Program Transformation for Development, Verification, and Synthesis of Programs

Fabio Fioravanti [a], Alberto Pettorossi [b],
Maurizio Proietti [c], and Valerio Senni [b]

[a] *Department of Sciences*
University of 'G. D'Annunzio'
Viale Pindaro 42, 65127, Pescara, Italy
E-mail: fioravanti@sci.unich.it
[b] *Department of Informatics, Systems, and Production*
University of Rome 'Tor Vergata'
Via del Politecnico 1, 00133, Rome, Italy
E-mail: {pettorossi,senni}@disp.uniroma2.it
[c] *CNR-IASI*
Viale Manzoni 30, 00185, Rome, Italy
E-mail: maurizio.proietti@iasi.cnr.it

Abstract. This paper briefly describes the use of the program transformation methodology for the development of correct and efficient programs. In particular, we will refer to the case of constraint logic programs and, through some examples, we will show how by program transformation, one can improve, synthesize, and verify programs.

Keywords: Constraint logic programming, model checking, program development, program synthesis, unfold/fold program transformation, software verification.

1. Introduction

The program transformation methodology has been introduced in the case of functional programs by Burstall and Darlington [4] and then it has been adapted to logic programs by Hogger [13] and Tamaki and Sato [30]. The main idea of this methodology is to transform, maybe in several steps and by applying various transformation rules, the given initial program into a final program with the aim of improving its efficiency and preserving its correctness. If the initial program encodes a declarative specification of a problem and the final program encodes an efficient algorithm to solve that problem, then program transformation is equivalent to program synthesis. Thus, program transformation can be viewed as a technique both: (i) for *program improvement* and advanced compilation, and (ii) for *program synthesis* and program derivation.

In recent years program transformation has also been used as a technique for *program verification*. It has been shown, in fact, that via program transformation one can perform *model checking* and, in general, one can prove properties of *infinite* state systems that cannot be analyzed by using standard model checking techniques.

In this paper we will illustrate the three uses of the program transformation methodology we mentioned above, namely, program improvement, program synthesis, and program verification. In particular, we will consider the case of specifications and algorithms written as *constraint logic programs* [14] and we will consider the following transformation rules: *definition*, *unfolding*, *folding*, *goal replacement*, and *clause splitting* [2,7,10,11,27,30]. These rules are correct in the sense that they preserve the perfect model semantics [27], and they are applied according to some specific strategies, which ensure that the objectives of the transformations are actually achieved. This approach to program transformation is, thus, called the '*rules + strategies approach*'.

Transformation rules similar to those we consider, have been proposed also for: (i) concurrent constraint logic programs [8], (ii) constraint handling rules [29], and (iii) functional logic programs [1], and thus, the transformation methodology we will present, can also be used for those classes of programs. In the literature one can find also other sets of transformation rules which are shown to be correct with respect to other semantics (see, for instance, [28] for rules which preserve the well-founded semantics).

2. Program Improvement

Programs are often written in a parametric form so that they can be reused in different contexts, and when a parametric program is reused, one may want to transform it for taking advantage of the new context of use. This transformation, called *program specialization* [12,15,18], often allows great efficiency improvements. Let us present an example of this transformation by deriving a deterministic, specialized pattern matcher starting from a given nondeterministic, parametric pattern matcher and a given specific pattern.

In this example we consider the matching relation $\mathit{le\_m}(P,S)$ on strings of numbers which holds between a pattern $P=[p_1,\ldots,p_n]$ and a string S iff in S there is a substring $Q=[q_1,\ldots,q_n]$ such that for $i=1,\ldots,n, p_i \leq q_i$. (This example can be generalized by considering, instead of $p_i \leq q_i$, any relation that can be expressed via a constraint logic program.) The following constraint logic program can be taken as the specification of our general pattern matching problem:

