Synthesizing Concurrent Programs using Answer Set Programming

Emanuele De Angelis[1], Alberto Pettorossi[2], and Maurizio Proietti[3]

[1] Dipartimento di Scienze, University 'G. D'Annunzio', Pescara, Italy
deangelis@sci.unich.it
[2] DISP, University of Rome Tor Vergata, Italy
pettorossi@disp.uniroma2.it
[3] CNR-IASI, Rome, Italy
proietti@iasi.cnr.it

Abstract. We address the problem of the automatic synthesis of concurrent programs within a framework based on Answer Set Programming (ASP). The concurrent program to be synthesized is specified by providing both the behavioural and the structural properties it should satisfy. Behavioural properties, such as safety and liveness properties, are specified by using formulas of the Computation Tree Logic, which are encoded as a logic program. Structural properties, such as the symmetry of processes, are also encoded as a logic program. Then, the program which is the union of these two encodings, is given as input to an ASP system which returns as output a set of answer sets. Finally, each answer set is decoded into a synthesized program that, by construction, satisfies the desired behavioural and structural properties.

1 Introduction

We consider *concurrent programs* consisting of finite sets of *processes* which interact with each other through *communication protocols*. Such protocols are based on a set of instructions, called *synchronization instructions*, operating on shared variables ranging over finite domains. The communication protocols are realized in a distributed manner, that is, every process includes one or more regions of code consisting of synchronization instructions, responsible for the interaction between processes.

Even for a small number of processes, communication protocols which guarantee a desired behaviour of the concurrent programs may be hard to design. In this paper we propose a method for automatically synthesizing correct concurrent programs starting from the formal specification of their desired behaviour.

Methods for the automatic synthesis of concurrent programs from *temporal logic specifications* have been proposed in the past by Clarke and Emerson [6], Manna and Wolper [15], and Attie and Emerson [1,2]. All these authors reduce the task of synthesizing a concurrent program to the task of synthesizing the synchronization instructions of each process. We follow their approach and everything which is irrelevant to the synchronization among processes, is abstracted away and each process is considered to be a finite state automaton.

CONCURRENCY, SPECIFICATION AND PROGRAMMING
M. Szczuka et al. (eds.): Proceedings of the international workshop CS&P 2011
September 28-30, Pułtusk, Poland, pp. 85-98

We introduce a framework, based on logic programming, for the automatic synthesis of concurrent programs. We assume that the *behavioural properties* of the concurrent programs, such as safety and liveness properties, are specified by using formulas of the Computation Tree Logic (CTL for short), which is a very popular propositional temporal logic over branching time structures (see, for instance, [5,6]). This temporal, behavioural specification φ is encoded as a set Π_φ of clauses. We also assume that the processes to be synthesized satisfy suitable *structural properties*, such as a *symmetry* property, and that those properties can be encoded as a set Π_Σ of clauses. Structural properties cannot be easily specified by using CTL formulas and we use, instead, a simple algebraic structure that we will present in the paper. Thus, the specification of a concurrent program to be synthesized consists of a logic program $\Pi = \Pi_\varphi \cup \Pi_\Sigma$ which encodes both the behavioural and the structural properties that the concurrent program should satisfy.

We show that every answer set (that is, every stable model) of the program Π represents a concurrent program satisfying the given specification. Thus, by using an Answer Set Programming (ASP) system, such as DLV [9] or smodels [19], which computes the answer sets of logic programs, we can synthesize concurrent programs which enjoy some desired properties.

We have performed some synthesis experiments and, in particular, we have synthesized some mutual exclusion protocols which are guaranteed to enjoy various properties, such as (i) bounded overtaking, (ii) absence of starvation, and (iii) maximal reactivity (their formal definition will be given in the paper). We finally compare our results with those presented in [1,2,12].

The paper is structured as follows. In Section 2 we recall some preliminary notions and terminology. In Section 3 we present our framework for synthesizing concurrent programs and we define the notion of a symmetric concurrent program. In Section 4 we describe our synthesis procedure and the logic program which we use for the synthesis. In Section 5 we present some examples of synthesis of symmetric concurrent programs. Finally, in Section 6 we discuss the related work and some topics that can be investigated in the future.

2 Preliminaries

Let us recall some basic notions and terminology we will use. We present: (i) the syntax of (a variant of) the *guarded commands* [7] which are used for defining concurrent programs, (ii) some basic notions of *group theory* which are required for defining symmetric concurrent programs, (iii) the syntax and the semantics of the Computation Tree Logic, and (iv) the syntax and the semantics of Answer Set Programming, which is the framework we use for our synthesis method.

Guarded commands. In our variant of the guarded commands we consider two basic sets: (i) variables, v in *Var*, each ranging over a finite domain D_v, and (ii) guards, g in *Guard*, of the form: $g ::= \mathit{true} \mid \mathit{false} \mid v = d \mid \neg g \mid g_1 \wedge g_2$, with $v \in \mathit{Var}$ and $d \in D_v$. We also have the following derived sets whose definitions are mutually recursive: (iii) commands, c in *Command*, of the form: $c ::= \mathit{skip} \mid v := d \mid c_1 \,;\, c_2 \mid \texttt{if}\ gc\ \texttt{fi} \mid \texttt{do}\ gc\ \texttt{od}$, where ‘;’ denotes the *sequential*

composition of commands, and (iv) guarded commands, gc in *GCommand*, of the form: $gc ::= g \rightarrow c \mid gc_1 \,[]\, gc_2$, where '$[]$' denotes the *parallel composition* of guarded commands.

The execution of `if` $gc_1 \,[]\, \ldots \,[]\, gc_n$ `fi` is performed as follows: one of the guarded commands $g \rightarrow c$ in $\{gc_1, \ldots, gc_n\}$ whose guard g evaluates to *true* is chosen, then c is executed; otherwise, if no guard in $\{gc_1, \ldots, gc_n\}$ evaluates to *true* then the whole command `if ... fi` terminates with failure.

The execution of `do` $gc_1 \,[]\, \ldots \,[]\, gc_n$ `od` is performed as follows: one of the guarded commands $g \rightarrow c$ in $\{gc_1, \ldots, gc_n\}$ whose guard g evaluates to *true* is chosen, then c is executed and the whole command `do...od` is executed again; otherwise, if no guard in $\{gc_1, \ldots, gc_n\}$ evaluates to *true* then the execution proceeds with the next command.

