Context-Sensitive Multivariant Assertion Checking in Modular Programs

Paweł Pietrzak[1], Jesús Correas[2], Germán Puebla[1], and Manuel V. Hermenegildo[1,3]

[1] School of Computer Science, Technical University of Madrid (UPM)
[2] School of Computer Science, Complutense University of Madrid
[3] CS and ECE Departments, University of New Mexico

Abstract. We propose a *modular*, assertion-based system for verification and debugging of large logic programs, together with several interesting models for checking assertions statically in modular programs, each with different characteristics and representing different trade-offs. Our proposal is a *modular* and *multivariant* extension of our previously proposed abstract assertion checking model and we also report on its implementation in the CiaoPP system. In our approach, the specification of the program, given by a set of assertions, may be partial, instead of the complete specification required by traditional verification systems. Also, the system can deal with properties which cannot always be determined at compile-time. As a result, the proposed system needs to work with *safe* approximations: all assertions proved correct are guaranteed to be valid and all errors actual errors. The use of modular, context-sensitive static analyzers also allows us to introduce a new distinction between assertions checked in a particular context or checked in general.

1 Introduction

Splitting program code into modules is widely recognized as a useful technique in the process of software development. In this paper we propose a framework for static (i.e., compile-time) checking of assertions in modular logic programs, based on information from global analysis. We assume a *strict* module system, i.e., a system in which modules can only communicate via their *interface*. The interface of a module contains the names of the *exported* predicates and the names of the *imported* modules.

Within our framework, the programmer is expected to write a (partial) specification for a module (or a set of modules) being subject to the verification process. The specification is written in terms of (Ciao) assertions [13]. From the programmer's viewpoint, these assertions resemble the type (and mode) declarations used in strongly typed logic languages such as Mercury [16] and in functional languages. However, when compared to the latter, note that in logic programming arguments of procedures behave differently in the sense that arguments might be either input or output, depending on the specific *usage* (i.e., the context) of the procedure. For instance, the classical predicate `append/3` can be

used for concatenating lists, for decomposing lists, for checking or finding a prefix of a given list, etc. Therefore, our assertion language and the checking procedure are designed to *allow various usages of a predicate*. Moreover, comparing to the former, herein we are interested in supporting a general setting in which, on one hand assertions can be of a quite general nature, including properties which are *undecidable*, and, on the other hand, only a small number of assertions may be present in the program, i.e., the assertions are *optional*.

Our approach is strongly motivated by the availability of powerful and mature static analyzers for (constraint) logic programs (see, e.g., [8] and its references), generally based on abstract interpretation [6]. Also, since we deal with modular programs, context-sensitive static analyses that handle modules (see, e.g., [7] and its references) provide us with suitable background. Especially relevant is our recent work on context sensitive, multivariant modular analysis (see [15, 5] among others). These analysis systems can statically infer a wide range of properties (from types to determinacy or termination) accurately and efficiently, for realistic modular programs. We would like to take advantage of such program analysis tools, rather than developing new abstract procedures, such as concrete [10] or abstract [3, 14] diagnosers and debuggers, or using traditional proof-based methods, e.g., [1, 9].

The work presented builds on [13] where the assertion language that we use was introduced, and on [14] where a proposal for the formal treatment of assertion checking, both at compile-time and at run-time, was presented. We extend the above-mentioned work in four main directions. Most importantly, the solution of [14] is not modular. We show herein how to check assertions in modular programs in a way that ensures the soundness of the approach. Also, the formalization is different to that of [14], the present one being based on generalized AND trees. In addition, in this work we exploit *multivariant* information generated by the analysis. This essentially means that multiple usages of a procedure can result in multiple descriptions in the analysis output. In consequence, this enables us to verify the code in a more accurate way.

Modular verification has also been studied within OO programming (e.g., [11]) where the importance of contextual correctness, as in our paper, has been recognized. Nevertheless this work differ from ours in several respects, the most important one being that they are based on traditional Hoare-like based verification techniques and the full specification is required, whereas our framework is based on abstract interpretation and allows for partial specifications.

In the context of Logic Programming [4] shows how to perform abstract diagnosis of incomplete logic programs. Our approach is similar to theirs, since the correctness of a modular program is established in terms of the correctness of its modules. However, in [4] the complete specification is needed and, more importantly, context-sensitive analysis information is not used, and therefore there is no concept of correctness in context. We claim that this is an important advantage of our approach, because it allows the validation of a module in a given program even when it is not possible to validate it in a context-independent way.

2 Preliminaries

An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. A *predicate descriptor* is an atom $p(X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ are distinct variables. We shall use predicate descriptors to refer to a certain form of atoms, as well as to predicate symbols. A *clause* is of the form $H :- B_1, \ldots, B_n$ where H, the *head*, is an atom and $B_1, \ldots, B_n$, the *body*, is a possibly empty finite conjunction of atoms. In the following we assume that all clause heads are normalized, i.e., H is of the form of a predicate descriptor. Furthermore, we require that each clause defining a predicate p has an identical sequence of variables $X_{p_1}, ..., X_{p_n}$ in the head. We call this the *base form* of p. This is not restrictive since programs can always be normalized, and it will facilitate the presentation of the algorithms later. However, both in the examples and in the implementation we handle non-normalized programs. A *definite logic program*, or *program*, is a finite sequence of clauses. *ren* denotes a set of renaming substitutions over variables in the program at hand.

