LA TRASFORMAZIONE DEI PROGRAMMI PER LO SVILUPPO, LA VERIFICA E LA SINTESI DEL SOFTWARE
PROGRAM TRANSFORMATION FOR DEVELOPMENT, VERIFICATION, AND SYNTHESIS OF SOFTWARE

Alberto Pettorossi, Maurizio Proietti, Valerio Senni

SOMMARIO/*ABSTRACT*

In questo articolo presentiamo brevemente la metodologia di trasformazione dei programmi per lo sviluppo di software corretto ed efficiente. Ci riferiremo, in particolare, al caso della trasformazione e dello sviluppo dei programmi logici con vincoli.

In this paper we briefly describe the use of the program transformation methodology for the development of correct and efficient programs. We will consider, in particular, the case of the transformation and the development of constraint logic programs.

Keywords: Constraint logic programming, model checking, 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 [3] and then it has been adapted to logic programs by Hogger [11] and Tamaki and Sato [26]. The main idea of this methodology is to transform, maybe in several steps and by applying different 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 is a *non*-executable specification of an algorithm, while the final program is executable, 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 development.

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 what follows we will illustrate the three uses of program transformation we have mentioned above, namely (i) program improvement, (ii) program synthesis, and (iii) program verification. In particular, we will consider the case of algorithms and specifications written as *constraint logic programs* [12] and we will focus our attention on the following transformation rules [1, 6, 8, 9, 25, 26]: *definition*, *unfolding*, *folding*, *goal replacement*, and *clause splitting* which will be applied according to some specific strategies. This approach to program transformation is, thus, called the *rules + strategies approach*.

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* [10, 13, 16], usually allows a great efficiency improvement. Let us present an example of this transformation by deriving a deterministic, specialized pattern matcher from a given nondeterministic, parametric pattern matcher and a specific pattern. In this example the matching relation on strings of numbers is the relation $le\_m(P,S)$ which holds between a pattern $P=[p_1,\ldots,p_n]$ and a string S iff in S there exists 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 any relation which 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. $le\_m(P,S) \leftarrow ap(B,C,S) \wedge ap(A,Q,B) \wedge le(P,Q)$
2. $ap([\,],\quad s,\quad s) \leftarrow$
3. $ap([X|\quad s],\quad s,[X|\quad s]) \leftarrow ap(\quad s,\quad s,\quad s)$
4. $le([\,],[\,]) \leftarrow$
5. $le([X|\quad s],[Y|\quad s]) \leftarrow X \leq Y \wedge le(Xs,Ys)$

where *ap* denotes list concatenation. Suppose that we want to use this general program in the case of the pattern $P =$ [1,0,2]. We start off our transformation by introducing the following clause by applying the so-called definition rule:

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

Clauses 1–6 constitute the initial program from which we begin our program transformation process by applying the transformation rules according to the so-called *Determinization Strategy* [8]. This strategy, which we will not present here, allows a fully automatic derivation of the deterministic, efficient pattern matcher. We unfold clause 6 w.r.t. the atom $\mathit{le\_m}([1,0,2], S)$, that is, we replace the atom $\mathit{le\_m}([1,0,2], S)$ which is an instance of the head of clause 1, by the corresponding instance of the body of clause 1. We get:

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

In order to fold clause 7, we introduce the following definition:

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

and then we fold clause 7, that is, we replace (the instance of) the body of a clause 8 which occurs in the body of clause 7 by (the corresponding instance of) the head of clause 8. We get:

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

Then we unfold clause 8 w.r.t. the atoms *ap* and *le* and we get:

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

Then we apply the clause splitting rule to clause 11, by separating the cases where $1 \leq X$ and $1 > X$. We get:

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

In order to fold clauses 10 and 12 we introduce the following two clauses defining the predicate $\mathit{ne}\;\;2$:

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

Then we fold clauses 10 and 12 by using the two clauses 14 and 15 and we also fold clause 13 by using clause 8. We get the following clauses which define $\mathit{ne}\;\;1$:

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

They are mutually exclusive because of the constraints $1 \leq X$ and $1 > X$. Now the program transformation continues in a similar way as above: we introduce the new predicates $\mathit{ne}\;\;3$ through $\mathit{ne}\;\;6$, we derive their defining clauses, and eventually, we get the following specialized, deterministic program:

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

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

3 Program Synthesis

Program synthesis is a technique for deriving programs from formal, possibly *non*-executable, specifications (see, for instance, [11, 24] for the derivation of logic programs from first-order logic specifications). In this section we present an example of use of the program transformation technique for performing program synthesis and deriving a constraint logic program from a first-order formula.