Symmetric Groups. A group G is a pair $\langle S, \circ \rangle$, where S is given a set and $\circ$ is a binary operation on S satisfying the following axioms: (i) $\forall x, y \in S.\, x \circ y \in S$ (closure), (ii) $\forall x, y, z \in S.\, (x \circ y) \circ z = x \circ (y \circ z)$ (associativity), (iii) $\exists e \in S. \forall x \in S. e \circ x = x \circ e = x$ (identity element), and (iv) $\forall x \in S. \exists y \in S.\, x \circ y = y \circ x = e$ (inverse element). The *order of a group* G is the cardinality of S. For any $x \in S$, for any $n \geq 0$, we write x^n to denote the term $x \circ \ldots \circ x$ with n occurrences of x. We stipulate that x^0 is e.

A group G is said to be *cyclic* iff there exists an element $x \in S$, called a *generator*, such that $S = \{x^n \mid n \geq 0\}$. We write G_x to denote the cyclic group generated by x.

We denote by $Perm(S)$ the set of all permutations (that is, bijections) on the set S. $Perm(S)$ is a group whose operation $\circ$ is function composition and the identity e is the identity permutation, denoted id. The *order of a permutation* p on a finite set S is the smallest natural number n such that $p^n = id$.

Computation Tree Logic. Computation Tree Logic (CTL) is a propositional branching time temporal logic [5].

Let *Elem* be a finite set of elementary propositions ranged over by b. The syntax of a CTL formula φ is as follows:

$$\varphi ::= b \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathsf{EX}\,\varphi \mid \mathsf{EG}\,\varphi \mid \mathsf{E}[\varphi_1 \,\mathsf{U}\, \varphi_2]$$

Let us introduce the following abbreviations: (i) $\varphi_1 \vee \varphi_2$ for $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, (ii) $\mathsf{EF}\varphi$ for $\mathsf{E}[true \,\mathsf{U}\, f]$ (iii) $\mathsf{AG}\,\varphi$ for $\neg\mathsf{EF}\,\neg\varphi$, (iv) $\mathsf{AF}\varphi$ for $\neg\mathsf{EG}\,\neg\varphi$, (v) $\mathsf{A}[\varphi_1 \,\mathsf{U}\, \varphi_2]$ for $\neg\mathsf{E}[\neg\varphi_2 \,\mathsf{U}\, (\neg\varphi_1 \wedge \neg\varphi_2)] \wedge \neg\mathsf{EG}\,\neg\varphi_2$, (vi) $\mathsf{AX}\,\varphi$ for $\neg\mathsf{EX}\,\neg\,\varphi$, (vii) $\mathsf{A}[\varphi_1 \,\mathsf{R}\, \varphi_2]$ for $\neg\mathsf{E}[\neg\varphi_1 \,\mathsf{U}\, \neg\varphi_2]$, and (viii) $\mathsf{E}[\varphi_1 \,\mathsf{R}\, \varphi_2]$ for $\neg\mathsf{A}[\neg\varphi_1 \,\mathsf{U}\, \neg\varphi_2]$.

We define the semantics of CTL by giving a Kripke structure $\mathcal{K} = \langle S, S_0, \lambda, R \rangle$, where: (i) S is a finite set of *states*, (ii) $S_0 \subseteq S$ is a set of *initial states*, (iii) $R \subseteq S \times S$ is a total *transition relation* (thus, $\forall u \in S, \exists v \in S, \langle u, v \rangle \in R$), and (iv) $\lambda : S \rightarrow \mathcal{P}(Elem)$ is a total, *labelling function* that assigns to every state $s \in S$ a subset $\lambda(s)$ of the set *Elem*.

For reasons of simplicity, when the set of the initial states is a singleton $\{u\}$, we will feel free to identify $\{u\}$ with u.

A path π in $\mathcal{K}$ from a state is an infinite sequence $\langle s_0, s_1, \ldots \rangle$ of states such that, for all $i \geq 0$, $\langle s_i, s_{i+1} \rangle \in R$. For $i \geq 0$, we denote by π_i the i-th element of π. The fact that a CTL formula φ holds in a state s of a Kripke structure $\mathcal{K}$

will be denoted by $\mathcal{K}, s \vDash \varphi$. For any CTL formula φ and state s, we define the relation $\mathcal{K}, s \vDash \varphi$ as follows:

$\mathcal{K}, s \vDash b$ iff $b \in \lambda(s)$
$\mathcal{K}, s \vDash \neg\, \varphi$ iff $\mathcal{K}, s \vDash \varphi$ does not hold
$\mathcal{K}, s \vDash \varphi_1 \wedge \varphi_2$ iff $\mathcal{K}, s \vDash \varphi_1$ and $\mathcal{K}, s \vDash \varphi_2$
$\mathcal{K}, s \vDash \mathsf{EX}\, \varphi$ iff there exists $\langle s, t\rangle \in R$ such that $\mathcal{K}, t \vDash \varphi$
$\mathcal{K}, s \vDash \mathsf{E}[\varphi_1\, \mathsf{U}\, \varphi_2]$ iff there exists a path $\pi = \langle s, s_1, \ldots\rangle$ in $\mathcal{K}$ and $i \geq 0$ such that $\mathcal{K}, \pi_i \vDash \varphi_2$ and for all $0 \leq j < i$, $\mathcal{K}, \pi_j \vDash \varphi_1$
$\mathcal{K}, s \vDash \mathsf{EG}\, \varphi$ iff there exists a path π such that $\pi_0 = s$ and for all $i \geq 0$, $\mathcal{K}, \pi_i \vDash \varphi$

2.1 Answer Set Programming

Answer set programming (ASP) is a declarative programming paradigm based on the answer set semantics of logic programs [10,14]. We assume the version of ASP with function symbols [3]. Now let us recall some basic definitions of ASP. For those not recalled here we refer to [3,10,14]. A *rule* r is an implication of the form:

$$a_1 \vee \ldots \vee a_k \leftarrow a_{k+1} \wedge \ldots \wedge a_m \wedge \mathtt{not}\, a_{m+1} \wedge \ldots \wedge \mathtt{not}\, a_n$$

where $a_1, \ldots, a_k, \ldots, a_n$ (for $k \geq 0$, $n \geq k$) are atoms and '$\mathtt{not}$' denotes negation as failure [11]. Given a rule r, we define the following sets: $\mathit{head}(r) = \{a_1, \ldots, a_k\}$, $\mathit{pos}(r) = \{a_{k+1}, \ldots, a_m\}$, and $\mathit{neg}(r) = \{a_{m+1}, \ldots, a_n\}$. An *integrity constraint* is a rule r such that $\mathit{head}(r) = \emptyset$. A *logic program* is a set of rules. When we write a rule r with variables, we actually mean all the ground instances of r. An *interpretation* I of a program Π is a subset of the Herbrand base. The *Gelfond-Lifschitz transformation* of a program Π with respect to an interpretation I is the program $\Pi^I = \{\mathit{head}(r) \leftarrow \mathit{pos}(r) \mid r \in \Pi \wedge \mathit{neg}(r) \cap I = \emptyset\}$. An interpretation M is said to be an *answer set* of Π iff M is the least Herbrand model of Π^M. The answer set semantics of Π assigns to Π a set of answer sets, denoted $\mathit{ans}(\Pi)$. Given an answer set $M \in \mathit{ans}(\Pi)$ and an atom a, we write $M \models a$ to denote that $a \in M$.