The concrete semantics used for reasoning about programs will use the notion of generalized AND trees, as they are described in [2]. Every node of a generalized AND tree, denoted $\langle \theta_c, P, \theta_s \rangle$, contains a call to a predicate P, with a call substitution θ_c and corresponding success subtitution θ_s. The concrete semantics of a program R for a given set of queries Q, $[\![R]\!]_Q$, is the set of generalized AND trees that represent the execution of the queries in Q for the program R.[4]

Definition 1. *calling_context*(P, R, Q) *of a predicate given by the predicate descriptor* P *defined in* R *for a set of queries* Q *is the set* $\{\theta_c | \exists T \in [\![R]\!]_Q$ *s.t.* $\exists \langle \theta'_c, P', \theta'_s \rangle$ *in* $T \wedge \exists \sigma \in ren$ *s.t.* $P = P'\sigma \wedge \theta_c = \theta'_c\sigma\}$

success_context(P, R, Q) *of a predicate given by the predicate descriptor* P *defined in* R *for a set of queries* Q *is the set of pairs* $\{(\theta_c, \theta_s) | \exists T \in [\![R]\!]_Q$ *s.t.* $\exists \langle \theta'_c, P', \theta'_s \rangle$ *in* $T \wedge \exists \sigma \in ren$ *s.t.* $P = P'\sigma \wedge \theta_c = \theta'_c\sigma \wedge \theta_s = \theta'_s\sigma\}$

Our basic tool for checking assertions is *abstract interpretation* [6]. Abstract interpretation is a technique for static program analysis in which semantics of the program is conservatively approximated using an *abstract domain* D_α (equipped with a partial order $\sqsubseteq$) which is simpler than the actual, *concrete domain* D. Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : D \to D_\alpha$, and *concretization* $\gamma : D_\alpha \to D$.

Goal-dependent abstract interpretation takes as input a program R and a call pattern[5] $P{:}\lambda$, where P is an atom, and λ is a restriction of the run-time bindings of P expressed as an abstract substitution in D_α. Such an abstract interpretation (denoted $analysis(R, P{:}\lambda)$) computes an *answer table* (AT) whose entries are of the form $P_i{:}\lambda_i^c \mapsto \lambda_i^s$, where P_i is an atom and λ_i^c and λ_i^s are, respectively, the abstract call and success substitutions. An analysis is said to be *multivariant (on*

[4] We find this formalization more suitable than the derivation-based one used in our previous work [14] because it simplifies the presentation of the subsequent material.

[5] Note that we shall use sets of call patterns instead in the subsequent sections –the extension is trivial.

calls) if more than one entry $P{:}\lambda_1^c \mapsto \lambda_1^s, \ldots, P{:}\lambda_n^c \mapsto \lambda_n^s$ $n \geq 0$ with $\lambda_i^c \neq \lambda_j^c$ for some i, j may be computed for the same predicate. As it is shown in this paper, multivariant analyzers may provide valuable information for assertion checking not obtainable otherwise. An abstract interpretation process is monotonic, in the sense that the more specific the initial call pattern is, the more precise the results of the analysis are.

The abstract semantics of a program (or module) R for a set of queries Q, $[\![R]\!]^{\alpha}_{Q_\alpha}$, can be represented as a set of abstract AND-OR trees [2]. A context-sensitive, multivariant static analyzer such as that in CiaoPP [12] actually computes this set of trees, and returns the set of nodes in such trees, kept in the answer table AT.

3 Modular programs and modular analysis

We start by introducing some notation. We will use m and n to denote *modules*. Given a module m, by $imports(m)$ we denote the set of modules which m imports. By $depends(m)$ we refer to the set generated by the transitive closure of $imports$. Note that there may be circular dependencies among modules. The *program unit* of a given module m is the finite set of modules containing m and the modules on which m depends: $program\_unit(m) = \{m\} \cup depends(m)$.[6] m is called the *top-level* module of its program unit. Finally, $exported(m)$ is the set of predicate names exported by module m, and $imported(m)$ is the set of predicate names imported by m. Given a program unit $program\_unit(m)$, we can always obtain a single-module program that behaves like $program\_unit(m)$. We will denote such program as $flatten(m)$.

In summary, the framework for modular analysis works as follows: given the top-level module m, analysis computes an intermodular fixed point by iterating through the modules in $program\_unit(m)$, and analyzing them one by one. When the intermodular fixed point has been reached, the analysis results for exported predicates are stored in a *Global Answer Table* (GAT for short), in the form of $P : CP \mapsto AP$ entries, where CP and AP are the call and the answer patterns of an exported predicate, respectively. In the rest of the paper we will use CP and AP to refer to abstract substituions stored in the GAT, and λ for other abstract substitutions.

