a distinct value in $1, \dots, k$. This notion will be used to consider sequences of events occurring at time instants with no gaps between them (i.e. a closed system). Some example of these notions are provided in Section B.2 in the Appendix.

Remark 1. It is important to notice that, given a store constraint φ and a subset U of its variables, the injective (closed) solutions of φ for U coincide with the solutions of the constraint $\varphi \wedge \bigwedge_{x,y \in U} x \neq y$ ($\varphi \wedge \bigwedge_{x \in U} 1 \leq x \leq |U|$). Thus, we can express these notions of solution in our logic.

Formal Definition of the Translation We are now ready to show how processes can be translated into $CLP(C_{E+Z})$ programs. The translation is inductively defined via a function \mathcal{T} that takes as input a process term P and a variable x , representing a time instant, and returns a CLP program $Prog$ and a goal G . The CLP program contains the clauses corresponding to the definitions of process constants used in P , together with the clauses defining some new process constants that we introduce in order to represent the choice operator and the conditionals. The goal, on the other hand, is the translation of P where we interpret actions in terms of store constraints, parallel in terms of conjunction and process constants as predicate symbols, where we assume that these predicate are in Π (and therefore not in Σ , as previously discussed). Process constant definitions are translated in terms of clause, in the obvious way. The map $\mathcal{T}r$ is inductively defined by the rules in Table 3. A few explanations for the rules in

In the following we assume $\mathcal{T}(P_1, y) = \langle Prog_1, G_1 \rangle$ and $\mathcal{T}(P_2, z) = \langle Prog_2, G_2 \rangle$:

Halt	$\mathcal{T}(halt, x) = \langle \emptyset, true \rangle$.
Out	$\mathcal{T}(out(a).P_1, x) = \langle Prog_1, out(a, x) \wedge x < y \wedge G_1 \rangle$, where $x \notin Var(G_1) \cup \{y\}$.
In	$\mathcal{T}(in(a).P_1, x) = \langle Prog_1, in(a, x) \wedge x < y \wedge G_1 \rangle$, where $x \notin Var(G_1) \cup \{y\}$.
Rdp	$\mathcal{T}(rdp(a)?P_1 : P_2, x) = \langle Prog_1 \cup Prog_2 \cup Prog_3, p(x) \rangle$, where $Prog_3 = \{p(u) \Leftarrow rdp(a, u) \wedge u < y \wedge G_1, p(u) \Leftarrow \overline{rdp}(a, u) \wedge u < z \wedge G_2\}$, p is a new symbol, and $u \notin Var(G_1 \wedge G_2) \cup \{y, z\}$.
Inp	$\mathcal{T}(inp(a)?P_1 : P_2, x) = \langle Prog_1 \cup Prog_2 \cup Prog_3, p(x) \rangle$, where $Prog_3 = \{p(u) \Leftarrow inp(a, u) \wedge u < y \wedge G_1, p(u) \Leftarrow \overline{inp}(a, u) \wedge u < z \wedge G_2\}$, p is a new symbol, and $u \notin Var(G_1 \wedge G_2) \cup \{y, z\}$.
Choice	$\mathcal{T}(P_1 + P_2, x) = \langle Prog_1 \cup Prog_2 \cup Prog_3, p(x) \rangle$, where $Prog_3 = \{p(y) \Leftarrow G_1, p(z) \Leftarrow G_2\}$, and p is a new symbol.
Par	$\mathcal{T}(P_1 \parallel P_2, x) = \langle Prog_1 \cup Prog_2, (x \leq y, z) \wedge G_1 \wedge G_2 \rangle$, where $x \notin Var(G_1, G_2)$, and $Var(G_1) \cap Var(G_2) = \emptyset$.
Call	$\mathcal{T}(p, x) = \langle \emptyset, p(x) \rangle$.
Def	$\mathcal{T}_d(p =_{def} B, x) = Prog \cup \{p(x) \Leftarrow G\}$, where $\mathcal{T}(B, x) = \langle Prog, G \rangle$.
Proc	$\mathcal{T}_r\langle P, D \rangle = \langle Prog \cup \bigcup_{d \in D} \{\mathcal{T}_d(d, x)\}, G \rangle$, where $\mathcal{T}(P, x) = \langle Prog, G \rangle$.

Table 3. Translation of processes into $CLP(C_{E+Z})$.

Table 3 are in order now.

(Halt) First of all, since the process halt does not produce any constraint on the store, its translation produces the empty program and the empty goal.