3 Specifying Concurrent Programs

Let $\mathcal{P} = \{P_1, \ldots, P_k\}$ be a finite set of *processes*. With every process $P_i \in \mathcal{P}$ we associate a variable $\mathtt{s}_i$, called the *local state*, ranging over a finite domain L, which is the same for all processes. The variable $\mathtt{s}_i$ can be tested and modified by P_i only. All processes may test and modify also a *shared variable* $\mathtt{x}$, which ranges over a finite domain D.

A *concurrent program* consists of a finite set $\mathcal{P}$ of processes that are executed in parallel and interact with each other through a communication protocol realized by a set of commands acting on the shared variable $\mathtt{x}$. Here is the formal definition of a concurrent program.

Definition 1 (k-Process Concurrent Program). Let L be a set of local states and D be a domain of the shared variable $\mathtt{x}$. For any $k > 1$, a *k-process concurrent program* C is a command of the form:

$C: \quad \mathtt{s}_1 := l_1; \ldots; \mathtt{s}_k := l_k;\ \mathtt{x} := d_0;\ \mathtt{do}\ P_1 \,[\!]\, \ldots \,[\!]\, P_k\ \mathtt{od}$

where $\mathtt{s}_1, \ldots, \mathtt{s}_k, \mathtt{x} \in \mathit{Var}$, $l_1, \ldots, l_k \in L$, and $d_0 \in D$.
Every process P_i in $P_1 \,[\!]\, \ldots \,[\!]\, P_k$ is a guarded command of the form:

$P_i: \quad \mathit{true} \rightarrow \mathtt{if}\ gc_1 \ [\!]\ \ldots\ [\!]\ gc_n\ \mathtt{fi}$

Every guarded command gc in $gc_1 \,[\!]\, \ldots \,[\!]\, gc_n$ is of the form:

$gc: \quad \mathtt{s}_i = l \wedge \mathtt{x} = d \ \rightarrow\ \mathtt{s}_i := l'; \mathtt{x} := d';$

where $l, l' \in L$ and $d, d' \in D$. □

We shall use the guarded command $\mathtt{s}_i = l \wedge \mathtt{x} = d \rightarrow \mathit{skip}$ as a shorthand for $\mathtt{s}_i = l \wedge \mathtt{x} = d \rightarrow \mathtt{s}_i := l; \mathtt{x} := d$. The command $\mathtt{s}_1 := l_1; \ldots; \mathtt{s}_k := l_k;\ \mathtt{x} := d_0;$ is called *initialization* of C.

Example 1. Let L be the set $\{\mathtt{t}, \mathtt{u}\}$ and D be the set $\{\mathtt{0}, \mathtt{1}\}$. A 2-process concurrent program C is:

$\mathtt{s}_1 := \mathtt{t};\ \mathtt{s}_2 := \mathtt{t};\ \mathtt{x} := \mathtt{0};\ \mathtt{do}\ P_1 \,[\!]\, P_2\ \mathtt{od}$

where P_1 and P_2 are defined as follows:

```
P1 : true → if                              P2 : true → if
       s1=t ∧ x=0 → s1:=u; x:=0;                  s2=t ∧ x=1 → s2:=u; x:=1;
    [] s1=t ∧ x=1 → skip;                      [] s2=t ∧ x=0 → skip;
    [] s1=u ∧ x=0 → s1:=t; x:=1;               [] s2=u ∧ x=1 → s2:=t; x:=0;
 fi                                          fi
```

This program is the familiar program for two processes, each of which either 'thinks' in its noncritical section ($\mathtt{s}_i = \mathtt{t}$) or 'uses a resource' in its critical section ($\mathtt{s}_i = \mathtt{u}$). The shared variable $\mathtt{x}$ gives each process its turn to enter the critical section: if $\mathtt{x} = \mathtt{0}$, process P_1 is in its critical section, and if $\mathtt{x} = \mathtt{1}$, process P_2 is in its critical section. □

Now we introduce the semantics of k-process concurrent programs by using Kripke structures. We model a state u of a k-process concurrent program C by a $(k+1)$-tuple $\langle l_1, \ldots, l_k, d\rangle$, where: (i) the first k components are the values of the local state variables $\mathtt{s}_1, \ldots, \mathtt{s}_k$, and (ii) d is the value of the shared variable $\mathtt{x}$.

Definition 2 (Kripke Structure Associated with a k-Process Concurrent Program). Let C be a k-process concurrent program of the form

$C: \quad \mathtt{s}_1 := l_1; \ldots; \mathtt{s}_k := l_k;\ \mathtt{x} := d_0;\ \mathtt{do}\ P_1 \,[\!]\, \ldots \,[\!]\, P_k\ \mathtt{od}$

where the l_i's belong to L and d_0 belongs to D. The *Kripke structure $\mathcal{K}$ associated with C* is the 4-tuple $\langle S, S_0, R, \lambda\rangle$, where:

(i) the set S of states is $L^k \times D$,
(ii) the set S_0 of initial states is the singleton $\{\langle l_1, \ldots, l_k, d_0\rangle\}$,
(iii) the set $R \subseteq S \times S$ of transitions
$\{\langle u, v\rangle \mid i, j \in \{1, \ldots, k\} \ \wedge\ \mathtt{s}_i = l \ \wedge\ \mathtt{x} = d \rightarrow \mathtt{s}_i := l';\ \mathtt{x} := d' \text{ in } P_i \ \wedge$
$u(\mathtt{s}_i) = l \wedge u(\mathtt{x}) = d \wedge v(\mathtt{s}_i) = l' \wedge v(\mathtt{x}) = d' \wedge u \neq v \wedge \forall j \neq i,\ u(\mathtt{s}_j) = v(\mathtt{s}_j)\}$,
where for all states $t \in S$, for all variables $\mathtt{x} \in \mathit{Var}$, $t(\mathtt{x})$ denotes the value of the variable $\mathtt{x}$ in t, and

(iv) for all states t of the form $\langle l_1, \ldots, l_k, d\rangle$, the value $\lambda(t)$ is defined to be $\{\mathtt{s}_1 = l_1, \ldots, \mathtt{s}_k = l_k, \mathtt{x} = d\}$.
The set *Elem* of the elementary propositions is the set $\bigcup_{t \in S} \lambda(t)$. □

We make the following assumptions about k-process concurrent programs.
(i) Since, by definition, the transition relation R of any Kripke structure is total, we have that every concurrent program C we consider, is *nonterminating*, in the sense that, in every state there exists a process P_i of C and a guarded command $g \rightarrow c$ of P_i such that: (i.1) g evaluates to *true*, and (i.2) c cannot be abbreviated to *skip*. This assumption restricts the class of concurrent programs we consider.
(ii) Every k-process concurrent program consists of *deterministic* processes, that is, for $i = 1, \ldots, k$, in every state, at most one guard of the guarded commands of process P_i evaluates to *true* (a similar assumption is made in [16]).