1. $\mathit{le\_m}(P,S) \leftarrow \mathit{ap}(B,C,S) \wedge \mathit{ap}(A,Q,B) \wedge \mathit{le}(P,Q)$
2. $\mathit{ap}([\,],\mathit{Ys},\mathit{Ys}) \leftarrow$
3. $\mathit{ap}([X|\mathit{Xs}],\mathit{Ys},[X|\mathit{Zs}]) \leftarrow \mathit{ap}(\mathit{Xs},\mathit{Ys},\mathit{Zs})$
4. $\mathit{le}([\,],[\,]) \leftarrow$
5. $\mathit{le}([X|\mathit{Xs}],[Y|\mathit{Ys}]) \leftarrow X \leq Y \wedge \mathit{le}(Xs,Ys)$

where *ap* denotes the familiar list concatenation predicate. Now let us specialize this general program for a specific pattern P, say $[1,0,2]$. The transformation starts off by applying the so-called definition rule, thereby introducing the following clause:

6. $\mathit{le\_m}_s(S) \leftarrow \mathit{le\_m}([1,0,2],S)$

The transformation rules will be applied according to the so-called *Determinization Strategy* [10]. This strategy, which will not be presented here, allows a fully automatic derivation of a deterministic, efficient pattern matcher. First, clause 6 is unfolded w.r.t. the atom $\mathit{le\_m}([1,0,2],S)$, that is, the atom $\mathit{le\_m}([1,0,2],S)$, which is an instance of the head of clause 1, is replaced by the corresponding instance of the body of clause 1. The result is:

7. $\mathit{le\_m}_s(S) \leftarrow \mathit{ap}(B,C,S) \wedge \mathit{ap}(A,Q,B) \wedge \mathit{le}([1,0,2],Q)$

Then, in order to fold clause 7, the following definition is introduced :

8. $\mathit{new}1(S) \leftarrow \mathit{ap}(B,C,S) \wedge \mathit{ap}(A,Q,B) \wedge \mathit{le}([1,0,2],Q)$

and then clause 7 is folded, that is, (the instance of) the body of clause 8 which occurs in the body of clause 7 is replaced by (the corresponding instance of) the head of clause 8. The result is:

9. $\mathit{le\_m}_s(S) \leftarrow \mathit{new}1(S)$

Then, clause 8 is unfolded w.r.t. the atoms *ap* and *le*, thereby obtaining:

10. $\mathit{new}1([X|\mathit{Xs}]) \leftarrow 1 \leq X \wedge \mathit{ap}(Q,C,\mathit{Xs}) \wedge \mathit{le}([0,2],Q)$
11. $\mathit{new}1([X|\mathit{Xs}]) \leftarrow \mathit{ap}(B,C,\mathit{Xs}) \wedge \mathit{ap}(A,Q,B) \wedge$ $\mathit{le}([1,0,2],Q)$

Now, the clause splitting rule is applied to clause 11. We have two cases: (i) $1 \leq X$, and (ii) $1 > X$, and we get the following two clauses:

12. $\mathit{new}1([X|\mathit{Xs}]) \leftarrow 1 \leq X \wedge \mathit{ap}(B,C,\mathit{Xs}) \wedge \mathit{ap}(A,Q,B) \wedge$ $\mathit{le}([1,0,2],Q)$
13. $\mathit{new}1([X|\mathit{Xs}]) \leftarrow 1 > X \wedge \mathit{ap}(B,C,\mathit{Xs}) \wedge \mathit{ap}(A,Q,B) \wedge$ $\mathit{le}([1,0,2],Q)$

Next, in order to fold clauses 10 and 12, we introduce these two clauses defining the predicate $\mathit{new}2$:

14. $\mathit{new}2(\mathit{Xs}) \leftarrow \mathit{ap}(Q,C,\mathit{Xs}) \wedge \mathit{le}([0,2],Q)$
15. $\mathit{new}2(\mathit{Xs}) \leftarrow \mathit{ap}(B,C,\mathit{Xs}) \wedge \mathit{ap}(A,Q,B) \wedge \mathit{le}([1,0,2],Q)$

Then clauses 10 and 12 are folded by using clauses 14 and 15. Moreover, clause 13 is folded by using clause 8. The resulting two clauses are the following:

16. $\mathit{new}1([X|\mathit{Xs}]) \leftarrow 1 \leq X \wedge \mathit{new}2(\mathit{Xs})$
17. $\mathit{new}1([X|\mathit{Xs}]) \leftarrow 1 > X \wedge \mathit{new}1(\mathit{Xs})$

