Compositional Verification of Asynchronous Processes via Constraint Solving

Giorgio Delzanno[1] and Maurizio Gabbrielli[2]

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

[2] Dip. di Scienze dell'Informazione, Università di Bologna, Mura Anteo Zamboni, 7 40127 Bologna, Italy
gabbri@cs.unibo.it

Abstract. 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 finite computations of a compositional model based on traces. The constraint system we use combines formulas of integer arithmetics with equalities over uninterpreted functions 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 hand, in order to apply verification methods developed for finite state systems we need suitable abstractions to finitely represent infinite sets of states. On the other hand, 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 in which software agents operate 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 like [4, 11, 16], where processes communicates 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.

Technically, our approach is based on the following steps. First of all, 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 [14, 8]), even though our technical treatment is different. 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. Following from the results on the combination of decision procedures of [15], we know that the quantifier-free satisfiability problem for constraints in C_{E+Z} is decidable. In order to deal with finite computations of recursive process definitions, the above mentioned constraint system is embedded into a Constraint Logic Programming (CLP) language [13]. 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. In this setting a recursive process definition naturally translates into a CLP recursive clause.

The resulting encoding has several benefits. Firstly, we introduce an implicit, compact representation of finite computations, which are described by C_{E+Z}formulas and which can be obtained explicitly by taking the solutions of these constraints. Moreover, we can use logical operators to represents 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. Thirdly, we can combine CLP systems with solvers like CVCL [5] and MathSAT [7] 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 conditions on the store, the verification of a property $\mathcal{P}$ for a process P can be reduced to the existence of a satisfiable answer constraint for the CLP translation of P in conjunction with the constraint defining $\mathcal{P}$.

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 [14] and imperative ones [8] to (timed) concurrent constraint programming [3] and Linda-like languages (expressed in terms of process algebras) [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. Finally, while in other CLP-based verification methods like [9] arithmetic constraints are used to represent infinite sets of states, in our approach we combine them with uninterpreted function symbols to represent set of traces.

Plan of the paper In Section 2 we present a process algebra for asynchronous processes. In Section 3 we present the logic language CLP(C_{E+Z}). In Section 3 we present the encoding from processes to CLP(C_{E+Z}) programs. Finally, in Section 5 we discuss related and future work.

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., [4, 11, 16]. 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.

Definition 1 (The language). Given a finite set of messages Msg, with typical elements $a, b \ldots$, and a set of process constants K with typical elements $p, q, \ldots$, the syntax of process terms is given by the following grammar:

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

where we assume that for each process constant p there exists a single definition $p =_{def} P$ where P is a process term and we assume guarded recursion. A *closed process* is a pair $\langle P, D\rangle$ where P is a process term and D is a set of definitions for all the 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*

Table 1. The transition system (symmetric rules omitted)

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	$\dfrac{\langle P, \mathcal{M}\rangle \rightarrow \langle P', \mathcal{M}'\rangle}{\langle P \mid\mid Q, \mathcal{M}\rangle \rightarrow \langle P' \mid\mid Q, \mathcal{M}'\rangle}$	
R7	$\langle p, \mathcal{M}\rangle \longrightarrow \langle P, \mathcal{M}\rangle$	provided $(p =_{def} P) \in D$

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 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. In the following we will work modulo the congruence relation $halt \equiv halt||halt$.

Operational Semantics 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 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 $\mathcal{M}'$. A computation is a sequence of configurations $C_1C_2 \ldots C_k$ such that $C_i \longrightarrow C_{i+1}$ for any $i : 1, \ldots, k-1$. In the following $\mathcal{S}tore$ will denote the set of possible stores, i.e. multiset over Msg, while $\mathcal{S}tore^*$ will indicate the set of finite sequences over $\mathcal{S}tore$. The observables are obtained by projecting runs over sequences in $\mathcal{S}tore^*$.

Definition 2. For any program $\langle P, D\rangle$, we define

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

Furthermore, we define the observable of *terminating computations* as follows.