Note that the usual assumption that every guarded command is executed *atomically* (in the sense that only one process at a time among the processes of a concurrent program is selected and executed) is taken into account in an implicit way when constructing the transition relation R of the Kripke structure.

Example 2. Given the 2-process symmetric concurrent program C of Example 1, the associated Kripke structure $\langle S, \{s_0\}, R, \lambda\rangle$ is depicted in Figure 1. We depict it as a graph whose nodes are the states in S and whose edges represent the transitions in R. The set S of states includes the four state depicted in Figure 1 and also the states $\langle \mathtt{t}, \mathtt{u}, \mathtt{0}\rangle$, $\langle \mathtt{u}, \mathtt{t}, \mathtt{1}\rangle$, $\langle \mathtt{u}, \mathtt{u}, \mathtt{0}\rangle$, and $\langle \mathtt{u}, \mathtt{u}, \mathtt{1}\rangle$, which have not been depicted because they are not reachable from the initial state $\langle \mathtt{t}, \mathtt{t}, \mathtt{0}\rangle$. Each transition from state u to state v is associated with the guarded command $g \rightarrow c$ whose guard g evaluates to *true* in u. For the labelling function λ, we have that $\lambda(\langle \mathtt{t}, \mathtt{t}, \mathtt{0}\rangle)$ is $\{\mathtt{s}_1 = \mathtt{t}, \mathtt{s}_2 = \mathtt{t}, \mathtt{x} = \mathtt{0}\}$ and, similarly, for the other states. □

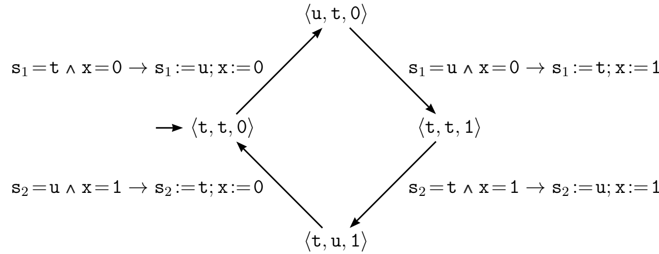


Fig. 1. The transition relation R of the Kripke structure $\mathcal{K} = \langle S, \{s_0\}, R, \lambda\rangle$ associated with the concurrent program C of Example 1. The initial state s_0 is $\langle \mathtt{t}, \mathtt{t}, \mathtt{0}\rangle$. The arcs are labelled by the guarded commands which are responsible for the transition.

Definition 3 (Satisfaction relation for a Concurrent Program). Let C be a k-process concurrent program, $\mathcal{K}$ be the Kripke structure associated with C, s_0 be the initial state of $\mathcal{K}$, and φ be a CTL formula. We say that C *satisfies* φ, denoted $C \models \varphi$, iff $\mathcal{K}, s_0 \models \varphi$. □

Example 3. Let us consider the 2-process concurrent program C defined in Example 1. We associate with the local states $\mathtt{t}$ (short for 'think') and $\mathtt{u}$ (short for 'use') two regions of code, called the noncritical section and the critical section, respectively. We require that the region of code associated with state $\mathtt{u}$ should be executed in a mutually exclusive way. This is formalized by the CTL formula $\varphi =_{def} \mathsf{AG}\neg(\mathtt{s}_1 = \mathtt{u} \wedge \mathtt{s}_2 = \mathtt{u})$, and we have that $C \models \varphi$ holds because for the Kripke structure $\mathcal{K}$ of Example 2 (see Figure 1), we have that $\mathcal{K}, s_0 \models \varphi$ (indeed, there is no path starting from the initial state $s_0 = \langle \mathtt{t}, \mathtt{t}, \mathtt{0} \rangle$ which leads the system to either the state $\langle \mathtt{u}, \mathtt{u}, \mathtt{0} \rangle$ or the state $\langle \mathtt{u}, \mathtt{u}, \mathtt{1} \rangle$). □

Often, in our setting a k-concurrent program consists of *symmetric processes*, the symmetry being determined by the fact that, for any two processes P_i and P_j, for $i \neq j$, we have that P_j can be obtained from P_i by permuting the values of the shared variable $\mathtt{x}$ in the guarded commands. Indeed, as shown in Example 1, the guarded commands in P_2 can be obtained from those in P_1 by interchanging $\mathtt{0}$ and $\mathtt{1}$. In practice, the property of symmetry is very common in many concurrent programs, and our task is precisely the one of automatically synthesizing symmetric processes. This observation motivates a notion of *symmetry* which we now introduce by using cyclic groups. A similar approach has been followed for the automated verification of concurrent systems in [8].

Definition 4 (k-Generating Function). Given an integer $k > 1$, and a finite domain D, we say that $f \in Perm(D)$ is *a k-generating function* iff either $f = id$ or f is a generator of a cyclic group $G_f = \{id, f, f^2, \ldots, f^{k-1}\}$ of order k. □

Let us introduce the following notation. Given a guarded command gc of the form:

$$\mathtt{s}_i = l \wedge \mathtt{x} = d \;\rightarrow\; \mathtt{s}_i := l';\; \mathtt{x} := d';$$

and a k-generating function f, we denote by $f(gc)$ the guarded command:

$$\mathtt{s}_{(i \bmod k)+1} = l \wedge \mathtt{x} = f(d) \;\rightarrow\; \mathtt{s}_{(i \bmod k)+1} := l';\; \mathtt{x} := f(d');$$

Definition 5 (k-Process Symmetric Concurrent Program). Given a k-generating function f, a *k-process symmetric concurrent program* C is a command of the form:

$$C: \quad \mathtt{s}_1 := l_0; \ldots; \mathtt{s}_k := l_0;\; \mathtt{x} := d_0;\; \mathtt{do}\; P_1 \,[\!]\, \ldots \,[\!]\, P_k\; \mathtt{od}$$

where, for all processes P_i, for all guarded commands gc, gc is in P_i iff $f(gc)$ is in $P_{(i \bmod k)+1}$. □