We will use the function $GAT = modular\_analysis(m)$ to refer to the analysis of the program unit m, that returns as result the global answer table, and $LAT = analysis(m, E, AT)$ to indicate the analysis of module m, with call patterns for exported predicates E and success patterns of imported predicates contained in AT, and returning the *Local Answer Table* (LAT), which contains the results of analyzing m. When computing the intermodular fixed point, $analysis(n, E, AT)$ is invoked for each module n in the program unit, where E is the set of calling patterns in GAT for predicates defined in n which need to be (re)analyzed, AT is the current state of the GAT, and the GAT is updated after

[6] Library modules and *builtins* require special treatment in order to avoid reanalysis of all used library predicates every time a user program is analyzed.

analysis with information from the resulting LAT. See [15] for details. We can define a partial ordering on answer tables over a given module in the following sense: $AT_1 \preceq AT_2$ iff $\forall(P : CP_1 \mapsto AP_1) \in AT_1, (\exists(P : CP_2 \mapsto AP_2) \in AT_2$ s.t. $CP_1 \sqsubseteq CP_2$ and $\forall(P : CP_2' \mapsto AP_2') \in AT_2$, if $CP_1 \sqsubseteq CP_2'$ then $AP_1 \sqsubseteq AP_2')$.

The computation performed by $analysis(m, E, AT)$ has the difficulty that, from the point of view of analysis of a given module m, the code to be analyzed is *incomplete* in the sense that the code for procedures imported from other modules is not available to analysis. During the analysis of a module m there may be calls $P : CP$ such that the procedure P is not defined in m but instead it is imported from another module n. There are several alternatives for computing a temporary answer pattern for $P : CP$, which are selected by means of the *success policy* (SP for short). SP is needed because given a call pattern $P : CP$ it will often be the case that no entry of exactly the form $P : CP \mapsto AP$ exists in the analysis results stored in the GAT for n (or there may be no entry at all). In such case, the information already present may be of value in order to obtain a (temporary) answer pattern AP, and continue the analysis of module m.

Several success policies can be defined which provide over- or under-approximations of the "exact" answer pattern $AP^{=}$ with different degrees of accuracy. By this exact value $AP^{=}$ we refer to the one which would be computed for the flattened program. As shown in [15], using over-approximating success policies (named SP^{+}) has the advantage that after analyzing any number of modules, even when a fixed point has not been reached yet, the information obtained for each module is always a correct over-approximation. The drawback is that when the fixed point is reached it may not be the least fixed point, i.e., information is not as precise as it could be. In contrast, under-approximating (SP^{-}) policies obtain the least fixed point (most precise information) but only produce correct results when the fixed point is reached. Therefore, SP^{-} policies are as accurate as performing the analysis of the flattened program. We will denote with $analysis_{SP}(m, E, AT)$ the analysis of a module m with respect to the set of call patterns E and using a success policy SP applied to the answer table AT.

4 Assertions

We consider two fundamental kinds of (basic) assertions [13].[7] The first one is `success` assertions, which are used to express properties which should hold on termination of a successful computation of a given predicate (*postconditions*). At the time of calling the predicate, the computation should satisfy a certain *precondition*. `success` assertions can be expressed in our assertion language using an expression of the form: `success` $P : Pre \Rightarrow Post$, where P is a predicate descriptor, and Pre and $Post$ are pre- and post-conditions respectively. Without loss of generality, we will consider that Pre and $Post$ correspond to abstract substitutions (λ_{Pre} and λ_{Post} resp.) over $vars(P)$. This kind of assertion should be interpreted as "in any invocation of P if Pre holds in the calling state and

[7] [13] presents other types of assertions, but they are outside the scope of this paper.

the computation succeeds, then *Post* should also hold in the success state." The postcondition stated in a `success` assertion refers to *all* the success states (possibly none). Note that `success` P : *true* $\Rightarrow$ *Post* can be abbreviated as `success` $P \Rightarrow$ *Post*.

A second kind of assertions expresses properties which should hold in any call to a given predicate. These properties are similar in nature to the classical *preconditions* used in program verification. These assertions have the form: `calls` $P : Pre$, and should be interpreted as "in all activations of P *Pre* should hold in the calling state." More than one assertion may be written for each predicate. That means that, in any invocation of P, at least one `calls` assertion for P should hold.

Finally, we write `pred` $P : Pre \Rightarrow Post$, as a shortcut for the two assertions: `calls` $P : Pre$ and `success` $P : Pre \Rightarrow Post$. We claim that the `pred` form is a natural way to describe a usage of the predicate. In what follows, we will use `calls` (resp. `success`) assertions when we want to refer to the calls part (resp. success part) of a `pred` assertion. We will assume, for simplicity and with no loss of generality, that all assertions referring to a predicate P defined in module m are also provided in that module. We will denote with $assertions(m)$ the set of assertions appearing in module m, and $assertions(P)$ refers to the assertions for predicate P.