Definition 3. For any program $\langle P, D\rangle$, we define

$$\mathcal{O}_h(\langle P, D\rangle) = \{\mathcal{M}_1 \cdot \ldots \cdot \mathcal{M}_n \mid \langle P_1, \mathcal{M}_1\rangle \to \ldots \to \langle halt, \mathcal{M}_n\rangle,\ P = P_1\}$$

For instance, let $P = out(b).in(a).halt$. Then, $\emptyset \cdot \{b\}$ is in $\mathcal{O}(Q)$ but not in $\mathcal{O}_h(Q)$, whereas $\{a\} \cdot \{a, b\} \cdot \{b\} \in \mathcal{O}(Q) \cap \mathcal{O}_h(Q)$.

2.1 Denotational Semantics

We can compositionally characterize the observables using the set $\mathcal{A}^*$ of sequences built on $\mathcal{A} = \{in(a), out(a), inp(a), rdp(a), \overline{inp}(a), \overline{rdp}(a) \mid a \in Msg\}$. To define the semantics of parallel, we consider the *shuffle operator* of [12]. The shuffling of s and t in $\mathcal{A}^*$, denoted $s \odot t$, is the set of all sequences of the form

$$s_1 \cdot t_1 \cdot \ldots \cdot s_k \cdot t_k$$

where $k > 0$, $s = s_1 \cdot \ldots \cdot s_k$, $t = t_1 \cdot \ldots \cdot t_k$, and the sequences s_i and t_i, $1 \leq i \leq k$, can be of arbitrary length (including the empty sequence). The operator $\odot$ associates less than concatenation; it can be extended to sets of sequences as follows: $S \odot T = \{s \odot t | s \in S, t \in T\}$.

As an example, $in(a) \cdot in(b) \odot in(c) = \{in(c) \cdot in(a) \cdot in(b), in(a) \cdot in(c) \cdot in(b)\}$.

Definition 4. The denotational semantics is the least function $\mathcal{D}$ from programs to $\wp(\mathcal{A}^*)$ which satisfies the equations in Table 2.

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

From the denotational semantics we can reconstruct the observables by resorting to the partial map $eval \in [\mathcal{A}^* \times \mathcal{S}tore \to \mathcal{S}tore^*]$ defined in Table 3. 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

Table 2. Denotational semantics for process terms with definitions in D

E1 $\mathcal{D}[\![halt]\!] = \{\epsilon\}$

E2 $\mathcal{D}[\![out(a).P]\!] = \{out(a) \cdot s \mid s \in \mathcal{D}[\![P]\!]\}$

E3 $\mathcal{D}[\![in(a).P]\!] = \{in(a) \cdot s \mid s \in \mathcal{D}[\![P]\!]\}$

E4 $\mathcal{D}[\![rdp(a)?P : Q]\!] = \{rdp(a) \cdot s \mid s \in \mathcal{D}[\![P]\!]\} \cup \{\overline{rdp}(a) \cdot s \mid s \in \mathcal{D}[\![Q]\!]\}$

E5 $\mathcal{D}[\![inp(a)?P : Q]\!] = \{inp(a) \cdot s \mid s \in \mathcal{D}[\![P]\!]\} \cup \{\overline{inp}(a) \cdot s \mid s \in \mathcal{D}[\![Q]\!]\}$

E6 $\mathcal{D}[\![P \parallel Q]\!] = \mathcal{D}[\![P]\!] \odot \mathcal{D}[\![Q]\!]$

E7 $\mathcal{D}[\![P + Q]\!] = \mathcal{D}[\![P]\!] \cup \mathcal{D}[\![Q]\!]$.

E6 $\mathcal{D}[\![p]\!] = \mathcal{D}[\![B]\!]$ provided $(p =_{def} B) \in D$

Table 3. The *eval* mapping

$$eval(\epsilon, \mathcal{M}) = \mathcal{M}$$

$$eval(out(a) \cdot s, \mathcal{M}) = \mathcal{M} \cdot eval(s, \mathcal{M} \oplus \{a\})$$

$$eval(in(a) \cdot s, \mathcal{M}) = \begin{cases} \mathcal{M} \cdot eval(s, \mathcal{M} \ominus \{a\}) & \text{if } a \in \mathcal{M} \\ \mathcal{M} \quad \text{otherwise} \end{cases}$$

$$eval(inp(a) \cdot s, \mathcal{M}) = \mathcal{M} \cdot eval(s, \mathcal{M} \ominus \{a\}) \quad \text{if } a \in \mathcal{M}$$

$$eval(rdp(a) \cdot s, \mathcal{M}) = \mathcal{M} \cdot eval(s, \mathcal{M}) \quad \text{if } a \in \mathcal{M}$$

$$eval(\overline{rdp}(a) \cdot s, \mathcal{M}) = eval(\overline{inp}(a) \cdot s, \mathcal{M}) = \mathcal{M} \cdot eval(s, \mathcal{M}) \quad \text{if } a \notin \mathcal{M}$$