Example 4. Let us consider the 2-process concurrent program C of Example 1. The group $Perm(D)$ of permutations over $D = \{\mathtt{0}, \mathtt{1}\}$ is made out of the following two permutations only: $f_1 = \{\langle \mathtt{0}, \mathtt{0} \rangle, \langle \mathtt{1}, \mathtt{1} \rangle\}$ and $f_2 = \{\langle \mathtt{0}, \mathtt{1} \rangle, \langle \mathtt{1}, \mathtt{0} \rangle\}$. The 2-generating function f_2 shows that the concurrent program C is symmetric.

```
P1 : true → if
       s1=t ∧ x=0 → s1:=u; x:=0;
     [] s1=t ∧ x=1 → skip;
     [] s1=u ∧ x=0 → s1:=t; x:=1;
     fi

P2 : true → if
       s2=t ∧ x=f2(0) → s2:=u; x:=f2(0);
     [] s2=t ∧ x=f2(1) → skip;
     [] s2=u ∧ x=f2(0) → s2:=t; x:=f2(1);
     fi
```

□

By definition, one can generate a k-process symmetric concurrent program C from one of the processes in C by applying the k-generating function f. Moreover, it is often the case that all processes of a given program C also share additional structural properties, besides those determined by f. For instance, in the case of Example 4, we have that both process P_1 and P_2 may move from the local state t to the local state u, or from t to t, or from u to t. These additional structural properties define a *local transition relation* $T \subseteq L \times L$ which together with the k-generating function f, defines a so called *symmetric program structure* $\Sigma = \langle f, T \rangle$. A pair $\langle l, l' \rangle$ in T will also be denoted by $l \mapsto l'$.

Definition 6 (Synthesis Problem of a k-Process Symmetric Concurrent Program). The *synthesis problem of a k-process symmetric concurrent program* C starting from: (i) a CTL formula φ, and (ii) a symmetric program structure $\Sigma = \langle f, T \rangle$, where f is a k-generating function and T is a local transition relation, consists in finding C such that $C \models \varphi$ holds. □

Note that there exists a CTL formula that characterizes the set of initial states. In particular, the initial state $\langle l_1, \ldots, l_k, d_0 \rangle$ can be characterized by the CTL formula $\mathtt{s}_1 = l_1 \wedge \ldots \wedge \mathtt{s}_k = l_k \wedge \mathtt{x} = d_0$, where we assume that each conjunct belongs to *Elem*. However, for reasons of simplicity, we assume that the initial state s_0 is given to our synthesis procedure as an additional input (see clause 1 of the logic program Π_φ of Definition 7).

4 Synthesising Concurrent Programs

In this section we present our synthesis procedure based on ASP. We encode the desired behavioural property φ of our k-process concurrent program to be synthesized as a logic programs Π_φ, and the desired structural property Σ as a logic programs Π_Σ. Programs Π_φ and Π_Σ are defined in the following Definition 7 and 8, respectively.

Definition 7 (Logic program encoding a behavioural property). Let φ be a CTL formula expressing a behavioural property. The logic program Π_φ encoding φ is as follows:

1. $\leftarrow \mathtt{not}\ sat(s_0, \varphi)$
2. $sat(U, F) \leftarrow elem(F, U)$
3. $sat(U, not(F)) \leftarrow \mathtt{not}\ sat(U, F)$
4. $sat(U, and(F_1, F_2)) \leftarrow sat(U, F_1) \wedge sat(U, F_2)$
5. $sat(U, ex(F)) \leftarrow tr(U, V) \wedge sat(V, F)$
6. $sat(U, eu(F_1, F_2)) \leftarrow sat(U, F_2)$
7. $sat(U, eu(F_1, F_2)) \leftarrow sat(U, F_1) \wedge tr(U, V) \wedge sat(V, eu(F_1, F_2))$
8. $sat(U, eg(F)) \leftarrow satpath(U, V, F) \wedge satpath(V, V, F)$
9. $satpath(U, V, F) \leftarrow sat(U, F) \wedge tr(U, V) \wedge sat(V, F)$
10. $satpath(U, Z, F) \leftarrow sat(U, F) \wedge tr(U, V) \wedge satpath(V, Z, F)$

11.1 $tr(s(S_1, \ldots, S_k, X), s(S'_1, \ldots, S'_k, X')) \leftarrow reachable(s(S_1, \ldots, S_k, X)) \wedge gc(1, S_1, X, S'_1, X') \wedge \langle S_1, X \rangle \neq \langle S'_1, X' \rangle$

...

11.k $tr(s(S_1, \ldots, S_k, X), s(S'_1, \ldots, S'_k, X')) \leftarrow reachable(s(S_1, \ldots, S_k, X)) \wedge gc(k, S_k, X, S'_k, X') \wedge \langle S_k, X \rangle \neq \langle S'_k, X' \rangle$

12. $\leftarrow \texttt{not}\ out(S) \land reachable(S)$
13. $out(S) \leftarrow tr(S, Z)$
14. $reachable(s_0) \leftarrow$
15. $reachable(S) \leftarrow tr(Z, S)$

where the predicates are defined as follows: (i) $sat(U, F)$ holds iff the formula F holds in state U, (ii) $elem(b, u)$ holds iff $b \in \lambda(u)$, that is, the elementary proposition b holds in state u, (iii) $satpath(U, V, F)$ holds iff there exists a path from state U to state V such that every state in that path satisfies the formula F, (iv) $tr(s(S_1, \ldots, S_k, X), s(S'_1, \ldots, S'_k, X'))$ holds iff the pair of states $\langle\langle S_1, \ldots, S_k, X\rangle, \langle S'_1, \ldots, S'_k, X'\rangle\rangle$ belongs to the transition relation R of the Kripke structure associated with the program C to be synthesized, and (v) the predicates *out* and *reachable* force the relation R to be total (in particular, $out(S)$ holds iff from state S there is an outgoing edge, and $reachable(S)$ holds iff there is a path from the initial state s_0 to state S.) □

Rule 1 is required for ensuring that φ holds in the initial state s_0 representing the initialization $\texttt{s}_1 := l_0; \ldots; \texttt{s}_k := l_0;\ \texttt{x} := d_0$ of the k-process symmetric concurrent program to be synthesized. Rule 11.i defines the interleaved execution of the guarded commands, that is, for all states U and V, $tr(U, V)$ holds iff U is a reachable state, and there exists a guarded command gc of process P_i whose guard evaluates to *true* in U and whose execution leads from state U to state V.