The example we will present is the N-queens example, which has been often considered in the literature for introducing programming techniques such as recursion and backtracking. The N-queens problem can be described as follows. We are required to place $N (\geq 0)$ queens 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. By using the fact that no two queens should lie on the same row, we represent the position of the N queens on the $N \times N$ chess board as a permutation $L = [i_1, \ldots, i_N]$ of the list $[1, \ldots, N]$ such that i_k is the column of the queen on row k.

A specification of the solution L for the N-queens problem is given by the following first-order formula $\varphi(N, L)$:

$\mathit{nat}(N) \wedge \mathit{nat\_list}(L) \wedge \mathit{length}(L, N) \wedge$
$\forall X\, (\mathit{mem\;\;er}(X, L) \rightarrow \mathit{in}(X, 1, N)) \wedge$
$\forall A, B, K, M$
$((1 \leq K \wedge K \leq M \wedge \mathit{occurs}(A, K, L) \wedge \mathit{occurs}(B, M, L))$
$\rightarrow (A \neq B \wedge A - B \neq M - K \wedge B - A \neq M - K))$

where the various predicates which occur in $\varphi(N, L)$ are defined by the following constraint logic program P:

$\mathit{nat}(0) \leftarrow$
$\mathit{nat}(N) \leftarrow N = M + 1 \wedge M \geq 0 \wedge \mathit{nat}(M)$
$\mathit{nat\_list}([\,]) \leftarrow$
$\mathit{nat\_list}([H|T]) \leftarrow \mathit{nat}(H) \wedge \mathit{nat\_list}(T)$

$length([\,],0) \leftarrow$
$length([H|T],N) \leftarrow N\!=\!M\!+\!1 \wedge M\!\geq\!0 \wedge length(T,M)$
$mem\ er(X,[H|T]) \leftarrow X\!=\!H$
$mem\ er(X,[H|T]) \leftarrow mem\ er(X,T)$
$in(X,M,N) \leftarrow X\!=\!N \wedge M\!\leq\!N$
$in(X,M,N) \leftarrow N\!=\!K\!+\!1 \wedge M\!\leq\!K \wedge in(X,M,K)$
$occurs(X,I,[H|T]) \leftarrow I\!=\!1 \wedge X\!=\!H$
$occurs(X,I\!+\!1,[H|T]) \leftarrow I\!\geq\!1 \wedge occurs(X,I,T)$

Now, we would like to synthesize a constraint logic program R which computes a predicate $ueens(N,L)$ such that the following Property π holds:

$(\pi) \quad M(R) \models\ ueens(N,L)$ iff $M(P) \models \varphi(N,L)$

where by $M(R)$ and $M(P)$ we denote the perfect model of the program R and P, respectively. By applying the technique presented in [9], we start off from the formula $ueens(N,L) \leftarrow \varphi(N,L)$. From that formula by applying a variant of the *Lloyd-Topor transformation* [17], we derive this stratified program F:

$ueens(N,L) \leftarrow nat(N) \wedge nat\_list(L) \wedge length(L,N) \wedge$
$\qquad \neg aux1(L,N) \wedge \neg aux2(L)$
$aux1(L,N) \leftarrow mem\ er(X,L) \wedge \neg in(X,1,N)$
$aux2(L) \leftarrow 1\!\leq\!K \wedge K\!\leq\!M \wedge$
$\qquad \neg(A\!\neq\!B \wedge A\!-\!B\!\neq\!M\!-\!K \wedge B\!-\!A\!\neq\!M\!-\!K) \wedge$
$\qquad occurs(A,K,L) \wedge occurs(B,M,L)$

