facultad de informática

universidad politécnica de madrid

Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging

M. Hermenegildo
G. Puebla
F. Bueno

TR Number CLIP8/98.0

Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging

Technical Report Number: CLIP8/98.0

Authors

M. Hermenegildo, *G. Puebla*, *F. Bueno*
{herme,german,bueno}@fi.upm.es
Department of Computer Science
Technical University of Madrid (UPM)

Acknowledgements

This work has been supported in part by ESPRIT project DiSCiPl and CICYT project ELLA. The authors would also like to thank Lee Naish, Jan Małuszyński, Wlodek Drabent, and Pierre Deransart for many interesting discussions on assertions and assertion-based debugging, Pawel Pietrzak for his adaptation of John Gallagher's type analysis for CLP($\mathcal{FD}$), and Abder Aggoun, Helmut Simonis, Eric Vetillard and Claude Lai for their feedback on the assertion language design.

Abstract

We discuss a framework for the application of abstract interpretation as an aid during program development, rather than in the more traditional application of program optimization. Program validation and detection of errors is first performed statically by comparing (partial) specifications written in terms of assertions against information obtained from (global) static analysis of the program. The results of this process are expressed in the user assertion language. Assertions (or parts of assertions) which cannot be checked statically are translated into run-time tests. The framework allows the use of assertions to be optional. It also allows using very general properties in assertions, beyond the predefined set understandable by the static analyzer and including properties defined by user programs. We also report briefly on an implementation of the framework. The resulting tool generates and checks assertions for Prolog, CLP(R), and CHIP/CLP(fd) programs, and integrates compile-time and run-time checking in a uniform way. The tool allows using properties such as types, modes, non-failure, determinacy, and computational cost, and can treat modules separately, performing incremental analysis.

Contents

1 Introduction

As (constraint) logic programming systems mature further and larger applications are built, an increased need arises for advanced development and debugging environments. Such environments will likely comprise a variety of co-existing tools ranging from declarative debuggers to execution visualizers (see, for example, [1] for a more comprehensive discussion of tools and possible debugging scenarios). In this paper we concentrate our attention on the particular issue of program validation and debugging via direct static and/or dynamic checking of user-provided *assertions* [17, 18, 8, 6, 30, 2]. Classical examples of assertions are the type declarations used in languages such as Gödel [24] or Mercury [36] (and in functional languages). But here, and encouraged by the capabilities of the currently available abstract interpreters, we depart in several ways from the traditional approaches.

We start by recalling some classical definitions (see, e.g., [10]) in program validation and debugging. Given a program P, we denote by $\mathcal{I}$ the *intended semantics* for P, i.e., the specification for P. We denote by $[\![P]\!]$ the *actual semantics* of the current implementation of program P. We say that

- P is *partially correct* w.r.t. $\mathcal{I}$ iff $[\![P]\!] \subseteq \mathcal{I}$.
- P is *complete* w.r.t. $\mathcal{I}$ iff $\mathcal{I} \subseteq [\![P]\!]$.
- P is *incorrect* w.r.t. $\mathcal{I}$ iff $[\![P]\!] \not\subseteq \mathcal{I}$.
- P is *incomplete* w.r.t. $\mathcal{I}$ iff $\mathcal{I} \not\subseteq [\![P]\!]$.

Performing these validation tasks can result in the validation of P w.r.t. $\mathcal{I}$, i.e., proving that P is partially correct and/or complete w.r.t. $\mathcal{I}$, or in the detection of incorrectness and/or incompleteness *symptoms*, which would flag the existance of errors in P, and in which case a process of diagnosis should be started to locate such errors.

There are many ways in which the validation task can be performed [3, 4, 16, 19, 38]. In general, direct application of the previous definitions is not practical for different reasons. First, providing the entire and exact intended semantics $\mathcal{I}$ may be a tedious task. Also, the actual semantics $[\![P]\!]$ of P may be an infinite object and it is often more convenient to use approximations of it. In the framework we propose, as in most existing debugging frameworks, we concentrate on *partial correctness* [1] debugging, i.e., we try to detect incorrectness symptoms or to prove that they do not exist.

We assume that the starting point for correctness validation and debugging is a set of user-provided assertions. At the same time, we would like our system to be as general as possible. First, we would like the assertions to be *optional*: specifications may be given only for some parts of the program (i.e., assertions may be given for only some procedures or program points), and even for those parts the information given may be incomplete (e.g., for a predicate we may perhaps have the type of one argument, the mode of another, and no information on other arguments). Also, we are interested in supporting assertions which are much more general than traditional type declarations, and such that it may be statically undecidable whether they hold or not for a given program. Finally, we would like the system to *generate* assertions, especially

[1] For brevity, we will usually write correctness/incorrectness when referring to *partial* correctess/incorrectness.