Definition 8 (Logic program encoding a structural property). Let L be the set of local states and D be the domain of the shared variable. Let $\Sigma = \langle f, T\rangle$ be a symmetric program structure of a k-process symmetric concurrent program. The logic program Π_Σ is defined as follows:

1.1 $\bigvee_{\langle S', X'\rangle \in Next(\langle S_1, X\rangle)} gc(1, S_1, X, S', X') \leftarrow reachable(S_1, S_2, \ldots, S_k, X)$
1.2 $\leftarrow gc(1, S, X, S', X') \land gc(1, S, X, S'', X'') \land \langle S', X'\rangle \neq \langle S'', X''\rangle$
2.1 $gc(2, S, f(X), S', f(X')) \leftarrow gc(1, S, X, S', X')$
2.2 $\leftarrow gc(2, S, X, S', X') \land \texttt{not}\ ps(2, S, X)$
2.3 $ps(2, S_2, X) \leftarrow reachable(S_1, S_2, \ldots, S_k, X)$
...
k.1 $gc(k, S, f(X), S', f(X')) \leftarrow gc(k-1, S, X, S', X')$
k.2 $\leftarrow gc(k, S, X, S', X') \land \texttt{not}\ ps(k, S, X)$
k.3 $ps(k, S_k, X) \leftarrow reachable(S_1, S_2, \ldots, S_k, X)$

where: (i) $gc(i, S, X, S', X')$ holds iff $\texttt{s}_\texttt{i} = l \land \texttt{x} = d \to \texttt{s}_\texttt{i} := l';\ \texttt{x} := d'$ is in P_i, (ii) f is a k-generating function, (iii) $ps(i, S, X)$ holds iff there exists a reachable state of the form $\langle S_1, \ldots, S_{i-1}, S, S_{i+1}, \ldots, S_k, X\rangle$, and (iv) for all $l \in L$, $d \in D$, $Next(l, d) = \{\langle l', d'\rangle \mid l \mapsto l' \in T \land d' \in D\}$. □

Rules 1.1 and 1.2 generate a set of guarded commands for process P_1. The disjunction in the head of Rule 1.1 is over all possible guarded commands that P_1 may execute. The set of those guarded commands is defined using the sets $Next(l, d)$, one for each $l \in L$ and $d \in D$. The integrity constraint 1.2 enforces the generation of a set of guarded commands in which any two guards of the

guarded commands in P_1 are mutually exclusive (recall that we consider only deterministic processes). For $j=2,\ldots,k$, Rules $j.1$, $j.2$ and $j.3$ realize Definition 5. We use Rule $j.1$ to derive a guarded command in P_j from a guarded command of the process P_{j-1}. Rule $j.2$ ensures that for every guarded command $g \rightarrow c$ derived by $j.1$, there exists a reachable state U such that in U the guard g evaluates to *true*.

Now we present a theorem establishing the correctness of our synthesis procedure. It relates the k-process symmetric concurrent programs satisfying φ with the answer sets of the logic program $\Pi_\varphi \cup \Pi_\Sigma$. Obviously, the correctness of the synthesis procedure implies also the correctness of the programs Π_φ and Π_Σ encoding the behavioural properties and the structural properties, as specified in Definition 7 and 8, respectively.

Theorem 1 (Correctness of Synthesis). *Let* $\Pi = \Pi_\varphi \cup \Pi_\Sigma$ *be the logic program obtained, as specified by Definitions* 7 *and* 8*, from:* (i) *a* CTL *formula* φ *and* (ii) *a symmetric program structure* $\Sigma = \langle f, T \rangle$*. Then,*

$$\big(\mathtt{s_1}\!:=l_0;\ldots;\mathtt{s_k}\!:=l_0;\ \mathtt{x}\!:=d_0;\ \mathtt{do}\ P_1 \,[\!]\, \ldots \,[\!]\, P_k\ \mathtt{od}\big) \ \models\ \varphi$$

iff there exists an answer set M *in* $ans(\Pi)$ *such that*

$\forall i \in \{1,\ldots,k\}$, $\forall l, l' \in L$, $\forall d, d' \in D$,

$\big(\mathtt{s}_i = l \wedge \mathtt{x} = d \rightarrow \mathtt{s}_i := l'; \mathtt{x} := d'\big)$ *is in* P_i *iff* $M \models \mathtt{gc}(i, l, d, l', d')$.

5 Experimental Results

In this section we present some experimental results obtained by applying our synthesis procedure to mutual exclusion protocols. All experiments have been performed on an Intel Core 2 Duo E7300 2.66GHz under the Linux operating system.

The first synthesis we did is the one of a simple program, called 2-*mutex*-1, for two processes enjoying the mutual exclusion property only, and then we progressively increased the number of properties that the synthesized program should satisfy (see Table 1). In that table the program k-*mutex*-p denotes a synthesized program for k processes satisfying p behavioural properties. For instance, program 2-*mutex*-4 is the synthesized program that works for 2 processes and enjoys the four behavioural properties: (i) *ME* (mutual exclusion), (ii) *SF* (starvation freedom), (iii) *BO* (bounded overtaking), and (iv) *MR* (maximal reactivity), defined by CTL formulas as follows.

(i) *Mutual Exclusion*, that is, it is not the case that process P_i is in its critical section ($\mathtt{s}_i = \mathtt{u}$), and process P_j is in its critical section ($\mathtt{s}_j = \mathtt{u}$) at the same time: for all i, j in $\{1,\ldots,k\}$, with $i \neq j$,

$$\mathsf{AG}\,\neg(\mathtt{s}_i = \mathtt{u} \wedge \mathtt{s}_j = \mathtt{u}) \qquad (ME)$$

(ii) *Starvation Freedom*, that is, if a process is waiting to enter the critical section ($\mathtt{s}_i = \mathtt{w}$), then after a finite amount of time, process P_i will execute its critical section ($\mathtt{s}_i = \mathtt{u}$): for all i in $\{1,\ldots,k\}$,

$$\mathsf{AG}\,(\mathtt{s}_i = \mathtt{w} \rightarrow \mathsf{AF}\,\mathtt{s}_i = \mathtt{u}) \qquad (SF)$$

(iii) *Bounded Overtaking*, that is, while process P_i is in its waiting section,any other process P_j exits from its critical section at most once: for all i, j in $\{1, \ldots, k\}$,

$$\mathsf{AG}\,((\mathtt{s}_i = \mathtt{w} \wedge \mathtt{s}_j = \mathtt{u}) \rightarrow \mathsf{AF}\,(\mathtt{s}_j = \mathtt{t} \wedge \mathsf{A}[\neg(\mathtt{s}_j = \mathtt{u})\,\mathsf{U}\,\mathtt{s}_i = \mathtt{u}])) \quad (BO)$$

(iv) *Maximal Reactivity*, that is, if process P_i is waiting to execute the critical section and all other processes are executing their noncritical sections, then in the next state P_i will enter its critical section: for all i in $\{1, \ldots, k\}$,

$$\mathsf{AG}\,((\mathtt{s}_i = \mathtt{w} \wedge \textstyle\bigwedge_{j \in \{1,\ldots,k\} \setminus \{i\}} \mathtt{s}_j = \mathtt{t}) \rightarrow \mathsf{EX}\,\mathtt{s}_i = \mathtt{u}) \quad (MR)$$

Table 1. Column named Program gives the names of the synthesized programs. *k-mutex-p* denotes the mutual exclusion program for k processes and p behavioural properties that are indicated in the column named Satisfied Properties. Column named $|D|$ gives the cardinality of the domain of the shared variable $\mathtt{x}$. Column named f gives the k-generating functions. Column named $|ans(\Pi)|$ gives the cardinality of $ans(\Pi)$, that is, the number of answer sets of program $\Pi = \Pi_\varphi \cup \Pi_\Sigma$. In column named Time we indicate the times (in seconds) taken for the synthesis using the smodels [19].