It can be shown that this variant of the Lloyd-Topor transformation preserves the perfect model semantics and, thus, we have that:

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

The derived program $P\cup F$ is not very satisfactory from a computational point of view, when using SLDNF resolution with the left-to-right selection rule. Indeed, for a query of the form $ueens(n,L)$, where n is a nonnegative integer and L is a variable, program $P\cup F$ works by first generating a value l for L and then testing whether or not $length(l,n) \wedge \neg aux1(l,n) \wedge \neg aux2(l)$ holds. This generate-and-test behavior is very inefficient and it may also lead to nontermination. Thus, the process of program synthesis proceeds by applying the definition, unfolding, folding, and goal replacement transformation rules (see [9] for details), with the objective of deriving a more efficient, terminating program. We derive the following definite logic program R:

$ueens(N,L) \leftarrow ne\ \ 2(N,L,0)$
$ne\ \ 2(N,[\,],K) \leftarrow N\!=\!K$
$ne\ \ 2(N,[H|T],K) \leftarrow \quad \geq \quad +1 \wedge ne\ \ 2(N,T,K\!+\!1) \wedge$
$\qquad ne\ \ 3(H,T,N,0)$
$ne\ \ 3(A,[\,],N,M) \leftarrow in(A,1,N) \wedge nat(A)$
$ne\ \ 3(A,[B|T],N,M) \leftarrow A\!\neq\!B \wedge A\!-\!B\!\neq\!M\!+\!1\ \wedge$
$\qquad B\!-\!A\!\neq\!M\!+\!1\ \wedge nat(B) \wedge$
$\qquad ne\ \ 3(A,T,N,M\!+\!1)$

together with the clauses defining the predicates in and nat.

Since the transformation rules preserve the perfect model semantics, we have that $M(R) \models\ ueens(N,L)$ iff $M(P \cup F) \models\ ueens(N,L)$ and, thus, Property (π) holds. Moreover, it can be shown that R terminates for all queries of the form $ueens(n,L)$ and it computes a solution for the N-queens problem in a clever way: each time a queen is placed on the board, program R tests whether or not it attacks every other queen already placed on the board.

4 Program Verification

The proof of program properties is often needed during program development for checking the correctness of software components w.r.t. their specifications. In this section we see the use of program transformation for proving program properties specified either by first-order formulas or by temporal logic formulas.

Proofs performed by using program transformation have strong relationships with proofs by mathematical induction (see [2] for a survey on inductive proofs). In particular, the unfolding rule can be used for decomposing a formula of the form $\varphi(f(X))$, where $f(X)$ is a complex term, into a combination of n formulas of the forms $\varphi_1(X),\ldots,\varphi_n(X)$, and the folding rule can be used for applying inductive hypotheses.

It has been shown that the unfold/fold transformations introduced in [3, 26] can be used for proving several kinds of program properties, such as equivalences of functions defined by recursive equation programs [14], equivalences of predicates defined by logic programs [20], first-order properties of predicates defined by constraint logic programs [21], and temporal properties of concurrent systems [7, 23].

4.1 The unfold/fold proof method

By using a simple example taken from [21], we illustrate a method based on program transformation, called *unfold/fold proof method*, for proving first-order properties of constraint logic programs. Consider the following constraint logic program *Member* which defines the membership relation for lists:

$mem\ er(X,[Y|L]) \leftarrow X\!=\!Y$
$mem\ er(X,[Y|L]) \leftarrow mem\ er(X,L)$

Suppose we want to show that every finite list of numbers has an upper bound, i.e., the following formula holds:

$\varphi:\ \forall L\, \exists U\, \forall X\, (mem\ er(X,L) \rightarrow X \leq U)$

The unfold/fold proof method works in two steps. In the first step, φ is transformed into a set of clauses by applying a variant of the Lloyd-Topor transformation [17], thereby deriving the following program $Prop_1$:

$prop \leftarrow \neg p$
$p \leftarrow list(L) \wedge \neg q(L)$
$q(L) \leftarrow list(L) \wedge \neg r(L,U)$
$r(L,U) \!\leftarrow\! X\!>\!U \wedge list(L) \wedge mem\ er(X,L)$

The predicate $prop$ is equivalent to φ in the sense that $M(Mem\ er) \models \varphi$ iff $M(Mem\ er \cup Prop_1) \models prop$.

