Synthesizing Concurrent Programs using Answer Set Programming

Emanuele De Angelis\textsuperscript{1}, Alberto Pettorossi\textsuperscript{2}, and Maurizio Proietti\textsuperscript{3}

\textsuperscript{1} Dipartimento di Scienze, University ‘G. D’Annunzio’, Pescara, Italy
dean@sci.unich.it
\textsuperscript{2} DISP, University of Rome Tor Vergata, Italy
pettorossi@disp.uniroma2.it
\textsuperscript{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 [16], 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.
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 $\varphi$ is encoded as a set $\Pi_\varphi$ 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 $\Pi_\Sigma$ 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 $\Pi$ represents a concurrent program satisfying the given specification. Thus, by using an Answer Set Programming (ASP) system, such as DLV [9] or smodels [20], 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 $\text{Var}$, each ranging over a finite domain $D_v$, and (ii) guards, $g$ in $\text{Guard}$, of the form: $g := \text{true} | \text{false} | v = d | \neg g | g_1 \land g_2$, with $v \in \text{Var}$ and $d \in D_v$. We also have the following derived sets whose definitions are mutually recursive: (iii) commands, $c$ in $\text{Command}$, of the form:
c ::= skip | v := d | c₁ ; c₂ | if gc f₁ | do gc od, where ‘;’ denotes the sequential composition of commands, and (iv) guarded commands, gc in GCommand, of the form: gc ::= g → c | gc₁ || gc₂, where ‘||’ denotes the parallel composition of guarded commands.

The execution of if gc₁ . . . || gcₙ fi is performed as follows: one of the guarded commands g → c in {gc₁, . . . , gcₙ} whose guard g evaluates to true is chosen, then c is executed; otherwise, if no guard in {gc₁, . . . , gcₙ} evaluates to true then the whole command if . . . fi terminates with failure.

The execution of do gc₁ . . . || gcₙ od is performed as follows: one of the guarded commands g → c in {gc₁, . . . , gcₙ} 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 {gc₁, . . . , gcₙ} evaluates to true then the execution proceeds with the next command.

Symmetric Groups. A group G is a pair ⟨S, o⟩, where S is given a set and o is a binary operation on S satisfying the following axioms: (i) ∀x, y ∈ S. x o y ∈ S (closure), (ii) ∀x, y, z ∈ S. (x o y) o z = x o (y o z) (associativity), (iii) ∃e ∈ S. ∀x ∈ S. e o x = x o e = x (identity element), and (iv) ∀x ∈ S. ∃y ∈ S. x o y = y o x = e (inverse element). The order of a group G is the cardinality of S. For any x ∈ S, for any n ≥ 0, we write xⁿ to denote the term x o . . . o x with n occurrences of x. We stipulate that x⁰ is e.

A group G is said to be cyclic if there exists an element x ∈ S, called a generator, such that S = {xⁿ | n ≥ 0}. We write Gₓ 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 o 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 pⁿ = 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:

ϕ ::= b | ϕ₁ ∧ ϕ₂ | ¬ϕ | EXϕ | EGϕ | E[ϕ₁ U ϕ₂]

Let us introduce the following abbreviations: (i) ϕ₁ ∨ ϕ₂ for ¬(¬ϕ₁ ∧ ¬ϕ₂), (ii) EFϕ for E[true U f] (iii) AGϕ for ¬EF¬ϕ, (iv) AFϕ for ¬EG¬ϕ, (v) A[ϕ₁ U ϕ₂] for ¬E[ϕ₁ U (¬ϕ₁ ∧ ¬ϕ₂)] ∧ ¬EG¬ϕ₂, (vi) AXϕ for ¬EX¬ϕ, (vii) A[ϕ₁ R ϕ₂] for ¬E[ϕ₁ U ¬ϕ₂], and (viii) E[ϕ₁ R ϕ₂] for ¬A[¬ϕ₁ U ¬ϕ₂].