Program	Satisfied Properties	$\|D\|$	f	$\|ans(\Pi)\|$	Time
2-*mutex*-1	*ME*	2	*id*	6	0.07
2-*mutex*-1	*ME*	2	f_1	7	0.70
2-*mutex*-2	*ME*, *SF*	2	f_1	3	0.71
2-*mutex*-3	*ME*, *SF*, *BO*	2	f_1	3	1.44
2-*mutex*-4	*ME*, *SF*, *BO*, *MR*	3	f_2	2	11.7
3-*mutex*-1	*ME*	2	*id*	5	0.95
3-*mutex*-1	*ME*	2	f_1	10	0.87
3-*mutex*-2	*ME*, *SF*	3	f_3	8	152
3-*mutex*-3	*ME*, *SF*, *BO*	3	f_3	8	1700

In our synthesis experiments we have made the following choices for s_0, L, D, f, and T.

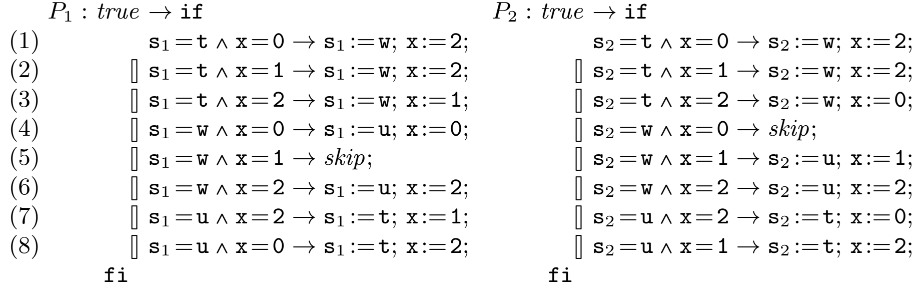
The initial state s_0 is $\langle \mathtt{t}, \mathtt{t}, 0 \rangle$ and $\langle \mathtt{t}, \mathtt{t}, \mathtt{t}, 0 \rangle$ for the 2- and 3-process symmetric concurrent programs, respectively.

The set L of the local states for the variables $\mathtt{s}_i$'s is $\{\mathtt{t}, \mathtt{w}, \mathtt{u}\}$, where $\mathtt{t}$ represents the *noncritical section*, $\mathtt{w}$ represents the *waiting section*, and $\mathtt{u}$ represents the *critical section*.

The domain D of the shared variable $\mathtt{x}$ is a finite set of natural numbers whose cardinality $|D|$ depends on: (i) the number k of the processes to be synthesized, and (ii) the properties that the concurrent program should satisfy. The value of $|D|$ is not known a priori, and we guess it at the beginning of our synthesis task. If the synthesis fails, we increase the value of $|D|$, hoping for a successful synthesis with a larger value of $|D|$.

The k-generating function f is chosen among the following ones: (i) *id* is the identity function, (ii) $f_1 = \{\langle 0, 1 \rangle, \langle 1, 0 \rangle\}$, (iii) $f_2 = \{\langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 2, 2 \rangle\}$, and (iv) $f_3 = \{\langle 0, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 0 \rangle\}$.

The local transition relation T is $\{\mathtt{t} \mapsto \mathtt{w},\ \mathtt{w} \mapsto \mathtt{w},\ \mathtt{w} \mapsto \mathtt{u},\ \mathtt{u} \mapsto \mathtt{t}\}$. The pair $\mathtt{t} \mapsto \mathtt{w}$ denotes that, once the noncritical section has been executed, a process enters the waiting section. The pairs $\mathtt{w} \mapsto \mathtt{w}$ and $\mathtt{w} \mapsto \mathtt{u}$ denote that a process may repeat



```
    P1 : true → if                                P2 : true → if
(1)       s1 = t ∧ x = 0 → s1 := w; x := 2;           s2 = t ∧ x = 0 → s2 := w; x := 2;
(2)     [] s1 = t ∧ x = 1 → s1 := w; x := 2;        [] s2 = t ∧ x = 1 → s2 := w; x := 2;
(3)     [] s1 = t ∧ x = 2 → s1 := w; x := 1;        [] s2 = t ∧ x = 2 → s2 := w; x := 0;
(4)     [] s1 = w ∧ x = 0 → s1 := u; x := 0;        [] s2 = w ∧ x = 0 → skip;
(5)     [] s1 = w ∧ x = 1 → skip;                   [] s2 = w ∧ x = 1 → s2 := u; x := 1;
(6)     [] s1 = w ∧ x = 2 → s1 := u; x := 2;        [] s2 = w ∧ x = 2 → s2 := u; x := 2;
(7)     [] s1 = u ∧ x = 2 → s1 := t; x := 1;        [] s2 = u ∧ x = 2 → s2 := t; x := 0;
(8)     [] s1 = u ∧ x = 0 → s1 := t; x := 2;        [] s2 = u ∧ x = 1 → s2 := t; x := 2;
        fi                                          fi
```

Fig. 2. The two synthesized processes P_1 and P_2 of the program 2-*mutex*-4: $\mathtt{s}_1 := \mathtt{t}$; $\mathtt{s}_2 := \mathtt{t}$; $\mathtt{x} := 0$; do P_1 [] P_2 od. It enjoys the following properties: ME, SF, BO, and MR.

(possibly an unbounded number of times) the execution of its waiting section and then may enter its critical section. The pair $\mathtt{u} \mapsto \mathtt{t}$ denotes that, once the critical section has been executed, a process enters its noncritical section.

In Figures 2 and 3 we present the syntax and the semantics of the synthesized program, called 2-*mutex*-4, for the 2-process mutual exclusion problem described in Example 3. (Program 2-*mutex*-4 is essentially the same as the Peterson algorithm [17], but it uses a single shared variable.)