They are mutually exclusive because of the constraints $1 \leq X$and $1>X$. Now the program transformation process continues in a way similar to the one we have followed above, when deriving clauses 16 and 17 starting from clause 8. By starting from clauses 14 and 15, the new predicates $\mathit{new}3$ through $\mathit{new}6$ are introduced, and by application of the unfold and fold rules, their defining clauses are derived. Eventually, the following deterministic, specialized program is obtained:

9. $\mathit{le\_m}_s(S) \leftarrow \mathit{new}1(S)$
16. $\mathit{new}1([X|\mathit{Xs}]) \leftarrow 1 \leq X \wedge \mathit{new}2(\mathit{Xs})$
17. $\mathit{new}1([X|\mathit{Xs}]) \leftarrow 1 > X \wedge \mathit{new}1(\mathit{Xs})$
18. $\mathit{new}2([X|\mathit{Xs}]) \leftarrow 1 \leq X \wedge \mathit{new}3(\mathit{Xs})$
19. $\mathit{new}2([X|\mathit{Xs}]) \leftarrow 0 \leq X \wedge 1 > X \wedge \mathit{new}4(\mathit{Xs})$
20. $\mathit{new}2([X|\mathit{Xs}]) \leftarrow 0 > X \wedge \mathit{new}1(\mathit{Xs})$
21. $\mathit{new}3([X|\mathit{Xs}]) \leftarrow 2 \leq X \wedge \mathit{new}5(\mathit{Xs})$
22. $\mathit{new}3([X|\mathit{Xs}]) \leftarrow 1 \leq X \wedge 2 > X \wedge \mathit{new}3(\mathit{Xs})$
23. $\mathit{new}3([X|\mathit{Xs}]) \leftarrow 0 \leq X \wedge 1 > X \wedge \mathit{new}4(\mathit{Xs})$
24. $\mathit{new}3([X|\mathit{Xs}]) \leftarrow 0 > X \wedge \mathit{new}1(\mathit{Xs})$
25. $\mathit{new}4([X|\mathit{Xs}]) \leftarrow 2 \leq X \wedge \mathit{new}6(\mathit{Xs})$
26. $\mathit{new}4([X|\mathit{Xs}]) \leftarrow 1 \leq X \wedge 2 > X \wedge \mathit{new}2(\mathit{Xs})$
27. $\mathit{new}4([X|\mathit{Xs}]) \leftarrow 1 > X \wedge \mathit{new}1(\mathit{Xs})$
28. $\mathit{new}5([X|\mathit{Xs}]) \leftarrow$
29. $\mathit{new}6([X|\mathit{Xs}]) \leftarrow$

This program is deterministic, in the sense that at most one clause at a time can be applied during the evaluation of any ground goal. The efficiency of this program is very high because it behaves like a deterministic finite automaton as the Knuth-Morris-Pratt matcher.

3. Program Synthesis

Program synthesis is a technique for deriving programs from formal specifications (see, for instance, [13, 26] for the derivation of logic programs from first-order logic specifications). In this section we present an example of use of program transformation for program synthesis and we derive a constraint logic program from its specification provided as a first-order formula.

We consider the N-queens problem, which is often used in the literature for introducing techniques such as recursion and backtracking. That problem can be described as follows: N (≥ 0) queens are to be placed on an $N \times N$ chess board, so that no two queens attack each other, that is, they do not lie on the same row, column, or diagonal. Since no two queens should lie on the same column, the positions of the N queens on the chess board can be denoted by the list $L = [i_1, \ldots, i_N]$ such that, for $1 \leq k \leq N$, i_k is the row where the queen on column k is placed. A specification of the solution L for the N-queens problem is given by the following first-order formula $\varphi(N, L)$:

$$\begin{array}{l}
\mathit{nat}(N) \wedge \mathit{nat\_list}(L) \wedge \mathit{length}(L, N) \wedge \\
\forall X\ (\mathit{member}(X, L) \rightarrow \mathit{in}(X, 1, N)) \wedge \\
\forall A,B,K,M \\
((1 \leq K \wedge K < M \wedge \mathit{occurs}(A, K, L) \wedge \mathit{occurs}(B,M,L)) \\
\qquad \rightarrow (A \neq B \wedge A-B \neq M-K \wedge B-A \neq M-K))
\end{array}$$