In the second step, we eliminate the *existential variables* occurring in $Prop_1$ (that is, the variables occurring in the body of a clause and not in its head) by applying the transformation strategy presented in [21]. We derive the following program $Prop_2$ which defines the predicate *prop*:

$prop \leftarrow \neg p$
$p \leftarrow p_1$
$p_1 \leftarrow p_1$

Now, $Prop_2$ is a propositional program and has a *finite* perfect model, which is $\{prop\}$. Since all transformations we have made can be shown to preserve the perfect model, we have that $M(Mem\ \ er) \models \varphi$ iff $M(Prop_2) \models prop$ and, therefore, we have completed the proof of φ because $prop$ belongs to $M(Prop_2)$.

Note that the unfold/fold proof method can be viewed as an extension to constraint logic programs of the *quantifier elimination* method, which has well-known applications in the field of automated theorem proving (see [22] for a brief survey).

4.2 Infinite-state model checking

Now we present a method for verifying temporal properties of infinite state systems based on the transformation of constraint logic programs [7].

As indicated in [4], 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* $t \subseteq S \times S$. We assume that t is a *total* relation, that is, for every state $s \in S$ there exists a state $s' \in S$, called *successor state* of s, such that $t(s, s')$ holds. A *computation path* starting from a (possibly not initial) 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}.

The properties of the evolution over time of a concurrent system are specified by using a temporal logic called *Computation Tree Logic* (or CTL, for short [4]) which specifies 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 connectives: $\neg$ ('not') and $\wedge$ ('and'), (ii) the following quantifiers along a computation path: g ('for all states on the path' or 'globally'), ('there exists a state on the path' or 'in the future'), x ('next time'), and u ('until'), and (iii) the quantifiers over computation paths: a ('for all paths') and e ('there exists a path').

Very efficient algorithms and tools exist for verifying temporal properties of *finite state systems*, that is, systems where the set S of states is finite [4]. However, many concurrent systems cannot be modeled by finite state systems. Unfortunately, the problem of verifying CTL properties of *infinite* state systems is undecidable, in general, and thus, it cannot be approached by traditional model checking techniques. For this reason various methods based on automated theorem proving have been proposed for enhancing model checking and allowing us to deal with infinite state

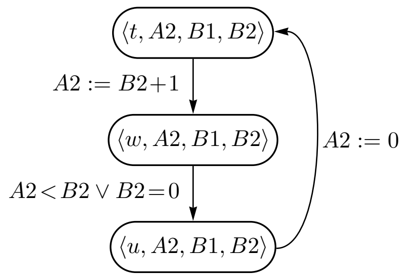
$\langle t, A2, B1, B2\rangle$
$A2 := B2{+}1$
$\langle w, A2, B1, B2\rangle$
$A2 := 0$
$A2 < B2 \vee B2 = 0$
$\langle u, A2, B1, B2\rangle$

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

systems (see [5] for a method based on constraint logic programming). Due to the above mentioned undecidability limitation, all these methods are incomplete.

As an example of use of program transformation for verifying CTL properties of infinite state systems, now we consider the *Bakery* protocol [15] and we verify that it satisfies the *mutual exclusion* and *starvation freedom* properties.