Example 1. A possible set of `calls` assertions for the traditional `length/2` predicate that relates a list to its length, might be:

```
:- calls length(L,N) : (var(L), int(N)).    %(1)
:- calls length(L,N) : (list(L), var(N)).   %(2)
```

These assertions describe different modes for calling that predicate: either for `(1)` generating a list of length `N`, or `(2)` to obtain the length of a list `L`.

Possible success assertions for that predicate are:

```
:- success length(L,N) : (var(L), int(N))  => list(L).
:- success length(L,N) : (var(N), list(L)) => int(N).
```

The following two assertions are equivalent to all the previous assertions for `length/2`:

```
:- pred length(L,N) : (var(L), int(N))  => list(L).
:- pred length(L,N) : (var(N), list(L)) => int(N).
```

We assign a *status* to each assertion. The status indicates whether the assertion refers to intended or actual properties, and the relation between the property and the program semantics. This section builds on [14], but it has been adapted to our use of generalized AND trees.

We say that a `calls` assertion A with predicate descriptor P is *applicable* to a node $N = \langle \theta_c, P', \theta_s \rangle$ of the generalized AND tree if there is $\sigma \in ren$ (a renaming substitution) s.t. $P' = P\sigma$ and N is adorned on the left, i.e., the call substitution θ_c of N has been already computed. A `success` assertion A with predicate descriptor P is applicable to a node N if $P' = P\sigma$ (where $\sigma \in ren$) and N is adorned on the right, i.e., the success substitution θ_s of the call at N

has been computed (the procedure exit has been completed). In what follows, we will denote with ρ a suitable renaming substitution.

If an assertion holds within a fixed set of queries Q then the assertion is said to be *checked* with respect to Q. If this is proved, the assertion receives the corresponding status `checked`. Formally:

Definition 2 (Checked assertions). *Let R be a program.*

- *An assertion $A =$* `calls` $P : Pre$ *in R is* checked *w.r.t. the set of queries Q iff $\forall \theta_c \in calling\_context(P, R, Q), \theta_c\rho \in \gamma(\lambda_{Pre})$.*
- *An assertion $A =$* `success` $P : Pre \Rightarrow Post$ *in R is* checked *w.r.t. a set of queries Q iff $\forall (\theta_c, \theta_s) \in success\_context(P, R, Q), \theta_c\rho \in \gamma(\lambda_{Pre}) \rightarrow \theta_s\rho \in \gamma(\lambda_{Post})$.*

A calls or success assertion can also be *false*, whenever it is known that there is at least one call (or success) pattern in the concrete semantics that violates the property in the assertion. If we can prove this, the assertion is given the status `false`. In addition, an error message will be issued by the preprocessor.

Definition 3 (False assertions). *Let R be a program.*

- *An assertion $A =$* `calls` $P : Pre$ *in R is* false *w.r.t. the set of queries Q iff $\exists \theta_c \in calling\_context(P, R, Q)$ s.t. $\theta_c\rho \notin \gamma(\lambda_{Pre})$.*
- *An assertion $A =$* `success` $P : Pre \Rightarrow Post$ *in R is* false *w.r.t. the calling context Q iff $\exists (\theta_c, \theta_s) \in success\_context(P, R, Q)$ s.t. $\theta_c\rho \in \gamma(\lambda_{Pre}) \wedge \theta_s\rho \notin \gamma(\lambda_{Post})$.*

Finally, an assertion which expresses a property which holds for any initial query is a *true* assertion. If it can be proven, independently on the calling context, during compile-time checking, the assertion is rewritten with the status `true`. Formally:

Definition 4 (True success assertion). *An assertion $A =$* `success` $P : Pre \Rightarrow Post$ *in R is* true *iff for every set of queries Q, $\forall (\theta_c, \theta_s) \in success\_context(P, R, Q), \theta_c\rho \in \gamma(\lambda_{Pre}) \rightarrow \theta_s\rho \in \gamma(\lambda_{Post})$.*

Note that the difference between checked assertions and true ones, is that the latter hold for any context. Thus, the fact that an assertion is true implies that it is also checked.

Assertions are subject to compile-time checking. An assertion which is not determined by compile-time checking to be given any of the above statuses is a *check* assertion. This assertion expresses an intended property. It may hold or not in the current version of the program. This is the default status, i.e., if an assertion has no explicitly written status, it is assumed that the status is `check`. Before performing a compile-time checking procedure all assertions written by the user have `check` status.

In our setting, checking assertions must be preceded by analysis, and basically it boils down to comparing assertions (whenever applicable) with the abstract information obtained by analysis. Below we present sufficient conditions

for compile-time assertion checking in a program not structured in modules. The following sections will deal with assertion checking of modules and modular programs. In the case of proving a calls assertion, we would like to ensure that all concrete calls are included in the description λ_{Pre}. For disproving calls assertions, i.e., turning them to false, we want to show that there is some concrete call which is not covered by λ_{Pre}.