where: $\mathit{nat}(N)$ holds iff N is a natural number, $\mathit{nat\_list}(L)$ holds iff L is a list of natural numbers, $\mathit{length}(L, N)$ holds iff L is a list of length N, $\mathit{member}(X, L)$ holds iff X is an element of the list L, $\mathit{in}(X, M, N)$ holds iff $M \leq X \leq N$, and $\mathit{occurs}(X, I, L)$ holds iff X occurs in the list L at position I, with $1 \leq I \leq N$, where N is the length of L. Let P be the program made out of the clauses defining the predicates nat, $\mathit{nat\_list}$, member, in, and occurs.

Now, our goal is to synthesize a constraint logic program R which computes a predicate $\mathit{queens}(N, L)$ such that the following Property (π) holds:

$$(\pi) \quad M(R) \models \mathit{queens}(N, L) \text{ iff } M(P) \models \varphi(N, L)$$

where $M(R)$ and $M(P)$ denote the *perfect model* of the program R and P, respectively. Now we apply the technique presented in [11] starting from the formula $\mathit{queens}(N, L) \leftarrow \varphi(N, L)$. The first step of that technique consists in applying a variant of the Lloyd-Topor transformation [19] which, starting from $\mathit{queens}(N, L) \leftarrow \varphi(N, L)$, derives the following constraint logic program F:

$$\begin{array}{l}
\mathit{queens}(N, L) \leftarrow \mathit{nat}(N) \wedge \mathit{nat\_list}(L) \wedge \mathit{length}(L, N) \wedge \\
\qquad\qquad\qquad \neg \mathit{aux}1(L, N) \wedge \neg \mathit{aux}2(L) \\
\mathit{aux}1(L, N) \leftarrow \mathit{member}(X, L) \wedge \neg \mathit{in}(X, 1, N) \\
\mathit{aux}2(L) \leftarrow 1 \leq K \wedge K < M \wedge \\
\qquad\qquad \neg(A \neq B \wedge A-B \neq M-K \wedge B-A \neq M-K) \wedge \\
\qquad\qquad \mathit{occurs}(A, K, L) \wedge \mathit{occurs}(B, M, L)
\end{array}$$

It can be shown that this variant of the Lloyd-Topor transformation preserves the perfect model semantics. As a consequence:

$$M(P \cup F) \models \mathit{queens}(N, L) \text{ iff } M(P) \models \varphi(N, L).$$

Unfortunately, the performance of the derived program $P \cup F$ is not satisfactory, because it exhibits an inefficient generate-and-test behavior, whenever we evaluate queries by applying the usual depth-first search strategy with the left-to-right computation rule. In particular, for any query of the form $\mathit{queens}(n, L)$, where n is a natural number and L is a variable list, program $P \cup F$ first generates a value l for the list L and then tests whether or not $\mathit{length}(l, n) \wedge \neg \mathit{aux}1(l, n) \wedge \neg \mathit{aux}2(l)$ holds. Note that this generate-and-test behavior may even lead to nontermination, because an infinite number of lists may be generated.

Thus, the process of program synthesis proceeds by applying the definition, unfolding, folding, and goal replacement rules (see [11] for details), with the objective of deriving a more efficient, terminating program. The result of this transformation is the following program R:

$$\begin{array}{l}
\mathit{queens}(N, L) \leftarrow \mathit{new}2(N, L, 0) \\
\mathit{new}2(N, [\,], K) \leftarrow N = K \\
\mathit{new}2(N, [H|T], K) \leftarrow N \geq K+1 \wedge \mathit{new}2(N, T, K+1) \wedge \\
\qquad\qquad\qquad\qquad \mathit{new}3(H, T, N, 0) \\
\mathit{new}3(A, [\,], N, M) \leftarrow \mathit{in}(A, 1, N) \wedge \mathit{nat}(A) \\
\mathit{new}3(A, [B|T], N, M) \leftarrow A \neq B \wedge A-B \neq M+1 \wedge \\
\qquad\qquad\qquad\qquad B-A \neq M+1 \wedge \mathit{nat}(B) \wedge \\
\qquad\qquad\qquad\qquad \mathit{new}3(A, T, N, M+1)
\end{array}$$

together with the clauses defining the predicates in and nat. Since the transformation rules preserve the perfect model semantics, $M(R) \models \mathit{queens}(N, L)$ iff $M(P \cup F) \models \mathit{queens}(N, L)$ and, thus, Property (π) holds. It can be shown that R terminates for all queries of the form $\mathit{queens}(n, L)$. Note that program R solves the N-queens problem in a clever way: each time a new queen is placed on the board, R tests whether or not it attacks another queen already on the board.

