A Method for Compiling and Executing Expressive Assertions

F.J. Galán Morillo and J.M. Cañete Valdeón

Dept. of Languages and Computer Systems. Faculty of Computer Science of Seville
Av. Reina Mercedes s/n 41012 Sevilla, Spain.
phone: (34) 95 455 27 73, fax: (34) 95 455 71 39
{galanm,canete}@lsi.us.es

Abstract. Programming with assertions constitutes an effective tool to detect and correct programming errors. The ability of executing formal specifications is essential in order to test automatically a program with respect to its assertions. However, formal specifications may describe recursive models which are difficult to identify so current assertion checkers limit, in a considerable way, the expressivity of the assertion language. In this paper, we are interested in showing how transformational synthesis can help to execute "expressive" assertions of the form $\forall x(r(x) \Leftrightarrow QyR(x,y))$ where x is a set of variables to be instantiated at execution time, Q is an existential or universal quantifier and R a quantifier free formula in the language of a particular first-order theory $\mathcal{A}$ we call assertion context. The class of assertion contexts is interesting because it presents a balance between expressiveness for writing assertions and existence of effective methods for executing them by means of synthesized (definite) logic programs.

Keywords: Assertion, correctness, formal specification, logic program, program synthesis, programming with assertions, meaning-preserving transformation, testing.

1 Introduction

Experience has shown that writing assertions while programming is an effective way to detect and correct programming errors. As an added benefit, assertions serve to document programs, enhancing maintainability. Programming languages such as Eiffel [18], SPARK [2] and recent extensions to the Java programming language, such as iContract [13], JML [16] and Jass [5], allow to write assertions into the program code in the form of pre-post conditions and invariants. Software components called assertion checkers are then used to decide if program assertions hold at execution time. However, and due to mechanization problems, current checkers do not accept the occurrence of unbounded quantification in assertions. This fact limits the expressivity of the assertion language and so the effectiveness of testing activities by means of runtime assertion checkers.

In order to motivate the problem, we show in Ex. 1 a (schematic) program which includes an unbounded quantified sub-formula in its post-condition.

Example 1. A program which returns a number different to zero if and only if the parameter l is a subset of the parameter s. Program types Nat and Set are used to represent natural numbers and sets of natural numbers respectively.

subset(l: Set, s: Set) *return* z: Nat
pre: $true$
  ... program code ...
post: $\neg idnat(z, 0) \Leftrightarrow \forall e(member(e, l) \Rightarrow member(e, s))$

where $idnat$ is the identity relation for natural numbers:

$$idnat:\ Nat \times Nat$$

$$idnat(0,0) \Leftrightarrow true$$
$$\forall x(idnat(s(x), 0) \Leftrightarrow false)$$
$$\forall y(idnat(0, s(y)) \Leftrightarrow false)$$
$$\forall x, y(idnat(s(x), s(y)) \Leftrightarrow idnat(x, y))$$

where $member$ is a relation to decide if a natural number is included or not in a set:

$$member:\ Nat \times Set$$

$$\forall e(member(e, []) \Leftrightarrow false)$$
$$\forall e(member(e, [x|y]) \Leftrightarrow (idnat(x, e) \vee member(e, y)))$$

At execution time, subset's program code will supply values (ground terms) to assertion variables l, s and z closing their interpretations. Thus, the correctness of a program behavior such as *subset*$([s(0)], [0, s(0)]) = s(0)$ will depend on the evaluation of $\neg idnat(s(0), 0) \Leftrightarrow \forall e(member(e, [s(0)]) \Rightarrow member(e, [0, s(0)]))$ (i.e. subset's post-condition after substituting l, s and z by values $[s(0)]$, $[s(0), 0]$ and $s(0)$ respectively). Due to the form of $idnat$'s axioms, it is not difficult to find a program able to evaluate ground atoms such as $idnat(s(0), 0)$, in fact, the if-part of $idnat$'s axioms can be considered one of such programs (Ex. 2). However, the occurrence of unbounded quantification in sub-formulas such as $\forall e(member(e, [s(0)]) \Rightarrow member(e, [0, s(0)]))$ complicates extraordinarily the search for such programs [9]. Due to this fact, current assertion checkers [18], [2], [5], [13], [16] does not consider the use of unbounded quantification in their assertion languages. Such a decision limits the expressivity of the assertion language and, therefore, the effectiveness of testing activities by means of runtime assertion checkers.

Example 2. A logic program which is able to evaluate ground atoms for $idnat$.

$$idnat(0, 0) \Leftarrow$$
$$idnat(s(x), s(y)) \Leftarrow idnat(x, y)$$

Our objective can be summarized in the following questions:

1. *Is it possible to extend current checkers to execute "expressive assertions"?* Firstly, we need to formalize what we call "expressive assertions". For us, an expressive assertion is a (new) relation r which represents a quantified

sub-formula $QyR(x,y)$ within a program assertion. Formally, r is defined by means of one axiom of the form $\forall x(r(x) \Leftrightarrow QyR(x,y))$ where x is a set of variables to be instantiated at execution time, Q is an existential or universal quantifier, and R a quantifier free formula in the language of a particular first order theory $\mathcal{A}$ called assertion context. For instance, $\forall l, s(r(l,s) \Leftrightarrow \forall e(member(e,l) \Rightarrow member(e,s)))$ is the definition of an expressive assertion r for the quantified sub-formula $\forall e(member(e,l) \Rightarrow member(e,s))$ in subset's post-condition (Ex. 1). Therefore, to answer "yes" to the question is equivalent to say that assertion checkers must be able to evaluate any ground atom $r(t)$ in $\mathcal{A}$ where t is the set of values supplied by the program at execution time.

2. *How can we do it?*
   Logic program synthesis constitutes an important aid in order to overcome the problem. Our intention is:
   - If $Q = \exists$, to synthesize a (definite) logic program $r_1^{\exists}$ of the form $r_1^{\exists}(x,y) \Leftarrow P^{\exists}(x,y)$ from a specification of the form $\forall x, y(r_1^{\exists}(x,y) \Leftrightarrow R(x,y))$.
   - If $Q = \forall$, to synthesize a (definite) logic program $r_1^{\forall}$ of the form $r_1^{\forall}(x,y) \Leftarrow P^{\forall}(x,y)$ from a specification of the $\forall x, y(r_1^{\forall}(x,y) \Leftrightarrow \neg R(x,y))$.

   In any case, synthesized programs r_1^{Q} must be *totally correct wrt goals of the form* $\Leftarrow r_1^{Q}(t,y)$. In order to synthesize logic programs, we have studied some of the most relevant synthesis paradigms (constructive, transformational and inductive) [7,8,9]. In particular, we are interested in transformational methods [4,14] however, some important problems are exposed in [6,8]: *"A transformation usually involves a sequence of unfolding steps, then some rewriting, and finally a folding step. The eureka about when and how to define a new predicate is difficult to find automatically. It is also hard when to stop unfolding. There is a need for loop-detection techniques to avoid infinite synthesis through symmetric transformations"*. In order to overcome these problems, we propose to develop program synthesis within assertion contexts [11]. Such a decision will allow us:
   - To *structure* the search space for new predicates.
   - To define a relation of similarity on formulas *for deciding when to define new predicates* and, from here, a particular folding rule *to define new predicates without human intervention.*
   - To define an incremental compilation method where *no symmetric transformations are possible.*
3. *How can assertion checkers evaluate ground atoms* $r(t)$ *from goals* $\Leftarrow r_1^{Q}(t,y)$ *in a refutation system such as Prolog?*
   - The execution of $\Leftarrow r_1^{\exists}(t,y)$ in a Prolog system will compute a set of substitutions $\{\theta_1, ..., \theta_j\}$ for y, thus:
     If $\{\theta_1, ..., \theta_j\} = \emptyset$, by total correctness of $r_1^{\exists}$, $\mathcal{A} \vdash \neg r(t)$ else $\mathcal{A} \vdash r(t)$.
   - The execution of $\Leftarrow r_1^{\forall}(t,y)$ in a Prolog system will compute a set of substitutions $\{\theta_1, ..., \theta_j\}$ for y, thus:
     If $\{\theta_1, ..., \theta_j\} = \emptyset$, by total correctness of $r_1^{\forall}$, $\mathcal{A} \vdash r(t)$ else $\mathcal{A} \vdash \neg r(t)$.

Our work is explained in the following manner. In Sect(s). 2 and 3 we introduce a set of preliminary definitions and a brief background on transformational synthesis respectively. Section 4 formalizes assertion contexts as a class of particular first-order theories to write expressive assertions. Then, Sect. 5 defines a compilation method for expressive assertions. Section 6 explains how to execute expressive assertions from compilation results. Finally, we conclude in Sect. 7.

2 Preliminary Definitions

This section introduces a set of preliminary definitions in order to clarify the vocabulary we will use in the rest of the paper.