$eval(\overline{inp}(a) \cdot out(c) \cdot out(b), \emptyset) = \emptyset \cdot \{c\} \cdot \{b, c\}$. 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). The following result states the correctness of the denotational semantics with respect to the notion of observables of Def. 2.

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

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 [13] 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. C_{E+Z} constraints are quantifier free formulas built over the signature $\Sigma_E \cup \Sigma_Z$, where Σ_E is a set of functions symbols, and $\Sigma_Z = \{0, 1, \ldots, +, -, <, \leq\}$ is the usual signature of integer arithmetics. An example of C_{E+Z} constraint is the formula $x \leq y \wedge f(x) = g(y) + 1 \wedge f(g(x) + 1) \leq 2$. Thus in C_{E+Z} we can represent arithmetics over variables that depend on other variables. The theory of equality is the Σ_E-theory with no axioms, whereas the theory of arithmetics 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. From the general results of Nelson and Oppen [15], quantifier-free satisfiability is decidable in the combined theory $\mathcal{T}_{E+Z}$. Solvers like CVCL [5] and MathSAT [7] implement decision procedures to check satisfiability of these constraints.

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. The language CLP(C_{E+Z}) is defined on top of C_{E+Z} as follows.

Definition 5 (Clauses, Goals, and Programs). A *clause* is an implicitly universally quantified formula of the form

$$A_0 \Leftarrow \varphi \wedge A_1 \wedge \ldots \wedge A_n,$$

and a *goal* is an implicitly existentially quantified formula of the form

$$\varphi \wedge A_1 \wedge \ldots \wedge A_n,$$

where φ is a C_{E+Z} constraint over $\mathcal{V}$, $A_i = p_i(x^i_1, \ldots, x^i_{n_i})$ is an atom with $p_i \in \Pi$, $0 \leq i \leq n$, and $x^k_j \in \mathcal{V}$ for $j, k \geq 0$. A *program* is a set of clauses.

We use $\Rightarrow_P$ to denote the usual notion of CLP derivation relation (see [13]) for goals and (renamings of) clauses taken from a CLP(C_{E+Z}) program P. 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. $\Rightarrow^*_P$ denotes its reflexive and transitive closure. CLP(C_{E+Z}) programs allow to compute *answer constraints* [10]. Formally, we have the following definition.

Definition 6. The *answer constraint semantics* of P and G is defined as

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

Thus an answer constraint is the conjunction of constraints which are left when all atomic goals have been resolved using program clauses.

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

First of all, given the set of messages $Msg = \{a_1, \ldots, a_k\}$, we build the signature $\Sigma_E = \{s_1, \ldots, s_k\}$ (contained in Σ_{E+Z} by definition). The term $s_i(t)$ is used to represent the number of occurrences of message a_i in the store at time t. The effect of an action like $out(a_i)$ executed at time x can then be described via the clause like that defining the *store atom* $out(a_i, x)$ of Fig. 4. Thus in the rest of the paper we will focus our attention on the subclass of C_{E+Z} constraints defined as follows.

Definition 7 (Store Constraints). A *store atom* is a formula $\alpha(a_i, x)$ with $\alpha \in \{out, in, inp, rdp, \overline{inp}, \overline{rdp}\}$ defined by the CLP clauses of Table 4. A *store constraint* is a conjunction of store atoms and arithmetic formulas.

The effect of an action like $out(a_i)$ executed at time x can then be described via the store atom $out(a_i, x)$. Its definition describes the addition of a new occurrence of message a_i to the store. More in general, store constraints will allow us to express in a logical way the composition operators of our process algebra. In order to formally relate store constraints to the denotational semantics of a process, we need to refine the notion of solutions as follows.