4. Program Verification

Proof of program properties are often needed for checking the correctness of software components with respect to their specifications. This section illustrates the use of program transformation for proving program properties specified by either first-order formulas or temporal logic formulas.

Proofs performed by using program transformation are strongly related to proofs by mathematical induction (see [3] for a survey on inductive proofs). In particular, the unfolding rule can be used for decomposing a formula of the form $\varphi(t(X))$, where $t(X)$ is a complex term, into a combination of n formulas of the form $\varphi_1(X), \ldots, \varphi_n(X)$, and the folding rule can be used for applying inductive hypotheses.

The unfold/fold transformations introduced in [4, 30] can be used for proving several kinds of program properties, such as equivalences of functions defined by recursive equation programs [16], equivalences of predicates defined by logic programs [22], first-order properties of predicates defined by constraint logic programs [23], and temporal properties of concurrent systems [9,25].

4.1. The unfold/fold proof method

Now we present, by means of a simple example taken from [23], a technique called *unfold/fold proof method* which is based on program transformation and can be used for proving first-order properties of constraint logic programs.

Let us consider the following program Member which defines the membership relation for lists:

$\mathit{member}(X, [Y|L]) \leftarrow X = Y$
$\mathit{member}(X, [Y|L]) \leftarrow \mathit{member}(X, L)$

The following formula φ states that every finite list of numbers has an upper bound:

$\varphi: \ \ \forall L\, \exists U\, \forall X\, (\mathit{member}(X, L) \rightarrow X \leq U)$

To show that φ holds, we apply the unfold/fold proof method which consists of the following two steps. In the first step, φ is transformed into a set of clauses by applying a variant of the Lloyd-Topor transformation [19], thereby deriving the following program Prop_1 which defines the predicate *prop*:

$\mathit{prop} \leftarrow \neg p$
$p \leftarrow \mathit{list}(L) \wedge \neg q(L)$
$q(L) \leftarrow \mathit{list}(L) \wedge \neg r(L, U)$
$r(L, U) \leftarrow X > U \wedge \mathit{list}(L) \wedge \mathit{member}(X, L)$

The predicate prop is equivalent to φ in the sense that $M(\mathit{Member}) \models \varphi$ iff $M(\mathit{Member} \cup \mathit{Prop}_1) \models \mathit{prop}$. In the second step, the *existential variables* occurring in Prop_1 (that is, the variables occurring in the body of a clause and not in its head) are eliminated by applying the transformation strategy presented in [23]. As a result we get the following program Prop_2:

$\mathit{prop} \leftarrow \neg p \qquad\qquad p \leftarrow p_1 \qquad\qquad p_1 \leftarrow p_1$

Now, Prop_2 is a propositional program and its *finite* perfect model $M(\mathit{Prop}_2)$ is $\{\mathit{prop}\}$. Since it can be shown that every transformation we have applied preserves the perfect model, we get that $M(\mathit{Member}) \models \varphi$ iff $M(\mathit{Prop}_2) \models \mathit{prop}$. Therefore, we have that φ holds for the program Member because prop belongs to $M(\mathit{Prop}_2)$.

Note that the unfold/fold proof method can be viewed as an instance of the *quantifier elimination* method which has well-known applications in the field of automated theorem proving (see [24] for a brief survey).