Definition 1 (Term, Formula). *A term of type τ is defined inductively as follows: (a) a variable of type τ is a term of type τ, (b) a constant of type τ is a term of type τ, and (c) if f is an n-ary function symbol of type $\tau_1, ..., \tau_n \to \tau$ and t_i is a term of type τ_i ($i = 1..n$), then $f(t_1, ..., f_n)$ is a term of type τ.*

A formula is defined inductively as follows: (a) if r is a relation symbol of type $\tau_1, ..., \tau_n$ and t_i is a term of type τ_i ($i = 1..n$), then $r(t_1, ..., t_n)$ is a typed atomic formula (or simply an atom), (b) if F and G are typed formulas, then so are $\neg F$, $F \wedge G$, $F \vee G$, $F \Rightarrow G$ and $F \Leftrightarrow G$ and (c) if F is a typed formula and x is a variable of type τ, then $\forall_\tau x\, F$ and $\exists_\tau x\, F$ are typed formulas (for legibility reasons we will omit subscripts in quantifiers). A typed literal is a typed atom or the negation of a typed atom.

A ground term (or value) is a term not containing variables. Similarly, a ground formula is a formula not containing variables.

A closed formula is a formula whose variables are quantified. A quantifier-free formula is a formula without quantifiers.

Definition 2 (Herbrand base). *The Herbrand universe of a first order language L is the set of all ground terms, which can be formed out of the constants and function symbols appearing in L. The Herbrand base for L is the set of all ground atoms which can be formed by using relation symbols from L with ground terms from the Herbrand universe of L.*

Definition 3 (Patterns). *A term pattern $t_{\mathcal{P}}$ is obtained from a term t by replacing each variable occurrence in t by the symbol $\_$. An atom pattern $r(t)_{\mathcal{P}}$ is obtained from an atom $r(t)$ by replacing every term occurrence in $r(t)$ by its respective term pattern. We say that $l_{\mathcal{P}}$ is a literal pattern for $r(t)$ if either $l_{\mathcal{P}} = r(t)_{\mathcal{P}}$ or $l_{\mathcal{P}} = \neg r(t)_{\mathcal{P}}$. A formula pattern $F_{\mathcal{P}}$ is obtained from a quantifier-free formula F by replacing every literal in F by its respective literal pattern.*

We say that $t_{1\mathcal{P}} > t_{2\mathcal{P}}$ if either $t_{1\mathcal{P}} = \_$ and $t_{2\mathcal{P}} = f(t_{2,1\mathcal{P}}, ..., t_{2,n\mathcal{P}})$ or $t_{1\mathcal{P}} = f(t_{1,1\mathcal{P}}, ..., t_{1,n\mathcal{P}})$ and $t_{2\mathcal{P}} = f(t_{2,1\mathcal{P}}, ..., t_{2,n\mathcal{P}})$ and there exists a non-empty subset $S \subseteq \{1..n\}$ such that $t_{1,i\mathcal{P}} > t_{2,i\mathcal{P}}$ for every $i \in S$ and $t_{1,j\mathcal{P}} = t_{2,j\mathcal{P}}$ for every $j \in (\{1..n\} - S)$.

Let $r(t_{1,1}, ..., t_{1,n})_{\mathcal{P}}$ and $r(t_{2,1}, ..., t_{2,n})_{\mathcal{P}}$ be two atom patterns, we say that $r(t_{1,1}, ..., t_{1,n})_{\mathcal{P}} > r(t_{2,1}, ..., t_{2,n})_{\mathcal{P}}$ if there exists a non-empty set $S \subseteq \{1..n\}$ such that $t_{1,i\mathcal{P}} > t_{2,i\mathcal{P}}$ for every $i \in S$ and $t_{1,j\mathcal{P}} = t_{2,j\mathcal{P}}$ for every $j \in (\{1..n\} - S)$.

Definition 4 (Definite Logic Program, Definite Goal). *A definite logic clause is a universally closed formula of the form $A \Leftarrow B_1, ..., B_n$ where A, B_1, ..., B_n are atoms. A definite logic program is a finite set of definite program clauses (Ex. 2 shows a definite logic program). A definite goal is a clause of the form $\Leftarrow B_1, ..., B_n$.*

3 Background on Transformational Synthesis

In transformational synthesis, a sequence of meaning preserving transformation rules is applied to a specification until a program is obtained [8]. This kind of stepwise forward reasoning is feasible with axiomatic specifications of the form $\forall x(r(x) \Leftrightarrow R(x))$. There are atomic transformation rules such as unfolding (replacing an atom by its definition), folding (replacing a sub-formula by an atom) and rewrite and simplification rules. The objective of applying transformations is to filter out a new version of the specification where recursion may be introduced by a folding step. This usually involves a sequence of unfolding steps, then some rewriting, and finally a folding step. These atomic transformation rules constitute a correct and complete set for exploring the search space, however they lead to very tedious synthesis due to the no existence of a guiding plan, except for the objective of introducing recursion. The "eureka" about when and how to define a new predicate is difficult to find automatically. It is hard to decide when to stop unfolding and also there is a need for detecting symmetric transformations to avoid infinite synthesis.

4 Assertion Contexts

As we said in Sect. 1, an expressive assertion is written in the language of a particular first order theory called assertion context. In Ex. 3 we show assertion context $\mathcal{A}$ from which subset's post-condition in Ex. 1 has been written. The Herbrand universe of $\mathcal{A}$ is formed out of the constants 0, $[\,]$ and function symbols s and $[\,|\,]$. Every assertion in $\mathcal{A}$ (*idnat* and *member*) is formalized by means of a relation symbol, a signature and a finite set of first order axioms. The Herbrand base of $\mathcal{A}$ is the set of all ground atoms which can be formed by using relation symbols *idnat* and *member* with ground terms from the Herbrand universe of $\mathcal{A}$.

Example 3 (Assertion context from the specifier's point of view).

Assertion Context $\mathcal{A}$
Nat generated by $0 :\rightarrow Nat, s : Nat \rightarrow Nat$
Set generated by $[\,] :\rightarrow Set, [\,|\,] : Nat \times Set \rightarrow Set$

Assertion $idnat : Nat \times Nat$
$idnat(0,0) \Leftrightarrow true$ $\quad \forall x(idnat(s(x),0) \Leftrightarrow false)$
$\forall y(idnat(0,s(y)) \Leftrightarrow false)$ $\quad \forall x,y(idnat(s(x),s(y)) \Leftrightarrow idnat(x,y))$

Assertion $member : Nat \times Set$
$\forall e(member(e, [\,]) \Leftrightarrow false)$
$\forall e, x, y(member(e, [x|y]) \Leftrightarrow (idnat(x, e) \vee member(e, y)))$

It is important to remark that assertion contexts must be processed by means of assertion checkers (i.e. software components) so reasonable restrictions have to be imposed on the form of axioms in order to make feasible their automatic processing. Example 4 shows $\mathcal{A}$ from the assertion checker's point of view where some redundant information (i.e. layers and patterns) are explicitly shown.

Example 4 (Assertion context from the assertion checker's point of view).

Assertion Context $\mathcal{A}$
Nat generated by $0 :\rightarrow Nat, s : Nat \rightarrow Nat$
Set generated by $[\,] :\rightarrow Set, [\;|\;] : Nat \times Set \rightarrow Set$

Assertion $idnat : Nat \times Nat$
$idnat(0, 0) \Leftrightarrow true$ $\quad\forall x(idnat(s(x), 0) \Leftrightarrow false)$
$\forall y(idnat(0, s(y)) \Leftrightarrow false)$ $\quad\forall x, y(idnat(s(x), s(y)) \Leftrightarrow idnat(x, y))$

Assertion $member : Nat \times Set$
$\forall e(member(e, [\,]) \Leftrightarrow false)$
$\forall e, x, y(member(e, [x|y]) \Leftrightarrow (idnat(x, e) \vee member(e, y)))$

Layers.
layer 0: $idnat$ **layer 1:** $member$
Patterns.
l-patterns: $idnat(0, 0)$, $idnat(0, s(\_))$, $idnat(s(\_), 0)$, $idnat(s(\_), s(\_))$
$member(\_, [\,])$, $member(\_, [\_|\_])$
i-patterns: $idnat(\_, 0)$, $idnat(\_, s(\_))$, $idnat(0, \_)$, $idnat(s(\_), \_)$
u-patterns: $idnat(\_, \_)$, $member(\_, \_)$

For the purpose of writing consistent contexts, the following restrictions have been imposed on assertions:

1. *Every axiom is of the form $\forall(a(x) \Leftrightarrow B(z))$ with $z \subseteq x$ where $a(x)$ is an atom called the left-hand side (lhs) of the axiom and $B(z)$ is a quantifier-free formula composed of literals and binary logical connectives called the right-hand side (rhs) of the axiom.*
2. *Every element in the Herbrand base of $\mathcal{A}$ unifies with the lhs of a unique axiom.*
3. *Every assertion is encapsulated in a layer. If a is located at layer i, a is a symbol of level i. Every positive atom occurring in $B(z)$ is defined on the symbol a or on a symbol of level $i - 1$ (if possible). Every negative atom occurring in $B(z)$ is defined on a symbol of level $i - 1$.*
4. *Every recursive specification is well-founded wrt a set of parameters.*

Theorem 1 (Ground Decidability). *For every $a(t)$ in the Herbrand base of $\mathcal{A}$ either $\mathcal{A} \vdash a(t)$ or $\mathcal{A} \vdash \neg a(t)$.* (A proof of this theorem can be found in [12]).