Definition 5 (Abstract assertion checking). *Let R be a program, and Q_α an abstract description of queries to R.*

- *An assertion $A =$* `success` *$P : Pre \Rightarrow Post$ in R is* abstractly true *iff $\exists P'{:}\lambda^c \mapsto \lambda^s \in analysis(R, \{P : \lambda_{Pre}\})$ s.t. $\exists \sigma \in ren,\ P' = P\sigma, \lambda^c = \lambda_{Pre} \wedge \lambda^s \sqsubseteq \lambda_{Post}$.*
- *An assertion $A =$* `success` *$P : Pre \Rightarrow Post$ in R is* abstractly checked *w.r.t. Q_α iff $\forall P'{:}\lambda^c \mapsto \lambda^s \in analysis(R, Q_\alpha)$ s.t. $\exists \sigma \in ren,\ P' = P\sigma, \lambda^c \sqsubseteq \lambda_{Pre} \rightarrow \lambda^s \sqsubseteq \lambda_{Post}$.*
- *An assertion $A =$* `calls` *$P : Pre$ in R is* abstractly checked *w.r.t. Q_α iff $\forall P'{:}\lambda^c \mapsto \lambda^s \in analysis(R, Q_\alpha)$ s.t. $\exists \sigma \in ren,\ P' = P\sigma, \lambda^c \sqsubseteq \lambda_{Pre}$.*
- *An assertion $A =$* `success` *$P : Pre \Rightarrow Post$ in R is* abstractly false *w.r.t. Q_α iff $\forall P'{:}\lambda^c \mapsto \lambda^s \in analysis(R, Q_\alpha)$ s.t. $\exists \sigma \in ren,\ P' = P\sigma, \lambda^c \sqsubseteq \lambda_{Pre} \wedge (\lambda^s \sqcap \lambda_{Post} = \bot)$.*
- *An assertion $A =$* `calls` *$P : Pre$ in R is* abstractly false *w.r.t. Q_α iff $\forall P'{:}\lambda^c \mapsto \lambda^s \in analysis(R, Q_\alpha)$ s.t. $\exists \sigma \in ren,\ P' = P\sigma, \lambda^c \sqcap \lambda_{Pre} = \bot$.*

In this definition $analysis(R, Q_\alpha)$ is a generic analysis computation, and therefore the definition is parametric with respect to the analysis actually performed for checking the assertions, as will be shown below. The sufficient conditions are the following:

Proposition 1 (Checking a calls assertion). *Let $A =$* `check calls` *$P : Pre$ be an assertion.*

- *If A is abstractly checked w.r.t. Q_α, then A is checked w.r.t. $\gamma(Q_\alpha)$.*
- *If A is abstractly false w.r.t. Q_α, then A is false w.r.t. $\gamma(Q_\alpha)$.*
- *otherwise, nothing can be deduced about A considered atomically (and it is left in check status).*

Soundness of the above statements can be derived directly from the correctness of abstract interpretation. In the case of checked assertions, we make sure that all call patterns that can appear at run-time belong to $\gamma(\lambda_{Pre})$. The "false" cases are a bit more involved. Due to the approximating nature of abstract interpretation, there is no guarantee that a given abstract call description λ^c corresponds to any call pattern that can appear at run-time. Thus, it is possible that the assertion is never applicable, but if it is, it will be invalid. What is known is that every run-time call pattern is described by one or more entries for P in AT. Thus, in order to ensure that no call pattern will satisfy λ_{Pre}, all λ^c's for P must be taken into account.

Finally, if a `calls` assertion is not abstractly checked nor abstractly false, we cannot deduce anything about A when it is considered atomically. However, we could still split it, and apply the same process to the parts.

Proposition 2 (Checking a success assertion). *Let* $A =$ `check success` $P : Pre \Rightarrow Post$ *be an assertion.*

- *If A is abstractly true, then A is true.*
- *If A is abstractly checked w.r.t. Q_α, then A is checked w.r.t. $\gamma(Q_\alpha)$.*
- *If A is abstractly false w.r.t. Q_α, then A is false w.r.t. $\gamma(Q_\alpha)$.*
- *otherwise, nothing can be deduced about A considered atomically (and it is left in check status).*

In the same way as before, a `success` assertion remains atomically `check` when it is not abstractly checked nor abstractly false. We can however simplify the assertion when part of the assertion can be proved to hold, like in a `calls` assertion. Note that the more precise analysis results are, the more assertions get status `true`, `checked` and `false`.

5 Checking assertions in a single module

The modular analysis framework described in Section 3 is independent from the assertion language. Nevertheless, assertions may contain relevant information for the analyzer. To this end when $analysis(m, E, AT)$ is computed for a module m, the parameters E and AT can also refer to information gathered directly from assertions, rather than from other analysis steps. This yields additional entry and success policies:

- E can be extracted from the call parts of `pred` assertions for exported predicates in m. Such set will be denoted as $\mathcal{CP}_m^{Asst} = \{P : \lambda_{Pre} \mid P \in exported(m) \wedge \texttt{pred}\ P : Pre \Rightarrow Post \in assertions(m)\} \cup \{P : \top \mid P \in exported(m) \wedge assertions(P) = \emptyset\}$.
- AT can also be extracted from `pred` (or `success`) assertions found in the imported modules. Given a module m, the answer table generated from the assertions for imported modules is denoted as $\mathcal{AT}_m^{Asst} = \bigcup_{n \in imports(m)}(\{P : \lambda_{Pre} \mapsto \lambda_{Post} \mid P \in exported(n) \wedge \texttt{pred}\ P : Pre \Rightarrow Post \in assertions(n)\} \cup \{P : \top \mapsto \top \mid P \in exported(n) \wedge assertions(P) = \emptyset\})$.