Let us consider two agents A and B which want to access a shared resource in a mutual exclusive way by using the Bakery protocol. The state of agent A is represented by a pair $\langle A1, A2\rangle$, where $A1$ is an element of the set $\{t, w, u\}$ of *control states* (where t, w, and u stand for *think*, *wait*, and *use*, respectively) and $A2$ is a *counter* that takes values from the set of natural numbers. Analogously, the state of agent B is represented by a pair $\langle B1, B2\rangle$. The *state* of the system consisting of the two agents A and B, whose states are $\langle A1, A2\rangle$ and $\langle B1, B2\rangle$, respectively, is represented by the 4-tuple $\langle A1, A2, B1, B2\rangle$. The transition relation t of the two agent system from an old state $\ \ ldS$ to a new state $\ \ e\ \ S$, is defined as follows:

$t(\ \ ldS,\ \ \ e\ \ S) \leftarrow t_A(\ \ ldS,\ \ \ e\ \ S)$
$t(\ \ ldS,\ \ \ e\ \ S) \leftarrow t_B(\ \ ldS,\ \ \ e\ \ S)$

where the transition relation t_A for the agent A is given by the following clauses whose bodies are conjunctions of constraints (see also Figure 1):

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

The following analogous clauses define the transition relation t_B for the agent B:

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

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 is true at state S. For instance, 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 $F_1 \wedge F_2$, (iv) a formula of the form $e\ \ (F)$, are the following ones:

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

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

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

that is, *unsafe* holds at a state where both agents A and B are in the control state u (both agents are accessing the shared resource at the same time).

Note that by *ef* we denote the composition of *e* (there exists a computation path) and *f* (there exists a state on the path) and, indeed, $sat(S, e\ \ (F))$ holds iff there exists a computation path π starting from S and a state s on π such that F is true at s.

The mutual exclusion property holds for the Bakery protocol if there is no computation path starting from the initial state such that at a state on this path the *unsafe* property holds. Thus, the mutual exclusion property holds if $sat(\langle t, 0, t, 0\rangle, \neg e\ \ (unsa\ e))$ 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 for the predicates t, t_A, t_B, *sat*, and *elem* defined above.

In order to show that $sat(\langle t, 0, t, 0\rangle, \neg e\ \ (unsa\ e)) \in M(P_{mex})$, we introduce a new predicate mex defined by the following clause:

$(\mu)\quad mex \leftarrow sat(\langle t, 0, t, 0\rangle, \neg e\ \ unsa\ e)$

and we transform the program $P_{mex} \cup \{\mu\}$ into a new program Q which contains a clause of the form $mex \leftarrow$. This transformation is performed by applying the definition, unfolding, and folding rules according to the specialization strategy, that is, a strategy that derives clauses specialized to the computation of predicate mex. From the correctness of the transformation rules we have that $mex \in M(Q)$ iff $mex \in M(P_{mex} \cup \{\mu\})$ and, hence, $sat(\langle t, 0, t, 0\rangle, \neg e\ \ (unsa\ e)) \in M(P_{mex})$, that is, the mutual exclusion property holds.

For the Bakery protocol we may also want to prove the starvation freedom property which ensures that an agent, say A, which requests the shared resource, will eventually get it. This property is expressed by the CTL formula: $ag(\ \ _A \rightarrow a\ \ (u_A))$, which is equivalent to: $\neg e\ \ ((\ \ _A \wedge \neg a\ \ (u_A))$. The clauses defining the elementary properties $\ \ _A$ and u_A are:

$elem(\langle\ \ , A2, B1, B2\rangle, \ \ _A) \leftarrow$
$elem(\langle u, A2, B1, B2\rangle, u_A) \leftarrow$

The clauses defining the predicate $sat(S, P)$ for the case where P is a CTL formula of the form $a\ \ (F)$ are:

$sat(X, a\ \ (F)) \leftarrow sat(X, F)$
$sat(X, a\ \ (F)) \leftarrow ts(X, \ \ s) \wedge sat\_all(\ \ s, a\ \ (F))$
$sat\_all([\,], F) \leftarrow$
$sat\_all([X|\ \ s], F) \leftarrow sat(X, F) \wedge sat\_all(\ \ s, F)$

where $ts(X, \ \ s)$ holds iff $\ \ s$ is a list of all successor states of X. For instance, one of the clauses defining predicate ts in our Bakery example is:

$$ts(\langle t, A2, t, B2\rangle, [\langle\ \ , A21, t, B2\rangle, \langle t, A2, \ \ , B21\rangle]) \leftarrow A21 = B2+1 \wedge B21 = A2+1$$

which states that the state $\langle t, A2, t, B2\rangle$ has two possible successor states: $\langle\ \ , A21, t, B2\rangle$ (with $A21 = B2+1$) and $\langle t, A2, \ \ , B21\rangle$ (with $B21 = A2+1$).