We define the semantics of CTL by giving a Kripke structure K = ⟨S, S₀, λ, R⟩, where: (i) S is a finite set of states, (ii) S₀ ⊆ S is a set of initial states, (iii) R ⊆ S × S is a total transition relation (thus, ∀u ∈ S, ∃v ∈ S, (u, v) ∈ R), and (iv) λ : S → P(Elem) is a total, labelling function that assigns to every state s ∈ S a subset λ(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 K from a state is an infinite sequence ⟨s₀, s₁, . . .⟩ of states such that, for all i ≥ 0, ⟨sᵢ, sᵢ₊₁⟩ ∈ R. For i ≥ 0, we denote by πᵢ the i-th element
of \( \pi \). The fact that a CTL formula \( \varphi \) holds in a state \( s \) of a Kripke structure \( K \) will be denoted by \( K, s \models \varphi \). For any CTL formula \( \varphi \) and state \( s \), we define the relation \( K, s \models \varphi \) as follows:

\[
K, s \models b \quad \text{iff} \quad b \in \lambda(s) \\
K, s \models \neg \varphi \quad \text{iff} \quad K, s \not\models \varphi \text{ does not hold} \\
K, s \models \varphi_1 \land \varphi_2 \quad \text{iff} \quad K, s \models \varphi_1 \text{ and } K, s \models \varphi_2 \\
K, s \models \text{EX} \varphi \quad \text{iff} \quad \text{there exists } (s, t) \in R \text{ such that } K, t \models \varphi \\
K, s \models \text{E}[\varphi_1 \cup \varphi_2] \quad \text{iff} \quad \text{there exists a path } \pi = (s, s_1, \ldots) \text{ in } K \text{ and } i \geq 0 \text{ such that } K, \pi_i \models \varphi_2 \text{ and for all } 0 \leq j < i, K, \pi_j \models \varphi_1 \\
K, s \models \text{EG} \varphi \quad \text{iff} \quad \text{there exists a path } \pi \text{ such that } \\
\pi_0 = s \text{ and for all } i \geq 0, K, \pi_i \models \varphi
\]

### 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 [3,10,14,15]. A rule \( r \) is an implication of the form:

\[
a_1 \lor \ldots \lor a_k \leftarrow a_{k+1} \land \ldots \land a_m \land \text{not} a_{m+1} \land \ldots \land \text{not} a_n
\]

where \( a_1, \ldots, a_k, \ldots, a_n \) (for \( k \geq 0, n \geq k \)) are atoms and ‘\( \text{not} \)’ denotes negation as failure [11]. Given a rule \( r \), we define the following sets: \( \text{head}(r) = \{a_1, \ldots, a_k\} \), \( \text{pos}(r) = \{a_{k+1}, \ldots, a_m\} \), and \( \text{neg}(r) = \{a_{m+1}, \ldots, a_n\} \). An integrity constraint is a rule \( r \) such that \( \text{head}(r) = \emptyset \). A logic program is a set of rules. When we write a rule with variables, we actually mean all the ground instances of that rule.

An interpretation \( I \) of a program \( \Pi \) is a subset of the Herbrand base. The Gelfond-Lifschitz transformation of a program \( \Pi \) with respect to an interpretation \( I \) is the program \( \Pi^I = \{\text{head}(r) \leftarrow \text{pos}(r) \mid r \in \Pi \land \text{neg}(r) \cap I = \emptyset\} \). An interpretation \( M \) is said to be an answer set of \( \Pi \) if \( M \) is the least Herbrand model of \( \Pi^M \). The answer set semantics of \( \Pi \) assigns to \( \Pi \) a set of answer sets, denoted \( \text{ans}(\Pi) \). Given an answer set \( M \in \text{ans}(\Pi) \) 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 \( s_i \), called the local state, ranging over a finite domain \( L \), which is the same for all processes. The variable \( s_i \) can be tested and modified by \( P_i \) only. All processes may test and modify also a shared variable \( x \), 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 \( x \). Here is the formal definition of a concurrent program.

---

4
Definition 1 (k-Process Concurrent Program). Let \( L \) be a set of local states and \( D \) be a domain of the shared variable \( x \). For any \( k > 1 \), a \( k \)-process concurrent program \( C \) is a command of the form:
\[
C : \quad s_1 := l_1; \ldots; s_k := l_k; \quad x := d_0; \quad \text{do } P_1 \[ \ldots \] P_k \text{ od}
\]
where \( s_1, \ldots, s_k, x \in \text{Var}, l_1, \ldots, l_k \in L, \) 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 \text{true} \rightarrow \text{if } gc_1 \[ \ldots \] gc_n \text{ fi}
\]
Every guarded command \( gc \) in \( gc_1 \[ \ldots \] gc_n \) is of the form:
\[
\quad gc : \quad s_i = l \land x = d \rightarrow s_i := l'; x := d';
\]
where \( l, l' \in L \) and \( d, d' \in D \).