(In/Out) The preconditions and effects of actions $out(a)$ and $in(a)$ performed at time x can be described via the store atoms $out(a, x)$ and $in(a, x)$, respectively. Furthermore, in a term like $\alpha.P$ (where $\alpha \in \{out(a), in(a)\}$) we must guarantee that the actions in the continuation P will be executed after α . If x is the time at which α occurs (hence $x + 1$ is the time at which the effect of α becomes visible) and y is the time starting from which actions in P may occur, we can enforce the sequentiality between α and the actions in P by requiring that $x < y$, as we do in the rules for *Out* and *In*. Notice that we also require that $x \notin Var(G_1) \cup \{y\}$, since x must represent a new time instant (this condition, as usual in CLP, can be enforced by simply using a suitable variable renaming). Here and in the following $Var(G_1)$ for $n \geq 1$ denotes the set of all variables occurring G_1 .

(Inp/Rdp) The preconditions and effects of the “if” branch of the conditional $rdp(a)?P_1 : P_2$ performed at time x can be described by using the store atoms $rdp(a, x)$ in conjunction with the formula representing P_1 . Similarly, the store atoms $\overline{rdp}(a, x)$ can be used to model the “else” branch. In order to describe the whole conditional construct we then use a new predicate symbol, say p , which models the choice point in terms of two different clauses (contained in $Prog_3$),

corresponding to the two different cases. The treatment for the $inp(a)?P_1 : P_2$ construct is analogous.

(Choice) To model internal choice we use the CLP non-determinism in the selection of the clause to apply to a given goal. Thus, also in this case we introduce a new predicate name, say p , and we define it using the two different CLP clauses contained in $Prog_3$, which correspond to the two branches of the choice construct. Notice that, differently from the case of the conditionals, here the two clauses need not to be mutually exclusive.

(Par) The translation of parallel composition is more subtle. The only constraint that we can put on the set of events occurring in two distinct processes running in parallel is that they will occur after the whole system started its execution. Now, suppose that y represent the time at which process P_1 starts, z the time at which process P_2 starts and assume that x is the starting point of $P_1 \parallel P_2$: Then the constraint $x \leq y \wedge x \leq z$ must hold. Notice that the encoding that we use does not forbid solutions that map time variables associated to distinct actions to the same value. This might lead to illegal traces, that is, traces where two different basic actions are performed on the store at the same time instant, which is not acceptable since we use an interleaving model for the parallel operator. This illegal traces however can easily be ruled out by considering solutions of E+Z constraints that are *injective* for the set of variables associated to actions, thus ensuring that different variables assumes distinct values.

(Call) A process constant p occurring in a term can be viewed as the invocation of a process definition. In our translation this maps naturally to a goal denoting a call to a clause defining p .

(Proc/Def) Closed processes are finally translated by the function \mathcal{T}_r which uses \mathcal{T}_d to translate the definitions and \mathcal{T} to translate the terms. Notice that the function \mathcal{T}_r introduces an arbitrary initial time instant x . Notice also that the result of \mathcal{T}_r is modified by adding the clauses produced by \mathcal{T} since, as previously mentioned, new predicates and related new clauses are used to model the conditionals and the internal choice.

An example of the encoding is shown in Section B.1 in the Appendix.

Properties of the Encoding The translation of process terms into $CLP(C_{E+Z})$ programs allows us to reconstruct the denotational semantics and the observables, by considering the answer constraint semantics. We first need some definitions. Given a (possibly instantiated) store constraint φ , $A(\varphi)$ is the set of store atoms occurring in φ . Now, given a store atom $\alpha(t, x)$, $T(\alpha(a, t)) = t$ and $E(\alpha(a, t)) = \alpha(a)$. T and E are extended to sets, formulas, and sequences in the natural way. In particular, T applied to a store constraints φ returns the set of variables (time-stamps) associated to the actions $E(\varphi)$. Furthermore, let ν be a partial solution of φ injective for $T(\varphi)$, and let $A(\nu(\varphi)) = \{A_1, \dots, A_n\}$, then

$$S_\nu(\varphi) = \{A_{i_1} \cdot A_{i_2} \cdot \dots \cdot A_{i_n} \mid T(A_{i_k}) \leq T(A_{i_{k+1}}) \text{ for } k : 1, \dots, n-1\}$$

We are now ready to state the adequacy of our encoding. The following theorem shows that the answer constraints of the CLP program associated with a process term is an implicit representation of its denotational semantics.

