ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
CONSIGLIO NAZIONALE DELLE RICERCHE

E. De Angelis, A. Pettorossi, M. Proietti

USING ANSWER SET PROGRAMMING SOLVERS TO SYNTHESIZE CONCURRENT PROGRAMS

R. 19 2012

Emanuele De Angelis – Dipartimento di Scienze, Università G. d'Annunzio di Chieti-Pescara, Viale Pindaro 42, I-65127 Pescara, Italy, and Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, I-00185 Roma, Italy.
Email : deangelis@sci.unich.it.
URL: http://www.sci.unich.it/~deangelis.

Alberto Pettorossi – Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, Via del Politecnico 1, I-00133 Roma, Italy, and Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, I-00185 Roma, Italy.
Email : pettorossi@info.uniroma2.it.
URL : http://www.iasi.cnr.it/~adp.

Maurizio Proietti – Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, I-00185 Roma, Italy.
Email : maurizio.proietti@iasi.cnr.it.
URL : http://www.iasi.cnr.it/~proietti.

ISSN: 1128–3378

Collana dei Rapporti dell'Istituto di Analisi dei Sistemi ed Informatica, CNR

viale Manzoni 30, 00185 ROMA, Italy

tel. ++39-06-77161
fax ++39-06-7716461
email: iasi@iasi.rm.cnr.it
URL: http://www.iasi.rm.cnr.it

Abstract

We address the problem of the automatic synthesis of concurrent programs within a framework based on Answer Set Programming (ASP). Every 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 encoding programs, 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 by using a shared variable ranging over a finite domain. The interaction protocol is realized in a distributed manner, that is, every process includes some instructions which operate on the shared variable.

Even for a small number of processes, interaction protocols which guarantee a desired behaviour of the concurrent programs may be hard to design and prove correct. Thus, people have been looking for methods for the automatic synthesis of concurrent programs from the formal specification of their behaviour. Among those methods we recall the ones proposed by Clarke and Emerson [7], Manna and Wolper [24], and Attie and Emerson [2, 3], which use tableau- based algorithms, and those proposed by Pnueli and Rosner [28], and Kupferman and Vardi [22], which use automata-based algorithms.

In contrast with those approaches we do not present an *ad-hoc* algorithm for synthesizing concurrent programs and, instead, we propose a framework based on logic programming by which we reduce the problem of synthesizing concurrent programs to the problem of computing models of a logic program encoding a given specification. We assume that *behavioural properties* of concurrent programs, such as safety or liveness properties, are specified by using formulas of the Computation Tree Logic (CTL), which is a very popular propositional temporal logic over branching time structures [7, 8]. This temporal, behavioural specification φ is encoded as a logic program Π_φ. We also assume that the processes to be synthesized satisfy suitable *structural properties*, such as *symmetry* properties, which specify that all processes follow the same cycling pattern of possible actions. Such structural properties cannot be easily specified by using CTL formulas and, in order to overcome this difficulty, we use, instead, a simple algebraic structure which can be specified in predicate logic and encoded as a logic program Π_σ. 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 enjoy.

In order to construct models of the program Π, we use logic programming with the answer set semantics and we show that every answer set of Π encodes a concurrent program satisfying the given specification. Thus, by using an Answer Set Programming (ASP) solver, such as the ones presented in [11, 12, 19, 21, 23, 30], which computes the answer sets of logic programs, we can synthesize concurrent programs which enjoy the desired behavioural and structural properties. We have performed some synthesis experiments and, in particular, we have synthesized some protocols which are guaranteed to enjoy behavioural properties such as mutual exclusion, starvation freedom, and bounded overtaking, and also suitable symmetry properties. However, the synthesis framework we propose is general and it can be applied to many other classes of concurrent systems and properties besides those mentioned above.

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 used for the synthesis and we also prove that this procedure has optimal time complexity. Then, in Section 5 we present some examples of synthesis of symmetric concurrent programs and we compare the results obtained by using different state-of-the-art ASP solvers. Finally, in Section 6 we discuss related work and, in particular, we compare our results with those obtained by the ASP-based procedure for the synthesis from temporal specifications introduced by Heymans, Van Nieuwenborgh and Vermeir in [20]. In the Appendix we show the

proofs of the results presented in the paper, and the ASP source code of our synthesis procedure.

2. Preliminaries

Let us recall some basic notions and terminology we will use. We will present: (i) the syntax of (a variant of) the *guarded commands* [10], which we use for defining concurrent programs, (ii) some basic notions of group theory, which are required for defining symmetric concurrent programs, and (iii) some fundamental concepts of Computation Tree Logic and of Answer Set Programming, which we use for our synthesis method.

2.1. Guarded commands

The guarded commands we consider are defined from the following 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 which is associative, 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 which is associative and commutative.

The operational semantics of commands can be described in an informal way as follows. *skip* does nothing. $v := d$ stores the value d in the location of the variable v. In order to execute $c_1; c_2$ the command c_1 is executed first, and then the command c_2 is executed. In order to execute $\texttt{if}\ gc_1 \,[]\, \ldots \,[]\, gc_n\ \texttt{fi}$, with $n \geq 1$, one of the guarded commands $g \rightarrow c$ in $\{gc_1, \ldots, gc_n\}$ whose guard g evaluates to *true*, is chosen, and then c is executed; otherwise, if no guard of a guarded command in $\{gc_1, \ldots, gc_n\}$ evaluates to *true*, then the whole command $\texttt{if}\ \ldots\ \texttt{fi}$ terminates with failure. In order to execute $\texttt{do}\ gc_1 \,[]\, \ldots \,[]\, gc_n\ \texttt{od}$, with $n \geq 1$, 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 $\texttt{do} \ldots \texttt{od}$ is executed again; otherwise, if no guard of a guarded command in $\{gc_1, \ldots, gc_n\}$ evaluates to *true*, then the execution proceeds with the next command. The formal semantics of commands will be given in the next section.

2.2. Groups

A group G is a pair $\langle S, \circ \rangle$, where S is a set and $\circ$ is a binary operation on S satisfying the following axioms: (i) $\forall x, y \in S.\, x \circ y \in S$, (ii) $\forall x, y, z \in S.\, (x \circ y) \circ z = x \circ (y \circ z)$, (iii) $\exists e \in S.\, \forall x \in S.\, e \circ x = x \circ e = x$, and (iv) $\forall x \in S.\, \exists y \in S.\, x \circ y = y \circ x = e$. The element e is the *identity* of the group G and the cardinality of S is the *order of the group* G. 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 $G = \langle S, \circ \rangle$ 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 denote by $\mathit{Perm}(S)$ the set of all permutations on the set S, that is, the set of all bijections from S to S. $\mathit{Perm}(S)$ is a group whose operation $\circ$ is function composition and the identity e is the identity permutation, denoted *id*. Given a finite set S, the *order of a permutation* p in $\mathit{Perm}(S)$ is the smallest natural number n such that $p^n = \mathit{id}$.

2.3. Computation Tree Logic

Computation Tree Logic (CTL) is a propositional branching time temporal logic [8]. The underling time structure is a *tree of states*. Every state denotes an instant in time and may have many successor states. There are quantifiers over paths of the tree: A (*for all paths*) and E (*for some path*), which are used for specifying properties that hold for all paths or for some path, respectively. Together with these quantifiers, there are temporal operators such as: X (*next state*), F (*eventually*), G (*globally*), and U (*until*), which are used for specifying properties that hold in the states along paths of the tree. Their formal semantics will be given below.

Given a finite nonempty set *Elem* of elementary propositions ranged over by p, the syntax of CTL formulas φ is as follows:

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

We introduce the following abbreviations:

(i) *true* for $\varphi \vee \neg\varphi$, where φ is any CTL formula,
(ii) *false* for $\neg$*true*,
(iii) $\varphi_1 \vee \varphi_2$ for $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$,
(iv) $\mathsf{EF}\varphi$ for $\mathsf{E}[\mathit{true}\,\mathsf{U}\,\varphi]$
(v) $\mathsf{AG}\,\varphi$ for $\neg\mathsf{EF}\,\neg\varphi$,
(vi) $\mathsf{AF}\,\varphi$ for $\neg\mathsf{EG}\,\neg\varphi$,
(vii) $\mathsf{A}[\,\varphi_1\mathsf{U}\varphi_2]$ for $\neg\mathsf{E}[\neg\varphi_2\,\mathsf{U}\,(\neg\varphi_1 \wedge \neg\varphi_2)] \wedge \mathsf{AF}\,\varphi_2$, and
(viii) $\mathsf{AX}\,\varphi$ for $\neg\mathsf{EX}\,\neg\,\varphi$.

The semantics of CTL is provided by a Kripke structure $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \lambda\rangle$, where: (i) $\mathcal{S}$ is a finite set of *states*, (ii) $\mathcal{S}_0 \subseteq \mathcal{S}$ is a set of *initial states*, (iii) $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is a *total transition relation* (thus, $\forall u \in \mathcal{S}.\ \exists v \in \mathcal{S}.\ \langle u, v\rangle \in \mathcal{R}$), and (iv) $\lambda : \mathcal{S} \to \mathcal{P}(\mathit{Elem})$ is a *total labelling function* that assigns to every state $s \in \mathcal{S}$ a subset $\lambda(s)$ of the set *Elem*. A path π in $\mathcal{K}$ from a state s_0 is an infinite sequence $\langle s_0, s_1, \ldots\rangle$ of states such that, for all $i \geq 0$, $\langle s_i, s_{i+1}\rangle \in \mathcal{R}$. 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:

$\mathcal{K},s \vDash p$	iff	$p \in \lambda(s)$
$\mathcal{K},s \vDash \neg\,\varphi$	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 $\langle s,t\rangle \in \mathcal{R}$ such that $\mathcal{K},t \vDash \varphi$
$\mathcal{K},s \vDash \mathsf{E}[\varphi_1\mathsf{U}\,\varphi_2]$	iff	there exists a path $\langle s_0,s_1,s_2,\ldots\rangle$ in $\mathcal{K}$ with $s_0 = s$ such that for some $i \geq 0$, $\mathcal{K},s_i \vDash \varphi_2$ and for all $0 \leq j < i$, $\mathcal{K},s_j \vDash \varphi_1$
$\mathcal{K},s \vDash \mathsf{EG}\,\varphi$	iff	there exists a path $\langle s_0,s_1,s_2,\ldots\rangle$ in $\mathcal{K}$ with $s_0 = s$ such that for all $i \geq 0$, $\mathcal{K},s_i \vDash \varphi$.

Thus, in particular we have that: (i) $\mathcal{K},s \vDash \mathsf{EX}\,\varphi$ holds iff in $\mathcal{K}$ there exists a successor of state s which satisfies φ, (ii) $\mathcal{K},s \vDash \mathsf{E}[\varphi_1\mathsf{U}\,\varphi_2]$ holds iff there exists a path in $\mathcal{K}$ starting at s along which there exists a state where φ_2 holds and φ_1 holds in every preceding state, and (iii) $\mathcal{K},s \vDash \mathsf{EG}\,\varphi$ holds iff in $\mathcal{K}$ there exists a path starting at s where φ holds in every state along that path.

2.4. Answer Set Programming

Answer set programming (ASP) is a declarative programming paradigm based on logic programs and their answer set semantics. Now we recall some basic definitions of ASP and for those not recalled here the reader may refer to [4, 5, 13, 17, 18, 32]. A *term* t is either a variable X or a function symbol f of arity n (≥ 0) applied to n terms $f(t_1, \ldots, t_n)$. If $n = 0$ then f is called

a *constant*. An *atom* is a predicate symbol p of arity n (≥ 0) applied to n terms $p(t_1, \ldots, t_n)$. A *rule* 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 $a_1, \ldots, a_k, a_{k+1}, \ldots, a_n$ (for $k \geq 0$, $n \geq k$) are atoms and '`not`' denotes negation as failure. A rule with $k > 1$ is said to be a *disjunctive rule* and each atom in $\{a_1, \ldots, a_k\}$ is called a *disjunct*. A rule with $k = 1$ is called *normal*. A rule with $k = 0$ is called an *integrity constraint*. A rule with $k = n$ is called a *fact*. A *logic program* Π is a set of rules. It is said to be a *disjunctive logic program* if there exists a disjunctive rule and it is said to be a *normal logic program* if for every rule $k \leq 1$.

Given a rule r, we define the following sets: $H(r) = \{a_1, \ldots, a_k\}$, $B^+(r) = \{a_{k+1}, \ldots, a_m\}$, $B^-(r) = \{a_{m+1}, \ldots, a_n\}$, and $B(r) = B^+(r) \cup B^-(r)$ and we introduce the following abbreviations: $\mathit{head}(r) = \bigvee_{a \in H(r)} a$, $\mathit{pos}(r) = \bigwedge_{a \in B^+(r)} a$, $\mathit{neg}(r) = \bigwedge_{a \in B^-(r)} \mathtt{not}\ a$, and $\mathit{body}(r) = \mathit{pos}(r) \wedge \mathit{neg}(r)$.

Given two logic programs Π_1 and Π_2, we say that Π_1 *is independent of* Π_2, denoted $\Pi_2 \rhd \Pi_1$, if for each rule r_2 in Π_2, for each predicate symbol p occurring in $H(r_2)$, there is no rule r_1 in Π_1 such that p occurs in $B(r_1)$.

A term, or an atom, or a rule, or a program is said to be *ground* if no variable occurs in it. A ground instance of a term, or an atom, or a rule, or a program is obtained by replacing every variable occurrence by a ground term constructed by using function symbols appearing in Π. The set of all the ground instances of the rules of a program Π is denoted by $\mathit{ground}(\Pi)$. Note that if a program Π has function symbols with positive arity, then $\mathit{ground}(\Pi)$ may be infinite. However, as indicated at the beginning of Section 5, for our purposes we only need a finite subset of that infinite set.

An *interpretation* I of a program Π is a (finite or infinite) set of ground atoms. By $\overleftarrow{I}$ we denote the set $\{p \leftarrow \mid p \in I\}$ of facts. The *Gelfond-Lifschitz transformation* of $\mathit{ground}(\Pi)$ with respect to an interpretation I is the program $\mathit{ground}(\Pi)^I = \{\mathit{head}(r) \leftarrow \mathit{pos}(r) \mid r \in \mathit{ground}(\Pi) \text{ and } B^-(r) \cap I = \emptyset\}$. For any rule $r \in \mathit{ground}(\Pi)$, we say that I *satisfies* r if ($B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) implies $H(r) \cap I \neq \emptyset$. An interpretation I is said to be an *answer set* of Π if I is a *minimal model* of $\mathit{ground}(\Pi)^I$, that is, I is a minimal set (with respect to set inclusion) which satisfies all rules in $\mathit{ground}(\Pi)^I$. The answer set semantics assigns to every program Π the set $\mathit{ans}(\Pi)$ of its answer sets.

Given a program $\Pi = \Pi_1 \cup \Pi_2$, the following fact holds [13]: if $\Pi_2 \rhd \Pi_1$, then $\mathit{ans}(\Pi) = \bigcup_{M \in \mathit{ans}(\Pi_1)} \mathit{ans}(\overleftarrow{M} \cup \Pi_2)$.

3. Specifying Concurrent Programs

A *concurrent program* consists of a finite set of *processes* that are executed in parallel in a shared-memory environment, that is, processes that interact with each other through a *shared variable*. We assume that the shared variable ranges over a finite domain. With every process we associate a distinct *local variable* ranging over a finite domain which is the same for all processes. Every process may test and modify the shared variable and its own local variable by executing guarded commands.

Definition 1 (k-process concurrent program) For $k > 1$, let $\mathtt{x}_1, \ldots, \mathtt{x}_k$ be *local variables* ranging over a finite domain $\mathcal{L}$ and let $\mathtt{y}$ be a *shared variable* ranging over a finite domain $\mathcal{D}$.