We shall use the guarded command \( s_i = l \land x = d \rightarrow \text{skip} \) as a shorthand for \( s_i = l \land x = d \rightarrow s_i := l; x := d \). The command \( s_1 := l_1; \ldots; s_k := l_k; \quad x := d_0 \) is called \textit{initialization} of \( C \).

Example 1. Let \( L \) be the set \( \{ t, u \} \) and \( D \) be the set \( \{ 0, 1 \} \). A 2-process concurrent program \( C \) is:
\[
s_1 := t; s_2 := t; x := 0; \quad \text{do } P_1 \[ \ldots \] P_2 \text{ od}
\]
where \( P_1 \) and \( P_2 \) are defined as follows:
\[
P_1 : \quad \text{true} \rightarrow \text{if}
\]
\[
\begin{align*}
& s_1 = t \land x = 0 \rightarrow s_1 := u; x := 0; \\
& s_1 = t \land x = 1 \rightarrow \text{skip}; \\
& s_1 = u \land x = 0 \rightarrow s_1 := t; x := 1;
\end{align*}
\]
\[
P_2 : \quad \text{true} \rightarrow \text{if}
\]
\[
\begin{align*}
& s_2 = t \land x = 1 \rightarrow s_2 := u; x := 1; \\
& s_2 = t \land x = 0 \rightarrow \text{skip}; \\
& s_2 = u \land x = 1 \rightarrow s_2 := t; x := 0;
\end{align*}
\]
This program is the familiar program for two processes, each of which either ‘thinks’ in its noncritical section \((s_i = t)\) or ‘uses a resource’ in its critical section \((s_i = u)\). The shared variable \( x \) gives each process its turn to enter the critical section: if \( x = 0 \), process \( P_1 \) is in its critical section, and if \( x = 1 \), process \( P_2 \) is in its critical section.

Now we introduce the semantics of concurrent programs by using Kripke structures. We model a state \( u \) of a \( k \)-process concurrent program \( C \) by a \((k + 1)\)-tuple \((l_1, \ldots, l_k, d)\), where: (i) the first \( k \) components are the values of the local state variables \( s_1, \ldots, s_k \), and (ii) \( d \) is the value of the shared variable \( x \).

Definition 2 (Kripke Structure Associated with a k-Process Concurrent Program). Let \( C \) be a \( k \)-process concurrent program of the form
\[
C : \quad s_1 := l_1; \ldots; s_k := l_k; \quad x := d_0; \quad \text{do } P_1 \[ \ldots \] P_k \text{ od}
\]
where the \( l_i \)'s belong to \( L \) and \( d_0 \) belongs to \( D \). The \textit{Kripke structure} \( K \) associated with \( C \) is the 4-tuple \((S, S_0, R, \lambda)\), where:
(i) the set \( S \) of states is \( L^k \times D \),
(ii) the set \( S_0 \) of initial states is the singleton \( \{ (l_1, \ldots, l_k, d_0) \} \),

5
transitions in
R
transition from state
⟨
and also the states
it as a graph whose nodes are the states in
S
atomically
(in the sense that only one process at a time among the processes o f a
k
(ii) Every
depicted because they are not reachable from the initial state
to
skip
is, for
i
(iv) for all states
of process
P
λ
evaluated
in

Definition 3 (Satisfaction relation for a Concurrent Program).

We make the following assumptions about
k-process concurrent programs.
i) Since, by definition, the transition relation
R
of every 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
1
of
C
and a guarded command
g→c
of
P
1
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 [17]).

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. We depict
⟨
s_0, \ldots, s_k\rangle
, the value \( \lambda(t) \) is defined to be
\{s_1=l_1, \ldots, s_k=l_k, x=d\}.
The set \( \text{Elem} \) of the elementary propositions is the set \( \bigcup_{t \in S} \lambda(t) \).

Having defined the Kripke structure associated with a given program, now we can define the notion of a program satisfying a given behavioural property.

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 \( \varphi \) be a CTL formula. We say that
C
satisfies \( \varphi \), denoted \( C \models \varphi \), if \( \mathcal{K}, s_0 \models \varphi \).