Theorem 2. *For any closed process $S = \langle D, T \rangle$ such that $\mathcal{T}_r(S) = \langle P, G \rangle$*

$$\mathcal{D}[\![S]\!] = \{ E(s) \mid s \in S_\nu(\varphi), \varphi \in \mathcal{A}(P, G), \nu \text{ partial solut. of } \varphi \text{ injective for } T(\varphi) \}$$

Now, given a solution of φ with store function s (see Section 4) from φ we can extract the history of updates by evaluating s on the time-stamps $T(\nu(\varphi)) = \{t_1 \leq \dots \leq t_k\}$ (recall that $s(t)$ is an alternative representation of a multiset):

$$O_\nu(\varphi) = \{ \mathcal{M}_1 \cdot \dots \cdot \mathcal{M}_k \mid \mathcal{M}_i = s(t_i), \text{ for } i : 1, \dots, k \}$$

The following theorem shows that in order to extract the observable of a process we need to consider solutions that are both injective and closed. Indeed a closed solution assigns to time-stamps values taken from a closed interval with a number of elements that correspond to the number of actions in the term.

Theorem 3. *For any closed process $S = \langle D, T \rangle$ such that $\mathcal{T}_r(S) = \langle P, G \rangle$*

$$\mathcal{O}_{halt}(S) = \{ s \mid s \in O_\nu(\varphi), \varphi \in \mathcal{A}(P, G), \\ \nu \text{ is a solution of } \varphi \text{ injective and closed for } T(\varphi) \}$$

5 Towards Constraint-based Compositional Verification

Theorem 2 and 3 are at the basis of a possible compositional verification method for our calculus. Firstly, we recall that CLP enjoys the *and-compositionality* property: an answer constraint for $G_1 \wedge G_2$ can be obtained by conjoining the answer constraints of G_1 and G_2 [14]. And-compositionality can be exploited for a compositional analysis of composed systems as follows. Suppose that the translation of processes P_1 and P_2 returns the goal G_1 and G_2 and a CLP program $Prog = Prog_1 \cup Prog_2$. Since the constraint encoding a combination of P_1 and P_2 can be expressed as a conjunction $G_1 \wedge G_2 \wedge \varphi$ (see Table 3) we can use and-compositionality to separately analyse G_1 and G_2 and then join the result. Indeed, from Theorem 2, we know that answer constraints characterize open traces of processes and thus they can be further combined with other traces.

Furthermore, Theorem 3 can be exploited to combine CLP and a C_{E+Z} solver for checking properties of observables. To illustrate, let us consider a process P encoded via the CLP program $Prog$ and goal G and suppose we are interested in checking if a given store can be reached starting from an initial one. We first notice that we can encode a configuration \mathcal{M} of the store at time t using the constraint $s_1(t) = n_1, \dots, s_k(t) = n_k$ where n_i is the number of occurrences of message a_i in \mathcal{M} . Reachability can be reduced then to constraint computing and solving as follows. We first exploit the CLP component to compute an answer constraint φ of G (by exploiting and-compositionality this can be done separately for the subcomponents). Suppose now that the time-stamps of actions in

φ are $T(\varphi) = \{x_1, \dots, x_m\}$, where x_1 is the initial point of the evaluation of G . Furthermore, let ψ_0 be a constraint expressing initial condition on the store at time x_1 , and ψ_1 be a constraint expressing conditions on the final store (i.e. at a time greater or equal than x_1, \dots, x_m). By Theorem 3, the existence of an injective and closed solution for $\psi_0 \wedge \varphi \wedge \psi_1$ can be used as sufficient condition for the original reachability question. Notice that to obtain a complete test we need to generate all answer constraints. From remark 1 and from the encoding of store atoms in C_{E+Z} of Fig. 1 the latter problem can be reduced to a satisfiability problem of a C_{E+Z} formula for which we can use a solver like ICS [12] and CVCL [6].

This kind of reasoning can be viewed as an extension of the bounded model checking paradigm in which the encoding of a bounded execution of the concurrent system into the formula passed to the constraint solver is constructed in a fully compositional way. We illustrate these ideas in Example B.3 in the Appendix. Starting from a preliminary work on Petri Nets [10], we are currently working on a CLP-CVCL prototype implementing this verification method for our process algebra. Furthermore, we are investigating how to apply alternative evaluation strategies for the CLP encoding (e.g. bottom-up evaluation) for verification of infinite-state processes.