For $i=1,\ldots,k$, a *process* P_i is a guarded command of the form:

P_i : $\quad true \rightarrow \texttt{if } gc_1 \ [] \ldots [] \ gc_{n_i} \texttt{ fi}$

where every guarded command gc in $gc_1 [] \ldots [] gc_{n_i}$ is of the form:

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

with $\langle l, d\rangle \neq \langle l', d'\rangle$. We assume that, for $i=1,\ldots,k$, the guards (that is, the expressions to the left of $\rightarrow$) of any two guarded commands of process P_i are mutually exclusive, that is, for all pairs $\langle l, d\rangle$, there is at most one occurrence of the guard '$\mathtt{x}_i = l \wedge \mathtt{y} = d$' in process P_i.
A *k-process concurrent program* C is a command of the form:

C : $\quad \mathtt{x}_1 := \overline{l}_1; \ldots; \mathtt{x}_k := \overline{l}_k; \ \mathtt{y} := \overline{d}; \ \texttt{do } P_1 [] \ldots [] P_k \texttt{ od}$

The $(k+1)$-tuple $\langle \overline{l}_1, \ldots, \overline{l}_k, \overline{d}\rangle$ is said to be the *initialization* of C. □

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

$\mathtt{x}_1 := \mathtt{t}; \quad \mathtt{x}_2 := \mathtt{t}; \quad \mathtt{y} := \mathtt{0}; \quad \texttt{do } P_1 [] P_2 \texttt{ od}$

where P_1 and P_2 are defined as follows:

P_1 : *true* → if
  $\mathtt{x}_1 = \mathtt{t} \wedge \mathtt{y} = \mathtt{0} \rightarrow \mathtt{x}_1 := \mathtt{u}; \mathtt{y} := \mathtt{0}$
  [] $\mathtt{x}_1 = \mathtt{u} \wedge \mathtt{y} = \mathtt{0} \rightarrow \mathtt{x}_1 := \mathtt{t}; \mathtt{y} := \mathtt{1}$
fi

P_2 : *true* → if
  $\mathtt{x}_2 = \mathtt{t} \wedge \mathtt{y} = \mathtt{1} \rightarrow \mathtt{x}_2 := \mathtt{u}; \mathtt{y} := \mathtt{1}$
  [] $\mathtt{x}_2 = \mathtt{u} \wedge \mathtt{y} = \mathtt{1} \rightarrow \mathtt{x}_2 := \mathtt{t}; \mathtt{y} := \mathtt{0}$
fi

This program realizes a protocol which ensures mutual exclusion between the two processes P_1 and P_2. For $i = 1, 2$, process P_i either 'uses a resource' in its critical section, that is, the value of $\mathtt{x}_i$ is $\mathtt{u}$, or 'thinks' in its noncritical section, that is, the value of $\mathtt{x}_i$ is $\mathtt{t}$. The shared variable $\mathtt{y}$ gives the processes P_1 and P_2 the turn to enter the critical section: if $\mathtt{y} = \mathtt{0}$, process P_1 enters the critical section ($\mathtt{x}_1 = \mathtt{u}$), while if $\mathtt{y} = \mathtt{1}$, process P_2 enters the critical section ($\mathtt{x}_2 = \mathtt{u}$).

Note that in a real concurrent program, while P_i is in its noncritical (or critical) section it may execute arbitrary commands not affecting the values of the local and the shared variables. However, for the sake of simplicity, we omit such arbitrary commands and we will consider only those commands which are relevant to the interaction between processes. (A similar approach is taken in [7] where *synchronization skeletons* are considered.) □

Now we introduce the semantics of k-process concurrent programs by using Kripke structures. Given a k-process concurrent program C, a *state* of C is any $(k+1)$-tuple $\langle l_1, \ldots, l_k, d\rangle$, where: (i) the first k components are values for the local variables $\mathtt{x}_1, \ldots, \mathtt{x}_k$ of C, one local variable for each process P_i, and (ii) d is a value for the shared variable $\mathtt{y}$ of C. Given any state s, by $s(\mathtt{x}_i)$ we denote the value of the local variable of process P_i in state s and, similarly, by $s(\mathtt{y})$ we denote the value of the shared variable in state s.

Definition 2 (Reachability) Let C be a k-process concurrent program. We say that state s_2 is *one-step reachable* from state s_1, and we write $Reach(s_1, s_2)$, if there exists a process P_i, for some $i \in \{1, \ldots, k\}$, with a guarded command of the form: $\mathtt{x}_i = s_1(\mathtt{x}_i) \wedge \mathtt{y} = s_1(\mathtt{y}) \rightarrow \mathtt{x}_i := s_2(\mathtt{x}_i); \mathtt{y} := s_2(\mathtt{y})$, and for all $j \in \{1, \ldots, k\}$ different from i, $s_1(\mathtt{x}_j) = s_2(\mathtt{x}_j)$. We say that s_2 is *reachable* from s_1 if $Reach^*(s_1, s_2)$, where by $Reach^*$ we denote the reflexive, transitive closure of $Reach$. □

Note that our definition of the transition relation $Reach$ formalizes the *interleaving* semantics of guarded commands.

Definition 3 (Kripke structure associated with a k-process concurrent program) Let C be a k-process concurrent program of the form

$C:\quad \mathtt{x}_1{:=}\overline{l}_1;\ldots;\mathtt{x}_k{:=}\overline{l}_k;\ \mathtt{y}{:=}\overline{d};\ \mathtt{do}\ P_1\ [\!]\ \ldots\ [\!]\ P_k\ \mathtt{od}$

Let *Reach* be the reachability relation associated with C which we assume to be total. The *Kripke structure* $\mathcal{K}$ *associated with* C is the 4-tuple $\langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \lambda\rangle$, where:
(i) $\mathcal{S} = \{s \mid \mathit{Reach}^*(s_0, s)\} \subseteq \mathcal{L}^k \times \mathcal{D}$ is the set of *reachable* states,
(ii) $\mathcal{S}_0 = \{s_0\} = \{\langle \overline{l}_1, \ldots, \overline{l}_k, \overline{d}\rangle\}$,
(iii) $\mathcal{R} = \mathit{Reach} \subseteq \mathcal{S}\times\mathcal{S}$, and
(iv) for all $\langle l_1, \ldots, l_k, d\rangle \in \mathcal{S}$, $\lambda(\langle l_1, \ldots, l_k, d\rangle) = \{\mathit{local}(P_1, l_1), \ldots, \mathit{local}(P_k, l_k), \mathit{shared}(d)\}$, where for $i = 1, \ldots, k$, the elementary proposition $\mathit{local}(P_i, l_i)$ denotes that the local variable $\mathtt{x}_i$ of process P_i has value l_i, and analogously, the elementary proposition $\mathit{shared}(d)$ denotes that the shared variable $\mathtt{y}$ has value d.
The set *Elem* of the elementary propositions is $\{\mathit{local}(P_i, l_i) \mid i = 1, \ldots, k\} \cup \{\mathit{shared}(d) \mid d \in \mathcal{D}\}$. □

Note that, since every state has a successor state, every concurrent program is a *nonterminating* program.

For every given state s, for every $i \in \{1, \ldots, k\}$, if $(\mathtt{x}_i = l \wedge \mathtt{y} = d \ \rightarrow\ \mathtt{x}_i := l';\ \mathtt{y} := d')$ is a guarded command in P_i such that $l = s(\mathtt{x}_i)$ and $d = s(\mathtt{y})$, then we say that P_i is *enabled* in s and the guard $\mathtt{x}_i = l \wedge \mathtt{y} = d$ *holds* in s.

Example 2. Given the 2-process concurrent program C of Example 1, the associated Kripke structure is depicted in Figure 1. We depict it as a graph whose nodes are the reachable states from the initial state $s_0 = \langle \mathtt{t}, \mathtt{t}, 0\rangle$. Each transition from state s to state t is associated with the guarded command whose guard holds in s. For the initial state s_0, we have that $\lambda(s_0) = \{\mathit{local}(P_1, \mathtt{t}),\ \mathit{local}(P_2, \mathtt{t}),\ \mathit{shared}(0)\}$ and, similarly, for the values of λ for the other states. □

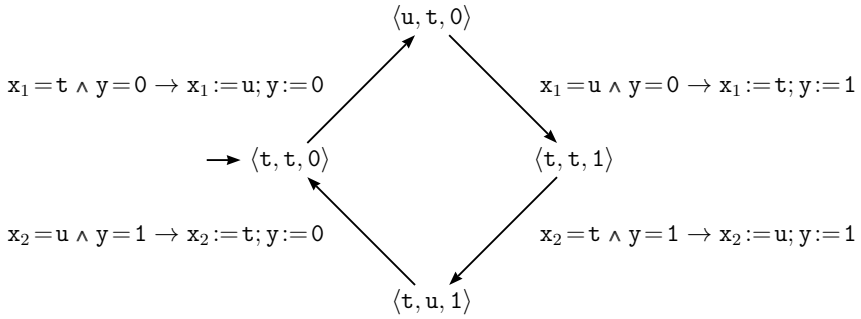

Figure 1: The graph representing the transition relation *Reach* of the Kripke structure associated with the concurrent program of Example 1. Each arc is labelled by the guarded command which causes that transition according to Definition 2. The initial state is $s_0 = \langle \mathtt{t}, \mathtt{t}, 0\rangle$.

Definition 4 (Satisfaction relation for a k-process concurrent program) Let C be a k-process concurrent program with initialization s_0, $\mathcal{K}$ be the Kripke structure associated with C, and φ be a CTL formula. We say that C *satisfies* φ, denoted $C \vDash \varphi$, if $\mathcal{K}, s_0 \vDash \varphi$. □

Example 3. Let us consider the 2-process concurrent program C defined in Example 1. The fact that the critical section associated with the value $\mathtt{u}$ of the local variable is executed in a mutually exclusive way, is formalized by the CTL formula $\varphi =_{def} \mathsf{AG}\,\neg(\mathit{local}(P_1,\mathtt{u}) \wedge \mathit{local}(P_2,\mathtt{u}))$. 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 $\langle \mathtt{t},\mathtt{t},\mathtt{0}\rangle$ which leads to either the state $\langle \mathtt{u},\mathtt{u},\mathtt{0}\rangle$ or the state $\langle \mathtt{u},\mathtt{u},\mathtt{1}\rangle$. □

In the literature (see, for instance, [2, 8, 15]) it is often considered the case where concurrent programs consist of similar processes, the similarity being determined by the fact that all processes follow the same cycling pattern of possible actions.

In this paper we formalize some *structural* properties which extend the notion of similarity. In particular, for any two distinct processes P_i and P_j in a concurrent program, we assume that process P_j can be obtained from process P_i by permuting the values of the shared variable $\mathtt{y}$. For instance, in Example 1 the guarded commands in P_2 can be obtained from those in P_1 by interchanging $\mathtt{0}$ and $\mathtt{1}$. Moreover, it is often the case that all processes of a given concurrent program C also share additional structural properties, such as the fact that the tests and the assignments performed on the local variables are the same for all processes in C. For instance, in Example 1 we have that both processes P_1 and P_2 may change state by changing the value of their local variables from $\mathtt{t}$ to $\mathtt{u}$ or from $\mathtt{u}$ to $\mathtt{t}$.

Now we formalize those structural properties by introducing the *k-symmetric program structures*.

Definition 5 (k-symmetric program structure) For $k>1$, let $\mathcal{L}$ be a finite domain for the local variables $\mathtt{x}_1,\ldots,\mathtt{x}_k$, and $\mathcal{D}$ be a finite domain for the shared variable $\mathtt{y}$. A *k-symmetric program structure* $\sigma = \langle f, T, l_0, d_0\rangle$ *over* $\mathcal{L}$ *and* $\mathcal{D}$ consists of: (i) a *k-generating function* $f \in \mathit{Perm}(\mathcal{D})$, which is either the identity function id or a generator of a cyclic group $\{id, f, f^2,\ldots, f^{k-1}\}$ of order k, (ii) a *local transition relation* $T \subseteq \mathcal{L}\times\mathcal{L}$ which is total over $\mathcal{L}$, (iii) an element $l_0 \in \mathcal{L}$, and (iv) an element $d_0 \in \mathcal{D}$. □

Definition 6 (k-process symmetric concurrent program) For any $k>1$, let $\sigma = \langle f, T, l_0, d_0\rangle$ be a k-symmetric program structure. A k-process concurrent program is said to be *symmetric* w.r.t. σ if it is of the form $\mathtt{x}_1{:=}l_0;\ldots;\mathtt{x}_k{:=}l_0;\mathtt{y}{:=}d_0;\ \mathtt{do}\ P_1 \,[]\, \ldots \,[]\, P_k\ \mathtt{od}$ and, for all $i \in \{1,\ldots,k\}$, for all guarded commands gc of the form $\mathtt{x}_i{=}l \wedge \mathtt{y}{=}d \rightarrow \mathtt{x}_i{:=}l';\mathtt{y}{:=}d'$, we have that:
(i) $\langle l, l'\rangle \in T$ and
(ii) gc is in P_i iff $\big(\mathtt{x}_{(i \bmod k)+1}{=}l \wedge \mathtt{y}{=}f(d) \rightarrow \mathtt{x}_{(i \bmod k)+1}{:=}l';\mathtt{y}{:=}f(d')\big)$ is in $P_{(i \bmod k)+1}$. □

Example 4. Let us consider the 2-process concurrent program C of Example 1. The group $\mathit{Perm}(\mathcal{D})$ of permutations over $\mathcal{D} = \{\mathtt{0},\mathtt{1}\}$ consists of the following two permutations: $id = \{\langle \mathtt{0},\mathtt{0}\rangle, \langle \mathtt{1},\mathtt{1}\rangle\}$ (that is, the identity permutation) and $f = \{\langle \mathtt{0},\mathtt{1}\rangle, \langle \mathtt{1},\mathtt{0}\rangle\}$. The program C is symmetric w.r.t. the 2-symmetric program structure $\langle f, T, \mathtt{t}, \mathtt{0}\rangle$, where the local transition relation T is $\{\langle \mathtt{t},\mathtt{u}\rangle,\ \langle \mathtt{u},\mathtt{t}\rangle\}$. Indeed, its initialization is: $\mathtt{x}_1{:=}\mathtt{t};\ \mathtt{x}_2{:=}\mathtt{t};\ \mathtt{y}{:=}\mathtt{0}$, and processes P_1 and P_2 are as follows:

P_1 : *true* $\rightarrow$ **if**
  $\mathtt{x}_1{=}\mathtt{t} \wedge \mathtt{y}{=}\mathtt{0} \rightarrow \mathtt{x}_1{:=}\mathtt{u};\ \mathtt{y}{:=}\mathtt{0}$
 [] $\mathtt{x}_1{=}\mathtt{u} \wedge \mathtt{y}{=}\mathtt{0} \rightarrow \mathtt{x}_1{:=}\mathtt{t};\ \mathtt{y}{:=}\mathtt{1}$
fi

P_2 : *true* $\rightarrow$ **if**
  $\mathtt{x}_2{=}\mathtt{t} \wedge \mathtt{y}{=}f(\mathtt{0}) \rightarrow \mathtt{x}_2{:=}\mathtt{u};\ \mathtt{y}{:=}f(\mathtt{0})$
 [] $\mathtt{x}_2{=}\mathtt{u} \wedge \mathtt{y}{=}f(\mathtt{0}) \rightarrow \mathtt{x}_2{:=}\mathtt{t};\ \mathtt{y}{:=}f(\mathtt{1})$
fi

□

4. Synthesizing Concurrent Programs

Now we present our method based on Answer Set Programming for synthesizing a k-process symmetric concurrent program from a CTL formula encoding a given behavioural property and a k-symmetric program structure encoding a given structural property.

Definition 7 (The synthesis problem) Given a CTL formula φ and a k-symmetric program structure σ over the finite domains $\mathcal{L}$ and $\mathcal{D}$, the *synthesis problem* consists in finding a k-process concurrent program C such that $C \vDash \varphi$ and C is symmetric with respect to σ. □

The synthesis problem can be solved by applying the following two-step procedure: (*Step* 1) we generate a k-process symmetric concurrent program C, and (*Step* 2) we verify whether or not C satisfies a given behavioural property φ. By Definition 6, from any process P_i, with $i = 1, \ldots, k$, we derive process $P_{(i \bmod k)+1}$ by applying the k-generating function f to the guarded commands of P_i, thereby deriving the guarded commands of $P_{(i \bmod k)+1}$. Thus, Step 1 can be performed by generating process P_1 and using f for generating the other $k-1$ processes. Then Step 2 reduces to the test of the satisfiability relation $\mathcal{K}, s_0 \vDash \varphi$, where: (i) $\mathcal{K}$ is the Kripke structure associated with C, and (ii) state s_0 is the initial state of $\mathcal{K}$ corresponding to the initialization of C.

We present a solution to the synthesis problem in a purely declarative manner by reducing it to the problem of computing the answer sets of a logic program Π encoding an instance of the synthesis problem. The logic program Π is the union of a program Π_σ which encodes a structural property σ and a program Π_φ which encodes a behavioural property φ.

In Theorem 4.1 we will prove that every answer set of Π encodes a k-process concurrent program satisfying φ and which is symmetric w.r.t. σ. We have that Π_σ is independent of Π_φ (that is, $\Pi_\varphi \rhd \Pi_\sigma$) and, thus, we can first compute the answer sets of Π_σ and then use those answer sets, together with program Π_φ, to test whether or not the encoded k-symmetric concurrent program satisfies φ.

Programs Π_σ and Π_φ are introduced by the following Definitions 8 and 9, respectively.