Definition 8 (Solutions). Given a store constraint φ with variables in V,

Table 4. Definition of store atoms: a_i is a message in Msg

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

- 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 $\overline{s_i} : \mathbb{Z} \to \mathbb{Z}$ for $i : 1,\ldots,k$;
- a (partial) solution ν is *injective* for $U \subseteq V$ if $\nu(x) \neq \nu(y)$ for every $x, y \in U$ such that $x \neq y$;
- given a subset $U \subseteq V$ with cardinality m, a (partial) solution ν is *closed* for U if $\nu(x) \in [1, m]$ for any $x \in U$.

A C_{E+Z} constraint is satisfiable if it has a solution. The *store function* of a solution ν is defined as $s(t) = \langle \overline{s_1}(t), \ldots, \overline{s_k}(t)\rangle$. 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 $i : 1,\ldots,k$. 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 $U = \{x_1,\ldots,x_k\}$ is assigned a distinct value in $1,\ldots,k$. This notion will be used to consider sequences of a closed system in which events occur at time instants with no gaps between them.

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 $\varphi \wedge \bigwedge_{x,y\in U} x \neq y$ ($\varphi \wedge \bigwedge_{x\in U} 1 \leq x \leq |U|$).

4.1 Formal Definition of the Translation

The translation of a process term P with definitions in D to a CLP(C_{E+Z}) program is inductively defined via the functions $\mathcal{T}_p$, $\mathcal{T}_d$ and $\mathcal{T}$. The function $\mathcal{T}_p$ takes as input the 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 defining some new process constants that we introduce in order to represent the choice operator and the conditionals. The goal G 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 taken from Π (and therefore not in Σ). The function $\mathcal{T}_d$ translates process definitions in D into CLP(C_{E+Z}) clauses. Finally, the function $\mathcal{T}$, defined on top of $\mathcal{T}_d$ and $\mathcal{T}_p$, is used to encode the closed process $\langle P, D\rangle$.

Definition 9 (Translation). Let $\mathcal{V}$ be a denumerable set of variables, then the translation functions $\mathcal{T}_p$, $\mathcal{T}_d$ and $\mathcal{T}$ are formally defined as follows.

Halt: The process halt does not produce any constraint on the store. Thus, for any $x \in \mathcal{V}$ we define $\mathcal{T}_p(halt, x) = \langle \emptyset, true \rangle$.

In/Out: The precondition and effect of the action $out(a)$ performed at time x can be described via the store atom $out(a, x)$ of Table 4. Since the effect of $out(a)$ is visible at time $x + 1$, if y is the time starting from which actions in P may occur, we can enforce the sequentiality between the execution of $out(a)$ and of the actions in P by requiring that $x < y$. Thus, if $\mathcal{T}_p(P, y) = \langle Prog, G \rangle$ for some variable $y \in \mathcal{V}$, then we define

$$\mathcal{T}_p(out(a).P, x) = \langle Prog, out(a, x) \wedge x < y' \wedge G' \rangle$$

for any variable $x \in \mathcal{V}$, and renamings y', G' of y, G such that $x \notin Var(G') \cup \{y'\}$. The treatment of the term $in(a).P$ is analogous (predicate out is replaced by in).

Conditional: 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 atom $rdp(a, x)$ in conjunction with the formula representing P_1. Similarly, the store atom $\overline{rdp}(a, x)$ can be used to model the "else" branch. In order to describe the whole conditional construct we then use a new predicate symbol which models the choice point in terms of two mutually exclusive clauses, corresponding to the two different cases. Formally, if $\mathcal{T}_p(P_1, y_1) = \langle Prog_1, G_1 \rangle$ and $\mathcal{T}_p(P_2, y_2) = \langle Prog_2, G_2 \rangle$ for two variables $y_1, y_2 \in \mathcal{V}$, then we define

$$\begin{aligned}
&\mathcal{T}_p(rdp(a)?P_1 : P_2, x) = \langle Prog \cup Prog_1 \cup Prog_2, p(x) \rangle \\
&Prog = \{p(u) \Leftarrow rdp(a, u) \wedge u < y_1' \wedge G_1',\ p(u) \Leftarrow \overline{rdp}(a, u) \wedge u < y_2' \wedge G_2'\}
\end{aligned}$$