References

1. W. Ackermann. Solvable cases of the decision procedure. North-holland Publishing Company, 1954.
2. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Denotational Semantics for a Timed Linda Language. In *Proc. PPDP 01*. ACM Press, 2001.
3. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161, 2000.
4. F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures in a paradigm for asynchronous communication. In *Proc. CONCUR'91*, LNCS 527, pages 111–126, 1991.
5. J.P Banatre and D. Le Metayer. Programming by Multiset Transformation. *Communication of the ACM*, 36(1) 98-111, 1993.
6. C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. *CAV '04*, p. 515-518, 2004.
7. A. Biere, A. Cimatti, E. Clarke, Y. Zhu Symbolic Model Checking without BDDs TACAS '99, LNCS 1579, p. 193–207, 1999.
8. S. Brookes. A fully abstract semantics of a shared variable parallel language. In *Proc. LICS 93*, 1993.
9. N. Busi, R. Gorrieri and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167-199, 1998.
10. G. Delzanno and M. Gabbrielli Compositional Bounded Model Checking for Petri Nets. Technical report available under the first author home page, 2005.
11. G. Delzanno and A. Podelski. Model checking in CLP. In *Proc. TACAS'99*, LNCS 1579, pp. 223–239, 1999.
12. J.-C. Filliatre, S. Owre, H. Rue, and N. Shankar. ICS: Integrated Canonizer and Solver. In *Proc. (CAV '01)*, LNCS 2102, pages 246–249, 2001.

13. L. Fribourg. Constraint Logic Programming Applied to Model Checking. In *Proc. LOPSTR'99*, LNCS 1817, pp. 30–41, 1999.
14. M. Gabbrielli and G. Levi. Modeling answer constraints in Constraint Logic Programs. In *Proc. ICLP '91*, 1991.
15. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 70(1): 80-112, 1985.
16. B. Jonsson. A model and a proof system for asynchronous processes. In *Proc. PODC '85*, pages 49–58. ACM Press, 1985.
17. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19-20:503–582, 1994.
18. Z. Manna and C. G. Zarba Combining Decision Procedures. In Formal Methods 46 at the Cross Roads: From Panacea to Foundational Support, LNCS 2757, pages 381–422. Springer, 2003.
19. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.
20. G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.
21. V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of POPL '90*, pages 232–245, 1990.

A Preliminaries

A.1 Σ -Theories

Let us first recall that a signature Σ consists of a set Σ_C of constants, a set Σ_F of function symbols, and a set Σ_P of predicate symbols. Σ -terms are first order term constructed using variables in V and the symbols in Σ in the usual way, while Σ atomic formulas are either objects of the form $p(t_1, \dots, t_n)$ where $p \in \Sigma_P$ is a predicate symbol and t_i is a Σ -term for $i = 1, \dots, n$, or of the form $t_1 = t_2$ where t_1 and t_2 are terms and $=$ is the logical equality symbol. Σ -formulas are built upon atomic formulas and standard logical connectives. Σ -sentences are Σ -formulas with no free variables. A Σ -theory T is any set of Σ -sentences. A Σ -interpretation (or Σ -structure) \mathcal{I} with domain D is defined as a map which interprets variables and constants of Σ as element of D , function symbols of arity k as functions from D^k to D and predicate symbols of arity k as relation over D^k . Equality is interpreted as identity on the domain D . Furthermore, given a Σ -theory T , a T -interpretation is a Σ -interpretation \mathcal{I} such that all the formulas in T evaluate to true under \mathcal{I} .

A Σ -formula φ is T -satisfiable (or simply satisfiable) if φ evaluates to true under some T -interpretation; it is T -valid if φ evaluates to true under all T -interpretation.

A.2 CLP Derivations

Let C be the clause $p(y_1, \dots, y_k) \leftarrow \psi \wedge B_1 \wedge \dots \wedge B_k$, and G be the goal $\varphi \wedge A_1 \wedge \dots \wedge A_i \wedge p(x_1, \dots, x_k) \wedge A_{i+1} \wedge \dots \wedge A_n$ such that G and C do not share any variable (renaming can be used here to ensure this condition), and ρ

be the formula $x_1 = y_1 \wedge \dots \wedge x_k = y_k$. The relation \Rightarrow_C is defined over goals and clauses as follows

$$G \Rightarrow_C \varphi \wedge \psi \wedge \rho \wedge A_1 \wedge \dots \wedge A_i \wedge \dots \wedge B_k \wedge A_{i+1} \wedge \dots \wedge A_n$$

provided $\rho \wedge \varphi \wedge \psi$ is satisfiable. Note that this notion of computation step is slightly different from that one of the usual CLP scheme, as in our case the interpretation for the function symbols used in the constraints is not fixed, while this is not the case in the usual CLP programs. This difference, however, does not affect the results presented in this paper.