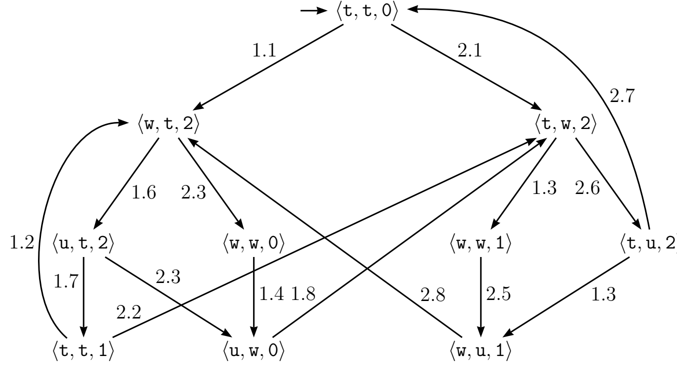


Fig. 3. The transition relation of the Kripke structure associated with the 2-process concurrent program 2-*mutex*-4. The initial state is $\langle \mathtt{t}, \mathtt{t}, 0 \rangle$. For $i = 1, 2$, an arc labelled $i.n$ indicates that the guarded command n of process P_i is responsible for that transition.

6 Related Work and Concluding Remarks

Two well known, early works on synthesis of concurrent programs were those by Clark and Emerson [6] and Manna and Wolper [15]. In [6] Clark and Emerson introduce the notion of a synchronization skeleton as an abstraction of the actual processes in concurrent programs. They synthesize concurrent programs for a shared-memory model of execution by extracting the synchronization skeletons from the models of temporal logic specifications. In particular they introduce a three-phase synthesis procedure: Phase (1) provide the CTL specification of the concurrent program; Phase (2) apply the tableau-based decision procedure for

the satisfiability of CTL formulas to obtain a model of the CTL specification; Phase (3) extract the synchronization skeletons from the model of the CTL specification. Similarly to [6] in [15] Manna and Wolper present a method for synthesizing synchronization instructions for processes in a message-passing model of execution from a Propositional Temporal Logic (PTL) using a tableau-based decision procedure for the satisfiability of PTL formulas. The instructions synthesized by their method are written as Communicating Sequential Processes [13]. In [18] Piterman, Pnueli, and Sa'ar consider the problem of the design of digital circuits from Linear Temporal Logic (LTL) specifications and give an $O(N^3)$ algorithm to construct an automaton satisfying a formula of a particular class of LTL specifications. We closely follow the approaches of [6] and [15]. In particular we synthesize concurrent processes that communicate with each other by means of shared variables starting from CTL specifications. The programs we synthesize are written as guarded commands [7].

In order to reduce the search space of our synthesis problem, we have used a notion of symmetric concurrent programs which is similar to the one which was introduced in [1,8] to overcome the state explosion problem. Our notion of symmetry is formalized using group theory, similarly to what has been done in [8] for model checking.

Similarly to Attie and Emerson [2], we also propose a method for the synthesis task and we separate the behavioural properties from the structural properties. However, in our approach the structural properties, such as symmetry, are represented in the symmetric program structures, rather than an automata based formalism.

We have implemented our synthesis method in Answer Set Programming (ASP). One advantage of our method over [1,6,15] is its generality: besides temporal properties, we can specify structural properties, such as the above mentioned symmetry, and our ASP program will automatically synthesize concurrent programs satisfying the desired properties without the need for ad hoc algorithms. To the best of our knowledge, there is only one paper by Heymans, Nieuwenborgh and Vermeir [12] who use Answer Set Programming for the synthesis of concurrent programs. They have extended the ASP paradigm by adding preferences among models and they have developed an answer set system, called OLPS. Using OLPS they perform the synthesis of concurrent programs following the approach proposed in [6]. They use (preferred) ASP only for the Phase (2) of the synthesis procedure introduced in [6]. The synchronization skeletons can be read from the model of Phase (2) as defined in [6]. We do not require any extension of the ASP paradigm, we use the by now standard ASP systems, such as DLV [9] and smodels [19], and every phase of our synthesis procedure is fully automated by using an ASP program. In particular we use a two-phase synthesis procedure: (i) provide the specification of the concurrent program to be synthesized by giving the desired behavioural (by using CTL) and structural properties, and (ii) use the ASP program to synthesize the concurrent program satisfying the given specification.

In practice our approach works for synthesizing k-Process Concurrent Program with a limited number k of processes. As future work we plan to explore

various techniques for reducing the search space of the synthesis procedure and, thus, we hope to synthesize protocols for a larger number of processes and more complex properties to be guaranteed. Among these techniques we envisage to apply those used in compositional model checking [4].

References

1. P. C. Attie and E. A. Emerson. Synthesis of Concurrent Programs with Many Similar Processes *ACM TOPLAS*, 51–115, 1998.
2. P. C. Attie and E. A. Emerson. Synthesis of Concurrent Programs for an Atomic Read/Write Model of Computation. *ACM TOPLAS*, 187–242, 2001.
3. F. Calimeri, S. Cozza, G. Ianni and N. Leone. Enhancing ASP by Functions: Decidable Classes and Implementation Techniques. *Proceedings of the* 24*-th AAAI Conference on Artificial Intelligence 2010*, 1666–1670, 2010.
4. E. M. Clarke Jr., D. E. Long, and K. L. McMillan. Compositional model checking. *Logic in Computer Science, LICS '89, Proceedings*, IEEE Computer Society, 353–362, 1989.
5. E. M. Clarke Jr., O. Grumber and D. A. Peled. *Model Checking*. The MIT Press, 1999.
6. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. *Workshop on Logic of Programs*, London, UK, Springer-Verlag, 52–71, 1982.
7. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
8. E. A. Emerson and A. P. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*: 9, 1–2, 105–131, 1996.
9. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri and F. Scarcello. The DLV system for knowledge representation and reasoning *ACM TOCL*: 7, 499–562, 2006.
10. M. Gelfond and V. Lifschitz. The Stable Model Semantics For Logic Programming. *Proc. of the Fifth Intern. Conf. and Symp. on Logic Programming*, Seattle, MIT Press, 1070–1080, 1988.
11. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*: 9, 365–385, 1991.
12. S. Heymans, D. Van Nieuwenborgh and D. Vermeir. Synthesis from Temporal Specifications using Preferred Answer Set Programming. *LNCS* no. 3701, Springer, 280–294, 2005.
13. C.A.R. Hoare. Communicating Sequential Processes. Prentice-Hall, 1985.
14. V. Lifschitz. Answer Set Programming and Plan Generation. *Artificial Intelligence* no. 138, 39-54, 2002.
15. Z. Manna and P. Wolper: Synthesis of Communicating Processes from Temporal Logic Specifications. *ACM TOPLAS*, 68–93, 1984.
16. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: Specification*. Springer-Verlag, 1991.
17. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
18. N. Piterman, A. Pnueli and Y. Sa'ar. Synthesis of Reactive(1) Designs. *LNCS* no. 3855, Springer, 364–380, 2006.
19. T. Syrjänen and I. Niemelä. The Smodels System. *LNCS* no. 2173, Springer, 434–438, 2001.