for variables $x, u \in \mathcal{V}$, a new predicate symbol p, and renamings y_1', y_2', G_1', G_2' of y_1, y_2, G_1, G_2 such that $x, u \notin Var(G_1') \cup Var(G_2') \cup \{y_1', y_2'\}$.
The treatment for the term $inp(a)?P_1 : P_2$ 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 symbol and we define it using two different CLP clauses, which correspond to the two branches of the choice construct. Formally, if $\mathcal{T}_p(P_1, y_1) = \langle Prog_1, G_1 \rangle$ and $\mathcal{T}_p(P_2, y_2) = \langle Prog_2, G_2 \rangle$ for $y_1, y_2 \in \mathcal{V}$, then

$$\begin{aligned}
&\mathcal{T}_p(P_1 + P_2, x) = \langle Prog \cup Prog_1 \cup Prog_2, p(x) \rangle \\
&Prog = \{p(y_1) \Leftarrow G_1,\ p(y_2) \Leftarrow G_2\}
\end{aligned}$$

for any $x \in \mathcal{V}$ and a new predicate symbol p. Notice that, differently from the case of the conditionals, here the two clauses in $Prog$ are not mutually exclusive.

Parallel: 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_1 represent the time at which process P_1 starts,

y_2 the time at which process P_2 starts and assume that x is the starting point of $P_1 \mid\mid P_2$. Then the constraint $x \leq y_1 \wedge x \leq y_2$ 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, i.e., 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. Illegal traces can be ruled out by considering solutions that are *injective* (see Def. 8) for the set of variables associated to actions, thus ensuring that different variables assume distinct values (see Theorem 3). Formally, if $\mathcal{T}_p(P_1, y_1) = \langle Prog_1, G_1\rangle$ and $\mathcal{T}_p(P_2, y_2) = \langle Prog_2, G_2\rangle$ for $y_1, y_2 \in \mathcal{V}$, then

$$\mathcal{T}_p(P_1 \mid\mid P_2, x) \;=\; \langle Prog_1 \cup Prog_2, (x \leq y_1' \wedge x \leq y_2') \wedge G_1' \wedge G_2'\rangle,$$

for any variable $x \in \mathcal{V}$, and renamings y_1', y_2', G_1', G_2' of y_1, y_2, G_1, G_2 such that $Var(G_1') \cap Var(G_2') = \emptyset$, and $x \notin Var(G_1') \cup Var(G_2') \cup \{y_1', y_2'\}$.

Constant: 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. Formally, for any $x \in \mathcal{V}$ we define

$$\mathcal{T}_p(p, x) = \langle \emptyset, p(x)\rangle.$$

Definition: Given $p =_{def} P$ and $\mathcal{T}_p(P, x) = \langle Prog, G\rangle$ for $x \in \mathcal{V}$, we define

$$\mathcal{T}_d(p =_{def} P) = Prog \cup \{p(x) \Leftarrow G\}.$$

Program: Now let $D = \{d_1, \ldots, d_n\}$ be process definitions, $y \in \mathcal{V}$, and P be a process term such that $\mathcal{T}_p(P, x) = \langle Prog, G\rangle$ for some $x \in \mathcal{V}$. Then we define

$$\mathcal{T}(\langle P, D\rangle) = \langle \{\mathcal{T}_d(d_1, y), \ldots, \mathcal{T}_d(d_n, y)\} \cup Prog, G\rangle.$$

Notice that the function $\mathcal{T}$ introduces an arbitrary initial time variable x.

4.2 Properties of the Encoding

The translation of a process term into a CLP(C_{E+Z}) program allows us to reconstruct the denotational semantics of Def. 4 and the observables of Def. 2, by considering the answer constraint semantics of Def. 6. To make this claim more formal, it will be convenient to use the following terminology.

Given a (possibly instantiated) store constraint φ, we will use $A(\varphi)$ to denote the set of store atoms occurring in φ, and given a store atom $\alpha(t, x)$, we define $T(\alpha(a, t)) = t$ 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 for an instantiated constraint) associated to the actions $E(\varphi)$. Furthermore, let ν be a partial solution of φ injective for $T(\varphi)$, and let $A(\nu(\varphi)) = \{A_1, \ldots, A_n\}$, then

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

We are now ready to state the adequacy of our encoding. The following theorem shows that the answer constraints of the CLP(C_{E+Z}) program associated with a closed process is an implicit representation of its denotational semantics.