Let P_{sf} denote the program obtained by adding to P_{mex} the clauses defining: (i) the elementary properties $\ \ _A$ and u_A, (ii) the atom $sat(X, a\ \ (F))$, (iii) the predicate sat_all, and (iv) the predicate ts. In order to verify the starvation freedom property we introduce the clause:

$(\sigma)\quad s\ \ \leftarrow sat(\langle t, 0, t, 0\rangle, \neg e\ \ (\ \ _A \wedge \neg a\ \ (u_A)))$

and, by applying the definition, unfolding, and folding rules according to the specialization strategy, we transform the program $P_{sf} \cup \{\sigma\}$ into a new program R which contains a clause of the form $s\ \ \leftarrow$.

The derivations needed for verifying the mutual exclusion and the starvation freedom properties were performed in a fully automatic way by using our experimental constraint logic program transformation system MAP [18].

5 Conclusions and Future Directions

We have presented the program transformation methodology and we have demonstrated that it is very effective 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 useful for constructing software products which are provably correct and whose time and space performance is very high.

In order to make program transformation effective in practice we need to increase the level of automation of the transformation strategies for program improvement, program synthesis, and program verification. Furthermore, these strategies should be incorporated into powerful tools for program development.

An important direction for future research is also the exploration of new areas of application of the transformation methodology. In this paper we have described the use of program transformation for verifying temporal properties of infinite state concurrent systems. Similar techniques could also be devised for verifying other kinds of properties and other classes of systems, such as security properties of distributed systems, safety properties of hybrid systems, and protocol conformance of multiagent systems. A more challenging issue is the fully automatic synthesis of software systems which are guaranteed to satisfy the properties specified by the designer.

6 Acknowledgements

We would like to thank the editors of the Intelligenza Artificiale Magazine for their invitation to write this paper in honor of Prof. Alberto Martelli. We also thank Prof. Martelli for teaching us the 'structural way' of presenting algorithms which we learned from his unification paper [19].

REFERENCES

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

[2] 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.

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

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

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

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

[7] 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.

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

[9] 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.

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

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

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

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

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

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

[16] 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.

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

[18] MAP. The MAP transformation system. `http://www.iasi.cnr.it/~proietti/system.html`, 1995–2008.

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

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

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

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

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

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

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

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

7 Contacts

Alberto Pettorossi
Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata"
Via del Politecnico 1,
00133 Roma, Italy
+39 0672597379
pettorossi@disp.uniroma2.it

Maurizio Proietti (Corresponding Author)
Istituto di Analisi dei Sistemi e Informatica, Consiglio Nazionale delle Ricerche
Viale Manzoni 30,
00185 Roma, Italy
+39 067716426
maurizio.proietti@iasi.cnr.it

Valerio Senni
Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata"
Via del Politecnico 1,
00133 Roma, Italy
+39 0672597717
senni@disp.uniroma2.it

8 Biography

Alberto Pettorossi is professor of Theoretical Computer Science at the Engineering Faculty of the University of Roma "Tor Vergata". His research activity has been in the area of rewriting systems and concurrent computation, and it is now concerned with the automatic derivation, transformation, and verification of programs.

Maurizio Proietti received his Laurea degree in Mathematics from the University of Roma "Sapienza". He is a senior researcher at the Istituto di Analisi dei Sistemi e Informatica "A. Ruberti" of the National Research Council (IASI-CNR) in Roma. His research interests include various aspects of constraint and logic programming, and automatic methods for the transformation, synthesis, and verification of programs.

Valerio Senni is a Ph. D. student in Information Engineering at the the University of Roma "Tor Vergata". He holds a research contract for the development of automatic methods for proving program correctness.

Photograph. A black and white glossy photograph, passport- sized of each author is also required. It will be printed in the corresponding biography box.