Definition 8 (Logic program encoding a structural property) Let $\sigma = \langle f, T, l_0, d_0 \rangle$ be a k-symmetric program structure over the finite domains $\mathcal{L}$ and $\mathcal{D}$ and s_0 be the $(k+1)$-tuple $\langle l_0, \ldots, l_0, d_0 \rangle$. The logic program Π_σ is as follows:

$$
\begin{array}{ll}
1.1 & \mathit{enabled}(1, X_1, Y) \vee \mathit{disabled}(1, X_1, Y) \leftarrow \mathit{reachable}(\langle X_1, \ldots, X_k, Y \rangle) \\
1.2 & \mathit{enabled}(2, X, Y) \leftarrow \mathit{gc}(2, X, Y, X', Y') \\
 & \qquad \vdots \\
1.k & \mathit{enabled}(k, X, Y) \leftarrow \mathit{gc}(k, X, Y, X', Y') \\
2.1 & \mathit{gc}(1, X, Y, X_1, Y_1) \vee \ldots \vee \mathit{gc}(1, X, Y, X_m, Y_m) \leftarrow \mathit{enabled}(1, X, Y) \wedge \\
 & \qquad\qquad \mathit{candidates}(X, Y, [\langle X_1, Y_1 \rangle, \ldots, \langle X_m, Y_m \rangle]) \\
2.2 & \mathit{gc}(2, X, Z, X', Z') \leftarrow \mathit{gc}(1, X, Y, X', Y') \wedge \mathit{perm}(Y, Z) \wedge \mathit{perm}(Y', Z') \\
 & \qquad \vdots \\
2.k & \mathit{gc}(k, X, Z, X', Z') \leftarrow \mathit{gc}(k-1, X, Y, X', Y') \wedge \mathit{perm}(Y, Z) \wedge \mathit{perm}(Y', Z') \\
3.1 & \mathit{reachable}(s_0) \leftarrow \\
3.2 & \mathit{reachable}(\langle X_1, \ldots, X_k, Y \rangle) \leftarrow \mathit{tr}(\langle X'_1, \ldots, X'_k, Y' \rangle, \langle X_1, \ldots, X_k, Y \rangle) \\
4.1 & \mathit{tr}(\langle X_1, \ldots, X_k, Y \rangle, \langle X'_1, \ldots, X_k, Y' \rangle) \leftarrow \mathit{reachable}(\langle X_1, \ldots, X_k, Y \rangle) \wedge \mathit{gc}(1, X_1, Y, X'_1, Y') \\
 & \qquad \vdots \\
4.k & \mathit{tr}(\langle X_1, \ldots, X_k, Y \rangle, \langle X_1, \ldots, X'_k, Y' \rangle) \leftarrow \mathit{reachable}(\langle X_1, \ldots, X_k, Y \rangle) \wedge \mathit{gc}(k, X_k, Y, X'_k, Y') \\
5. & \leftarrow \mathit{reachable}(\langle X_1, \ldots, X_k, Y \rangle) \wedge \texttt{not}\ \mathit{enabled}(1, X_1, Y) \wedge \ldots \wedge \texttt{not}\ \mathit{enabled}(k, X_k, Y)
\end{array}
$$

together with the following two sets of ground facts:

(i) $\{candidates(l, d, L(l, d)) \leftarrow \mid\ l \in \mathcal{L} \wedge d \in \mathcal{D}\}$,where $L(l, d)$ is any list representing the set of pairs $\{\langle l', d' \rangle \ \mid\ \langle l, l' \rangle \in T \ \wedge\ d' \in \mathcal{D} \ \wedge\ \langle l, d \rangle \neq \langle l', d' \rangle\}$

(ii) $\{perm(d, d') \leftarrow \mid d, d' \in \mathcal{D} \wedge f(d) = d'\}$. □

In this program, for $i = 1, \ldots, k$, the predicate $gc(i, l, d, l', d')$ holds iff in process P_i there exists the guarded command $\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d'$ (see also Definition 10).

Rule 1.1 states that in every reachable state, process P_1 is either enabled (that is, one of its guards holds) or disabled. Rule 1.1 is used to derive atoms either of the form $enabled(1, X_1, Y)$ or of the form $disabled(1, X_1, Y)$. If an atom of the form $enabled(1, X_1, Y)$ is derived, then a guarded command for process P_1 (that is, an atom of the form $gc(1, X, Y, X_i, Y_i)$) is generated by using Rule 2.1. Note that, without Rule 1.1, no atom for the predicates *enabled* and *gc* could be generated and, therefore, no concurrent program would be synthesized.

Rules 1.i, with $i = 2, \ldots, k$, state that any process P_i is enabled in state s if P_i has a guarded command of the form $\mathtt{x}_i = X \ \wedge\ \mathtt{y} = Y \rightarrow \mathtt{x}_i := X';\ \mathtt{y} := Y'$, for some values of X' and Y', such that $X = s(\mathtt{x}_i)$ and $Y = s(\mathtt{y})$.

The disjunctive Rule 2.1 generates a guarded command for process P_1 by first enumerating all candidate guarded commands for that process (through the predicate *candidates*) and then selecting one candidate which corresponds to a disjunct of its head. Each guarded command consists of the guard $\mathtt{x}_1 = X \ \wedge\ \mathtt{y} = Y$, encoded by using the atom $enabled(1, X, Y)$, and a command $\mathtt{x}_1 := X_i;\ \mathtt{y} := Y_i$, encoded by a pair $\langle X_i, Y_i \rangle$ in the list which is the third argument of $candidates(X, Y, L(l, d))$.

The number m of pairs $\langle X_i, Y_i \rangle$ in the list $L(l, d)$ is uniquely determined by the values l and d of the variables X and Y, respectively, in $enabled(1, X, Y)$. (It can be shown that $|\mathcal{D}| - 1 \leq m \leq |\mathcal{L}| \cdot |\mathcal{D}| - 1$.) Thus, Rule 2.1 actually stands for a set of rules, one rule for each value of m, and this set of rules can effectively be derived only when the set of facts for the predicate *candidates* is computed.

For instance, let us consider the sets $T = \{\langle \mathtt{a}, \mathtt{b} \rangle,\ \langle \mathtt{a}, \mathtt{a} \rangle,\ \langle \mathtt{b}, \mathtt{a} \rangle\}$ and $\mathcal{D} = \{\mathtt{0}, \mathtt{1}\}$. For $X = \mathtt{b}$, $Y = \mathtt{0}$, we have that $candidates(\mathtt{b}, \mathtt{0}, [\langle \mathtt{a}, \mathtt{0} \rangle, \langle \mathtt{a}, \mathtt{1} \rangle])$ holds (recall that a guarded command should change either the value of the local variable or the value of the shared variable), and for $X = \mathtt{a}$, $Y = \mathtt{0}$, we have that $candidates(\mathtt{a}, \mathtt{0}, [\langle \mathtt{a}, \mathtt{1} \rangle, \langle \mathtt{b}, \mathtt{0} \rangle, \langle \mathtt{b}, \mathtt{1} \rangle])$ holds. Hence, when $Y = \mathtt{0}$, we have two instances of Rule 2.1, one for $m = 2$ and one for $m = 3$.

Rules 2.2–2.k realize Definition 6. In particular, it allows us to derive the guarded command for processes $P_2, \ldots, P_k$ from the guarded commands generated for process P_1. Note that, due to our definition of a symmetric program structure, the subscript of the process used for the initial choice (1 in our case) is immaterial, in the sense that any other choice for that subscript produces a solution satisfying the same behavioural and structural properties.

Rules 3.1, 3.2, and 4.1–4.k define, in a mutually recursive way, the reachability relation (encoded by the predicate *reachable*) and the transition relation $\mathcal{R}$ (encoded by the predicate *tr*) of the Kripke structure associated with the concurrent program to be synthesized.

Rule 5 is an integrity constraint enforcing that any answer set of Π_σ is a model of $\Pi_\sigma - \{\text{Rule 5}\}$ which does not satisfy the body of Rule 5. Thus, Rule 5 guarantees that the transition relation $\mathcal{R}$ is total, that is, in every reachable state there exists at least one enabled process.

Now let us present the logic program Π_φ which encodes a given behavioural property φ. Note that program Π_φ depends on program Π_σ for the definition of the transition relation $tr(S, T)$ and for the initial state s_0, which is assumed to be the $(k+1)$-tuple $\langle l_0, \ldots, l_0, d_0 \rangle$.

Definition 9 (Logic program encoding a behavioural property) Let φ be a CTL formula. The logic program Π_φ encoding φ is as follows:

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

together with the following two sets of ground facts:

(i) $\{elem(s, local(P_i, l)) \leftarrow \mid 1 \leq i \leq k \wedge s \in \mathcal{L}^k \times \mathcal{D}\ \wedge s(\mathtt{x}_i) = l\}$

(ii) $\{elem(s, shared(d)) \leftarrow \mid s \in \mathcal{L}^k \times \mathcal{D} \wedge s(\mathtt{y}) = d\}$. □

Note that in the ground facts defining *elem*, for $i = 1, \ldots, k$, by $s(\mathtt{x}_i)$ we denote the i-th component of s and by $s(\mathtt{y})$ we denote the $(k+1)$-th component of s (see Section 3 for this notational convention). In Rule 1 of program Π_φ, by abuse of language, we use φ to denote the ground term representing the CTL formula φ. In particular, in the ground term φ we use the function symbols *not*, *and*, *ex*, *eu* and *eg* to denote the operators $\neg$, $\wedge$, EX, EU, and EG, respectively.

Rules 2–10, taken from [26], encode the semantics of CTL formulas as follows: (i) $sat(s, \psi)$ holds iff the formula ψ holds in state s, and (ii) $satpath(s, t, \psi)$ holds iff there exists a path from state s to state t such that every state in that path (except possibly the last one) satisfies the formula ψ. Rule 1 is an integrity constraint enforcing that any answer set of Π is a model of $(\Pi_\varphi \cup \Pi_\sigma) - \{\text{Rule 1}\}$ satisfying $sat(s_0, \varphi)$.

Now we establish the correctness (that is, the soundness and completeness) of our synthesis procedure. It relates the k-process symmetric (w.r.t. σ) concurrent programs satisfying φ with the answer sets of the logic program $\Pi_\varphi \cup \Pi_\sigma$. Let us first introduce the following definition.

Definition 10 (Encoding of a k-process concurrent program) Let C be a k-process concurrent program of the form $\mathtt{x}_1 := \overline{l}_1; \ldots; \mathtt{x}_k := \overline{l}_k;\ \mathtt{y} := \overline{d};\ \texttt{do}\ P_1 \,[\!]\, \ldots \,[\!]\, P_k\ \texttt{od}$. Let M be a set of ground atoms. We say that M *encodes* C if, for all i, l, d, l', d', the following holds:

$gc(i, l, d, l', d') \in M$ iff $\big(\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d'\big)$ is a guarded command in P_i. □

Theorem 4.1 (Soundness and completeness of synthesis) Let φ be a CTL formula and σ be a k-symmetric program structure over the finite domains $\mathcal{L}$ and $\mathcal{D}$. Then, there exists a k-process concurrent program C such that (i) $C \vDash \varphi$ and (ii) C is symmetric w.r.t. σ iff there exists an answer set $M \in ans(\Pi_\varphi \cup \Pi_\sigma)$ such that M encodes C. □

The following theorem establishes the complexity of our synthesis procedure as a function of the synthesis parameters, that is, (i) the number k of processes, (ii) the size $|\varphi|$ of the CTL behavioural property φ defined to be the number of operators and elementary propositions occurring in φ, and (iii) the cardinalities of $\mathcal{L}$ and $\mathcal{D}$ which are the domains of f and T, respectively. When we state the complexity result with respect to one parameter, we assume that the others remain constant.

Theorem 4.2 (Complexity of synthesis) For any number $k > 1$ of processes, for any symmetric program structure σ over $\mathcal{L}$ and $\mathcal{D}$, and for any CTL formula φ, an answer set of the logic program $\Pi_\varphi \cup \Pi_\sigma$ can be computed in (i) exponential time w.r.t. k, (ii) linear time w.r.t. $|\varphi|$, and (iii) nondeterministic polynomial time w.r.t. $|\mathcal{L}|$ and w.r.t. $|\mathcal{D}|$. □

It is known (see, for instance, [22]) that the problem of synthesis from a CTL specification φ is EXPTIME-complete w.r.t. $|\varphi|$. In order to compare the complexity of our synthesis procedure with that of other techniques which can be found in the literature [2, 3, 7, 22, 20], note that the parameters of our synthesis procedure are not mutually independent. In particular, as we will see in the following section, the usual behavioural properties considered for the mutual exclusion problem, determine a CTL specification whose size depends on the number k of processes. However, since our ASP synthesis procedure has time complexity which is exponential w.r.t. k, it turns out that our translation yields a synthesis procedure which still belongs to the EXPTIME class and, thus, it matches the complexity of the synthesis problem.

5. Synthesizing Concurrent Programs using ASP solvers

In this section we present some experimental results obtained by applying our procedure for the synthesis of various mutual exclusion protocols.

In order to compute the answer sets of a logic program P with an ASP solver, we should first construct the set $\mathit{ground}(P)$. This set is constructed by a *grounder* which is *either* a standalone tool, such as *gringo* [12] or *lparse* [31], independent of the ASP solver, *or* is a built-in module of the ASP solver, as in the *DLV* system [23].

If a logic program P has function symbols with positive arity, then $\mathit{ground}(P)$ is infinite. Thus, in particular, $\mathit{ground}(\Pi)$ is infinite. However, in order to compute the answer sets of Π, we only need some finite subsets of $\mathit{ground}(\Pi)$. These subsets are constructed by most grounders by means of the so called *domain predicates*, which specify the finite domains over which the variables should range [12, 23, 31].

In our case, a finite set of ground rules is obtained from program Π_φ by introducing in the body of each of the Rules 2–10 a domain predicate so that terms representing CTL formulas are restricted to range over subterms of φ. (Here and in what follows, when we refer to a subterm, we mean a non necessarily proper subterm.) In particular, a rule of the form $\mathit{sat}(S, \psi) \leftarrow \mathit{Body}$ is replaced by $\mathit{sat}(S, \psi) \leftarrow \mathit{Body} \wedge d(\psi)$, where d is the domain predicate defined by the set $\{d(\psi) \leftarrow \mid \psi \text{ is a subterm of } \varphi\}$ of ground facts. The correctness of this replacement relies on the fact that, in order to prove $\mathit{sat}(s_0, \varphi)$ by using Rules 2–10, it is sufficient to consider only the instances of these rules where subterms of φ occur.

Note that, by using a grounder after the introduction of domain predicates, we get a set of ground instances of Rules 2–10 whose cardinality is linear in the number of subterms of φ and, hence, in the size of φ. This fact is relevant for the complexity results stated in Theorem 4.2.

5.1. Synthesis examples

In our synthesis experiments, in order to define the k-symmetric program structures of the programs to be synthesized, we have made the following choices for: (i) the domain $\mathcal{L}$ of the local variables $\mathtt{x}_i$'s, (ii) the domain $\mathcal{D}$ of the shared variable $\mathtt{y}$, (iii) the k-generating function f, (iv) the set T, (v) the value of $l_0 \in \mathcal{L}$, and (vi) the value of $d_0 \in \mathcal{D}$.

We have taken the domain $\mathcal{L}$ to be $\{\mathtt{t}, \mathtt{w}, \mathtt{u}\}$, where $\mathtt{t}$ represents the *noncritical section*, $\mathtt{w}$ represents the *waiting section*, and $\mathtt{u}$ represents the *critical section*.

We have taken the domain $\mathcal{D}$ to be $\{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{n}\}$, where $\mathtt{n}$ depends on: (i) the number k of the processes in the concurrent program to be synthesized, and (ii) the properties that the concurrent program should satisfy. At the beginning of every synthesis experiment we have taken $\mathtt{n}=\mathtt{1}$ and, if the synthesis failed, we have increased the value of $\mathtt{n}$ by one unity at a time, hoping for a successful synthesis with a larger value of $\mathtt{n}$.

We have taken the k-generating function f to be either (i) the identity function id, or (ii) a permutation among the $|\mathcal{D}|!/\big(k \cdot (|\mathcal{D}| - k)!\big)$ permutations of order k defined over $\mathcal{D}$.

We have taken the local transition relation T to be $\{\langle \mathtt{t}, \mathtt{w}\rangle, \langle \mathtt{w}, \mathtt{w}\rangle, \langle \mathtt{w}, \mathtt{u}\rangle, \langle \mathtt{u}, \mathtt{t}\rangle\}$. The pair $\langle \mathtt{t}, \mathtt{w}\rangle$ denotes that, once the noncritical section $\mathtt{t}$ has been executed, a process may enter the waiting section $\mathtt{w}$. The pairs $\langle \mathtt{w}, \mathtt{w}\rangle$ and $\langle \mathtt{w}, \mathtt{u}\rangle$ denote that a process may repeat (possibly an unbounded number of times) the execution of its waiting section $\mathtt{w}$ and then may enter its critical section $\mathtt{u}$. The pair $\langle \mathtt{u}, \mathtt{t}\rangle$ denotes that, once the critical section $\mathtt{u}$ has been executed, a process may enter its noncritical section $\mathtt{t}$.

Finally, we have taken l_0 to be $\mathtt{t}$ and d_0 to be $\mathtt{0}$.

For $k = 2, \ldots, 6$, we have synthesized (see Column 1 of Table 1) various k-process symmetric concurrent programs of the form $\mathtt{x}_1 := \mathtt{t};\ \ldots ;\mathtt{x}_k := \mathtt{t};\ \mathtt{y} := \mathtt{0};\ \mathtt{do}\ P_1 \,[\!]\, \ldots \,[\!]\, P_k\ \mathtt{od}$, which satisfies some behavioural properties among those defined by the following CTL formulas (see Column 2 of Table 1).