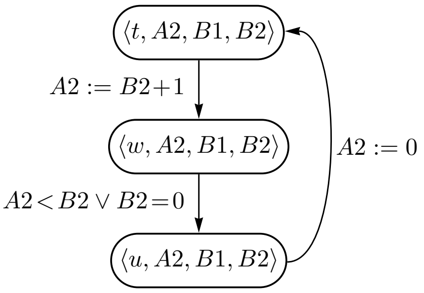
4.2. Infinite-state model checking

In this section we present a method [9] for verifying temporal properties of infinite state systems by transforming constraint logic programs.

As indicated in [5], the behavior of a concurrent system that evolves over time according to a given protocol can be modeled by means of a *state transition system*, that is, (i) a set S of *states*, (ii) an *initial state* $s_0 \in S$, and (iii) a *transition relation* $t \subseteq S \times S$. Let us assume that t is a *total* relation, that is, for every state $s \in S$ there exists a state $s' \in S$, called the *successor state* of s, such that $t(s, s')$ holds. A *computation path* starting from a state s_1 is an *infinite* sequence of states $s_1\, s_2 \ldots$ such that, for every $i \geq 1$, there is a transition from s_i to s_{i+1}, that is, $t(s_i, s_{i+1})$ holds.

The properties of the evolution over time of a concurrent system are specified by using formulas of the temporal logic called *Computation Tree Logic* (or CTL, for short [5]) which specify the properties of the computation paths. The formulas of CTL are built from a given set of *elementary properties* of the states by using: (i) the usual connectives: $\neg$ (‘not’) and $\wedge$ (‘and’), (ii) the following quantifiers along single computation paths: g (‘for all states on the path’ or ‘globally’), f (‘there exists a state on the path’ or ‘in the future’), x (‘next time’), and u (‘until’), and (iii) the two quantifiers over computation paths: a (‘for all paths’) and e (‘there exists a path’).

There exist very efficient algorithms and tools which use model checking techniques for verifying temporal properties of *finite state systems*, that is, systems where the set S of states is finite [5]. Unfortunately, many concurrent systems cannot be modeled by finite state systems and one should look for verification methods for *infinite state systems*. Various methods based on automated theorem proving have been proposed in the literature and, in particular, a method based on constraint logic programming is described in [6]. Note that all of the verification methods for infinite state systems are



$\langle t, A2, B1, B2\rangle$
$A2 := B2{+}1$
$\langle w, A2, B1, B2\rangle$
$A2 := 0$
$A2 < B2 \vee B2 = 0$
$\langle u, A2, B1, B2\rangle$

Fig. 1. The Bakery protocol: a graphical representation of the transition relation t_A for agent A.

necessarily incomplete, because the problem of verifying CTL properties of those systems is in general undecidable.

As an example of use of program transformation for verifying CTL properties of infinite state systems, now we will consider the *Bakery* protocol [17] and we will verify that it satisfies the *mutual exclusion* property. Let A and B be two agents which want to access a shared resource in a mutual exclusive way by using the Bakery protocol. A state of agent A is represented by the pair $\langle A1, A2\rangle$, where $A1$ is a *control state* that takes values in the set $\{t,w,u\}$ (where t, w, and u stand for *think*, *wait*, and *use*, respectively), and $A2$ is a *counter* that takes values in the set of natural numbers. Analogously, the state of agent B is represented by a pair $\langle B1, B2\rangle$ of the control state $B1$ and the counter $B2$. Then the *state* of the system consisting of the two agents A and B is represented by the 4-tuple $\langle A1,A2,B1,B2\rangle$, and the transition relation t from an old state $OldS$ to a new state $NewS$, is defined by:

$t(OldS,\ NewS) \leftarrow t_A(OldS,\ NewS)$
$t(OldS,\ NewS) \leftarrow t_B(OldS,\ NewS)$

where the transition relation t_A for the agent A is given by the following clauses whose bodies are atomic constraints (in general, they may be conjunctions of atomic constraints) (see also Figure 1):

$t_A(\langle t, A2, B1, B2\rangle, \langle w, A21, B1, B2\rangle) \leftarrow A21 = B2{+}1$
$t_A(\langle w, A2, B1, B2\rangle, \langle u, A2, B1, B2\rangle) \leftarrow A2 < B2$
$t_A(\langle w, A2, B1, B2\rangle, \langle u, A2, B1, B2\rangle) \leftarrow B2 = 0$
$t_A(\langle u, A2, B1, B2\rangle, \langle t, A21, B1, B2\rangle) \leftarrow A21 = 0$