Note that we assume the topmost patterns if no assertions are present.

When checking assertions of modular programs, a given module can be considered either in the context of a program unit or separately, taking into account only the imported predicates. When treated in the context of a program unit, the calling context of a module m is called the *set of initial queries* Q_m. We say that the set of initial queries Q_m to a module m is *valid* iff for every imported predicate p all the calls assertions related to p are checked w.r.t. Q_m.

Definition 6 (Partially correct in context module). *A module m is* partially correct in context *with respect to a set of initial queries Q_m iff (1) every calls assertion in m is checked w.r.t. Q_m, and (2) every success assertion in m is true, or checked w.r.t. Q_m, and (3) every calls assertion for a predicate imported by m is checked with respect to Q_m.*

Definition 7 (Partially correct module). *A module m is* partially correct *iff m is partially correct in context w.r.t. any valid set of initial queries.*

Assertions are checked, as explained above, w.r.t. all analysis information available for a given (call or success of a) predicate after executing the analysis of the code. Such analysis information is multivariant, and covers all the program points in the analyzed code where a given predicate is called and how it succeeds. If available, a GAT table can be used to improve the analysis results with information from previous analyses of imported modules.

In our system, when checking a module, `calls` assertions for imported predicates are visible, and can therefore also be checked. This enables verifying whether a particular call pattern to an imported predicate satisfies its assertions. Of course, a `calls` assertion cannot be given status `true` or `checked`, as in general not all call patterns for the imported predicate occur in the calling module. Nevertheless, a warning or error is issued whenever the assertion is violated and/or cannot be shown to hold.

Proposition 3. *Let m be a module, and $LAT = analysis_{SP^+}(m, \mathcal{CP}_m^{Asst}, AT)$, where AT is an over-approximating answer table for (some modules in) $imports(m)$.*

The module m is partially correct if all success assertions are abstractly true w.r.t. LAT and all calls assertions for predicates in m and $imported(m)$ are abstractly checked w.r.t. LAT.

This proposition considers correctness of a single module regardless of the calling context of the module, since the starting point of the analysis is the set of preconditions in `pred` assertions. Note that LAT must be computed using an over-approximating success policy, in order to obtain correct results (provided that AT is correct). The answer table AT used for the analysis may be incomplete, or even an empty set: this approach allows us to check the assertions of a given module even when there is no information available from the imported modules. However, the more accurate AT is, the more assertions get status `true` or `checked`. This proposition is especially useful during the development of a modular program (i.e., the "edit-check cycle"), when different programmers develop different modules of the program. A programmer can check the assertions of his/her module as soon as it is syntactically correct. If other programmers in the team have analyzed their modules already, a shared GAT can be used to generate the answer table AT for checking the module more accurately.

Unfortunately, if the modules imported by m are not implemented yet, there is no possibility to analyze them in order to provide more accurate information to the analyzer. In order to overcome that, we can use the assertion information for the exported predicates in imported modules to obtain a more precise LAT. In this case, correctness of the module cannot be guaranteed, but a weaker notion of correctness, conditional partial correctness, may be proved. Note that in this case the analysis relies on possibly unverified assertions written by the user.

Proposition 4. *Let m be a module, and $LAT = analysis_{SP^+}(m, \mathcal{CP}_m^{Asst}, \mathcal{AT}_m^{Asst})$.*

The module m is conditionally partially correct if all success assertions are abstractly true, and all calls assertions for predicates in m and imported(m) are abstractly checked w.r.t. LAT.

This conditional partial correctness turns into partial correctness when the program unit is taken as a whole, as we will see in Section 6.

Example 2. Consider the standard `functor/3` predicate. The ISO standard for Prolog states that `functor/3` can only be invoked using two possible calling modes, and any other mode will raise a run-time error. The first mode allows obtaining the functor name and arity of a structure, while the second calling mode builds up a structure given its functor name and arity.

Our assertion checking system is able to statically detect such calling patterns because several assertions are allowed for a given predicate, and the underlying analyzer captures context-sensitive, multivariant abstract information. They can be expressed by means of the following assertions:

```
:- pred functor(+T,Name,Arity) => (atomic(Name), nat(Arity)).
:- pred functor(T,+Name,+Arity) : (atomic(Name), nat(Arity)) => nonvar(T).
```

In these assertions, the plus sign before an argument has the usual meaning of a Prolog mode, i.e., that the argument cannot be a free variable on calls. The calls parts of these assertions will be used when analyzing and checking any module that uses this library predicate, in order to check the calling modes to it.