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

$$\mathsf{AG}\,\neg(\mathit{local}(\mathtt{P}_i, \mathtt{u}) \wedge \mathit{local}(\mathtt{P}_j, \mathtt{u})) \qquad (ME)$$

(ii) *Progression and Starvation Freedom*, that is, (progression) every process P_i which is in the noncritical section, may enter its waiting section (that is, modify the local variable $\mathtt{x}_i$ from $\mathtt{t}$ to $\mathtt{w}$), thereby requesting to enter the critical section, and (starvation freedom) if a process P_i is in waiting section ($\mathtt{x}_i = \mathtt{w}$), then after a finite amount of time, it will enter its critical section ($\mathtt{x}_i = \mathtt{u}$): for all i in $\{1, \ldots, k\}$,

$$\mathsf{AG}\,((\mathit{local}(\mathtt{P}_i, \mathtt{t}) \rightarrow \mathsf{EX}\,\mathit{local}(\mathtt{P}_i, \mathtt{w})) \wedge (\mathit{local}(\mathtt{P}_i, \mathtt{w}) \rightarrow \mathsf{AF}\,\mathit{local}(\mathtt{P}_i, \mathtt{u}))) \qquad (SF)$$

(iii) *Bounded Overtaking*, that is, while process P_i is in its waiting section, every other process P_j leaves its critical section *at most once*, that is, P_j should not be in its critical section $\mathtt{u}$ and then in its waiting section $\mathtt{w}$ and then again in its critical section $\mathtt{u}$, while P_i is always in its waiting section $\mathtt{w}$ (see the underlined subformulas): for all i, j in $\{1, \ldots, k\}$, with $i \neq j$,

$$\begin{array}{l}\mathsf{AG}\,\neg\big[\underline{\mathit{local}(\mathtt{P}_i, \mathtt{w}) \wedge \mathit{local}(\mathtt{P}_j, \mathtt{u})} \wedge \\ \quad \mathsf{E}\,\big[\mathit{local}(\mathtt{P}_i, \mathtt{w})\,\mathsf{U}\,\big(\mathit{local}(\mathtt{P}_i, \mathtt{w}) \wedge \mathit{local}(\mathtt{P}_j, \mathtt{w}) \wedge \\ \qquad \mathsf{E}\,\big[\mathit{local}(\mathtt{P}_i, \mathtt{w})\,\mathsf{U}\,\underline{(\mathit{local}(\mathtt{P}_i, \mathtt{w}) \wedge \mathit{local}(\mathtt{P}_j, \mathtt{u}))}\big]\big)\big]\big]\end{array} \qquad (BO)$$

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

$$\mathsf{AG}\,((\mathit{local}(\mathtt{P}_i, \mathtt{w}) \wedge \textstyle\bigwedge_{j \in \{1,\ldots,k\} - \{i\}} \mathit{local}(\mathtt{P}_j, \mathtt{t})) \rightarrow \mathsf{EX}\,\mathit{local}(\mathtt{P}_i, \mathtt{u})) \qquad (MR)$$

First, we have synthesized a simple protocol, called 2-*mutex*-1, for two processes enjoying the mutual exclusion property (see row 1 of Table 1), and then we synthesized various other protocols for two or more processes which enjoy other properties. In that table the identifier k-*mutex*-p occurring in the first column, denotes the synthesized protocol for k processes satisfying the $p\,(\geq 1)$ behavioural properties listed in the second column Properties. For instance, program 2-*mutex*-4 is the synthesized protocol for 2 processes which enjoys the four behavioural properties

ME, *SF*, *BO*, and *MR*. In each row of Table 1 we have shown the minimal cardinality (in Column $|\mathcal{D}|$) and the k-generating function (in Column f) for which the synthesis of the program of that row succeeds.

The synthesis of program 2-*mutex*-1 succeeds with $|\mathcal{D}|$=2 and both the identity function and the permutation $f_1 = \{\langle 0, 1\rangle, \langle 1, 0\rangle\}$ (see rows 1 and 2). The syntheses of programs 2-*mutex*-2 and 2-*mutex*-3 fail for $|\mathcal{D}| = 2$ and the identity function, but they succeed for $|\mathcal{D}| = 2$ and f_1 (see rows 3 and 4). The synthesis of 2-*mutex*-4 fails for $|\mathcal{D}|=2$ and any choice of a 2-generating function. Thus, we increased $|\mathcal{D}|$ from 2 to 3. For $|\mathcal{D}|=3$ and the identity function the synthesis fails, but it succeeds for the permutation $f_2 = \{\langle 0, 1\rangle, \langle 1, 0\rangle, \langle 2, 2\rangle\}$ of order 2 (see row 5). If we use different permutations of order 2, instead of f_2, we get programs which are equal to the program 2-*mutex*-4 (presented in Figure 2), modulo a permutation of the values of the shared variable y.

The synthesis of 3-*mutex*-1 succeeds for $|\mathcal{D}| = 2$ and the identity function (see row 6). The synthesis of 3-*mutex*-2 fails for $|\mathcal{D}|=2$ (the only choice for the 3-generating function is the identity function) and, thus, we increased $|\mathcal{D}|$ from 2 to 3. By using $|\mathcal{D}| = 3$ and the identity function, the synthesis fails, but it succeeds for $|\mathcal{D}|=3$ and the permutation $f_3 = \{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 0\rangle\}$ of order 3 (see row 7). This synthesis succeeds also by using different permutations of order 3, and in all these cases we get programs which are equal to 3-*mutex*-2, modulo a permutation of the values of the shared variable y.

The synthesis of 3-*mutex*-3 (see row 8) is analogous to that of 3-*mutex*-2 to which row 7 refers.

The synthesis of 3-*mutex*-4 fails for $|\mathcal{D}| = 4, 5$, and 6, while it succeeds for $|\mathcal{D}| = 7$ and the permutation $f_4 = \{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 0\rangle, \langle 3, 4\rangle, \langle 4, 5\rangle, \langle 5, 3\rangle, \langle 6, 6\rangle\}$ which is of order 3 (see row 9).

The last rows 10, 11, and 12 of Table 1 refer, respectively, to the programs 4-*mutex*-1, 5-*mutex*-1, and 6-*mutex*-1 whose syntheses succeed for $|\mathcal{D}|=2$ and the identity function.

Table 1: Column 'Program' shows the names of the synthesized programs. k-*mutex*-p is the name of the k-process program satisfying the p behavioural properties shown in column 'Properties'. Column $|\mathcal{D}|$ shows the cardinality of the domain $\{0, 1, \ldots, n\}$ of the shared variable y. Column f shows the k-generating function used for the synthesis. Column $|ans(\Pi)|$ shows the number of answer sets of $\Pi = \Pi_\varphi \cup \Pi_\sigma$. Column 'Time' shows the time expressed in seconds (unless otherwise specified) to generate all answer sets of Π, by using the ASP solver *claspD* [12].

	Program	Properties	$\lvert\mathcal{D}\rvert$	f	$\lvert ans(\Pi)\rvert$	Time
(1)	2-*mutex*-1	*ME*	2	*id*	10	0.01
(2)	2-*mutex*-1	*ME*	2	f_1	10	0.01
(3)	2-*mutex*-2	*ME*, *SF*	2	f_1	2	0.03
(4)	2-*mutex*-3	*ME*, *SF*, *BO*	2	f_1	2	0.05
(5)	2-*mutex*-4	*ME*, *SF*, *BO*, *MR*	3	f_2	2	0.17
(6)	3-*mutex*-1	*ME*	2	*id*	9	0.05
(7)	3-*mutex*-2	*ME*, *SF*	3	f_3	6	3.49
(8)	3-*mutex*-3	*ME*, *SF*, *BO*	3	f_3	4	4.32
(9)	3-*mutex*-4	*ME*, *SF*, *BO*, *MR*	7	f_4	2916	$\approx$ 4.4 hours
(10)	4-*mutex*-1	*ME*	2	*id*	9	0.35
(11)	5-*mutex*-1	*ME*	2	*id*	9	2.89
(12)	6-*mutex*-1	*ME*	2	*id*	9	20.43

In Figure 2 we have presented the synthesized program, called 2-*mutex*-4, for the 2-process mutual exclusion problem described in Example 3. In Figure 3 we present the transition relation of the associated Kripke structure. Program 2-*mutex*-4 is basically the same as Peterson algorithm [27], but, instead of using three shared variables, each of which ranges over a domain of two values, program 2-*mutex*-4 uses two local variables $\mathtt{x}_1$ and $\mathtt{x}_2$ which range over $\{\mathtt{t}, \mathtt{w}, \mathtt{u}\}$, and a single shared variable $\mathtt{y}$ which ranges over $\{0, 1, 2\}$.

The comparison between Peterson algorithm and our program 2-*mutex*-4 is illustrated in Figure 4, where we have presented in the upper part the original Peterson algorithm for two processes and in the lower part our synthesized Peterson-like algorithm derived by hand from the transition relation of program 2-*mutex*-4 depicted Figure 3. Note that in Peterson algorithm the three shared variables are assigned constant values, while in our algorithm we pay the price of using a single shared variable $\mathtt{y}$ by the need of performing some operations on that variable.

However, in Peterson algorithm if a process, say P_1, is in its waiting section and the other process P_2 fails after assigning to q and never does the assignment to s, then P_1 cannot enter its critical section. This problem can be avoided by assuming that the sequence of assignments to q and s is atomic. In our algorithm this problem does not arise simply because we have not a sequence of assignments, but a single assignment to $\mathtt{y}$.

Let us briefly explain our hand derivation of the Peterson-like algorithm from the algorithm described by guarded commands in Figure 2. Let us consider process P_1. We have that:
(i) when P_1 enters the waiting section or leaves the critical section, $\mathtt{y}$ is modified as follows:
$\mathtt{y} := \mathit{if}\ \mathtt{y}{=}2\ \mathit{then}\ 1\ \mathit{else}\ 2$ (see the guarded commands (1), (2), (3), (6), and (7)), and
(ii) when P_1 enters the critical section, $\mathtt{y}$ should be different from $\mathtt{1}$ and the value of $\mathtt{y}$ is not modified (see the guarded commands (4) and (5)).

For process P_2 we replace $\mathtt{1}$ by $\mathtt{0}$.

As indicated in Figure 4 the two assignments to $\mathtt{y}$ (that is, $\mathtt{y} := \mathit{if}\ \mathtt{y} = 2\ \mathit{then}\ 1\ \mathit{else}\ 2$, and $\mathtt{y} := \mathit{if}\ \mathtt{y} = 2\ \mathit{then}\ 0\ \mathit{else}\ 2$) can be expressed as assignments in Kleene 3-valued logic whose values are $\mathtt{0}$, $\mathtt{1}$, and $\mathtt{2}$. In that logic we have that:

(i) $\neg\mathtt{x} =_{def} 2 - \mathtt{x}$,
(ii) $\mathtt{x} \wedge \mathtt{y} =_{def} \min(\mathtt{x}, \mathtt{y})$,
(iii) $\mathtt{x} \vee \mathtt{y} =_{def} \max(\mathtt{x}, \mathtt{y})$, and
(iv) $\mathtt{x} \rightarrow \mathtt{y} =_{def} \mathit{if}\ \mathtt{x} \leq \mathtt{y}\ \mathit{then}\ 2\ \mathit{else}\ 0$.

Note that in that logic $\mathtt{x} \rightarrow \mathtt{y} \neq \neg\mathtt{x} \vee \mathtt{y}$.

	P_1 : *true* → `if`	P_2 : *true* → `if`
(1)	$\mathtt{x}_1{=}\mathtt{t} \wedge \mathtt{y}{=}0 \rightarrow \mathtt{x}_1{:=}\mathtt{w};\ \mathtt{y}{:=}2$	$\mathtt{x}_2{=}\mathtt{t} \wedge \mathtt{y}{=}0 \rightarrow \mathtt{x}_2{:=}\mathtt{w};\ \mathtt{y}{:=}2$
(2)	$[\!]\ \mathtt{x}_1{=}\mathtt{t} \wedge \mathtt{y}{=}1 \rightarrow \mathtt{x}_1{:=}\mathtt{w};\ \mathtt{y}{:=}2$	$[\!]\ \mathtt{x}_2{=}\mathtt{t} \wedge \mathtt{y}{=}1 \rightarrow \mathtt{x}_2{:=}\mathtt{w};\ \mathtt{y}{:=}2$
(3)	$[\!]\ \mathtt{x}_1{=}\mathtt{t} \wedge \mathtt{y}{=}2 \rightarrow \mathtt{x}_1{:=}\mathtt{w};\ \mathtt{y}{:=}1$	$[\!]\ \mathtt{x}_2{=}\mathtt{t} \wedge \mathtt{y}{=}2 \rightarrow \mathtt{x}_2{:=}\mathtt{w};\ \mathtt{y}{:=}0$
(4)	$[\!]\ \mathtt{x}_1{=}\mathtt{w} \wedge \mathtt{y}{=}0 \rightarrow \mathtt{x}_1{:=}\mathtt{u};\ \mathtt{y}{:=}0$	$[\!]\ \mathtt{x}_2{=}\mathtt{w} \wedge \mathtt{y}{=}1 \rightarrow \mathtt{x}_2{:=}\mathtt{u};\ \mathtt{y}{:=}1$
(5)	$[\!]\ \mathtt{x}_1{=}\mathtt{w} \wedge \mathtt{y}{=}2 \rightarrow \mathtt{x}_1{:=}\mathtt{u};\ \mathtt{y}{:=}2$	$[\!]\ \mathtt{x}_2{=}\mathtt{w} \wedge \mathtt{y}{=}2 \rightarrow \mathtt{x}_2{:=}\mathtt{u};\ \mathtt{y}{:=}2$
(6)	$[\!]\ \mathtt{x}_1{=}\mathtt{u} \wedge \mathtt{y}{=}2 \rightarrow \mathtt{x}_1{:=}\mathtt{t};\ \mathtt{y}{:=}1$	$[\!]\ \mathtt{x}_2{=}\mathtt{u} \wedge \mathtt{y}{=}2 \rightarrow \mathtt{x}_2{:=}\mathtt{t};\ \mathtt{y}{:=}0$
(7)	$[\!]\ \mathtt{x}_1{=}\mathtt{u} \wedge \mathtt{y}{=}0 \rightarrow \mathtt{x}_1{:=}\mathtt{t};\ \mathtt{y}{:=}2$	$[\!]\ \mathtt{x}_2{=}\mathtt{u} \wedge \mathtt{y}{=}1 \rightarrow \mathtt{x}_2{:=}\mathtt{t};\ \mathtt{y}{:=}2$
	`fi`	`fi`

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

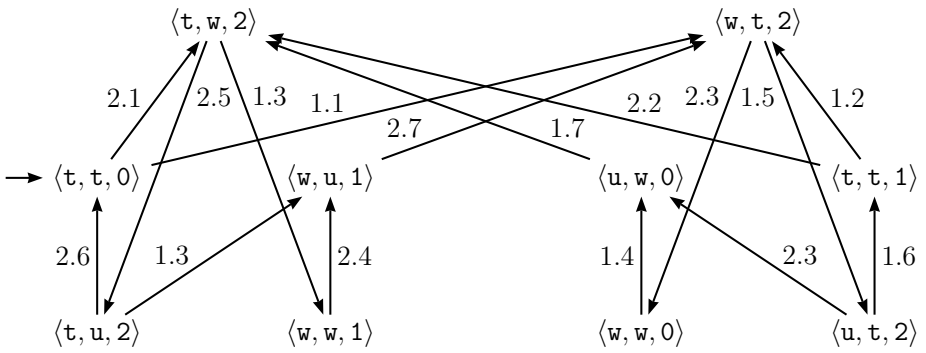

Figure 3: The transition relation of the Kripke structure associated with the 2-process concurrent program 2-*mutex*-4 of Figure 2. An arc $s \xrightarrow{i.n} t$ indicates that the guarded command (n) of process P_i (see Figure 2) causes the transition from state s to state t. The initial state is $\langle \mathtt{t}, \mathtt{t}, 0 \rangle$.

5.2. Comparison of ASP solvers on the synthesis examples

We have implemented our synthesis procedure by using the following ASP solvers:

- *clasp* [11] (`http://potassco.sourceforge.net/`)
- *claspD* [12] (`http://potassco.sourceforge.net/`)
- *cmodels* [19] (`http://www.cs.utexas.edu/~tag/cmodels/`)
- *DLV* [23] (`http://www.dlvsystem.com/dlvsystem/index.php/DLV`)
- *GnT* [21] (`http://www.tcs.hut.fi/Software/gnt/`)
- *smodels* [30] (`http://www.tcs.hut.fi/Software/smodels/`)

The ground instances of Π given as input to *clasp*, *claspD*, *cmodels*, *GnT*, and *smodels*, have been generated by *gringo* (`http://potassco.sourceforge.net/`). All experiments have been performed on an Intel Core 2 Duo E7300 2.66GHz under the Linux operating system. In order to compare the performance of the ASP solvers listed above, we have implemented the synthesis procedure by using the following encodings of Π_σ (i.e., the program which generates the guarded commands): (i) Disjunctive Logic Program (Π_σ^{DLP}), (ii) Normal Logic Program (Π_σ^{NLP}), and (iii) Stratified Choice Integrity constraint Program (Π_σ^{SCI}) [25], and we have executed each solver on each example with a 600 second timeout. All times required to generate the first solution and all solutions are reported in Tables 2, 3, and 4, for Π_σ^{DLP}, Π_σ^{NLP}, and Π_σ^{SCI}, respectively (each table includes only the ASP solvers which are able to deal with the considered encoding).