Example 3. Let us consider the 2-process concurrent program
C
defined in Example 1. We associate with the local states \( t \) (short for ‘think’) and \( u \) (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 \( u \) should be executed in a mutually exclusive way. This is formalized by the CTL formula
Fig. 1. The transition relation $R$ of the Kripke structure $\mathcal{K} = (S, \{s_0\}, R, \lambda)$ associated with the concurrent program $C$ of Example 1. The initial state $s_0$ is $\langle t, t, 0 \rangle$. The arcs are labelled by the guarded commands which are responsible for the transition.

\[ \varphi \overset{\text{def}}{=} \text{AG}[\neg(s_1 = u \land s_2 = u)] \], 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 $s_0 = \langle t, t, 0 \rangle$ which leads the system to either the state $\langle u, u, 0 \rangle$ or the state $\langle u, u, 1 \rangle$).

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 $x$ 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 0 and 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 $f \in \text{Perm}(D)$ 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:

\[ s_i = l \land x = d \rightarrow s_i := l'; \ x := d'; \]

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

\[ s_{(i \mod k)+1} = l \land x = f(d) \rightarrow s_{(i \mod k)+1} := l'; \ x := f(d'); \]

**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:

\[ C : s_1 := l_0; \ldots; s_k := l_0; \ x := d_0; \text{ do } P_1 \parallel \ldots \parallel P_k \text{ od} \]

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

7
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 $\varphi$, 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 $s_1 = l_1 \land \ldots \land s_k = l_k \land x = d_0$, where we assume that each conjunct belongs to $\text{Elem}$. 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 $\Pi_\varphi$ of Definition 7).

4 Synthesising Concurrent Programs

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

Definition 7 (Logic program encoding a behavioural property). Let $\varphi$ be a CTL formula expressing a behavioural property. The logic program $\Pi_\varphi$ encoding $\varphi$ is as follows:

1. $\leftarrow \text{not sat}(s_0, \varphi)$
2. \( sat(U, F) \leftarrow elem(F, U) \)
3. \( sat(U, \text{not}(F)) \leftarrow \text{not} \ sat(U, F) \)
4. \( sat(U, \text{and}(F_1, F_2)) \leftarrow sat(U, F_1) \land sat(U, F_2) \)
5. \( sat(U, \text{ex}(F)) \leftarrow tr(U, V) \land sat(V, F) \)
6. \( sat(U, \text{ev}(F_1, F_2)) \leftarrow sat(U, F_2) \)
7. \( sat(U, \text{ev}(F_1, F_2)) \leftarrow sat(U, F_1) \land tr(U, V) \land sat(V, \text{ev}(F_1, F_2)) \)
8. \( sat(U, \text{eg}(F)) \leftarrow \text{satpath}(U, V, F) \land \text{satpath}(V, V, F) \)
9. \( \text{satpath}(U, V, F) \leftarrow sat(U, F) \land tr(U, V) \land sat(V, F) \)
10. \( \text{satpath}(U, Z, F) \leftarrow sat(U, F) \land tr(U, V) \land \text{satpath}(V, Z, F) \)
11. \( tr(s(S_1, \ldots, S_k, X), s(S'_1, \ldots, S'_{k'}', X')) \leftrightarrow \text{reachable}(s(S_1, \ldots, S_k, X)) \land gc(1, S_1, X, S'_1, X') \land \langle S_1, X \rangle \neq \langle S'_1, X' \rangle \)
12. \( \leftarrow \text{not} \ \text{out}(S) \land \text{reachable}(S) \)
13. \( \text{out}(S) \leftarrow tr(S, Z) \)
14. \( \text{reachable}(s_0) \leftarrow \)
15. \( \text{reachable}(S) \leftarrow tr(Z, S) \)

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) \( \text{satpath}(U, V, F) \) 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 \( \langle S_1, \ldots, S_k, X \rangle, \langle S'_1, \ldots, S'_{k'}, X' \rangle \) belongs to the transition relation \( R \) of the Kripke structure associated with the program \( C \) to be synthesized, and (v) the predicates \( \text{out} \) and \( \text{reachable} \) force the relation \( R \) to be total (in particular, \( \text{out}(S) \) holds iff from state \( S \) there is an outgoing edge, and \( \text{reachable}(S) \) holds iff there is a path from the initial state \( s_0 \) to state \( S \)).