From Theorem 1, we formalize the semantics of assertion contexts. Our proposal is borrowed from previous results in the field of deductive synthesis [3], [14], [15].

Definition 5 (Consistency). *A model for $\mathcal{A}$ is defined in the following terms:*

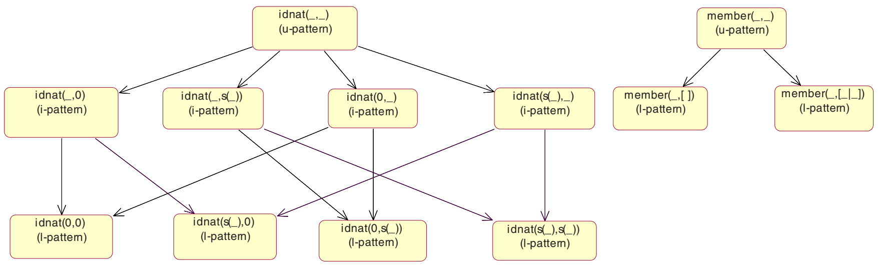
$$\mathcal{A} \models a(t) \;\; \textit{iff} \;\; \mathcal{A} \vdash a(t) \;\; \textit{and} \;\; \mathcal{A} \models \neg a(t) \;\; \textit{iff} \;\; \mathcal{A} \vdash \neg a(t)$$

for every $a(t)$ in the Herbrand base of $\mathcal{A}$.

For the purpose of structuring the search space for new predicates [8], assertion contexts are enriched by means of a set of atom patterns. We classify atom patterns into three categories: lower patterns (l-patterns), intermediate pattern (i-patterns) and upper patterns (u-patterns). A lower pattern is calculated from the atom on the lhs of an axiom and a upper pattern is calculated from an atom on the rhs of an axiom. The rest of atom patterns (i.e. intermediate patterns) are calculated from upper and lower patterns via $>$ (Sect. 2, Def.3): every intermediate pattern is lesser than any upper pattern and greater than any lower pattern.

5. *Every atom occurring on the rhs of an axiom presents an intermediate atom pattern or a upper atom pattern.*

For the purpose of legibility, we display atom patterns by means of directed graphs where atom patterns are nodes and directed links are instances of the relation $>$. For instance, Fig. 1 shows the set of atom patterns in $\mathcal{A}$ (Ex. 4).



idnat(_,_) (u-pattern)
member(_,_) (u-pattern)
idnat(_,0) (i-pattern)
idnat(_,s(_)) (i-pattern)
idnat(0,_) (i-pattern)
idnat(s(_),_) (i-pattern)
member(_,[ ]) (l-pattern)
member(_,[_|_]) (l-pattern)
idnat(0,0) (l-pattern)
idnat(s(_),0) (l-pattern)
idnat(0,s(_)) (l-pattern)
idnat(s(_),s(_)) (l-pattern)

Fig. 1. Graph-based description of the set of atom patterns in $\mathcal{A}$.

Once we have formalized the notion of assertion context, we can formalize the notion of expressive assertion. Roughly speaking, an expressive assertion is a (new) relation r intended to represent a quantified sub-formula of the form $QyR(x, y)$ within a program assertion.

Definition 6 (Expressive Assertion). *We say that r is an expressive assertion in $\mathcal{A}$ if and only if r is a new symbol not occurring $\mathcal{A}$ which is defined by means of a unique axiom of the form $\forall x(r(x) \Leftrightarrow Q\, yR(x,y))$ where x is a set of variables to be instantiated at execution time, Q is a (existential or universal) quantifier and R is a quantifier-free formula in the language of $\mathcal{A}$ where every atom presents an intermediate atom pattern or an upper atom pattern in $\mathcal{A}$.*

Example 5 (Expressive assertion for $\forall e(member(e,l) \Leftrightarrow member(e,s))$ in subset*'s post-condition (Ex. 1)).*

$$\textsf{Assertion} \quad r : Set \times Set$$
$$\forall l, s(r(l,s) \Leftrightarrow \forall e(member(e,l) \Rightarrow member(e,s)))$$

5 Compilation Method

This section explains how transformational synthesis can help to compile an expressive assertion $\forall x(r(x) \Leftrightarrow Q\, yR(x,y))$. As we said in Sect. 1, our intentions are: If $Q = \exists$, to synthesize a *totally correct* (definite) logic program $r_1^{\exists}$ of the form $r_1^{\exists}(x,y) \Leftarrow P^{\exists}(x,y)$ from an auxiliary specification $\forall x, y(r_1^{\exists}(x,y) \Leftrightarrow R(x,y))$ and if $Q = \forall$, to synthesize a *totally correct* (definite) logic program $r_1^{\forall}$ of the form $r_1^{\forall}(x,y) \Leftarrow P^{\forall}(x,y)$ from an auxiliary specification $\forall x, y(r_1^{\forall}(x,y) \Leftrightarrow \neg R(x,y))$.

Example 6 (Auxiliary specification for r (Ex. 5)). In order to normalize the form of formulas (Def. 9), an equivalent formula $\forall(r_1^{\forall}(e,l,e,s) \Leftrightarrow \neg(member(e,l) \Rightarrow member(e,s)))$ is considered for $\forall(r_1^{\forall}(e,l,e,s) \Leftrightarrow (member(e,l) \wedge \neg member(e,s)))$.

$$\textsf{Assertion} \quad r_1^{\forall} : Nat \times Set \times Nat \times Set$$
$$\forall e, l, s(r_1^{\forall}(e,l,e,s) \Leftrightarrow (member(e,l) \wedge \neg member(e,s)))$$

A compilation is done by means of a finite sequence of meaning-preserving transformation steps. Each transformation step is composed of an expansion phase followed by a reduction phase. An expansion phase is intended to decompose a formula into a set of formulas and a reduction phase is intended to replace sub-formulas by new predicates. As we will show later, the set of new predicates ("recursive predicates") is computable.

5.1 Expansion Phase

Expansion phase decomposes a formula F into a set of formulas by means of instantiations and unfolding steps. Our intention is to decomposed F in a guided manner by using a particular rule called instantiation. To implement instantiations, we will use atom patterns in the following terms: if $r(y)$ is a selected atom to be instantiated in F and $r(y)_{\mathcal{P}}$ is a pattern which dominates a subgraph in the graph-based description of atom patterns for r then lower patterns in such a subgraph will induce a set of substitutions for variables in $r(y)$. Such sets of

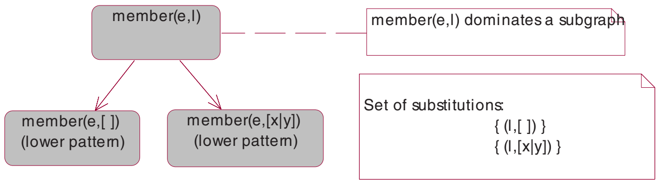


Fig. 2. Set of substitutions for $member(e,l)$.

substitutions will be the basis to construct instantiations. In Fig. 2 we show an example of a set of substitutions for an atom.

In Def(s). 7-9, we consider that F is a formula of the form $\forall(r_i(x) \Leftrightarrow R(x))$ with $R(x)$ as a quantifier-free formula written in (the language of an assertion context) $\mathcal{A}$.