6 Checking assertions in a program unit

Checking assertions in a program unit consisting of several modules differs from checking assertions in a single module in some ways. First of all, the most accurate initial queries to a given module m are provided by the calls to m made by other modules in the program unit (except the top-level one). Secondly, the success patterns of imported predicates may also be more accurate if we consider a given program unit. This leads us to the notion of correctness for program units. Note that the following definition concerns the concrete semantics.

Definition 8 (Partially correct program unit). *Let m_{top} be a module defining a program unit $U = program\_unit(m_{top})$. U is partially correct iff m_{top} is partially correct and $\forall m \in depends(m_{top})$, m is partially correct in context w.r.t. the sets of initial queries induced by the initial queries to m_{top}.*

Verifying a program unit with no intermodular analysis information

As explained in the previous section, every assertion A of the form `check calls` $P : \lambda_{Pre} \in assertions(m)$ where $P \in exported(m)$ is verified in every module that imports P from m. If such calls assertions are abstractly true in all importing modules (i.e., for every call pattern CP found in a module importing P we have that $CP \sqsubseteq \lambda_{Pre}$), then that means that λ_{Pre} approximates all possible calling

Algorithm 1 Checking assertions without modular analysis

Input: top-level module m_{top}
Output: Warning/Error messages, new status in the assertions in *program_unit* (m_{top})

```
for all m ∈ program_unit(m_top) do
    LAT_m := analysis_{SP+}(m, CP_m^Asst, AT_m^Asst)
    check_assertions(m, LAT_m)
end for
```

patterns to P from outside m. Therefore, the `calls` assertions can be used as starting points for analyzing every module in the program unit for checking the assertions. This leads us to a scenario for checking assertions, shown in Algorithm 1, where no prior intermodular analysis is required, and which aims at proving every module to be conditionally correct rather than correct in context.

Observe that Algorithm 1 does not use the modular analysis results as input. Instead, `pred` assertions of exported predicates are taken as input to the single-module analysis phase, $\mathcal{CP}_m^{Asst}$. A similar policy is applied when collecting success patterns of imported predicates.

This scenario can be viewed as proving conditional correctness of each module $m \in \mathit{program\_unit}(m_{top})$, where the conditions are the corresponding `pred` assertions from imported modules, as stated in Proposition 4. On the other hand, since we check all the modules in the program unit, and the program unit is self-contained, the `pred` assertions from imported modules are also the subject of checking. Assume that after checking all the modules in $\mathit{program\_unit}(m_{top})$ all the `pred` assertions get status `checked` or `true`.[8] This means that for every exported/imported predicate P, the analysis information $P : CP \mapsto AP$ generated when analyzing individual modules satisfies the checking conditions of Propositions 1 and 2. Thus, the following result holds:

Proposition 5. *Let m_{top} be a module defining a program unit $U = \mathit{program\_unit}(m_{top})$. If each module $m \in U$ is conditionally partially correct, and m_{top} is partially correct, then U is partially correct.*

If the assertions get true or checked using Algorithm 1, it is easy to see that they would also get true or checked if the (full) modular analysis were used, as modular analysis computes the least fixed point, i.e., it returns the most accurate analysis information. Consequently, if the `calls` assertions receive status `checked` and the `success` assertions receive status `true` when checking with Algorithm 1, there is no need to run a costly modular analysis.

Interleaving analysis and checking

Algorithm 1 may not be able to determine that a program unit is partially correct if the user has provided either too few assertions for exported predicates or they

[8] In this case the `calls` part originated from the `pred` assertion receives status `checked`, and the `success` part status `true`.

Algorithm 2 Interleaving analysis and checking

Input: top module m_{top}
Output: GAT, Warning/Error messages, new status in the assertions in *program_unit* (m_{top})

Set initial GAT with marked entries for call patterns from $\mathcal{CP}^{Asst}_{m_{top}}$
while there are modules with marked entries in GAT **do**
1 select module m
$LAT_m := analysis_{SP}(m, \mathcal{CP}^{GAT}_m, GAT)$
$check\_assertions(m, LAT_m)$
if an error is detected in m **then**
STOP
end if
2 update GAT with LAT_m
end while

are not accurate enough. In this case we have to replace information from the missing assertions and to incorporate a certain degree of automatic propagation of call/success patterns among modules during the checking process. The basic idea is to interleave analysis and compile-time checking during modular analysis. The main advantage of this approach is that errors will be detected as soon as possible, without computing an expensive intermodular fixpoint, yet having call and success patterns being propagated among modules. The whole process terminates as soon as an error is detected or when the modular analysis fixed point has been reached, as shown in Algorithm 2. Concrete procedures in steps 1 and 2 depend on a specific intermodular analysis algorithm, success and entry policies, etc. Note that in Algorithm 2 every module is analyzed for $\mathcal{CP}^{GAT}_m$, the set of all call patterns for a module m in the GAT.[9]

If an SP^+ success policy is used in Algorithm 2, then $LAT^1_m \succeq LAT^2_m \succeq \cdots \succeq LAT^n_m$, where LAT^n_m coincides with the analysis results of module m when the intermodular fixed point has been reached, and each of the LAT^i_m corresponds to the status of the analysis answer table for m at every iteration of the algorithm that schedules m for analysis.