In a similar way the following clauses define the transition relation t_B for the agent B:

$t_B(\langle A1, A2, t, B2\rangle, \langle A1, A2, w, B21\rangle) \leftarrow B21 = A2{+}1$
$t_B(\langle A1, A2, w, B2\rangle, \langle A1, A2, u, B2\rangle) \leftarrow B2 < A2$
$t_B(\langle A1, A2, w, B2\rangle, \langle A1, A2, u, B2\rangle) \leftarrow A2 = 0$
$t_B(\langle A1, A2, u, B2\rangle, \langle A1, A2, t, B21\rangle) \leftarrow B21 = 0$

Note that the system has an infinite number of states, because counters may increase in an unbounded way.

The temporal properties of a transition system are specified by defining a predicate $sat(S,P)$ which holds if and only if the temporal formula P holds at state S. The clauses defining $sat(S,P)$ for the cases where P is: (i) an elementary formula F, (ii) a formula of the form $\neg F$, (iii) a formula of the form $F_1 \wedge F_2$, and (iv) a formula of the form $ef(F)$, are the following:

$sat(S,F) \leftarrow elem(S,F)$
$sat(S,\neg F) \leftarrow \neg sat(S,F)$
$sat(X,F_1 \wedge F_2) \leftarrow sat(X,F_1) \wedge sat(X,F_2)$
$sat(S, ef(F)) \leftarrow sat(S,F)$
$sat(S, ef(F)) \leftarrow t(S,T) \wedge sat(T, ef(F))$

where $elem(S,F)$ holds iff F is an elementary property which holds at state S. In particular, for the Bakery protocol, the following clause:

$elem(\langle u, A2, u, B2\rangle, unsafe) \leftarrow$

encodes that *unsafe* holds at a state where both agents A and B are in the control state u (that is, both agents are accessing the shared resource at the same time). Note that *ef* denotes the composition of *e* (there exists a computation path) and f (there exists a state on the path) and, indeed, $sat(S, ef(F))$ holds iff there exists a computation path π starting from S and a state s on π such that F holds at s.

The mutual exclusion property holds for the Bakery protocol if there is no computation path π starting from the initial state such that at a state on this path π the *unsafe* property holds. Thus, the mutual exclusion property holds if $sat(\langle t,0,t,0\rangle, \neg ef(unsafe))$ belongs to the perfect model $M(P_{mex})$, where: (i) $\langle t,0,t,0\rangle$ is the initial state of the system, and (ii) P_{mex} is the program consisting of the clauses, given above, defining the predicates t, t_A, t_B, *sat*, and *elem*. In order to show that $sat(\langle t,0,t,0\rangle, \neg ef(unsafe)) \in M(P_{mex})$, the following clause, defining the new predicate mex, is introduced:

$(\mu) \qquad mex \leftarrow sat(\langle t,0,t,0\rangle, \neg ef(unsafe))$

and the program $P_{mex} \cup \{\mu\}$ is transformed into a new program Q which contains a clause with empty body of the form: $mex \ \leftarrow$ (see [9] for details). This transformation is performed by using the definition, unfolding, and folding rules according to the specialization strategy [12,15,18] that in our case derives the clauses specialized to the evaluation of the predicate mex. By the correctness of the rules, $mex \in M(Q)$ iff $mex \in M(P_{mex} \cup \{\mu\})$ and, hence, $sat(\langle t,0,t,0\rangle, \neg ef(unsafe)) \in M(P_{mex})$, that is, the mutual exclusion property holds, because $mex \leftarrow$ is a clause of Q.

The derivation needed for verifying the mutual exclusion property was performed in a fully automatic way by using our experimental constraint logic program transformation system MAP [20].

5. Conclusions and Future Directions

This paper presents a brief overview of the program transformation methodology and illustrates its use for: (i) the derivation of correct software modules from their formal specifications, and (ii) the proof of properties of programs. Since program transformation preserves correctness and improves efficiency, it is very effective for constructing programs which are provably correct and whose performance is very high.