Definition 7 (i-Instantiation). *We say that $inst(F, i, a(y)) = \{\forall(r_i(x)\theta_1 \Leftrightarrow R(x)\theta_1), ..., \forall(r_i(x)\theta_j \Leftrightarrow R(x)\theta_j)\}$ is the i-instantiation of an atom $a(y)$ in F if and only if*

1. *$a(y)$ is an atom in $R(x)$ of level i where $a(y)_{\mathcal{P}}$ dominates a subgraph of patterns with lower patterns $\{a(y_1)_{\mathcal{P}}, ..., a(y_j)_{\mathcal{P}}\}$.*
2. *$\{\theta_1, ..., \theta_j\}$ is the set of substitutions such that $(a(y)\theta_k)_{\mathcal{P}} = a(y_k)_{\mathcal{P}}$ $(k = 1..j)$.*
3. *Every atom in $R(x)\theta_k$ presents an atom pattern in $\mathcal{A}$.*

Example 7 (Instantiation of $member(e,l)$ in the axiom of $r^{\forall}$ in Ex. 6).

$$\boxed{\begin{array}{ll} \forall\,(r_1^{\forall}(e,[\,],e,s) \Leftrightarrow (member(e,[\,]) \wedge \neg member(e,s))) & \theta_1 = \{(l,[\,])\} \\ \forall\,(r_1^{\forall}(e,[x|y],e,s) \Leftrightarrow (member(e,[x|y]) \wedge \neg member(e,s))) & \theta_2 = \{(l,[x|y])\} \end{array}}$$

Definition 8 (Unfolding Step). *Let $Ax = \forall(a(x) \Leftrightarrow B(z))$ be an axiom in an assertion context $\mathcal{A}$ and $a(y)$ an atom occurring in F with $a(x)\theta = a(y)$. We say that $unf(F, a(y), Ax)$ is the unfolding step of $a(y)$ in F wrt Ax if and only if $a(y)$ is replaced in F by $B(z)\theta$.*

Definition 9 (Normalization Rules). *To avoid negations in front of formulas, we normalize them by using the following set of rewrite rules where G and H are quantifier-free formulas.*

$$\begin{array}{ll} (1)\ \neg false \rightarrow true, & (2)\ \neg true \rightarrow false \\ (3)\ \neg\neg G \rightarrow G, & (4)\ \neg(G \Rightarrow H) \rightarrow (G \wedge \neg H) \\ (5)\ \neg(G \wedge H) \rightarrow (\neg G \vee \neg H) & (6)\ \neg(G \vee H) \rightarrow (\neg G \wedge \neg H) \\ (7)\ \neg(G \Leftrightarrow H) \rightarrow (\neg(G \Rightarrow H) \vee \neg(H \Rightarrow G)) & \end{array}$$

Definition 10 (i-Expansion). *We say that the set of formulas $exp(F, i)$ is the i-expansion of F if and only if every formula in $exp(F, i)$ is constructed by applying all i-instantiations (at least one) to F and then all unfolding steps (at least one) to each resulting formula. After unfolding steps, it can appear negative sub-formulas (i.e. presence of negation in front of unfolded sub-formulas). To avoid negations in front of such sub-formulas, we normalize them.*

Example 8 (Expansion of the axiom for $r_1^\forall$ in Ex. 6).

$$\boxed{\begin{array}{l}
(1)\ \forall\,(r_1^\forall(e,[\,],e,[\,]) \Leftrightarrow (\mathit{false} \wedge \mathit{true}))\\
(2)\ \forall\,(r_1^\forall(e,[\,],e,[v|w]) \Leftrightarrow (\mathit{false} \wedge (\neg \mathit{idnat}(v,e) \wedge \neg \mathit{member}(e,w))))\\
(3)\ \forall\,(r_1^\forall(e,[x|y],e,[\,]) \Leftrightarrow ((\mathit{idnat}(x,e) \vee \mathit{member}(e,y)) \wedge \mathit{true}))\\
(4)\ \forall\,(r_1^\forall(e,[x|y],e,[v|w]) \Leftrightarrow ((\mathit{idnat}(x,e) \vee \mathit{member}(e,y))\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\wedge\\
\qquad\qquad\qquad\qquad\qquad(\neg \mathit{idnat}(v,e) \wedge \neg \mathit{member}(e,w))))
\end{array}}$$

5.2 Structuring the Search Space

Once an expressive assertion has been "decomposed" into a set of formulas (expansion), we are interested in *finding recursive compositions*. This can be done by identifying sub-formulas and replacing them by new predicates (reduction). Our intention is to anticipate and organize the search space for sub-formulas and new predicates in order to manage reductions automatically. Thus, after expanding an expressive assertion, we must be able for predicting the set of all possible resulting formulas.

Definition 11 (Search Space). *Let $\forall x(r(x) \Leftrightarrow Q\,yR(x,y))$ be an expressive assertion in an assertion context $\mathcal{A}$ and $\forall x,y(r^Q(x,y) \Leftrightarrow R(x,y))$ the auxiliary specification from which a logic program has to be synthesized for r. We say that the set of formula patterns $\Omega(r)$ is the search space for r if and only if it includes the set of all formula pattern combinations which results of replacing all literals in R by atom patterns in $\mathcal{A}$ and by negative forms of atoms patterns in $\mathcal{A}$.*

Every element $E_\mathcal{P}$ in $\Omega(r)$ encodes a sort of formulas. Such a codification depends on the sequence of relation symbols in $E_\mathcal{P}$. Our method considers that every formula pattern $E_\mathcal{P}$ is equivalent to a fresh atom pattern whose relation symbol, say $r_{E_\mathcal{P}}$, represents such a codification. In order to establish a precise codification, a bijection is proposed between term patterns in $E_\mathcal{P}$ and parameter positions in $r_{E_\mathcal{P}}$. We say that $\Omega_{ext}(r)$ is the *extended search space* for r if and only if $\Omega_{ext}(r)$ is formed from $\Omega(r)$ by including an element of the form $r_{E_\mathcal{P}} \Leftrightarrow E_\mathcal{P}$ for each element $E_\mathcal{P} \in \Omega(r)$. Thus, an extended search space represents a repository of new predicates (i.e. $r_{E_\mathcal{P}}$) and sub-formulas (i.e. $E_\mathcal{P}$) to be considered at reduction time.

Experimentally, it is important to note that no complete extended search spaces are needed when compiling expressive assertions. For instance, from a theoretical point of view, $|\Omega_{ext}(r_1^\forall)| = 196$ but only 9 of these patterns have been needed when compiling $r_1^\forall$ (Table 1). A practical result is proposed in [10] where we show that search spaces can be constructed on demand using tabulation techniques.

5.3 Similarity. A Criterion for Identifying New Predicates

In order to automate reductions, we propose a method to decide when a formula is similar to an element in a search space. We supply "operational" definitions to justify the mechanization of our proposal.

Table 1. $\Omega_{ext}(r_1^{\forall})$ (partial).

(1) $r_1^{\forall}(\_1,\_2,\_3,\_4) \Leftrightarrow (member(\_1,\_2) \wedge \neg member(\_3,\_4))$
(2) $r_2^{\forall}(\_1,\_2) \Leftrightarrow (false \wedge \neg member(\_1,\_2))$
(3) $r_3^{\forall}(\_1,\_2) \Leftrightarrow (member(\_1,\_2) \wedge false)$
(4) $r_4^{\forall}(\_1,\_2) \Leftrightarrow (true \wedge \neg member(\_1,\_2))$
(5) $r_5^{\forall}(\_1,\_2) \Leftrightarrow (member(\_1,\_2) \wedge true)$
(6) $r_6^{\forall} \Leftrightarrow (false \wedge true)$
(7) $r_7^{\forall} \Leftrightarrow (false \wedge false)$
(8) $r_8^{\forall} \Leftrightarrow (true \wedge false)$
(9) $r_9^{\forall} \Leftrightarrow (true \wedge true)$

By $tree(R_{\mathcal{P}})$ we denote the *parse tree* of a formula pattern $R_{\mathcal{P}}$ where each leaf node contains a literal pattern and each internal node contains a binary logical connective. We say that a node in $tree(R_{\mathcal{P}})$ is *preterminal* if it has, at least, one leaf node.

We say that $R_{\mathcal{P}}$ is *similar wrt connectives* to $E_{\mathcal{P}}$ if and only if every binary logical connective in $tree(E_{\mathcal{P}})$ is located at the same place in $tree(R_{\mathcal{P}})$. In Fig. 3, we show that $R_{\mathcal{P}}$ is similar wrt connectives to $E_{\mathcal{P}}$ (but $E_{\mathcal{P}}$ is not similar wrt connectives to $R_{\mathcal{P}}$). Similarity wrt connectives induces a mapping f from preterminal nodes in $tree(E_{\mathcal{P}})$ to subtrees in $tree(R_{\mathcal{P}})$ (for instance, f in Fig. 3).

Definition 12 (Similar Pattern). *We say that a formula pattern $R_{\mathcal{P}}$ is similar to a formula pattern $E_{\mathcal{P}}$ if and only if*

1. *$R_{\mathcal{P}}$ is similar wrt connectives to $E_{\mathcal{P}}$ via mapping f,*
2. *a) For each preterminal node $n \in tree(E_{\mathcal{P}})$ with two leaf nodes, $e_{left_{\mathcal{P}}}$ and $e_{right_{\mathcal{P}}}$, there exist two leaf nodes, $l_{left_{\mathcal{P}}}$ on the left subtree of $f(n)$ and $l_{right_{\mathcal{P}}}$ on the right subtree of $f(n)$, where $e_{left_{\mathcal{P}}} = l_{left_{\mathcal{P}}}$ and $e_{right_{\mathcal{P}}} = l_{right_{\mathcal{P}}}$ and*
   b) For each preterminal node $n \in tree(E_{\mathcal{P}})$ with one leaf node, $e_{left_{\mathcal{P}}}/e_{right_{\mathcal{P}}}$, there exists a leaf node $l_{left_{\mathcal{P}}}/l_{right_{\mathcal{P}}}$ on the left/right subtree of $f(n)$, where $e_{left_{\mathcal{P}}}/e_{right_{\mathcal{P}}} = l_{left_{\mathcal{P}}}/l_{right_{\mathcal{P}}}$.