Proposition 6. *Let LAT_m be an answer table for module m. If an assertion is abstractly checked (resp. abstractly true or abstractly false) w.r.t. LAT_m it will also be abstractly checked (resp. abstractly true or abstractly false) w.r.t. any answer table LAT'_m s.t. $LAT'_m \preceq LAT_m$.*

Thus, the conclusions drawn about the assertions are sound in the following sense: if an assertion is detected to be checked or false in an intermediate step, it will surely remain checked or false at the end of the process. If the assertion is not yet proved not disproved, its status might change in the subsequent steps as the analysis information might be more accurate in future iterations.

[9] $\mathcal{CP}^{GAT}_m$ is used for simplicity of the presentation. In the actual implementation the modules are analyzed just for the *marked* entries, and only the assertions related to those entries are checked.

Algorithm 2 can be adapted to apply the SP^- success policy. The sequence of answer tables generated during the analysis using that policy is now $LAT_m^1 \preceq LAT_m^2 \preceq \cdots \preceq LAT_m^n$, where only LAT_m^n, i.e. the one corresponding with the global fixpoint, is guaranteed to safely approximate the module's semantics.

Proposition 7. *Let LAT_m be an answer table for module m. If an assertion A is not abstractly checked w.r.t. LAT_m, then $\forall LAT'_m$ s.t. $LAT_m \preceq LAT'_m$, A will not be abstractly checked w.r.t. LAT'_m.*

Therefore, in this case the following conclusions can be made about the final status of assertions: if at any intermediate step the status of an assertion remains as `check` or becomes `false`, it will at most be `check` at the end of the whole process. Therefore, Algorithm 2 must stop and issue an error as soon as `false` or `check` assertions are detected (instead of stopping only when there are `false` assertions, as above).

Sufficient condition for partial correctness follows:

Proposition 8. *Let m_{top} be a module defining a program unit $U = program\_unit(m_{top})$. If Algorithm 2 terminates without issuing error messages, then (1) if SP^+ is used and Algorithm 2 decides that an assertion A is abstractly true (resp. checked), then A is true (resp. checked); and (2) if SP^- is used then all assertions in U are checked.*

7 Conclusions

Algorithms 1 and 2 have different levels of accuracy, computing cost, and verification power. The advantages of Algorithm 2 are that it is potentially more accurate and it does not impose any burden on the user, since no assertions are compulsory. On the other hand, Algorithm 1 has low computing cost, since modules only need to be analyzed once and it can be applied to incomplete programs. All this at the price of a development policy where module interfaces are accurately described using assertions.

Comparing this paper with related work, the scenario described in Section 6 can be seen as an instance of the analysis with user-provided interface of [7]. Our goal is however different than theirs: instead of computing the most precise analysis information we try to prove or disprove assertions, which makes this method more related in fact to the one of [4], focused on program verification. Nevertheless, unlike [4] we do not require the user to provide a complete specification, specially in Algorithm 2 –the missing parts are either described by topmost values or infered by the interleaved analysis algorithm.

Acknowledgements: This work was funded in part by the IST programme of the European Commission, FET project FP6 IST-15905 *MOBIUS*, by Ministry of Education and Science (MEC) projects TIN2005-09207-C03 *MERIT-COMVERS* and *MERIT-FORMS*, and CAM project S-0505/TIC/0407

PROMESAS. M. Hermenegildo is also supported in part by the Prince of Asturias Chair in Information Science and Technology at UNM. P. Pietrzak is supported by a 'Juan de la Cierva' grant provided by the Spanish MEC.

References

1. K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.
2. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *JLP*, 10:91–124, 1991.
3. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *JLP*, 39(1–3):43–93, 1999.
4. M. Comini, G. Levi, and G. Vitiello. Modular abstract diagnosis. In *APPIA-GULP-PRODE'98*, pages 409–420, 1998.
5. J. Correas, G. Puebla, M. Hermenegildo, and F. Bueno. Experiments in Context-Sensitive Analysis of Modular Programs. In *LOPSTR'05*, LNCS. Springer-Verlag, September 2006.
6. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
7. P. Cousot and R. Cousot. Modular Static Program Analysis, invited paper. In *CC 2002*, number 2304 in LNCS, pages 159–178. Springer, 2002.
8. M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Trans. on Programming Languages and Systems*, 18(5):564–615, 1996.
9. P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.
10. W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.
11. K. R. M. Leino and P. Müller. Modular verification of static class invariants. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Formal Methods (FM)*, volume 3582 of *LNCS*, pages 26–42. Springer-Verlag, 2005.
12. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *JLP*, 13(2/3):315–347, July 1992.
13. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.
14. G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *LOPSTR'99*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.
15. G. Puebla, J. Correas, M. Hermenegildo, F. Bueno, M. García de la Banda, K. Marriott, and P. J. Stuckey. A Generic Framework for Context-Sensitive Analysis of Modular Programs. In M. Bruynooghe and K. Lau, editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, number 3049 in LNCS, pages 234–261. Springer-Verlag, 2004.
16. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.