The experimental results reported in Table 2 show that *claspD* is the ASP solver with the best performances on all synthesis examples obtained by using the Π_σ^{DLP} encoding. Regarding the other solvers we have that *cmodels* provides better timings than *GnT*, but the former crashes when exercised on Programs (2), (6), and (7) where *GnT* succeed.

Concerning the results obtained by using the Π_σ^{NLP} encoding (Table 3), we observe that *clasp* and *claspD*, which is an extension of *clasp* for solving disjunctive logic programs, are the tools that perform better on almost all examples. We also have that they provide approximately the same performances on small instances (i.e., the number of processes) of the synthesis problem (Programs (1) to (6)). However, the performance gap between *clasp* and *claspD* increases as the size of the instance of the synthesis problem increases (Programs (10) and (11)). Despite of worse timing results, *cmodels*, *GnT* and *smodels* succeed in synthesizing Program (12) where

Peterson Algorithm for the 2 processes P_1 and P_2 [27]

$q_1 := \mathit{false}; \quad q_2 := \mathit{false}; \quad s := 1;$

P_1 : **while** *true* **do**	P_2 : **while** *true* **do**
l_1 : non-critical section 1;	m_1 : non-critical section 2;
l_2 : $q_1 := \mathit{true}$; $s := 1$;	m_2 : $q_2 := \mathit{true}$; $s := 2$;
l_3 : **await** $(\neg q_2) \vee (s = 2)$;	m_3 : **await** $(\neg q_1) \vee (s = 1)$;
l_4 : critical section 1;	m_4 : critical section 2;
l_5 : $q_1 := \mathit{false}$; **od**	m_5 : $q_2 := \mathit{false}$; **od**

Synthesized Peterson-like Algorithm for the 2 processes P_1 and P_2 (2-*mutex*-4)

$\mathtt{y} := 0;$

P_1 : **while** *true* **do**	P_2 : **while** *true* **do**
l_1 : non-critical section 1;	m_1 : non-critical section 2;
l_2 : $\mathtt{y} := (\mathtt{y} \rightarrow 1) \vee 1$;	m_2 : $\mathtt{y} := (\mathtt{y} \rightarrow 1)$;
l_3 : **await** $\mathtt{y} \neq 1$;	m_3 : **await** $\mathtt{y} \neq 0$;
l_4 : critical section 1;	m_4 : critical section 2;
l_5 : $\mathtt{y} := (\mathtt{y} \rightarrow 1) \vee 1$; **od**	m_5 : $\mathtt{y} := (\mathtt{y} \rightarrow 1)$; **od**

Figure 4: The original Peterson algorithm for 2 processes (above) compared with our synthesized Peterson-like algorithm for 2 processes 2-*mutex*-4 (below) derived by hand from the transition relation of Figure 3. Implication and disjunctions are performed in Kleene 3-valued logic.

both *clasp* and *claspD* fail (we want also to point out that, however, *claspD* is able to synthesize all solutions for Program (12) in 20.43 second if we consider the disjunctive logic program encoding).

Finally, concerning the results obtained by using Π_σ^{SCI} (4) we have that *clasp* is the ASP solver which performs better and, on large instances (Programs (7)-(12)) it outperforms *smodels*.

6. Related Work and Concluding Remarks

We have proposed a framework based on Answer Set Programming (ASP) for the synthesis of concurrent programs satisfying some given behavioural and structural properties. Behavioural properties are specified by formulas of the Computational Tree Logic (CTL) and structural properties are specified by simple algebraic structures. The desired behavioural and structural properties are encoded as logic programs which are given as input to an ASP solver which, then, computes the answer sets of those programs. Every answer set encodes a concurrent program satisfying the given properties.

Pioneering works on the synthesis of concurrent programs from temporal specifications are those by Clarke and Emerson [7] and Manna and Wolper [24]. In both these works the authors reduce the synthesis problem to the satisfiability problem of the given temporal specifications. Their synthesis methods exploit the finite model property for propositional temporal logics which asserts that if a given formula is satisfiable, then it is satisfiable in a finite model (whose size depends on the size of the formula).

Table 2: Synthesis times using Π_σ^{DLP}. Numbers in column 'Program' refer to the synthesized programs listed in Table 1. Column 'First' and column 'All' show the time expressed in second to generate, respectively, the first answer set and all answer sets of $\Pi = \Pi_\sigma^{DLP} \cup \Pi_\varphi$. ∞ means 'no answer within 600 second'. e means 'tool crashes'.

Program	*claspD*		*cmodels*		*DLV*		*GnT*	
	First	All	First	All	First	All	First	All
(1)	0.01	0.01	0.01	0.02	0.01	0.02	0.02	0.04
(2)	0.01	0.01	0.02	e	0.01	0.09	0.02	0.05
(3)	0.03	0.03	0.06	0.08	0.21	1.18	0.12	0.18
(4)	0.03	0.05	0.07	0.09	0.16	3.19	0.13	0.23
(5)	0.14	0.17	0.26	0.57	∞	∞	0.84	5.92
(6)	0.04	0.05	e	e	0.40	∞	0.12	0.37
(7)	2.57	3.49	e	e	∞	∞	12.65	308.06
(8)	2.82	4.32	4.82	8.14	∞	∞	14.01	361.50
(9)	460.39	∞	544.03	∞	∞	∞	∞	∞
(10)	0.29	0.35	0.61	3.27	73.50	∞	1.17	3.65
(11)	2.07	2.89	3.10	106.04	∞	∞	10.90	71.22
(12)	12.39	20.43	18.37	∞	∞	∞	376.87	∞

In [7] Clarke and Emerson propose the following three-phase method for the synthesis of concurrent programs for a shared-memory model of execution: Phase 1 consists in providing the CTL specification of the concurrent program; Phase 2 consists in applying the tableau-based decision procedure for the satisfiability of CTL formulas to obtain a model of the CTL specification; and Phase 3 consists in extracting the synchronization skeletons from the model of the CTL specification.

Similarly, in [24] Manna and Wolper present a method that uses a tableau-based decision procedure for linear temporal logic (LTL) for the synthesis of synchronization instructions for processes in a message-passing model of execution.

However, the approaches proposed in [7, 24] have some drawbacks. In particular, they suffer from the state space explosion problem in that the models from which the synchronization instructions are extracted, have sizes which are exponential with respect to the number of processes. Moreover, the synthesized instructions work for models of computation which require further refinements for their use in a realistic architecture. Extensions of the synthesis methods of [7, 24] have been proposed by Attie and Emerson in [2] to deal with the state space explosion problem and allow an arbitrarily large number of processes by exploiting similarities among them. Also Attie and Emerson in [3] present an extension of their synthesis method to deal with a finer, more realistic atomicity of instructions so that only read and write operations are required to be atomic.

The papers we have considered so far refer to the synthesis of the so called closed systems, that is, the synthesis of programs whose processes are all specified by some given formulas. A different approach to the synthesis of concurrent programs has been presented by Pnueli and Rosner in [28]. These authors propose a method for synthesizing reactive modules of so called open systems, that is, systems in which the designer has no control over the inputs which come

Table 3: Synthesis times using Π_σ^{NLP}. Numbers in column 'Program' refer to the synthesized programs listed in Table 1. Column 'First' and column 'All' show the time expressed in seconds to generate, respectively, the first answer set and all answer sets of $\Pi = \Pi_\sigma^{NLP} \cup \Pi_\varphi$. ∞ means 'no answer within 600 second'.

Program	*clasp*		*claspD*		*cmodels*		*DLV*		*GnT*		*smodels*	
	First	All	First	All	First	All	First	All	First	All	First	All
(1)	0.02	0.02	0.01	0.01	0.02	0.02	0.01	0.02	0.02	0.03	0.02	0.02
(2)	0.01	0.01	0.01	0.02	0.01	0.02	0.02	0.04	0.02	0.03	0.01	0.01
(3)	0.03	0.04	0.03	0.04	0.06	0.08	0.09	0.15	0.09	0.09	0.04	0.04
(4)	0.04	0.04	0.04	0.04	0.08	0.08	0.13	0.20	0.11	0.11	0.50	0.60
(5)	0.13	0.16	0.15	0.19	0.24	0.31	∞	∞	0.93	1.06	0.26	0.31
(6)	0.05	0.07	0.04	0.06	0.06	0.09	0.77	∞	0.12	0.17	0.07	0.09
(7)	2.11	3.08	3.34	4.38	2.94	12.21	∞	∞	32.90	36.51	5.56	18.86
(8)	4.90	6.23	3.30	6.29	7.18	18.90	∞	∞	89.79	96.79	12.76	61.22
(9)	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
(10)	0.29	1.39	0.31	0.70	0.47	0.74	126.21	∞	0.84	1.61	0.63	1.27
(11)	2.48	40.57	2.34	12.60	3.07	6.57	∞	∞	11.84	41.63	18.74	31.10
(12)	∞	∞	∞	∞	18.23	55.18	∞	∞	158.02	442.69	359.58	596.53

Table 4: Synthesis times using Π_σ^{SCI}. Numbers in column 'Program' refer to the synthesized programs listed in Table 1. Column 'First' and column 'All' show the time expressed in seconds to generate, respectively, the first answer set and all answer sets of $\Pi = \Pi_\sigma^{SCI} \cup \Pi_\varphi$. ∞ means 'no answer within 600 second'. $\perp$ means 'no models at all'. (*) means '*GnT* is able to generate one out of two models'. e means 'tool crashes'.

Program	*clasp*		*claspD*		*cmodels*		*GnT*		*smodels*	
	First	All	First	All	First	All	First	All	First	All
(1)	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.06	0.01	0.04
(2)	0.01	0.01	0.01	0.01	0.01	0.03	0.03	0.06	0.01	0.04
(3)	0.04	0.05	0.03	0.04	0.06	0.07	$\perp$	$\perp$	0.04	0.04
(4)	0.04	0.04	0.06	0.06	0.07	0.08	0.16	0.21	0.05	0.06
(5)	0.13	0.17	0.13	0.18	0.25	0.58	0.36	(*)	0.72	4.27
(6)	0.05	0.04	0.05	0.06	0.06	0.12	$\perp$	$\perp$	0.90	0.13
(7)	1.33	2.10	2.35	3.83	3.17	e	$\perp$	$\perp$	135.63	267.07
(8)	2.66	4.28	3.61	5.69	e	e	$\perp$	$\perp$	406.59	∞
(9)	444.14	∞	∞	∞	262.45	∞	∞	∞	∞	∞
(10)	0.30	0.34	0.30	1.70	0.48	3.20	1.60	4.49	0.95	1.59
(11)	2.40	2.89	2.25	26.45	3.19	106.34	22.15	91.42	20.76	105.15
(12)	22.62	25.61	18.71	406.76	19.28	∞	598.60	∞	499.96	∞

from an external environment. They introduce an automata-based synthesis procedure from a specification given as a linear temporal logic formula. The synthesis of open systems has also been studied by Kupferman and Vardi in [22]. Also the method they propose is based on automata-theoretical techniques. Paper [22] is important because it also presents some basic complexity results for the synthesis problems when specifications are given by CTL formulas or LTL formulas.

Our synthesis procedure follows the lines of [2, 7, 24] and considers concurrent programs to be closed systems. The advantage of our method resides in the fact that we solve the synthesis problem in a purely declarative manner. We reduce the problem of synthesizing a concurrent program to the problem of finding the answer sets of a logic program without the need for any *ad hoc* algorithm. Moreover, besides temporal properties, we can specify for the programs to be synthesized, some structural properties, such as various symmetry properties. Then, our ASP program automatically synthesizes concurrent programs which satisfy the desired properties. In order to reduce the search space when solving the synthesis problem, we have used the notion of symmetric concurrent programs which is similar to the one which was introduced in [2] to overcome the state space explosion problem. Our notion of symmetry is formalized within group theory, similarly to what has been done in [15] for the problem of model checking.

To the best of our knowledge, there is only one paper [20] by Heymans, Nieuwenborgh and Vermeir who make use, as we do, of Answer Set Programming for the synthesis of concurrent programs. The authors of [20] have extended the ASP paradigm by adding preferences among models and they have realized an answer set system, called OLPS. They perform the synthesis of concurrent programs following the approach proposed in [7] and, in particular, they use OLPS for Phase 2 of the synthesis procedure, having reduced the satisfiability problem of CTL formulas to the problem of constructing the answer sets of logic programs. The encoding proposed by [20] yields a synthesis procedure with NEXPTIME time complexity and, thus, it is not optimal because the complexity of the problem of CTL satisfiability is EXPTIME [14].

On the contrary, our technique for reducing the satisfiability problem to the construction of the answer sets of logic programs, does not require any extension of the ASP paradigm. Indeed, we use standard ASP solvers, such as *claspD* [12], and every phase of our synthesis procedure is fully automatic. In particular, from any answer set we can mechanically derive the guarded commands which, by construction, guarantee that the synthesized program satisfies the given behavioural and structural properties. Moreover, we show that our method has optimal time complexity because it has EXPTIME complexity with respect to the size of the temporal specification.

In practice our approach works for synthesizing k-process concurrent programs with a limited number k of processes because the grounding phase needed to compute the answer sets, requires very large memory space for large values of k. As a future work we plan to explore various techniques for reducing both the search space of the synthesis procedure and the impact of the grounding phase on the memory requirements. Among these techniques we envisage to apply those used in the compositional model checking technique [9].

Acknowledgements

We thank Ludwik Czaja and Andrzej Skowron for the invitation to submit to Fundamenta Informaticae our contribution to the Concurrency, Specification, and Programming Workshop held in Pułtusk, Poland, 28–30 September 2011. Many thanks also to the anonymous referees for their helpful comments and constructive criticism.

A. Proofs

We first introduce the following notions which will be used in the proofs.

A nonempty set I of ground atoms is *elementary* [16] for a program *ground*(Π) if for all nonempty proper subsets S of I there exists a rule r in *ground*(Π) such that: (i) $H(r)\cap S\neq\emptyset$, (ii) $B^+(r)\cap(I-S)\neq\emptyset$, (iii) $H(r)\cap(I-S)=\emptyset$, and (iv) $B^+(r)\cap S=\emptyset$. A program *ground*(Π) is said to be *Head Elementary set Free* (HEF, for short) if, for every rule r in *ground*(Π), there is no elementary set Z for *ground*(Π) such that $|H(r)\cap Z|>1$. We say that Π is HEF if *ground*(Π) is HEF. With any given HEF program Π we associate a normal logic program Π^n obtained from Π by replacing every rule r of Π of the form:

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

for some $k>1$, by the following k normal rules:

$a_j \leftarrow \bigwedge_{i\in\{1,\ldots,k\}-\{j\}} \texttt{not}\, a_i \wedge\ a_{k+1} \wedge \ldots \wedge a_m \wedge\ \texttt{not}\, a_{m+1} \wedge \ldots \wedge \texttt{not}\, a_n$

for $j=1,\ldots,k$. It can be shown that $ans(\Pi) = ans(\Pi^n)$ [16].

The following Proposition A.1 is required for the proofs of Theorem 4.1 and Theorem 4.2.

Proposition A.1. The logic program Π_σ is Head Elementary set Free.

Proof 1. We assume by contradiction that there exists a rule r in *ground*(Π_σ) and there exists a set Z which is an elementary set for *ground*(Π_σ) such that $|H(r)\cap Z|>1$. If $|H(r)\cap Z|>1$, then either:
(i) r is an instance of Rule 1.1 of Definition 8 and there exist $l\in\mathcal{L}$, $d\in\mathcal{D}$ such that $\{enabled(1,l,d),\ disabled(1,l,d)\}\subseteq Z$, or
(ii) r is an instance of Rule 1.2 of Definition 8 and there exist $l,l',l''\in\mathcal{L}$, $d,d',d''\in\mathcal{D}$ such that $\{gc(1,l,d,l',d'), gc(1,l,d,l'',d'')\}\subseteq Z$.
Let us consider Case (i). Let S be a nonempty proper subset of Z such that $\{enabled(1,l,d)\}\in S$ and $\{disabled(1,l,d)\}\notin S$. Clearly, $H(r)\cap(Z-S)\neq\emptyset$. This contradicts Condition (iii) for Z to be an elementary set for *ground*(Π_σ).

Case (ii) is analogous to Case (i). Thus, we get that *ground*(Π_σ) is HEF and, by definition, also Π_σ is HEF.