Figure 3 shows an example of similarity.

At this point, two interesting results are presented. The first one (Theorem 2) establishes that expansions preserve semantics and the second one (Theorem 3), which is needed to ensure termination, establishes that every formula resulting from an expansion can be reduced to a new predicate in the (extended) search space.

Theorem 2 (Expansion Preserves Correctness). *Let $\{\forall(r_i(x_1) \Leftrightarrow R_1^{exp}(x_1))$, ..., $\forall(r_i(x_j) \Leftrightarrow R_j^{exp}(x_j))\}$ be the set of resulting formulas in the i-expansion of a formula $\forall(r_i(x) \Leftrightarrow R(x))$. For every ground atom $r_i(x)\phi$ there exists a ground atom $r_i(x_k)\delta$ with $k \in \{1..j\}$ such that*

$$\mathcal{A} \models R(x)\phi \Leftrightarrow R_k^{exp}(x_k)\delta$$

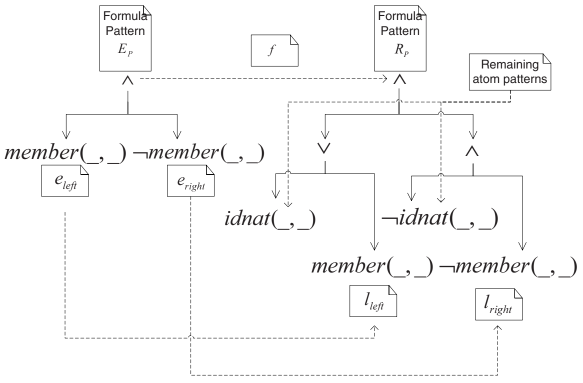


Fig. 3. $R_{\mathcal{P}}$ is similar to $E_{\mathcal{P}}$.

(A proof of this theorem can be found in [12]).

Theorem 3 (Expansion is an Internal Operation in Ω_{ext}). *Let $\{\forall(r_i(x_1)\Leftrightarrow R_1^{exp}(x_1)),...,\forall(r_i(x_j)\Leftrightarrow R_j^{exp}(x_j))\}$ be the i-expansion of a formula $\forall(r_i(x) \Leftrightarrow R(x))$. If $R(x)_{\mathcal{P}}$ is similar to the rhs of some pattern in $\Omega_{ext}(r_i)$ then $R_k^{exp}(x_k)_{\mathcal{P}}$ is also similar to the rhs of some pattern in $\Omega_{ext}(r_i)$ $(k = 1..j)$.* (A proof of this theorem can be found in [12]).

5.4 Reduction Phase

Reduction phase is intended to replace sub-formulas by atoms. To identify and replace sub-formulas by equivalent atoms are two key activities in a transformation step. Once a formula is similar to an element in a search space then it is rewritten (rewriting step), preserving its semantics, in order to facilitate an automatic replacement of sub-formulas by new predicates (folding step). In Def(s). 14, 15 and 16, we consider that F is a formula of the form $\forall(r_i(x) \Leftrightarrow R(x))$ with $R(x)$ a quantifier-free formula written in (the language of an assertion context) $\mathcal{A}$ and r a symbol not defined in $\mathcal{A}$.

Definition 13 (Simplification Rules). *In order to simplify formulas in presence of propositions $true$ and $false$, we consider the following set of rewrite rules where H is a formula.*

(1) $\neg true \to false$	(2) $\neg false \to true$	(3) $true \vee H \to true$
(4) $false \vee H \to H$	(5) $true \wedge H \to H$	(6) $false \wedge H \to false$
(7) $false \Rightarrow H \to true$	(8) $true \Rightarrow H \to H$	(9) $false \Leftrightarrow H \to \neg H$
(10) $H \Rightarrow true \to true$	(11) $H \Rightarrow false \to \neg H$	(12) $true \Leftrightarrow H \to H$
(13) $\forall(true) \to true$	(14) $\forall(false) \to false$	

In the following definition, we use $R(a_1, ..., a_p, ..., a_n)$ to refer to $R(x)$ where $\{a_1, ..., a_p, ..., a_n\}$ is the set of all the atoms occurring in $R(x)$.

Definition 14 (Rewriting Step). *Let $P_{\mathcal{P}}$ be a pattern in $\Omega_{ext}(r_i)$ such that $R(x)_{\mathcal{P}}$ is similar to $E_{\mathcal{P}} = rhs(P_{\mathcal{P}})$ with f as the induced mapping for deciding similarity wrt connectives and $\{a_1, a_2, ..., a_p\}$ as the set of atoms in $R(a_1, ...,$- $a_p, ..., a_n)$ which have not been used for deciding similarity (for instance, remaining atoms in Fig. 3). We say that $rew(F, P_{\mathcal{P}})$ is the rewriting step of F wrt $P_{\mathcal{P}}$ if and only if*

(Step 1) *We calculate the set of all the possible evaluations for $a_1, a_2, ..., a_p$ in $R(x)$ in the following schematic manner:*

$$
\begin{array}{l}
rew(F, P_{\mathcal{P}}) = \forall(r_i(x) \Leftrightarrow \\
\qquad \begin{array}{lllll}
(R(true, true, ..., true, a_{p+1}, ..., a_n) & \wedge\, a_1 & \wedge\, a_2 & \wedge ... \wedge\, a_p) & \vee \\
(R(false, true, ..., true, a_{p+1}, ..., a_n) & \wedge \neg a_1 & \wedge\, a_2 & \wedge ... \wedge\, a_p) & \vee \\
(R(true, false, ..., true, a_{p+1}, ..., a_n) & \wedge\, a_1 & \wedge \neg a_2 & \wedge ... \wedge\, a_p) & \vee \\
... & & & & \vee \\
\multicolumn{5}{l}{(R(false, false ..., false, a_{p+1}, ..., a_n) \wedge \neg a_1 \wedge \neg a_2 \wedge ... \wedge \neg a_p))}
\end{array}
\end{array}
$$

where each $R(c_1, c_2, ..., c_p, a_{p+1}, ..., a_n)$ denotes the replacement in $R(x)$ of the set of atoms $\{a_1, a_2, ..., a_p\}$ by the combination $\{c_1, c_2, ..., c_p\}$ of propositions true and $false$.

(Step 2) *We simplify each $R(c_1, c_2, ..., c_p, a_{p+1}, ..., a_n)$ in the following form:*

1. *For each preterminal node $n \in tree(E_{\mathcal{P}})$ with two leaf nodes, we simplify (Def. 13) sub-formulas in $R(c_1, c_2, ..., c_p, a_{p+1}, ..., a_n)$ which correspond to left and right subtrees of $f(n)$ in $R(c_1, c_2, ..., c_p, a_{p+1}, ..., a_n)_{\mathcal{P}}$.*
2. *For each preterminal node $n \in tree(E_{\mathcal{P}})$ with one left/right leaf node, we simplify (Def. 13) the sub-formula in $R(c_1, c_2, ..., c_p, a_{p+1}, ..., a_n)$ which corresponds to the left/right subtree of $f(n)$ in $R(c_1, c_2, ..., c_p, a_{p+1}, ..., a_n)_{\mathcal{P}}$.*

This selective simplification is intended to preserve similarity wrt connectives between $R(c_1, c_2,\ ..., c_p, a_{p+1}, ..., a_n)_{\mathcal{P}}$ and $E_{\mathcal{P}}$.

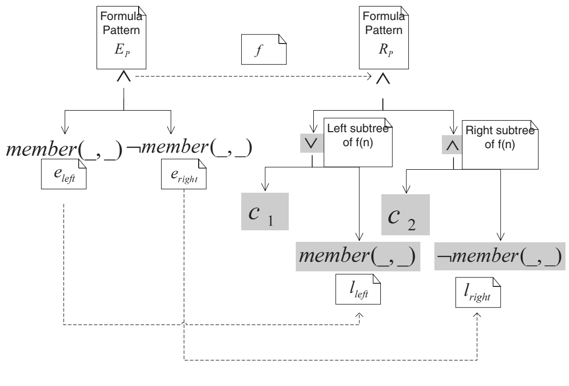


Fig. 4. Rewriting step. Sub-formulas to be simplified.