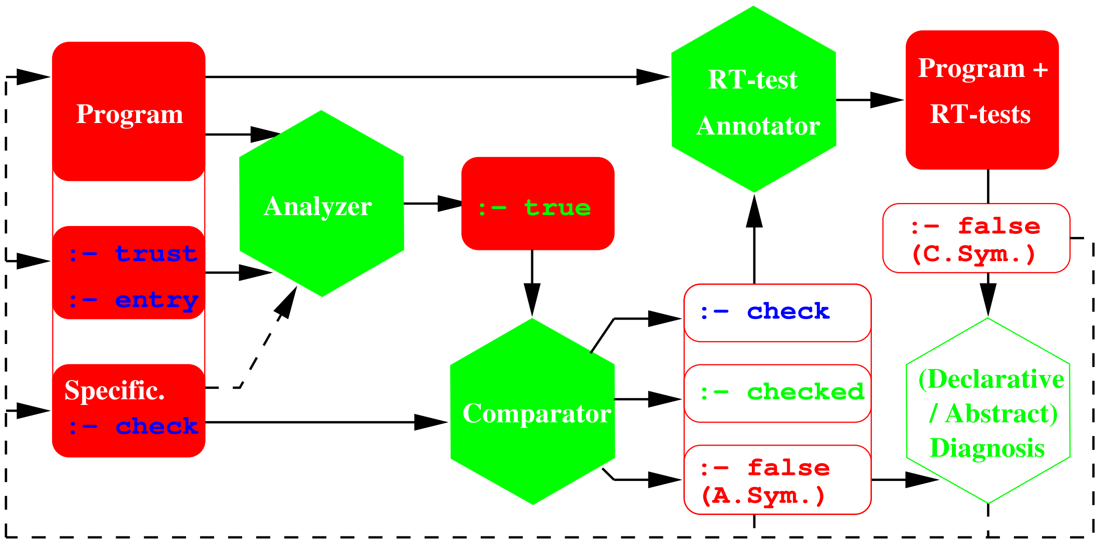

Figure 1: A Combined Framework for Program Development and Debugging

for parts of the program for which there are no `check` assertions. These assertions will have the status `true` and can be visually inspected by the user for checking correctness.[2]

As a consequence of our assumptions, the overall framework needs to deal throughout with approximations [10, 14, 22]. Thus, while the system can be complete with respect to statically decidable properties (e.g., certain type systems), it cannot be complete in general, and analysis may or may not be able to prove in general that a given assertion holds. The overall operation of the system will be sometimes imprecise but must always be *safe*. This means that all violations of assertions flagged by the system should indeed be violations, but perhaps there are assertions which the system cannot statically determine to hold or not. This means that the compiler cannot in general reject a program because it has not been able to prove that the complete specificiation holds. In order to limit the impact of this and at the same time detect as many errors as possible, we would like to design the framework and the assertion language in such a way that dynamic checking of assertions (run-time tests) is supported in addition to static checking. Furthermore, we would like to use (to the extent possible) the source language to perform such run-time tests, so that (at least conceptually) the addition of run-time tests to a program can be viewed as a source to source transformation.

Our approach is strongly motivated by the availability of powerful and mature static analyzers for (constraint) logic programs, generally based on abstract interpretation [14]. These analyzers have proved quite effective in statically *inferring* a wide range of program properties accurately and efficiently, for realistic programs (see, e.g., [23, 28, 11, 20, 21, 25, 8, 9] and their references). Such properties can range from types and modes to determinacy, non-failure, computational cost, independence, or termination, to name a few. Traditionally the results of static analyses have been applied primarily to program optimization: parallelization, partial evaluation, low level code optimization, etc. However, here we are interested instead in the applications of static analysis in *program development* (see, e.g., [5, 10, 22]), and, in particular in validation and error detection.

[2]Note however that if `check` assertions exist for such parts of the program, such checking is automated in the system by either compile-time or run-time checking.