B Main Example

Consider the process term $p||q$ and the following definitions

$$\begin{aligned} p &= out(b).out(c).halt \\ q &= out(a).halt \end{aligned}$$

The denotation of p and q and $p||q$ are

$$\begin{aligned} \mathcal{D}[p] &= \{out(b).out(c)\} \\ \mathcal{D}[q] &= \{out(a)\} \\ \mathcal{D}[p||q] &= \{out(a).out(b).out(c), out(b).out(a).out(c), out(b).out(c).out(a)\} \end{aligned}$$

The evaluation of the sequence $out(a).out(b).out(c)$ on the empty store returns the observable $\emptyset \cdot \{a\} \cdot \{a, b\} \cdot \{a, b, c\}$.

B.1 CLP encoding

The CLP encoding of definition $p = out(b).out(c).halt$ returns the clause

$$C_1 = p(z) \Leftarrow out(b, z) \wedge z < v \wedge out(c, v).$$

The CLP encoding of definition $q = out(a).halt$ returns the clause

$$C_2 = q(x) \Leftarrow out(a, x)$$

Thus, the CLP encoding of $p||q$ returns the program $Prog$ with clauses C_1 and C_2 and the goal

$$G = u \leq z \wedge u \leq x \wedge p(z) \wedge q(x)$$

B.2 Solutions and Observables

By and-compositionality of CLP, the answer constraint of G can be obtained by conjoining the answer constraints of its subgoals. Despite of the parallel operator, in our example G has only one derivation $G \Rightarrow_{Prog} \varphi$ returning the answer constraint

$$\varphi \doteq u \leq z \wedge u \leq x \wedge out(a, x) \wedge out(b, z) \wedge z < v \wedge out(c, v)$$

where $A(\varphi) = \{out(a, x), out(b, z), out(c, v)\}$ and $T(\varphi) = \{x, z, v\}$. The solution injective for $T(\varphi)$ defined as $\nu = [u \mapsto 1, x \mapsto 1, z \mapsto 5, v \mapsto 7]$ gives us the sequence (in $S_\nu(\varphi)$) ordered with respect to the time-stamps:

$$out(a, 1) \wedge out(b, 5) \wedge out(c, 7)$$

which corresponds to the sequence $out(a) \cdot out(b) \cdot out(c)$.

On the other hand, an example of solution ν' injective and closed for $T(\varphi)$ (i.e. such that the variables x, z, v range in the interval $[1, 3]$) is (evaluation of integer variables):

$$u \mapsto 1, x \mapsto 1, z \mapsto 2, v \mapsto 3$$

and (evaluation of uninterpreted functions):

$$s(1) = \langle 0, 0, 0 \rangle, s(2) = \langle 1, 0, 0 \rangle, s(3) = \langle 1, 0, 1 \rangle, s(4) = \langle 1, 1, 1 \rangle$$

where $s(t) = \langle s_1(t), s_2(t), s_3(t) \rangle$ and $a_1 = a, a_2 = b, a_3 = c$. The solution ν' defines the observable in $\mathcal{O}_{\text{hat}}(\langle D, p||q \rangle)$ defined as $\emptyset \cdot \{a\} \cdot \{a, c\} \cdot \{a, b, c\}$.

B.3 From Verification to Constraint Solving

Given the constraint

$$\varphi \doteq u \leq z \wedge u \leq x \wedge out(a, x) \wedge out(b, z) \wedge z < v \wedge out(c, v)$$

we can formulate a reachability problem for $p||q$ by simply adding constraints on initial and final configuration of the store. As an example, to check if the store contains at least an occurrence of message c at the end of the computation we define the constraint

$$\psi = s_1(u) = s_2(u) = s_3(u) = 0 \wedge \varphi \wedge v < w \wedge s_3(w) \geq 1$$

and then take solutions injective for $T(\varphi)$ and closed for $T(\varphi) \cup \{u, w\}$. Since injective and closed solution can be represented within the constraint language, we can expand the constraint as:

$$\psi' = \psi \wedge x \neq v \wedge x \neq z \wedge v \neq z \wedge 1 \leq u, x, y, z, w \leq 5$$

This way, the original verification problem is polynomially reduced to a constraint problem on the logic C_{E+Z} . Solvers like ICS and CVCL can be used to test satisfiability for ψ' .