Example 9 (Rewriting Step). Let F be the formula (4) in Ex. 8, $P_{\mathcal{P}}$ the pattern (1) in $\Omega_{ext}(r_1^{\forall})$ and $E_{\mathcal{P}} = rhs(P_{\mathcal{P}})$. From Fig. 3 we can verify that $\{a_1 = idnat(x,e), a_2 = idnat(v,e)\}$ is the set of atoms which has not been used for deciding similarity (i.e. remaining atoms). After rewriting (step 1) we obtain the following formula:

$$\boxed{\begin{array}{l} rew(F,P_{\mathcal{P}}) = \forall (r_1^{\forall}(e,[x|y],e,[v|w]) \Leftrightarrow \\ \quad (true \vee member(e,y)) \wedge (false \wedge \neg member(e,w)) \;\; \wedge idnat(x,e) \;\;\; \wedge idnat(v,e) \;\;\; \vee \\ \quad (false \vee member(e,y)) \wedge (false \wedge \neg member(e,w)) \wedge \neg idnat(x,e) \wedge idnat(v,e) \;\;\; \vee \\ \quad ((true \vee member(e,y)) \wedge (true \wedge \neg member(e,w)) \;\;\; \wedge idnat(x,e) \;\;\; \wedge \neg idnat(v,e) \vee \\ \quad (false \vee member(e,y)) \wedge (true \wedge \neg member(e,w)) \;\;\; \wedge \neg idnat(x,e) \wedge \neg idnat(v,e) \end{array}}$$

For preterminal node $\wedge$ in $tree(E_{\mathcal{P}})$, we simplify sub-formulas in $R(c_1, c_2, ..., c_p, a_{p+1}\ , ..., a_n)$ which correspond to left and right subtrees of $f(\wedge)$ in $R(c_1, c_2, ..., c_p, a_{p+1}, ..., a_n)_{\mathcal{P}}$. In Fig. 4 we have highlighted such subtrees. After rewriting (step 2), we obtain the formula:

$$\boxed{\begin{array}{llll} \multicolumn{4}{l}{rew(\ F, P_{\mathcal{P}}) = \forall (r_1^{\forall}(e,[x|y],e,[v|w]) \Leftrightarrow} \\ \quad true \wedge false & \wedge\, idnat(x,e) & \wedge\, idnat(v,e) & \vee \\ \quad member(e,y) \wedge false & \wedge \neg idnat(x,e) & \wedge\, idnat(v,e) & \vee \\ \quad true \wedge \neg member(e,w) & \wedge\, idnat(x,e) & \wedge \neg idnat(v,e) & \vee \\ \quad member(e,y) \wedge \neg member(e,w) & \wedge \neg idnat(x,e) & \wedge \neg idnat(v,e) & \end{array}}$$

In order to apply automatic folding to formulas, we need to instantiate patterns in extended search spaces. We say that a quantifier-free formula $pi(P_{\mathcal{P}}, B)$ is the *pattern instantiation* of $P_{\mathcal{P}} \in \Omega_{ext}(r_i)$ wrt B if and only if $B_{\mathcal{P}} = rhs(P_{\mathcal{P}})$ and $pi(P_{\mathcal{P}}, B)$ is obtained from $P_{\mathcal{P}}$ by replacing every term pattern in $P_{\mathcal{P}}$ by its respective term in B.

Example 10 (Pattern instantiation in $\Omega_{ext}(r_1^{\forall})$).

$$\boxed{\begin{array}{ll} P_{\mathcal{P}} & = r_1^{\forall}(\_{_1},\_{_2},\_{_3},\_{_4}) \Leftrightarrow (member(\_{_1},\_{_2}) \wedge \neg member(\_{_3},\_{_4})) \\ B & = member(e,y) \wedge \neg member(e,w) \\ pi(P_{\mathcal{P}},B) & = r_1^{\forall}(e,y,e,w) \Leftrightarrow (member(e,y) \wedge \neg member(e,w)) \end{array}}$$

Definition 15 (Folding Step). *Let F be a formula in $\mathcal{A}$ of the form $\forall(r_i(x) \Leftrightarrow R(x))$ and B a sub-formula in $R(x)$ with $B_{\mathcal{P}} = rhs(P_{\mathcal{P}})$ and $P_{\mathcal{P}} \in \Omega_{ext}(r_i)$. We say that $fold(F, P_{\mathcal{P}})$ is the folding step of F wrt $P_{\mathcal{P}}$ if and only if it is obtained by replacing B by $lhs(pi(P_{\mathcal{P}}, B))$ in $R(x)$.*

Although search spaces are finite, to identify sub-formulas to be folded constitutes a highly non-deterministic task. In order to guide an automatic identification of such sub-formulas we introduce the notion of encapsulation and then explain how rewriting and folding rules contribute to automate reductions.

We say that a formula/formula pattern $R/R_{\mathcal{P}}$ is *completely encapsulated* in $layer_i$ of $\mathcal{A}$ if and only if every atom/atom pattern occurring in $R/R_{\mathcal{P}}$ is defined on a relation symbol of level i. We say that a formula/formula pattern $R/R_{\mathcal{P}}$ is *partially encapsulated* in $layer_i$ if and only if some atom/atom patterns occurring in $R/R_{\mathcal{P}}$ is defined on a relation symbol of level i and the remaining atom/atom patterns are defined on relation symbols of lower level.

Definition 16 (i-Reduction). *Let $F_k \in exp(F, i)$ be a formula of level i. The i-reduction of F_k wrt $\Omega_{ext}(r_i)$, $red(F_k, i, \Omega_{ext}(r_i))$, is implemented in the following steps:*

1. *(Searching). To search for patterns $P_\mathcal{P} \in \Omega_{ext}(r_i)$ with $rhs(P_\mathcal{P})$ as a completely encapsulated pattern of level i. Literal patterns in $rhs(F_{k\mathcal{P}})$ can be used to accelerate this search. If this search fails then to continue by searching for partially encapsulated patterns of level i. If this search fails then to continue in a similar way by searching for patterns of level $i-1$ and so on.*
2. *(Rewriting, Step 1). Let $rhs(F_{k\mathcal{P}})$ be similar to $rhs(P_\mathcal{P})$. We fix in $rhs(F_{k\mathcal{P}})$ those atom patterns which are responsible of similarity and then a set A of remaining atoms in $rhs(F_k)$ is then selected to be evaluated.*
3. *(Rewriting, Step 2). After evaluating wrt A, we simplify by preserving the structure of logical connectives in $P_\mathcal{P}$.*
4. *(Folding) To identify sub-formula B to be folded (i.e. $B_\mathcal{P} = rhs(P_\mathcal{P})$), to construct a new predicate (i.e. $lhs(pi(P_\mathcal{P}, B))$) and then to replace B in $rew(F_k, rhs(P_\mathcal{P}))$ by the new predicate.*

Example 11 (i-Reduction). Let F_k be the formula (4) in Ex. 8.

(Searching) We search for patterns $P_\mathcal{P} \in \Omega_{ext}(r_1^\forall)$ such that $rhs(P_\mathcal{P})$ is a completely encapsulated pattern of level 1:

$$P_\mathcal{P} = r_1^\forall(\_{}_1, \_{}_2, \_{}_3, \_{}_4) \Leftrightarrow (member(\_{}_1, \_{}_2) \wedge \neg member(\_{}_3, \_{}_4))$$

(Rewriting, Step 1) If $rhs(F_{k\mathcal{P}})$ is similar to (the rhs of) several patterns then a non-deterministic choice must be done. In our example, the choice is deterministic (i.e. $P_\mathcal{P}$ is the unique candidate). We fix in $rhs(F_{k\mathcal{P}})$ those atom patterns which are responsible of similarity.

$$rhs(F_{k\mathcal{P}}) = (idnat(\_,\_) \vee \underline{member(\_,\_)}) \wedge (\neg idnat(\_,\_) \wedge \underline{\neg member(\_,\_)})$$

The set of remaining atoms $A = \{a_1 = idnat(x, e), a_2 = idnat(v, e)\}$ is then selected to be evaluated.