Theorem 2. For any closed process $S = \langle P, D\rangle$ such that $\mathcal{T}(S) = \langle Prog, G\rangle$, we have that $\mathcal{D}[\![S]\!] = \{E(s) \mid s \in S_\nu(\varphi),\ \varphi \in \mathcal{A}(Prog, G)$, and ν is a partial solution of φ 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 $T(\nu(\varphi)) = \{t_1 \leq \ldots \leq t_k\}$ (recall that $s(t)$ is an alternative representation of a multiset):

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

The following theorem shows that in order to extract the observables of a *terminating* process we need to consider solutions that are both injective and closed (see Def. 8). Indeed a closed solution assigns to time variables values taken from a closed interval whose cardinality corresponds to the number of possible actions of the original process term.

Theorem 3. For any closed process $S = \langle P, D\rangle$ such that $\mathcal{T}(S) = \langle Prog, G\rangle$, we have that $\mathcal{O}_h(S) = \{s \mid s \in O_\nu(\varphi),\ \varphi \in \mathcal{A}(Prog, G)$, and ν is a solution of φ injective and closed for $T(\varphi)\}$.

5 Conclusions

In this paper we have shown how to use CLP to obtain a constraint-based symbolic representation of the set of finite computations of asynchronous processes communicating via a common store. Theorem 2 and 3 are at the basis of a possible compositional verification method for this computational model. Indeed 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 [10]. 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 combination of P_1 and P_2 can be expressed as a constraint $G_1 \wedge G_2 \wedge \psi$ (ψ depends on the composition operator) we can use and-compositionality to separately analyze G_1 and G_2 and then join the results. Indeed, from Theorem 2, we know that answer constraints characterize open traces of processes that 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 $s_1(t) = n_1, \ldots, s_k(t) = n_k$ where n_i is the number of occurrences of message a_i in $\mathcal{M}$ for $i : 1, \ldots, k$. Reachability can be reduced then 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 $T(\varphi) = \{x_1, \ldots, x_m\}$, 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 $x_1, \ldots, x_m$. 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. 4 the latter problem can be reduced to a satisfiability problem of a C_{E+Z} formula. This kind of reasoning can be viewed as an extension of the bounded model checking paradigm [6] in which the encoding of a bounded execution of a system is constructed in a compositional way. We are currently working on a prototype implementing this verification method for our process algebra.

References

1. W. Ackermann. Solvable cases of the decision procedure. North-holland, 1954.
2. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Denotational Semantics for a Timed Linda Language. PPDP 2001: 28-36.
3. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161(1): 45-83, 2000.
4. J.P Banatre and D. Le Metayer. Programming by Multiset Transformation. *Communication of the ACM*, 36(1): 98-111, 1993.
5. C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. CAV 2004: 515-518.
6. A. Biere, A. Cimatti, E. Clarke, Y. Zhu Symbolic Model Checking without BDDs TACAS 1999: 193-207.
7. M.Bozzano, R.Bruttomesso, A.Cimatti, T.Junttila, P.v.Rossum, S.Schulz, and R.Sebastiani. An incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic. TACAS 2005: 317-333.
8. S. Brookes. A Fully Abstract Semantics of a Shared Variable Parallel Language. LICS 1993: 98-109.
9. G. Delzanno and A. Podelski. Model Checking in CLP. TACAS 1999: 223–239.
10. M. Gabbrielli and G. Levi. Modeling Answer Constraints in Constraint Logic Programs. ICLP 1991: 238-252.
11. D. Gelernter. Generative Communication in Linda. *TOPLAS*, 70(1): 80-112, 1985.
12. S. Ginsburg and E. H. Spanier. Mappings of languages by two-tape devices *JACM*, 12(3): 423-434, 1965.
13. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *JLP*, 19-20:503-582, 1994.
14. B. Jonsson. A Model and a Proof System for Asynchronous Processes. PODC 1985: 49-58.
15. G. Nelson and D. C. Oppen. Fast Decision Procedures based on Congruence Closure. JACM, 27(2):356-364, 1980.
16. V.A. Saraswat and M. Rinard. Concurrent Constraint Programming. POPL 1990: 232–245.