In order to make program transformation even more effective in the future, it is necessary to further develop and automate the transformation strategies for program improvement, program synthesis, and program verification, and incorporate them into powerful program development tools.

Topics for future research include the investigation of new application areas concerning, for instance, security properties of distributed systems, safety properties of hybrid systems, and protocol conformance of multi-agent systems. A more challenging long term goal is the fully automatic synthesis of programs which, by construction, are guaranteed to enjoy the properties expressed by their specification.

6. Acknowledgements

Many thanks to the guest editors of the Intelligenza Artificiale Journal for their kind invitation to write a paper in honor of Alberto Martelli. His paper [21] taught us how to write algorithms in a structural way.

References

[1] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A transformation system for lazy functional logic programs. In A. Middeldorp and T. Sato, editors, *Proc. 4th Fuji Int. Symp. Functional and Logic Programming*, LNCS 631, 147–162. Springer-Verlag, 1999.

[2] A. Bossi, N. Cocco, and S. Etalle. Transforming normal programs by replacement. In *Meta '92*, LNCS 649, 265–279. Springer-Verlag, 1992.

[3] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, 845–911. North Holland, 2001.

[4] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

[5] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[6] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Int. J. Software Tools for Technology Transfer*, 3(3):250–270, 2001.

[7] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theo. Comp. Sci.*, 166:101–146, 1996.

[8] S. Etalle, M. Gabbrielli, and M. C. Meo. Transformations of CCP programs. *ACM TOPLAS*, 23(3):304–395, 2001.

[9] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proc. VCL'01, Florence, Italy*, 85–96. Univ. Southampton, UK, 2001.

[10] F. Fioravanti, A. Pettorossi, and M. Proietti. Specialization with clause splitting for deriving deterministic constraint logic programs. In *IEEE Int. Conf. on Systems, Man and Cybernetics, Hammamet, Tunisia*, Vol. 1, 188–193. IEEE Computer Society Press, 2002.

[11] F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In *Program Development in Computational Logic*, LNCS 3049, 292–340. Springer-Verlag, 2004.

[12] J. P. Gallagher. Tutorial on specialisation of logic programs. In *PEPM '93, Copenhagen, Denmark*, 88–98. ACM Press, 1993.

[13] C. J. Hogger. Derivation of logic programs. *J. ACM*, 28(2): 372–392, 1981.

[14] J. Jaffar and M. Maher. Constraint logic programming: A survey. *J. of Logic Programming*, 19/20:503–581, 1994.

[15] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[16] L. Kott. The McCarthy's induction principle: 'oldy' but 'goody'. *Calcolo*, 19(1):59–69, 1982.

[17] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications ACM*, 17(8):453–455, 1974.

[18] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4&5):461–515, 2002.

[19] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.

[20] MAP. The MAP transformation system. http://www.iasi.cnr.it/~proietti/system.html, 1995–2010.

[21] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM TOPLAS*, 4(12):258–282, 1982.

[22] A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *J. of Logic Programming*, 41(2&3):197–230, 1999.

[23] A. Pettorossi, M. Proietti, and V. Senni. Proving properties of constraint logic programs by eliminating existential variables. In *ICLP '06*, LNCS 4079, 179–195. Springer, 2006.

[24] M. O. Rabin. Decidable theories. In Jon Barwise, editor, *Handbook of Mathematical Logic*, 595–629. North-Holland, 1977.

[25] A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, and S.A. Smolka. Verification of parameterized systems using logic program transformations. In *TACAS 2000, Berlin, Germany*, LNCS 1785, 172–187. Springer, 2000.

[26] T. Sato and H. Tamaki. Transformational logic program synthesis. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, 195–201. ICOT, 1984.

[27] H. Seki. Unfold/fold transformation of stratified programs. *Theo. Comp. Sci.*, 86:107–139, 1991.

[28] H. Seki. Unfold/fold transformation of general logic programs for well-founded semantics. *Journal of Logic Programming*, 16(1&2):5–23, 1993.

[29] P. Tacchella, M. Gabbrielli, and M. C. Meo. Unfolding in CHR. In *Proc. 9th Int. ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, 179–186, 2007.

[30] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proc. ICLP'84*, 127–138, Uppsala, Sweden. Uppsala University, 1984.