(Rewriting, Step 2) After evaluating and simplifying:

$$\boxed{\begin{array}{llll} rew(\ F_k, P_\mathcal{P}) = \forall(r_1^\forall(e, [x|y], e, [v|w]) \Leftrightarrow & & & \\ \quad true \wedge false & \wedge\, idnat(x, e) & \wedge\, idnat(v, e) & \vee \\ \quad member(e, y) \wedge false & \wedge \neg idnat(x, e) & \wedge\, idnat(v, e)) & \vee \\ \quad true \wedge \neg member(e, w) & \wedge\, idnat(x, e) & \wedge \neg idnat(v, e) & \vee \\ \quad member(e, y) \wedge \neg member(e, w) & \wedge \neg idnat(x, e) & \wedge \neg idnat(v, e) & \end{array}}$$

(Folding) At this point, it is easy to identify B as a sub-formula in $rew(F_k, P_\mathcal{P})$ whose pattern is equal to the $rhs(P_\mathcal{P})$.

$$\boxed{\begin{array}{llll} rew(\ F_k, P_\mathcal{P}) = \forall(r_1^\forall(e, [x|y], e, [v|w]) \Leftrightarrow & & & \\ \quad true \wedge false & \wedge\, idnat(x, e) & \wedge\, idnat(v, e) & \vee \\ \quad member(e, y) \wedge false & \wedge \neg idnat(x, e) & \wedge\, idnat(v, e)) & \vee \\ \quad true \wedge \neg member(e, w) & \wedge\, idnat(x, e) & \wedge \neg idnat(v, e) & \vee \\ \quad \underbrace{member(e, y) \wedge \neg member(e, w)}_{B} & \wedge \neg idnat(x, e) & \wedge \neg idnat(v, e) & \end{array}}$$

A new predicate is obtained by pattern instantiation (Ex. 10):

$$\boxed{lhs(pi(P_{\mathcal{P}}, B)) = r_1^{\forall}(e, y, e, w)}$$

Finally, the replacement of B by the new predicate produces the formula:

$$\boxed{\begin{array}{llll} \multicolumn{4}{l}{\forall(r_1^{\forall}(e, [x|y], e, [v|w]) \Leftrightarrow} \\ \quad true \wedge false & \wedge\, idnat(x, e) & \wedge\, idnat(v, e) & \vee \\ \quad member(e, y) \wedge false & \wedge \neg idnat(x, e) & \wedge\, idnat(v, e)) & \vee \\ \quad true \wedge \neg member(e, w) & \wedge\, idnat(x, e) & \wedge \neg idnat(v, e) & \vee \\ \quad \underbrace{r_1^{\forall}(e, y, e, w)}_{lhs(pi(P_{\mathcal{P}}, B))} & \wedge \neg idnat(x, e) & \wedge \neg idnat(v, e) & \end{array}}$$

We say that an i-reduction $red(F_k, i, \Omega_{ext}(r_i))$ is *complete* when all possible folding steps have been applied to $rew(F_k, P_{\mathcal{P}})$. We say that a reduction phase has been *completed* for F if and only if a complete i-reduction has been applied to every formula in $exp(F, i)$.

Theorem 4 (Reduction Preserves Correctness). *Let* $\forall(r_i(x) \Leftrightarrow R^{red}(x))$ *be the* i*-reduction of an expanded formula* $\forall(r_i(x) \Leftrightarrow R^{exp}(x))$ *wrt* $\Omega_{ext}(r_i)$*. For every ground atom* $r_i(x)\phi$*,*

$$\mathcal{A} \models R^{red}(x)\phi \Leftrightarrow R^{exp}(x)\phi$$

(A proof of this theorem can be found in [12]).

5.5 Compilation as an Incremental and Terminating Process

The compilation of an axiom is completed by a finite sequence of meaning-preserving transformation steps. Each transformation step is composed of an expansion phase followed by a (complete) reduction phase. Table 2 shows $r_1^{\forall}$ (Ex. 6) after a transformation step.

Theorem 5 (Forms of Compiled Axioms). *After a transformation step, every resulting formula presents one of the following forms.* (A proof of this theorem can be found in [12])*:*

1. $\forall(r_i(x) \Leftrightarrow r_j(x))$ *where* $r_j(x)_{\mathcal{P}}$ *is equal to the lhs of some element in* $\Omega_{ext}(r_i)$*.*
2. $\forall(r_i(x) \Leftrightarrow \bigvee r_j(x) \wedge G_j(x))$ *where* $r_j(x)_{\mathcal{P}}$ *is equal to the lhs of some element in* $\Omega_{ext}(r_i)$ *and* $G_j(x)$ *a conjunctive formula of literals defined on atoms whose patterns are included in* $\mathcal{A}$*.*

Each transformation step represents an *increment* in the overall compilation process. Due to Theorem 5, each successive increment compiles either an axiom for r_j (e.g. $\forall(r_3^{\forall}(e, y) \Leftrightarrow (member(e, y) \wedge false))$) or an axiom for a new assertion from a literal in $G_j(x)$ (e.g. $\forall(r_{10}^{\forall}(x, e) \Leftrightarrow \neg idnat(x, e))$).

Theorem 6 (Termination). *The compilation of an expressive assertion is completed in a finite amount of increments.* (A proof of this theorem can be found in [12]).

For instance, the compilation of $r_1^{\forall}$ (Ex. 6) has been completed by means of 11 increments ($r_1^{\forall}$,...,$r_{11}^{\forall}$).

Table 2. Compilation results after one transformation step for $r_1^\forall$ in Ex. 6.

(1) $\forall\,(r_1^\forall(e,[\,],e,[\,]) \Leftrightarrow r_5^\forall)$				
(2) $\forall\,(r_1^\forall(e,[\,],e,[v\mid w]) \Leftrightarrow (r_6^\forall$		$\wedge\ idnat(v,e)$	$\vee$	
	$r_2^\forall(e,w)$	$\wedge\ \neg idnat(v,e)))$		
(3) $\forall\,(r_1^\forall(e,[x\mid y],e,[\,]) \Leftrightarrow (r_9^\forall$		$\wedge\ idnat(x,e)$	$\vee$	
	$r_5^\forall(e,y)$	$\wedge\ \neg idnat(x,e)))$		
(4) $\forall\,(r_1^\forall(e,[x\mid y],e,[v\mid w]) \Leftrightarrow (r_7^\forall$		$\wedge\ idnat(x,e)$	$\wedge\ idnat(v,e)$	$\vee$
	$r_3^\forall(e,y)$	$\wedge\ \neg idnat(x,e)$	$\wedge\ idnat(v,e)$	$\vee$
	$r_4^\forall(e,w)$	$\wedge\ idnat(x,e)$	$\wedge\ \neg idnat(v,e)$	$\vee$
	$r_1^\forall(e,y,e,w)$	$\wedge\ \neg idnat(x,e)$	$\wedge\ \neg idnat(v,e)))$	

6 Executing Expressive Assertions from Synthesized Logic Programs

Once a compilation process has been completed, a finite set of new assertions have been produced (Theorem 6). The form of their axioms (i.e. universal closure, mutually-exclusive disjunctions of conjunctions, absence of negated atoms and stratification [17]) allow to define a simple translation method from compiled assertions to definite logic programs.

Definition 17 (Translation Method). *For every resulting axiom Ax from a compilation:*

1. *If Ax is of the form $\forall(r \Leftrightarrow P)$ where P is a propositional formula formed out from constants $true$ and $false$, two situations are possible:*
   a) *If the evaluation of P is equal to $false$ then Ax is translated to an empty clause.*
   b) *If the evaluation of P is equal to $true$ then Ax is translated to a clause of the form $r \Leftarrow$*
2. *If Ax is of the form $\forall(r(x) \Leftrightarrow \bigvee_k(r_1(x) \wedge ... \wedge r_n(x))$ then it is translated to a set of k clauses of the form $r(x) \Leftarrow r_1(x), ..., r_n(x)$. Every clause, which includes an atom occurrence r with axiom of the form $\forall(r \Leftrightarrow P)$ and P equal to $false$, is deleted.*

Example 12 (Synthesized logic program for $r_1^\forall$ (Ex. 6)).

$r_1^\forall(e,[\,],e,[v|w]) \Leftarrow r_2^\forall(e,w),\ r_{10}^\forall(v,e)$
$r_1^\forall(e,[x|y],e,[\,]) \Leftarrow r_{11}^\forall(x,e)$
$r_1^\forall(e,[x|y],e,[\,]) \Leftarrow r_5^\forall(e,y),\ r_{10}^\forall(x,e)$
$r_1^\forall(e,[x|y],e,[v|w]) \Leftarrow r_3^\forall(e,y),\ r_{10}^\forall(x,e), r_{11}^\forall(v,e)$
$r_1^\forall(e,[x|y],e,[v|w]) \Leftarrow r_4^\forall(e,w),\ r_{11}^\forall(x,e), r_{10}^\forall(v,e)$
$r_1^\forall(e,[x|y],e,[v|w]) \Leftarrow r_1^\forall(e,y,e,w),\ r_{10}^\forall(x,e), r_{10}^\forall(v,e)$

$r_2^\forall(e,[x|y]) \Leftarrow r_2^\forall(e,y),\ r_{10}^\forall(x,e)$

$r_3^\forall(e,[x|y]) \Leftarrow r_3^\forall(e,y),\ r_{10}^\forall(x,e)$

$r_4^\forall(e, [\,]) \Leftarrow r_9^\forall$
$r_4^\forall(e, [v|w]) \Leftarrow r_4^\forall(e, w),\ r_{10}^\forall(v, e)$

$r_5^\forall(e, [v|w]) \Leftarrow r_{11}^\forall(v, e)$
$r_5^\forall(e, [v|w]) \Leftarrow r_5^\forall(e, w),\ r_{10}^\forall(v, e)$

$r_9^\forall \Leftarrow$

$r_{10}^\forall(0, s(y)) \Leftarrow$
$r_{10}^\forall(s(x), 0) \Leftarrow$
$r_{10}^\forall(s(x), s(y)) \Leftarrow r_{10}^\forall(x, y)$