By this proposition and the fact that the transformation from Π into Π^n presented above, preserves the answer set semantics when applied to HEF programs [16], we have that $ans(\Pi_\sigma) = ans(\Pi^n_\sigma)$, where program Π^n_σ is obtained from program Π_σ as follows:
(i) Rule 1.1 of program Π_σ is replaced by the following two normal rules:

$enabled(1,X_1,Y) \leftarrow \texttt{not}\ disabled(1,X_1,Y) \wedge reachable(\langle X_1,\ldots,X_k,Y\rangle)$
$disabled(1,X_1,Y) \leftarrow \texttt{not}\ enabled(1,X_1,Y) \wedge reachable(\langle X_1,\ldots,X_k,Y\rangle)$, and

(ii) Rule 1.2 of program Π_σ is replaced by m normal rules, for $i=1,\ldots,m$, each of which is of the form:

$$gc(1,X,Y,X_i,Y_i) \leftarrow \bigwedge_{j\in\{1,\ldots,m\}-\{i\}} \texttt{not}\ gc(1,X,Y,X_j,Y_j) \wedge enabled(1,X,Y) \wedge candidates(X,Y,[\langle X_1,Y_1\rangle,\ldots,\langle X_m,Y_m\rangle]).$$

From the fact that $\Pi_\varphi \rhd \Pi_\sigma$ and $ans(\Pi_\sigma)=ans(\Pi^n_\sigma)$, we get that (see end of Section 2):
$ans(\Pi)=ans(\Pi_\varphi\cup\Pi_\sigma)=\bigcup_{M\in ans(\Pi_\sigma)} ans(\Pi_\varphi\cup\overleftarrow{M})=\bigcup_{M\in ans(\Pi^n_\sigma)} ans(\Pi_\varphi\cup\overleftarrow{M})=ans(\Pi_\varphi\cup\Pi^n_\sigma)$.

Therefore, in order to compute all answer sets of program $\Pi_\varphi\cup\Pi_\sigma$, we can give $\Pi_\varphi\cup\Pi^n_\sigma$ as input to an answer set solver which does not support disjunctive logic programs.

Proof of Theorem 4.1

In order to prove Theorem 4.1 we first introduce the following Lemma A.2 stating the correctness of the logic program Π_φ (see Definition 9).

Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \lambda \rangle$ be a Kripke structure. We introduce the program $P_\mathcal{K}$, called encoding of $\mathcal{K}$, consisting of $ground(\Pi_\varphi - \{\text{Rule 1}\}) \cup \{tr(s,t) \leftarrow | \ \langle s,t \rangle \in \mathcal{R}\}$. □

Note that, $P_\mathcal{K}$ is a *locally stratified* normal program, and thus it has a unique *stable model* [1] which coincides with its unique answer set $M|_{\{sat,satpath,elem,tr\}}$.

Lemma A.2 (Correctness of Π_φ) Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \lambda \rangle$ be a Kripke structure and $P_\mathcal{K}$ be the encoding of $\mathcal{K}$. For all states $s \in \mathcal{S}$ and CTL formulas φ we have that: $\mathcal{K}, s \vDash \varphi$ iff $sat(s,\varphi) \in M(P_\mathcal{K})$ □

Proof of A.2

The proof is by structural induction on φ. By induction hypothesis we assume that, for all states $s \in D$ and for all proper subformulas ψ of φ

$\mathcal{K}, s \models \psi$ iff $sat(s,\psi) \in M(P_\mathcal{K})$ (†)

Now we consider the following cases.

Case 1. (φ is an elementary proposition e of the form $local(P_i, l)$ or of the form $shared(d)$)
For all states $s \in \mathcal{S}$ we have that:
$\mathcal{K}, s \models e$
iff $s(\mathtt{x}_i) = l$ (or $s(\mathtt{y}) = d$) (by Point (iv) of Def. 3)
iff $elem(s,e) \in M(P_\mathcal{K})$ (by Points (i) and (ii) of Def. 9 and def. of $M(P_\mathcal{K})$)
iff $sat(s,e) \in M(P_\mathcal{K})$ (by Rule 2 of Π_φ and def. of $M(P_\mathcal{K})$)

Case 2. (φ is $\neg\psi$)
For all states $s \in \mathcal{S}$ we have that:
$\mathcal{K}, s \models \neg\psi$
iff $\mathcal{K}, s \models \psi$ does not hold (by def. of $\mathcal{K}, s \models \neg\psi$, see Sect. 2.3)
iff $sat(s,\psi) \notin M(P_\mathcal{K})$ (by (†))
iff $sat(s,\neg\psi) \in M(P_\mathcal{K})$ ($M(P_\mathcal{K})$ is an answer set of $P_\mathcal{K}$)

Case 3. (φ is $\psi_1 \wedge \psi_2$)
For all states $s \in \mathcal{S}$ we have that:
$\mathcal{K}, s \models \psi_1 \wedge \psi_2$
iff $\mathcal{K}, s \models \psi_1$ and $\mathcal{K}, s \models \psi_2$ (by def. of $\mathcal{K}, s \models \psi_1 \wedge \psi_2$, see Sect. 2.3)
iff $sat(s,\psi_1) \in M(P_\mathcal{K})$ and $sat(s,\psi_2) \in M(P_\mathcal{K})$ (by (†))
iff $sat(s,\psi_1 \wedge \psi_2) \in M(P_\mathcal{K})$ (by Rule 4 of Π_φ and $M(P_\mathcal{K})$ is an answer set of $P_\mathcal{K}$)

Case 4. (φ is $\mathsf{EX}\,\psi$)
For all states $s \in \mathcal{S}$ we have that:
$\mathcal{K}, s \models \mathsf{EX}\,\psi$
iff there exists a state $s' \in \mathcal{S}$ such that
$\langle s, s' \rangle \in \mathcal{R}$ and $\mathcal{K}, s' \models \psi$ (by def. of $\mathcal{K}, s \vDash \mathsf{EX}\,\psi$, see Sect. 2.3)
iff there exists a state $s' \in \mathcal{S}$ such that:
(i) $tr(s,s') \in M(P_\mathcal{K})$, and
(ii) $sat(s',\psi) \in M(P_\mathcal{K})$ (by def. of $P_\mathcal{K}$ and (†))

iff there exist a state $s' \in \mathcal{S}$ and
a clause of the form $sat(s, ex(\psi)) \leftarrow t(s, s') \wedge sat(s', \psi)$ in $P_{\mathcal{K}}$ such that:
(i) $t(s, s') \in M(P_{\mathcal{K}})$, and
(ii) $sat(s', \psi) \in M(P_{\mathcal{K}})$ (by def. of $P_{\mathcal{K}}$)
iff $sat(s, ex(\psi)) \in M(P_{\mathcal{K}})$ (by Rule 5 of $P_{\mathcal{K}}$ and $M(P_{\mathcal{K}})$ is an answer set of $P_{\mathcal{K}}$)

Case 5. (φ is $\mathsf{EU}[\psi_1, \psi_2]$)
For any set Y of states, let $\tau_{\mathsf{EU}}(Y)$ denote the set $\{s \in \mathcal{S} \,|\, \mathcal{K}, s \models \psi_2\} \cup (\{s \in \mathcal{S} \mid \mathcal{K}, s \models \psi_1\} \cap \{s \in \mathcal{S} \mid \exists s' \in \mathcal{S}$ such that $\langle s, s' \rangle \in \mathcal{R}$ and $s' \in Y\})$. From [8] we have that $\mathcal{K}, s \models \mathsf{EU}[\psi_1, \psi_2]$ holds iff $s \in lfp(\tau_{\mathsf{EU}})$, where $lfp(\tau_{\mathsf{EU}}) = \tau^{\omega}_{\mathsf{EU}}$ Thus, we have to show that, for all states $s \in \mathcal{S}$, $s \in lfp(\tau_{\mathsf{EU}})$ iff $sat(s, eu(\psi_1, \psi_2)) \in M(P_{\mathcal{K}})$. Let P_{EU} be $ground(\{$Rule 6, Rule 7$\}) \cup \{sat(s, \psi_1) \in M(P_{\mathcal{K}}) \mid s \in \mathcal{S}\} \cup \{sat(s, \psi_2) \in M(P_{\mathcal{K}}) \mid s \in \mathcal{S}\} \cup \{tr(s, t) \in M(P_{\mathcal{K}}) \mid s, t \in \mathcal{S}\}$, and T^h_{EU} be the immediate consequence operator [1]. We proceed by induction on h: for all $h \geq 0$, for all $s \in \mathcal{S}$, $s \in \tau^h_{\mathsf{EU}}(\emptyset)$ iff $sat(s, eu(\psi_1, \psi_2)) \in T^h_{\mathsf{EU}}(\emptyset)$. The base case trivially holds because $\tau^0_{\mathsf{EU}}(\emptyset) = \emptyset = \tau^h_{\mathsf{EU}}(\emptyset)$. Now, we assume the following inductive hypothesis:
for all $s \in \mathcal{S}$, $s \in \tau^h_{\mathsf{EU}}(\emptyset)$ iff $sat(s, eu(\psi_1, \psi_2)) \in T^h_{\mathsf{EU}}(\emptyset)$ (††)
and we prove that, for all $s \in \mathcal{S}$, $s \in \tau^{h+1}_{\mathsf{EU}}(\emptyset)$ iff $sat(s, eu(\psi_1, \psi_2)) \in T^h_{\mathsf{EU}}(\emptyset)$.
We have that:
$s \in \tau^{h+1}_{\mathsf{EU}}(\emptyset)$
iff *either* $\mathcal{K}, s \models \psi_2$ (by def. of τ_{EU})
or $\mathcal{K}, s \models \psi_1$ and there exists a state $s' \in \mathcal{S}$ such that
$\langle s, s' \rangle \in \mathcal{R}$ and $s' \in \tau^h_{\mathsf{EU}}(\emptyset)$ (by def. of τ_{EU})
iff *either* $sat(s, \psi_2) \in T^h_{\mathsf{EU}}(\emptyset)$ (by (††))
or $sat(s, \psi_1) \in T^h_{\mathsf{EU}}(\emptyset)$ and there exists a state $s' \in \mathcal{S}$ such that
$\langle s, s' \rangle \in \mathcal{R}$ and $sat(s', eu(\psi_1, \psi_2) \in T^h_{\mathsf{EU}}(\emptyset)$ (by (††))
iff there exists a clause $\gamma \in P_{\mathcal{K}}$ such that:
either
(i) γ is of the form $sat(s, eu(\psi_1, \psi_2)) \leftarrow sat(s, \psi_2)$, and
(ii) $sat(s, \psi_2) \in T^h_{\mathsf{EU}}(\emptyset)$ ($T^h_{\mathsf{EU}}(\emptyset)$ is a model of P_{EU})
or there exists a state $s' \in \mathcal{S}$ such that:
(i) γ is of the form $sat(s, eu(\psi_1, \psi_2)) \leftarrow sat(s, \psi_1) \wedge t(s, s') \wedge sat(s', eu(\psi_1, \psi_2))$,
(ii) $sat(s, \psi_1) \in T^h_{\mathsf{EU}}(\emptyset)$,
(iii) $t(s, s') \in T^h_{\mathsf{EU}}(\emptyset)$, and
(iv) $sat(s', eu(\psi_1, \psi_2)) \in T^h_{\mathsf{EU}}(\emptyset)$ (by def. of P_{EU} and $T^h_{\mathsf{EU}}(\emptyset)$ is a model of P_{EU})
iff $sat(s, eu(\psi_1, \psi_2)) \in T^h_{\mathsf{EU}}(\emptyset)$ ($T^h_{\mathsf{EU}}(\emptyset)$ is a model of P_{EU})

Case 6. (φ is $\mathsf{EG}(\psi)$)
In order to prove this case we make use of the following Proposition A.3 [8]. Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \lambda \rangle$ be a Kripke structure and $\mathcal{K}' = \langle \mathcal{S}', \mathcal{S}'_0, \mathcal{R}', \lambda' \rangle$ be the Kripke structure obtained from $\mathcal{K}$ as follows: $\mathcal{S}' = \{s \in S \mid \mathcal{K}, s \vDash \psi\}$, $\mathcal{S}'_0 = \{s \in S_0 \mid \mathcal{K}, s \vDash \psi\}$, $\mathcal{R}' = \mathcal{R} \mid_{\mathcal{S}' \times \mathcal{S}'}$, and $\lambda' = \lambda \mid_{S'}$.

Proposition A.3. $\mathcal{K}, s \vDash \mathsf{EG}\,\psi$ iff (i) $s_1 \in S'$, and (ii) there exists a path in $\mathcal{K}'$ that leads from s_1 to some node s_k in a nontrivial strongly connected component of the graph $\langle \mathcal{S}', \mathcal{R}' \rangle$.

Now, let us consider the program $P_{\mathcal{K}}$ constructed as indicated at the beginning of this section. We prove that conditions (i) and (ii) of Lemma A.3 hold iff $sat(s, eg(\psi)) \in M(P_{\mathcal{K}})$:
$sat(s_1, eg(\psi)) \in M(P_{\mathcal{K}})$ iff (by Rule 8)

there exists a state $s_k \in S'$ such that
$\{\ satpath(s_1, s_k, \psi),\ satpath(s_k, s_k, \psi)\ \} \subseteq M(P_{\mathcal{K}})$ iff (by def. of $\mathcal{P}_{\mathcal{K}}$)
$\{\ satpath(s_1, s_k, \psi) \leftarrow sat(s_1, \psi) \wedge tr(s_1, s_2) \wedge satpath(s_2, s_k, \psi),$
$satpath(s_2, s_k, \psi) \leftarrow sat(s_2, \psi) \wedge tr(s_2, s_3) \wedge satpath(s_3, s_k, \psi),$
$\ldots$
$satpath(s_{k-2}, s_k, \psi) \leftarrow sat(s_{k-2}, \psi) \wedge tr(s_{k-2}, s_{k-1}) \wedge satpath(s_{k-1}, s_k, \psi),$
$satpath(s_{k-1}, s_k, \psi) \leftarrow sat(s_{k-1}, \psi) \wedge tr(s_{k-1}, s_k)\ \} \subseteq P_{\mathcal{K}}$ iff
the following conditions holds:
(a) $\{\ sat(s_1, \psi), sat(s_2, \psi), \ldots, sat(s_{k-1}, \psi)\ \} \subseteq M(P_{\mathcal{K}})$ (by (†))
(b) $\{\ tr(s_1, s_2), \ldots, tr(s_{k-1}, s_k)\ \} \subseteq M(P_{\mathcal{K}})$ (by def. of $\mathcal{P}_{\mathcal{K}}$)
(c) $\{\ satpath(s_k, s_k, \psi)\ \} \subseteq M(P_{\mathcal{K}'})$
iff the following conditions holds:
(a) $\{\ sat(s_1, \psi), sat(s_2, \psi), \ldots, sat(s_{k-1}, \psi)\ \} \subseteq M(P_{\mathcal{K}})$ (by (†))
(b) $\{\ tr(s_1, s_2), \ldots, tr(s_{k-1}, s_k)\ \} \subseteq M(P_{\mathcal{K}})$ (by def. of $\mathcal{P}_{\mathcal{K}}$)
(c) $\{\ satpath(s_k, s_k, \psi) \leftarrow sat(s_k, \psi) \wedge tr(s_k, s_{k_1}) \wedge satpath(s_{k_1}, s_k, \psi),$
$satpath(s_{k_1}, s_k, \psi) \leftarrow sat(s_{k_1}, \psi) \wedge tr(s_{k_1}, s_{k_2}) \wedge satpath(s_{k_2}, s_k, \psi),$
$\ldots$
$satpath(s_{k_{n-2}}, s_k, \psi) \leftarrow sat(s_{k_{n-2}}, \psi) \wedge tr(s_{k_{n-2}}, s_{k_{n-1}}) \wedge satpath(s_{k_{n-1}}, s_n, \psi),$
$satpath(s_{k_{n-1}}, s_k, \psi) \leftarrow sat(s_{k_{n-1}}, \psi) \wedge tr(s_{k_{n-1}}, s_k)\ \} \subseteq P_{\mathcal{K}}$
iff the following conditions holds:
(a) $\{\ sat(s_1, \psi), sat(s_2, \psi), \ldots, sat(s_{k-1}, \psi)\ \} \subseteq M(P_{\mathcal{K}})$ (by (†))
(b) $\{\ tr(s_1, s_2), \ldots, tr(s_{k-1}, s_k)\ \} \subseteq M(P_{\mathcal{K}})$ (by def. of $\mathcal{P}_{\mathcal{K}}$)
(c') $\{\ sat(s_{k_1}, \psi), sat(s_{k_2}, \psi), \ldots, sat(s_{k_{n-1}}, \psi)\ \} \subseteq M(P_{\mathcal{K}})$ (by (†))
(c") $\{\ tr(s_k, s_{k_1}), tr(s_{k_1}, s_{k_2}), \ldots, tr(s_{k_{n-1}}, s_k)\ \} \subseteq M(P_{\mathcal{K}})$ (by def. of $\mathcal{P}_{\mathcal{K}}$)
iff there exists a path $\pi = s_1, \ldots, s_k$ of length $k \geq 1$ and s_k belongs to a nontrivial strongly connected component of $\langle \mathcal{S}', \mathcal{R}' \rangle$. □

Now we prove Theorem 4.1.

Let Π be the program $\Pi_\varphi \cup \Pi_\sigma$. We need the following notation. Given a set P of predicate symbols and a set M of atoms, we define $M|_P$ to be the set $\{A \in M \mid$ the predicate of A is in $P\}$.