Rule 1 is required for ensuring that \( \varphi \) holds in the initial state \( s_0 \) representing the initialization \( s_1 := \ell_0, \ldots; s_k := \ell_0; x := d_0 \) 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 \( F_i \) whose guard evaluates to \( \text{true} \) 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 \( \Sigma = (f, T) \) be a symmetric program structure of a \( k \)-process symmetric concurrent program. The logic program \( \Pi_\Sigma \) is defined as follows:

\[
1.1 \ \bigvee_{\langle S', X' \rangle \in \text{Next}(\langle s_1, X \rangle)} \text{gc}(1, S_1, X, S', X') \leftarrow \text{reachable}(S_1, S_2, \ldots, S_k, X) \\
1.2 \ \leftarrow \text{gc}(1, S, X, S', X') \land \text{gc}(1, S, X, S'', X''') \land \langle S', X' \rangle \neq \langle S'', X''' \rangle \\
2.1 \ \text{gc}(2, S, f(X), S', f(X')) \leftarrow \text{gc}(1, S, X, S', X') \\
2.2 \ \leftarrow \text{gc}(2, S, X, S', X') \land \text{not} \ ps(2, S, X)
\]
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 ($s_i = u$), and process $P_j$ is in its critical section ($s_j = u$) at the same time: for all $i, j$ in $\{1, \ldots, k\}$, with $i \neq j$,

$$\text{AG} \neg (s_i = u \land s_j = u)$$  \hfill (ME)

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

$$\text{AG} (s_i = w \rightarrow \text{AF} s_i = u)$$  \hfill (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 $\{1, \ldots, k\}$,

$$\text{AG} ((s_i = w \land s_j = u) \rightarrow \text{AF} (s_j = t \land \text{A} \neg (s_j = u) \cup s_i = u))$$  \hfill (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 $\{1, \ldots, k\}$,

$$\text{AG} ((s_i = w \land \bigwedge_{j \in \{1, \ldots, k\}\backslash\{i\}} s_j = t) \rightarrow \text{EX} s_i = u)$$  \hfill (MR)

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

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

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

The domain $D$ of the shared variable $x$ 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) $f_1 = \{(0, 1), (1, 0)\}$, (iii) $f_2 = \{(0, 1), (1, 0), (2, 2)\}$, and (iv) $f_3 = \{(0, 1), (1, 2), (2, 0)\}$.

The local transition relation $T$ is $\{t \mapsto w, w \mapsto w, w \mapsto u, u \mapsto t\}$. The pair $t \mapsto w$ denotes that, once the noncritical section has been executed, a process enters the waiting section. The pairs $w \mapsto w$ and $w \mapsto u$ denote that a process may repeat (possibly an unbounded number of times) the execution of its waiting section and then may enter its critical section. The pair $u \mapsto t$ denotes that, once the critical section has been executed, a process enters its noncritical section.
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. ME, SF, BO and MR stand for ‘mutual exclusion’, ‘starvation freedom’, ‘bounded overtaking’, and ‘maximal reactivity’, respectively. Column named $|D|$ gives the cardinality of the domain of the shared variable $x$. Column named $f$ gives the $k$-generating functions (they are defined in the text). Column named $|\text{ans}(\Pi)|$ gives the cardinality of $\text{ans}(\Pi)$, that is, the number of answer sets of program $\Pi = \Pi_p \cup \Pi_S$. In column named Time we indicate the times (in seconds) taken for the synthesis using the smodels [20].

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 [18], but it uses a single shared variable.)

6 Related Work and Concluding Remarks

Two well known, early works on synthesis of concurrent programs were those by Emerson and Clark [6] and Manna and Wolper [16].

In [6] Emerson and Clark introduce the notion of a synchronization skeleton as an abstraction of the actual processes in concurrent programs. They synthesize programs for a shared-memory model of execution by extracting the synchronization skeletons from the models of CTL specifications using a tableau-based decision procedure for the satisfiability of CTL formulas. This extraction procedure is not completely mechanized.

Similarly to [6] in [16] 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 [19] Piterman, Pnueli, and Sa’ar consider the problem of the design of digital circuits from Linear Temporal Logic (LTL) specifications and give an
We closely follow the approaches of [6] and [16]. 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 [1,6,16] 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, Nieuwborgh 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]. The synthesis method is not completely automatic and, in particular, the shared variables are manually introduced during the extraction of the synchronization skeleton. We do not require any extension of the ASP paradigm, we use the by now standard ASP systems, such as DLV [9] and smodels [20], and every steps of our synthesis procedure is fully automatic.

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

http://www.dlvsystem.com/dlvsystem/index.php/DLV


http://www.tcs.hut.fi/Software/smodels/