$r_{11}^\forall(0, 0) \Leftarrow$
$r_{11}^\forall(s(x), s(y)) \Leftarrow r_{11}^\forall(x, y)$

For the purpose of verifying that synthesized logic programs are totally correct wrt goals $\Leftarrow r_1^Q(t, y)$, a set of execution modes can be calculated for each synthesized predicate. An execution mode is formed by replacing each parameter in a signature by a mark $\downarrow$ to refer to 'a ground term as input parameter' or $\uparrow$ to refer to 'an existentially quantified variable as output parameter'. For instance, a logic program such as the one shown in Ex. 2 is a totally correct program for *idnat* (Ex. 3) wrt goals $\Leftarrow idnat_1^\forall(t, y)$ (i.e. $idnat(\downarrow, \uparrow)$), $\Leftarrow idnat_1^\forall(x, t)$ (i.e. $idnat(\uparrow, \downarrow)$) and $\Leftarrow idnat_1^\forall(t_1, t_2)$ (i.e. $idnat(\downarrow, \downarrow)$) where t, t_1, t_2 are ground Nat-terms. Static analysis techniques can be used to calculate and/or verify sets of execution modes for a logic program [1], [6]. Table 3 shows the set of execution modes calculated for the synthesized program in Ex. 12. Thus, if a synthesized logic program $\forall x, y(r_1^Q(x, y) \Leftarrow P(x, y))$ for an expressive assertion $\forall x(r(x) \Leftrightarrow Q\, yR(x, y))$ presents $r_1^Q(\downarrow, \uparrow)$ as one of its execution modes then r_1^Q can be used to execute ground instances of r. For instance, such a condition holds for $r_1^\forall$ (Table 3), hence, the synthesized logic program in Ex. 12 can be used to execute ground instances of r in Ex. 5.

Table 3. Execution modes for the synthesized program in Ex. 12.

$r_{11}^\forall(\downarrow, \downarrow)$	$r_{11}^\forall(\uparrow, \downarrow)$	$r_{11}^\forall(\downarrow, \uparrow)$
$r_{10}^\forall(\downarrow, \downarrow)$	$r_{10}^\forall(\uparrow, \downarrow)$	$r_{10}^\forall(\downarrow, \uparrow)$
$r_5^\forall(\downarrow, \downarrow)$	$r_5^\forall(\uparrow, \downarrow)$	
$r_4^\forall(\downarrow, \downarrow)$	$r_4^\forall(\uparrow, \downarrow)$	
$r_3^\forall(\downarrow, \downarrow)$	$r_3^\forall(\uparrow, \downarrow)$	
$r_2^\forall(\downarrow, \downarrow)$	$r_2^\forall(\uparrow, \downarrow)$	
$r_1^\forall(\downarrow, \downarrow, \downarrow, \downarrow)$	$r_1^\forall(\uparrow, \downarrow, \uparrow, \downarrow)$	

How can assertion checkers evaluate ground atoms $r(t)$ from goals $\Leftarrow r_1^Q(t, y)$ in a refutation system such as Prolog?

The execution of $\Leftarrow r_1^Q(t, y)$ in a Prolog system will compute a set of substitutions $\{\theta_1, ..., \theta_j\}$ for y, thus:

- For $Q = \exists$:
  1. If $\{\theta_1, ..., \theta_j\} = \emptyset$ then $\mathcal{A} \vdash \neg\exists y(r_1^{\exists}(t,y))$ (by logical consequence), $\mathcal{A} \vdash \neg\exists y(R(t,y))$ (by total correctness of $r_1^{\exists}$), $\mathcal{A} \vdash \neg r(t)$ (by equivalence)
  2. If $\{\theta_1, ..., \theta_j\} \neq \emptyset$ then, by a similar reasoning, $\mathcal{A} \vdash \exists y(r_1^{\exists}(t,y))$, $\mathcal{A} \vdash \exists y(R(t,y))$, $\mathcal{A} \vdash r(t)$.
- For $Q = \forall$:
  1. If $\{\theta_1, ..., \theta_j\} = \emptyset$ then $\mathcal{A} \vdash \neg\exists y(r_1^{\forall}(t,y))$ (by logical consequence), $\mathcal{A} \vdash \neg\exists y(\neg R(t,y))$ (by total correctness of $r_1^{\forall}$),$\mathcal{A} \vdash \forall y(R(t,y))$ (by equivalence), $\mathcal{A} \vdash r(t)$ (by equivalence).
  2. If $\{\theta_1, ..., \theta_j\} \neq \emptyset$ then, by a similar reasoning, $\mathcal{A} \vdash \exists y(r_1^{\forall}(t,y))$, $\mathcal{A} \vdash \exists y(\neg R(t,y))$,$\mathcal{A} \vdash \neg\forall y(R(t,y))$, $\mathcal{A} \vdash \neg r(t)$.

7 Conclusions and Future Work

In this paper, we have formalized a class of assertions we call expressive assertions in the sense that they describe recursive models which are no directly translatable into executable forms. Due to this fact, current assertion checkers are not able to execute expressive assertions. The existence of mature studies in the field of transformational synthesis constitutes an important aid to overcome the problem. Recurrent problems in transformational synthesis have been revisited, for instance, the “eureka problem” (i.e. non-automatic steps about when and how to define recursive predicates). In order to overcome the problem, we restrict our attention to a particular class of first-order theories we call assertion contexts. This sort of theories is interesting because it presents a balance between expressiveness for writing assertions and existence of effective methods for compiling and executing them via synthesized (definite) logic programs.

Finally, we consider that our work can also be used to construct assertion contexts in an incremental manner. In fact, assertions contexts can be extended with expressive assertions in a conservative way without losing execution capabilities. For instance, $\mathcal{A} \cup r \cup r_1^{\forall}$ is more expressive than $\mathcal{A}$ while preserving consistency and execution capabilities (i.e. $r_1^{\forall}$ can be used to execute ground atoms of r). This issue is essential from a practical view point in order to reach expressive assertion languages. We plan to study it as future work.

References

1. Arts, T., Zantema, H.: Termination of Logic Programs Using Semantic Unification. LOPSTR’95. Springer-Verlag, (1996) 219–233.
2. Barnes, J.: High Integrity Ada: The SPARK Approach. Addison-Wesley, (1997).
3. Bertoni, A., Mauri G., Miglioli, P.: On the Power of Model Theory in Specifying Abstract Data Types and in capturing their Recursiveness. Fundamenta Informaticae, **VI(2)** (1983) 27–170.
4. Burstall, R. M., Darlington, J.: A Transformational System for Developing Recursive Programs. Journal of the ACM **24(1)** (1977) 44–67.
5. Bartetzko, D., Fischer, C., Möller, M. Wehrheim, H.: Jass-Java with Assertions. 1st Workshop on Runtime Verification. Paris. France. ENTCS Elsevier, (1999).

6. Deville, Y.: Logic Programming. Systematic Program Development. Addison-Wesley. (1990)
7. Deville, Y., Lau, K. K.: Logic Program Synthesis. J. Logic Programming **19,20** (1994) 321–350.
8. Flener, P.: Logic Program Synthesis from Incomplete Information. Kluwer Academic Publishers, Massachusetts, (1995).
9. Flener, P.: Achievements and Prospects of Program Synthesis. LNAI 2407. Springer-Verlag, (2002) 310–346.
10. Galán, F. J., Cañete, J. M.: Improving Constructive Synthesizers by Tabulation Techniques and Domain Ordering. In David Warren (ed.), Tabulation and Parsing Deduction, (2000) 37-49.
11. Galán, F. J, Díaz, V. J., Cañete, J. M.: Towards a Rigorous and Effective Functional Contract for Components. Informatica. An International Journal of Computing and Informatics. 25(4), (2001) 527–533.
12. Galán, F. J., Cañete, J. M.: Compiling and Executing Assertions via Synthesized Logic Programs. Tecnical Report LSI-2004-01. Dept. of Language and Computer Systems. Faculty of Computer Science, Univ. of Seville. (2004)
13. Kramer, R.: iContract-The Java Design by Contract Tool. TOOLS 26: Technology of Object-Oriented Languages and Systems. IEEE Computer Society Press, (1998).
14. Lau, K., Ornaghi, M.: On Specification Frameworks and Deductive Synthesis of Logic Programs. LOPSTR'94. LNCS 883 Springer-Verlag, (1994) 104–121.
15. Lau, K., Ornaghi, M.: Towards an Object-Oriented Methodology for Deductive Synthesis of Logic Programs. LOPSTR'95. LNCS 1048 Springer-Verlag, (1995) 152–169.
16. Leavens, G., Baker, A., Ruby, C. Preliminary Design of JML. TR 98-06u, Dept. of Computer Science, Iowa State Univ., USA (2003).
17. Lloyd, J. W.: Foundations of Logic Programming, 2nd ed. Springer-Verlag, (1987).
18. Meyer, B.: Eiffel: The Language. Prentice-Hall, (1992).