(*if.* Soundness) Let M be an answer set of Π. Recall that σ is of the form $\langle f, T, l_0, d_0 \rangle$. Let us consider a command C of the form: $\mathtt{x}_1 := l_0; \ldots; \mathtt{x}_k := l_0;\ \mathtt{y} := d_0;\ \mathtt{do}\ P_1 \,[\!]\, \ldots \,[\!]\, P_k\ \mathtt{od}$, where for $i = 1, \ldots, k$, $\big(\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d'\big)$ is in P_i iff $gc(i, l, d, l', d') \in M$.
We have the following two properties of C.

(CP1) For $i = 1, \ldots, k$, every guarded command in P_i is of the form $\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d'$ with $\langle l, d \rangle \neq \langle l', d' \rangle$. Indeed, M is a model of Π_σ and, in particular, of the ground facts defining the predicate *candidates* (see Definition 8).

(CP2) For $i = 1, \ldots, k$, the guards of any two guarded commands of process P_i are mutually exclusive. Indeed, the following holds. By Proposition A.1, Π_σ is HEF. Hence, by Rule 2.1, for every $l \in \mathcal{L}$ and $d \in \mathcal{D}$, at most one atom of the form $gc(1, l, d, l', d')$ belongs to M. Since M is a supported model [6], by Rule 2.*i*, for $i = 2, \ldots, k$, we get that $gc(i, l, f(d), l', f(d')) \in M$ iff $gc(i-1, l, d, l', d') \in M$. By using this fact we get that, for $i = 1, \ldots, k$, for every $l \in \mathcal{L}$ and $d \in \mathcal{D}$, at most one atom of the form $gc(i, l, d, l', d')$ belongs to M.

By Properties (CP1) and (CP2), C is a k-process concurrent program (see Definition 1).

Now, we prove that: (i) C satisfies φ and (ii) C is symmetric w.r.t. σ.

Point (i). Let $\mathcal{K} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \lambda \rangle$ be the Kripke structure associated with C, constructed as indicated in Definition 3 and $P_{\mathcal{K}}$ be the encoding of $\mathcal{K}$. We have that:

M is an answer set of Π

iff M is an answer set of $\Pi - \{\leftarrow \texttt{not}\ sat(s_0, \varphi)\}$ and $sat(s_0, \varphi) \in M$
(by def. of integrity constraint)

iff $M = M(\Pi_\varphi - \{\leftarrow \texttt{not}\ sat(s_0, \varphi)\} \cup M\ |_{\{gc, enabled, disabled, reachable, tr, candidates, perm\}})$
(by def. of $\Pi_\varphi \cup \Pi_\sigma$)

iff $M = M(\mathcal{P}_k) \cup M\ |_{\{gc, enabled, disabled, reachable, tr, candidates, perm\}})$
(by def. of $P_{\mathcal{K}}$)

iff $M\ |_{\{sat, satpath, tr, elem\}} = M(\mathcal{P}_k)$.

Since $sat(s, \varphi) \in M$, we obtain that $sat(s, \varphi) \in M(\mathcal{P}_k)$ and, thus, $\mathcal{K}, s \vDash \varphi$.

Point (ii). By construction, C is of the form $\mathtt{x}_1{:=}l_0; \ldots; \mathtt{x}_k{:=}l_0; \mathtt{y}{:=}d_0;$ do $P_1 \,[\!]\, \ldots \,[\!]\, P_k$ od. Let us now prove that Conditions (i) and (ii) of Definition 6 hold.

For all $gc(i, l, d, l', d') \in M$ we have that the pair $\langle l', d' \rangle$ belongs to the list L which is the third argument of $candidates(l, d, L)$. By Point (i) of Definition 8, for every pair $\langle l', d' \rangle$ in L we have that $\langle l, l' \rangle \in T$ and, therefore, C satisfies Condition (i) of Definition 6.

Since M is a supported model of $ground(\Pi)^M$ and Rule 1.i, for $1 < i \leq k$, is the only rule in Π whose head is unifiable with $gc(i, l, d, l', d')$ we have that $gc(i-1, l, d, l', d') \in M$ iff $gc(i, l, f(d), l', f(d')) \in M$. Thus, Condition (ii) of Definition 6 holds for C because f is a permutation of order k.

(*only if*. Completeness) Let C be a k-process concurrent program which satisfies φ and is symmetric w.r.t. σ, and $\mathcal{K}$ be the Kripke structure $\langle \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \lambda \rangle$ associated with C whose processes are $P_1, \ldots, P_k$. We have to prove that there exists an answer set $M \in ans(\Pi_\varphi \cup \Pi_\sigma)$ which encodes C. Let M be defined as follows.

$$
\begin{array}{lll}
M = & \{reachable(s) \mid s \in \mathcal{S}\} & (M.1) \\
\cup & \{tr(s, s') \mid \langle s, s' \rangle \in \mathcal{R}\} & (M.2) \\
\cup & \{gc(i, l, d, l', d') \mid \big(\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d'\big) \text{ is in } P_i \wedge 1 \leq i \leq k\} & (M.3) \\
\cup & \{enabled(i, l, d) \mid \exists\, l', d'\ \big(\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d'\big) \text{ is in } P_i \wedge 1 \leq i \leq k\} & (M.4) \\
\cup & \{disabled(1, s(\mathtt{x}_1), s(\mathtt{y})) \mid s \in \mathcal{S} \wedge \neg\exists\, c\ \big(\mathtt{x}_1 = s(\mathtt{x}_1) \wedge \mathtt{y} = s(\mathtt{y}) \rightarrow c\big) \text{ is in } P_1\} & (M.5) \\
\cup & \{sat(s, \psi) \mid s \in \mathcal{S} \wedge \mathcal{K}, s \vDash \psi\} & (M.6) \\
\cup & \{satpath(s_0, s_n, \psi) \mid \exists \langle s_0, \ldots, s_n \rangle\ \forall i\ (0 \leq i \leq n \rightarrow \mathcal{K}, s_i \vDash \psi)\ \} & (M.7) \\
\cup & \{elem(p, s) \mid s \in \mathcal{S} \wedge p \in \lambda(s)\} & (M.8) \\
\cup & \{perm(d, d') \mid d, d' \in \mathcal{D} \wedge f(d) = d'\} & (M.9) \\
\cup & \{candidates(l, d, L(l, d)) \leftarrow \mid\ l \in \mathcal{L} \wedge d \in \mathcal{D}\} & (M.10)
\end{array}
$$

where $L(l, d)$ is any list representing the set $\{\langle l', d' \rangle \mid \langle l, l' \rangle \in T\ \wedge\ d' \in \mathcal{D}\ \wedge\ \langle l, d \rangle \neq \langle l', d' \rangle\}$ of pairs.

By M.3 and Definition 4.4 we have that M encodes C. Now we prove that M is an answer set of Π, that is, (i) M is a model of $ground(\Pi_\varphi \cup \Pi_\sigma)^M$ and (ii) M is a minimal such model.

(i) We prove that for every rule $r \in ground(\Pi_\varphi \cup \Pi_\sigma)^M$ if $B^+(r) \subseteq M$ then $H(r) \cap M \neq \emptyset$. We proceed by cases. Let us first consider the rules in $ground(\Pi_\sigma)$.

(Rule 1.1) Assume that r is $enabled(1, l_1, d) \vee disabled(1, l_1, d) \leftarrow reachable(\langle l_1, \ldots, l_k, d \rangle)$. If $reachable(\langle l_1, \ldots, l_k, d \rangle) \in M$ then, by M.1, we have that $\langle l_1, \ldots, l_k, d \rangle \in \mathcal{S}$. Since $\mathcal{R}$ is a total relation, either P_1 is enabled in $\langle l_1, \ldots, l_k, d \rangle$ and consequently, by M.4, $enabled(1, l_1, d) \in M$, or

it is not enabled and thus, by $M.5$, $\mathit{disabled}(1, l_1, d) \in M$.

(Rule 1.i, $i > 1$) Assume that r is $\mathit{enabled}(i, l, d) \leftarrow \mathit{gc}(i, l, d, l', d')$. If $\mathit{gc}(i, l, d, l', d') \in M$ then, by $M.3$, $\big(\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d'\big)$ is in P_i, and consequently, by $M.4$, $\mathit{enabled}(i, l, d) \in M$.

(Rule 2.1) Assume that r is of the form $\mathit{gc}(1, l, d, l_1, d_1) \vee \ldots \vee \mathit{gc}(1, l, d, l_m, d_m) \leftarrow \mathit{enabled}(1, l, d) \wedge \mathit{candidates}(l, d, [\langle l_1, d_1\rangle, \ldots, \langle l_m, d_m\rangle])$ for some $m \geq 1$. If $\mathit{enabled}(1, l, d) \in M$ then, by $M.4$, there exists in P_1 a guarded command whose guard is $\mathtt{x}_1 = l \wedge \mathtt{y} = d$ and the associated command is encoded as a pair $\langle l', d'\rangle$ occurring in the third argument of $\mathit{candidates}(l, d, [\langle l_1, d_1\rangle, \ldots, \langle l_m, d_m\rangle])$. Hence, by $M.3$, we have that $\mathit{gc}(1, l, d, l', d') \in M$.

(Rule 2.i, $i > 1$) Assume that r is $\mathit{gc}(i, l, e, l', e') \leftarrow \mathit{gc}(i{-}1, l, d, l', d') \wedge \mathit{perm}(d, e) \wedge \mathit{perm}(d', e')$, with $i > 1$. By Definition 6 we have that $\big(\mathtt{x}_i = l \wedge \mathtt{y} = f(d) \rightarrow \mathtt{x}_i := l'; \mathtt{y} := f(d')\big)$ is in P_i iff $\big(\mathtt{x}_{i-1} = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_{i-1} := l'; \mathtt{y} := d'\big)$ is in P_{i-1} and, therefore, if $\mathit{gc}(i{-}1, l, d, l', d') \in M$, $f(d) = e$, and $f(d') = e'$ then, by $M.3$, $\mathit{gc}(i, l, e, l', e') \in M$.

(Rule 3.1) Assume that r is $\mathit{reachable}(s_0) \leftarrow$. Since $s_0 \in \mathcal{S}$, we have that by $M.1$, $\mathit{reachable}(s_0) \in M$.

(Rule 3.2) Assume that r is $\mathit{reachable}(\langle l_1, \ldots, l_k, d\rangle) \leftarrow \mathit{tr}(\langle l'_1, \ldots, l'_k, d'\rangle, \langle l_1, \ldots, l_k, d\rangle)$. If we have that $\mathit{tr}(\langle l'_1, \ldots, l'_k, d'\rangle, \langle l_1, \ldots, l_k, d\rangle) \in M$ then, by $M.2$, $\langle \langle l'_1, \ldots, l'_k, d'\rangle, \langle l_1, \ldots, l_k, d\rangle\rangle \in \mathcal{R}$. Thus, $\langle l_1, \ldots, l_k, d\rangle \in \mathcal{S}$ and consequently, by $M.1$, $\mathit{reachable}(\langle l_1, \ldots, l_k, d\rangle) \in M$.

(Rule 4.1–4.k) Assume that r is $\mathit{tr}(s, t) \leftarrow \mathit{reachable}(s) \wedge \mathit{gc}(i, l, d, l', d')$, with $s(\mathtt{x}_i) = l$, $s(\mathtt{y}) = d$, $t(\mathtt{x}_i) = l'$, and $t(\mathtt{y}) = d'$. If $\{\mathit{reachable}(s), \mathit{gc}(i, l, d, l', d')\} \subseteq M$ then $s \in \mathcal{S}$ and there exists a guarded command of the form $\big(\mathtt{x}_i = l \wedge \mathtt{y} = d \rightarrow \mathtt{x}_i := l'; \mathtt{y} := d'\big)$ in P_i. Thus, by Definition 2, $\langle s, t\rangle \in \mathcal{R}$ and consequently, by $M.2$, we get that $\mathit{tr}(s, t) \in M$.

(Rule 5) Assume that r is $\leftarrow \mathit{reachable}(\langle l_1, \ldots, l_k, d\rangle)$. We show that $\mathit{reachable}(\langle l_1, \ldots, l_k, d\rangle) \notin M$. Let us assume, by contradiction, that $\mathit{reachable}(\langle l_1, \ldots, l_k, d\rangle) \in M$ and, thus, by M.1, $\langle l_1, \ldots, l_k, d\rangle \in \mathcal{S}$. Since $\mathcal{R}$ is total, for every reachable state s, there exists a process P_i which is enabled in s, that is, by $M.4$, $\mathit{enabled}(i, l_i, d) \in M$, contradicting the hypothesis that $r \in \mathit{ground}(\Pi_\sigma)^M$, that is, for all $i \in \{1, \ldots, k\}$, $\mathit{enabled}(i, l_i, d) \notin M$.

Now we consider the rules in $\mathit{ground}(\Pi_\varphi)$.

(Rule 1) Since C satisfies φ, by $M.6$, $\mathit{sat}(s_0, \varphi) \in M$ and, hence, $\{\leftarrow \mathtt{not}\ \mathit{sat}(s_0, \varphi)\}^M = \emptyset$. Thus, no rule of $\mathit{ground}(\Pi_\varphi)^M$ is obtained from Rule 1 by the Gelfond-Lifschitz transformation.

(Rules 2-8) Let $P_\mathcal{K}$ be the encoding of $\mathcal{K}$. By definition $M.6 = \{\mathit{sat}(s, \psi) \mid s \in \mathcal{S} \wedge \mathcal{K}, s \vDash \psi\}$. Moreover, by Lemma A.2 we have that, for all $s \in \mathcal{S}$ and CTL formulas ψ, if $\mathcal{K}, s \vDash \psi$ then $\mathit{sat}(s, \psi) \in M(P_\mathcal{K})$. Thus, $M.2 \cup M.6 \cup M.7 \cup M.8 = M(P_\mathcal{K})$ is a model of Rules 2-8.

(Rule 9) Assume that r is $\mathit{satpath}(s, t, \psi) \leftarrow \mathit{sat}(s, \psi) \wedge \mathit{tr}(s, t)$. Assume that $\{\mathit{sat}(s, \psi), \mathit{tr}(s, t)\} \subseteq M$. Then, $\mathcal{K}, s \vDash \psi$ and $\langle s, t\rangle \in \mathcal{R}$ hold. Hence, by $M.7$, we have that $\mathit{satpath}(s, t, \psi) \in M$.

(Rule 10) Assume that r is $\mathit{satpath}(u_0, u_n, \psi) \leftarrow \mathit{sat}(u_0, \psi) \wedge \mathit{tr}(u_0, u_1) \wedge \mathit{satpath}(u_1, u_n, \psi)$. Assume that $\{\mathit{sat}(u_0, \psi), \mathit{tr}(u_0, u_1), \mathit{satpath}(u_1, u_n, \psi)\} \subseteq M$. Then, $\mathcal{K}, u_0 \vDash \psi$, $\langle u_0, u_1\rangle \in \mathcal{R}$, and there exists a finite path $\langle u_1, \ldots, u_n\rangle$, with $n > 1$, such that for all $1 \leq i \leq n$, $\mathcal{K}, u_i \vDash \psi$. Thus, by $M.7$, $\mathit{satpath}(u_0, u_n, \psi) \in M$.

(ii) We have to prove that M is a minimal (w.r.t. set inclusion) model of $\mathit{ground}(\Pi)^M$. We prove it by contradiction. Let us assume that M' is a model of $\mathit{ground}(\Pi)^M$ such that $M' \subset M$. Let z be a ground atom in $M - M'$. We proceed by cases.

(Case A) Assume that z is $\mathit{gc}(i, l, d, l', d')$. Thus, by $M.3$, there exists a guarded command in C whose encoding does not belong to M', and consequently, M' does not encode C.

(Case B) For every $s \in \mathcal{S}$, we define $h(s)$ to be the least integer $k \geq 0$ such that $Reach^k(s_0, s)$ holds. Assume that z is $reachable(s)$. Without loss of generality, we may assume that s is a state such that $\forall r \in \mathcal{S}$ if $reachable(r) \in M - M'$, then $h(r) \geq h(s)$. We have the following two cases.
(Case B.1) $s = s_0$. We get a contradiction from the fact that M' is a model of $ground(\Pi)^M$ and, thus, M' satisfies Rule 3.1.
(Case B.2) $s \neq s_0$. We have that there exists no $t \in \mathcal{S}$ such that $tr(t, s) \in M'$ (otherwise, since M' satisfies Rule 3.2, we would have $reachable(s) \in M'$). Take any $t \in \mathcal{S}$ such that $Reach^{h(s)-1}(s_0, t)$. Since M' satisfies Rules 4.1–4.k and $tr(t, s) \notin M'$, one of the following two facts holds.
Either (B.2.1) $reachable(t) \notin M'$. By M.1 we have that $reachable(t) \in M$, and thus, $reachable(t) \in M - M'$. Since $h(t) < h(s)$, we get a contradiction with the assumption that $\forall r \in \mathcal{S}$ if $reachable(r) \in M - M'$, then $h(r) \geq h(s)$.
Or (B.2.2) there exists no process i such that $gc(i, t(\mathtt{x}_i), t(\mathtt{y}), s(\mathtt{x}_i), s(\mathtt{y})) \in M'$. Therefore, the proof proceeds as in Case (A).

(Case C) Assume that z is $enabled(i, l, d)$. Since M' satisfies Rule 1.i, there exist no l' and d', such that $gc(i, l, d, l', d') \in M'$. Therefore, the proof proceeds as in Case (A).

(Case D) Assume that z is $disabled(1, l, d)$. By M.4 and M.5, we have that $enabled(1, l, d) \notin M$. Since M' satisfies Rule 1.1, one of the following two facts hold.
Either (D.1) No atom of the form $reachable(\langle l, l_2, \ldots, l_k, d\rangle)$ belongs to M'. Therefore, the proof proceeds as in Case (B).
Or (D.2) $enabled(1, l, d)$ belongs to M'. Therefore, we get a contradiction with the facts that $M' \subset M$ and $enabled(1, l, d) \notin M$.

(Case E) Assume that z is $tr(t, s)$. Since M' satisfies Rules 4.1–4.k, one of the following two facts hold.
Either (E.1) $reachable(t) \notin M'$. Therefore, the proof proceeds as in Case (B).
Or (E.2) There is no process i such that $gc(i, t(\mathtt{x}_i), t(\mathtt{y}), s(\mathtt{x}_i), s(\mathtt{y})) \in M'$. Therefore, the proof proceeds as in Case (A).

(Case F) Assume that z is of one of the forms $sat(s, \psi)$, or $satpath(s, t, \psi)$, or $elem(s, p)$. By M.6, M.7, M.8, and Lemma A.2, we have that $M|_{\{sat, satpath, elem, tr\}}$ is the least Herbrand model of $ground(\Pi_\varphi)^M \cup \overleftarrow{M|_{\{tr\}}}$. Now, since M' is an Herbrand model of $ground(\Pi_\varphi)^M \cup \overleftarrow{M|_{\{tr\}}}$, we get that $M|_{\{sat, satpath, elem, tr\}} \subseteq M'$, thereby contradicting the assumption that $z \in M - M'$. □

Proof of Theorem 4.2

Let $|ground(\Pi)|$ denote the size (that is, the number of rules) of $ground(\Pi)$. We have that $|ground(\Pi)|$ is $\mathcal{O}(|\mathcal{L}|^{3k} \cdot |\mathcal{D}|^3 \cdot |\varphi|)$, where $k > 1$. Moreover, since program Π_σ is an HEF (see Proposition A.1) logic program, Π_σ can be transformed into a normal logic program Π_σ^n such that $ans(\Pi_\sigma) = ans(\Pi_\sigma^n)$. We have that $|ground(\Pi_\sigma^n)| = \alpha_1 + \alpha_2 + |ground(\Pi_\sigma)|$, where α_1 depends on the number of the ground instances of Rule 1.1 and α_2 depends on the number of the ground instances of Rule 2.1. Now we have that: (i) α_1 is at most $|\mathcal{L}|^k \cdot |\mathcal{D}|$ (indeed, the ground instances of Rule 1.1 are at most $|\mathcal{L}|^k \cdot |\mathcal{D}|$), and (ii) α_2 is $\mathcal{O}(|\mathcal{L}|^2 \cdot |\mathcal{D}|^2)$ (indeed, the ground instances of Rule 2.1 are at most $|\mathcal{L}| \cdot |\mathcal{D}|$, and in any instance of Rule 2.1 the value of m is at most $|\mathcal{L}| \cdot |\mathcal{D}|$). Thus, $\alpha_1 + \alpha_2$ is $\mathcal{O}(|\mathcal{L}|^k \cdot |\mathcal{D}|^2)$ and $|ground(\Pi^n)|$ is $\mathcal{O}(|\mathcal{L}|^{3k} \cdot |\mathcal{D}|^3 \cdot |\varphi|)$.

Given a set I of ground atoms, (i) to compute $ground(\Pi^n)^I$ takes linear time w.r.t. $|ground(\Pi^n)|$, (ii) to generate the minimal model M of $ground(\Pi^n)^I$ takes linear time w.r.t. $|ground(\Pi^n)^I|$,

and (iii) to check whether or not $I = M$ also takes linear time w.r.t. $|ground(\Pi^n)^I|$ (for more information on these results the reader may refer to [29]). Hence, to verify whether or not a given set of ground atoms is an answer set of Π takes linear time w.r.t. $|ground(\Pi^n)|$. Thus, the verification that I is an answer set of Π takes exponential time w.r.t. k, linear time w.r.t. $|\varphi|$, and polynomial time w.r.t. $\mathcal{L}$ and w.r.t. $\mathcal{D}$.

Now, the choice of a candidate answer set I can be done by: (i) choosing, for each $\langle l, d\rangle \in \mathcal{L}\times\mathcal{D}$, at most one ground atom in the set $\{gc(1, l, d, l', d') \mid \langle l, l'\rangle \in T \wedge d' \in \mathcal{D} \wedge \langle l, d\rangle \neq \langle l', d'\rangle\}$, (ii) computing in $\mathcal{O}(k)$ time a ground atom of the form $gc(i, \ldots)$, for $i = 2, \ldots, k$, (iii) computing in $\mathcal{O}(|\mathcal{L}|^{3k} \cdot |\mathcal{D}|^3 \cdot |\varphi|)$ time the ground instances of the rules in Π, where the truth values of the gc atoms are fixed as indicated at Steps (i) and (ii), thereby obtaining a stratified program, and (iv) finally, computing in $\mathcal{O}(|\mathcal{L}|^{3k} \cdot |\mathcal{D}|^3 \cdot |\varphi|)$ the unique stable model of that stratified program.

Since Step (i) can be done in nondeterministic polynomial time w.r.t. $|\mathcal{L}| \times |\mathcal{D}|$, we get the thesis. □

B. Source code

In this section we list the source code used to synthesize 2-*mutex*-1 with *claspD* (Program (2) in Table 1).

B.1. The (disjunctive) logic program Π_σ^{DLP} encoding a structural property

```
enabled(1,X1,Y) | disabled(1,X1,Y):- reachable(X1,X2,Y).
enabled(2,X2,Y) :- gc(2,X2,Y,X2p,Yp).

gc(1,t,Y,w,0) | gc(1,t,Y,w,1) | gc(1,t,Y,w,2):- enabled(1,t,Y).
gc(1,w,0,w,1) | gc(1,w,0,w,2) |
gc(1,w,0,u,0) | gc(1,w,0,u,1) | gc(1,w,0,u,2):- enabled(1,w,0).
gc(1,w,1,w,0) | gc(1,w,1,w,2) |
gc(1,w,1,u,0) | gc(1,w,1,u,1) | gc(1,w,1,u,2):- enabled(1,w,1).
gc(1,w,2,w,0) | gc(1,w,2,w,1) |
gc(1,w,2,u,0) | gc(1,w,2,u,1) | gc(1,w,2,u,2) - enabled(1,w,2).
gc(1,u,Y,t,0) | gc(1,u,Y,t,1) | gc(1,u,Y,t,2):- enabled(1,u,Y).
gc(2,X2,Z,X2p,Zp) :- gc(1,X2,Y,X2p,Yp), perm(Y,Z), perm(Yp,Zp).

reachable(X1,X2,Y) :- s0(X1,X2,Y).
reachable(X1p,X2p,Yp) :- reachable(X1,X2,Y), tr(X1,X2,Y,X1p,X2p,Yp).

tr(X1,X2,Y,X1p,X2,Yp):- reachable(X1,X2,Y), gc(1,X1,Y,X1p,Yp).
tr(X1,X2,Y,X1,X2p,Yp):- reachable(X1,X2,Y), gc(2,X2,Y,X2p,Yp).

 :- reachable(X1,X2,Y), not enabled(1,X1,Y), not enabled(2,X2,Y).
```

where: (i) variables `X1`, `X2`, `X1p`, and `X2p` range over $\mathcal{L} = \{\mathtt{t}, \mathtt{w}, \mathtt{u}\}$, (ii) variables `Y`, `Yp`, and `Z` range over $\mathcal{L} = \{0, 1, 2\}$, (iii) the 2-generating function $f_2 = \{\langle 0, 1\rangle, \langle 1, 0\rangle, \langle 2, 2\rangle\}$ is encoded by the following facts `perm(0,1)`, `perm(1,0)` and `perm(2,2)`. (iv) the initial state is encoded by `s0`$(\mathtt{t}, \mathtt{t}, 0)$. Note that the *local transition relation* $T = \{\langle \mathtt{t}, \mathtt{w}\rangle, \langle \mathtt{w}, \mathtt{w}\rangle, \langle \mathtt{w}, \mathtt{u}\rangle, \langle \mathtt{u}, \mathtt{t}\rangle\}$ is directly embedded into the guarded commands for process P_1.

B.2. The logic program Π_φ encoding a behavioural property

```
sat(X1,X2,Y,local(p1,X1)) :- elem(local(p1,X1),X1,X2,Y).
sat(X1,X2,Y,local(p2,X2)) :- elem(local(p2,X2),X1,X2,Y).
sat(X1,X2,Y,shared(Y)) :- elem(shared(Y),X1,X2,Y).
sat(X1,X2,Y,n(F)) :- not sat(X1,X2,Y,F), dp(n(F)), l(X1), l(X2), d(Y).
sat(X1,X2,Y,o(F,G)) :- sat(X1,X2,Y,F), dp(o(F,G)).
sat(X1,X2,Y,o(F,G)) :- sat(X1,X2,Y,G), dp(o(F,G)).
sat(X1,X2,Y,a(F,G)) :- sat(X1,X2,Y,F), sat(X1,X2,Y,G), dp(a(F,G)).
sat(X1,X2,Y,ex(F)) :- tr(X1,X2,Y,X1p,X2p,Yp), sat(X1p,X2p,Yp,F), dp(ex(F)).
sat(X1,X2,Y,eu(F,G)) :- sat(X1,X2,Y,G), dp(eu(F,G)).
sat(X1,X2,Y,eu(F,G)) :- sat(X1,X2,Y,F), tr(X1,X2,Y,X1p,X2p,Y),
    sat(X1p,X2p,Y,eu(F,G)), dp(eu(F,G)).
sat(X1,X2,Y,ef(F)) :- sat(X1,X2,Y,F), dp(ef(F)).
sat(X1,X2,Y,ef(F)) :- tr(X1,X2,Y,X1p,X2p,Yp), sat(X1p,X2p,Yp,ef(F)), dp(ef(F)).
sat(X1,X2,Y,eg(F)) :- satpath(X1,X2,Y,X1p,X2p,Yp,F),
    satpath(X1p,X2p,Yp,X1p,X2p,Yp,F), dp(eg(F)).
satpath(X1,X2,Y,X1p,X2p,Yp,F) :- sat(X1,X2,Y,F), tr(X1,X2,Y,X1p,X2p,Yp),
    dp(eg(F)).
satpath(X1,X2,Y,X1p,X2p,Yp,F) :- sat(X1,X2,Y,F), tr(X1,X2,Y,X1pp,X2pp,Ypp),
    satpath(X1pp,X2pp,Ypp,X1p,X2p,Yp,F), dp(eg(F)).
```

where: (i) variables `X1`, `X1p`, `X1pp`, `X2`, and `X2p` range over $\mathcal{L} = \{\mathtt{t}, \mathtt{w}, \mathtt{u}\}$, (ii) variables `Y`, `Yp`, and `Z` range over $\mathcal{L} = \{0, 1, 2\}$, (iii) variables `F` and `G` range over the term encoding the formula φ and its subterms, and (iv) predicate `dp` encodes the domain predicate which restricts the domain of variables `F` and `G`.

As an example we list: (i) the domain predicates (lines 1-5), (ii) the elementary properties (lines 8-9), and (iii) the integrity constraint (line 10) needed to ensure that processes P_1 and P_2, obtained by using $\Pi = \Pi_\sigma^{DLP} \cup \Pi_\varphi$, enjoy the mutual execution property.

```
dp(n(ef(n(n(a(local(p1,u),local(p2,u))))))).
dp(ef(n(n(a(local(p1,u),local(p2,u)))))).
dp(n(n(a(local(p1,u),local(p2,u))))).
dp(n(a(local(p1,u),local(p2,u)))).
dp(a(local(p1,u),local(p2,u))).

elem(local(p1,u),u,X2,Y) :- l(X2), d(Y).
elem(local(p2,u),X1,u,Y) :- l(X1), d(Y).

:- not sat(X1,X2,Y,n(ef(n(n(a(local(p1,u),local(p2,u))))))), s0(X1,X2,Y).
```

References

[1] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.

[2] P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems*, 20:51–115, January 1998.

[3] P. C. Attie and E. A. Emerson. Synthesis of concurrent programs for an atomic read-/write model of computation. *ACM Transactions on Programming Languages and Systems*, 23:187–242, 2001.

[4] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.

[5] P. Bonatti, F. Calimeri, N. Leone, and F. Ricca. Answer Set Programming. In A. Dovier and E. Pontelli, editors, *A 25-Year Perspective on Logic Programming*, Lecture Notes in Computer Science 6125. Springer, Berlin, 2010.

[6] S. Brass and J. Dix. Characterizations of the Stable Semantics by Partial Evaluation. In *Logic Programming and Nonmonotonic Reasoning*, Lecture Notes in Computer Science 928, pages 85–98. Springer, 1995.

[7] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, Lecture Notes in Computer Science 131, pages 52–71. Springer, Berlin, 1982.

[8] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[9] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Logic in Computer Science, LICS '89, Proceedings*, pages 353–362. IEEE Computer Society, 1989.

[10] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewod Cliffs, N.J., 1976.

[11] C. Drescher, B. Kaufmann, A. Neumann and T. Schaub. Conflict-driven answer set solving. In G. Brewka and J. Lang, editors, *Proc. 20th Int. Joint Conf. of Artificial Intelligence* (*IJCAI '07*), pages 386–392. AAAI Press, 2007.

[12] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub. Conflict-driven disjunctive answer set solving. In G. Brewka and J. Lang, editors, *Proc. 11th Int. Conf. on Principles of Knowledge Representation and Reasoning* (*KR '08*), pages 422–432. AAAI Press, 2008.

[13] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22:364–418, 1997.

[14] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 997–1072. Elsevier, 1990.

[15] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9:105–131, 1996.

[16] M. Gebser, J. Lee, and Y. Lierler. Head-elementary-set-free logic programs. In C. Baral, G. Brewka, and J. S. Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Nonmotonic Reasoning (LPNMR 2007)*, Lecture Notes in Computer Science 4483, pages 149–161. Springer, Berlin, 2007.

[17] M. Gelfond. Answer sets. In F. van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*, chapter 7, pages 285–316. Elsevier, 2007.

[18] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[19] E. Giunchiglia and Y. Lierler and M. Maratea. Answer Set Programming Based on Propositional Satisfiability. *Journal of Automated Reasoning*, 345–377, Springer, 2006.

[20] S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Synthesis from temporal specifications using preferred answer set programming. In M. Coppo, E. Lodi, and G. Pinna, editors, *Proceedings of the 9th Italian Conference on Theoretical Computer Science*, Lecture Notes in Computer Science 3701, pages 280–294. Springer, Berlin, 2005.

[21] T. Janhunen and I. Niemelä. GNT – A Solver for Disjunctive Logic Programs. In *Logic Programming and Nonmonotonic Reasoning*, Lecture Notes in Computer Science 2923, pages 331–335. Springer, 2004.

[22] O. Kupferman and M. Y. Vardi. Synthesis with incomplete information. In D. M. Gabbay, editor, *Applied Logic #16: Advances in Temporal Logic*, pages 109–127. Kluwer Academic Publishers, 2000.

[23] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transaction on Computational Logic*, 7:499–562, 2006.

[24] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984.

[25] I. Niemelä, Answer Set Programming without Unstratified Negation. *Proceedings of the 24th International Conference on Logic Programming* (*ICLP'08*), Lecture Notes in Computer Science 5366, Springer, 2008.

[26] U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In J. W. Lloyd, editor, *Proceedings of the First International Conference on Computational Logic (CL 2000), London, UK, 24-28 July*, Lecture Notes in Artificial Intelligence 1861, pages 384–398. Springer-Verlag, 2000.

[27] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[28] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL '89)*, pages 179–190, New York, NY, USA, 1989.

[29] J. S. Schlipf. Complexity and undecidability results for logic programming. *Annals of Mathematics and Artificial Intelligence*, 15:257–288, 1995.

[30] T. Syrjänen and I. Niemelä. The Smodels System *Proceedings of the 6th International Conference on Logic Programming and Nonmotonic Reasoning* (*LPNMR '01*), Lecture Notes in Computer Science 2173, pages 434–438. Springer, Berlin, 2001.

[31] T. Syrjänen. *Lparse 1.0 user's manual.* `http://www.tcs.hut.fi/Software/smodels/`, 2002.

[32] M. Truszczyński. Logic programming for knowledge representation. In V. Dahl and I. Niemelä, editors, *Proceedings of the 23rd International Conference on Logic Programming (ICLP '07)*, Lecture Notes in Computer Science 4670, pages 76–88. Springer, 2007.