2 Overall Framework Architecture and Operation

Figure 1 depicts the overal architecture of the proposed framework. Hexagons represent the different tools involved and arrows indicate the communication paths among the different tools. It is a design objective of the framework that most of such communication be performed also in terms of assertions. This has the advantage that at any point in the debugging process the information is easily readable by the user. Also, rather than having different *assertion languages* for each tool, we propose the use of a common assertion language for all of them, since this facilitates the above mentioned communication among the different tools, enables easy reuse of information (i.e., once a property has been stated there is no need to repeat it for the different tools), and facilitates understanding by the user, who only needs to learn a single assertion language. Note that not all tools need to be capable of dealing with all properties expressible in the assertion language. Rather, each tool only makes use of the part of the information given as assertions which the tool "understands" (this being allowed by the approximation-based approach used throughout the system). We now provide an overview of some of the characteristics of the assertion language used to describe the (partial) specification, we discuss how such assertions are used to perform run-time and compile-time program correctness validation and debugging, and we provide pointers to the rest of the paper, where each individual topic is informally discussed in more detail.

2.1 Check Assertions

As mentioned before, we assume that the user provides a set of assertions (the assertion language itself will be introduced in sections 3 and 4). All these assertions (and those which will be mentioned later) are written in the same syntax, with a prefix denoting their *status*. Because the user assertions are to be checked we say that such assertions have status "`check`" and refer to them as "`check` assertions" (see Figure 1).[3] The fact that an assertion has `check` status may be made explicit by prepending the `check` keyword to it, but `check` is the default assertion status and is therefore not required.

Intuitively, `check` assertions are just necessary conditions for the program to be correct. I.e., if they do not hold then the program is definitely incorrect. However, and as we do not require that `check` assertions encode a complete specification of P, the fact that all `check` assertions hold does not necessarily mean that the program is correct w.r.t. the semantics the user has in mind, much in the same way that a type correct program may produce incorrect results. Another way of looking at these assertions is as integrity constraints: if they do not hold then something is definitely wrong. Such assertions may be included in the program itself or provided separately.

We make a conceptual distinction between the notions of *property* and *assertion*. Properties are logic predicates, in the sense that the evaluation of each property either succeeds or fails (returns the value *true* or the value *false*). Properties are used to say that "X is a list of integers," "Y is ground," "p(X) does not fail," etc. The truth value of the assertion is that obtained by combining the truth values of the individual properties. Each individual assertion is constructed as a logic formula in a restricted syntax (to be described later) whose components are properties. The language of assertions we propose is structured around a relatively small and

[3] In addition, the user may optionally provide additional information to the analyzer by means of "`entry`" assertions (which describe the external calls to a module) and "`trust`" assertions (which provide abstract information on a predicate that the analyzer can use even if it cannot prove such information to be true) [8, 31].

fixed set of (classes of) assertions (which will be discussed in more detail in Section 3). The use of one or another class of assertion will indicate in which sets of execution states the assertion is *applicable*, such as for example: the success states or the call states of a predicate, the states corresponding to a program point between two clause body literals, the whole computation of a given call, etc. A program P is correct w.r.t. an assertion A if in all execution states reachable from valid input values for P either A is not applicable or the truth value of A is true. The assertion language leaves open the set of properties which may be used. The properties of interest may differ from one case to another and we allow the user to define such properties. There are two main kinds of properties: properties of execution states and properties of a computation. Such properties will be discussed in Section 4.

2.2 Run-time checking of assertions

The above mentioned assertions can then be checked at run-time, in the classical way, i.e., run-time tests will be added to the program which encode in some way (parts of) the given assertions. In the proposed framework, this is performed by the run-time test *annotator* module (Figure 1). This module takes the program and the `check` assertions as input (we assume for now that the *comparator* module of Figure 1 simply passes the `check` assertions through). The transformation (discussed in Section 5) must be such that, whenever the transformed program is executed, the assertions are checked for the data actually being explored by the program during execution, and this is done at the execution points that the assertions refer to. If the checking of any of the assertions fails, this implies that the assertion is `false`. Thus, a concrete[4] incorrectness symptom has been detected and some kind of error message is given to the user (and a procedure for localizing the cause of the error, such as standard or declarative diagnosis can be started).[5] *Correctness* of the transformation requires that the transformed program only produce an error if the specification is in fact violated.

2.3 Compile-time checking of assertions

Even though run-time checking can be very useful for detecting violations of specifications, it also has important drawbacks. First, run-time checking clearly introduces overhead into program execution. Also, it requires test cases (sample input data), which typically have an incomplete *coverage* of the program execution paths. Also, run-time checking cannot be used in general for proving that a program is correct w.r.t. an assertion, i.e., that the assertion is `checked`, as this would require testing the program with all possible input values, which is in general unrealistic.

Compile-time checking of assertions allows proving automatically at compile-time that (parts of) such assertions are implied by the program (i.e., that they hold in the program model, in the case of declarative properties) or that they hold for all possible program executions (i.e., in all SLD trees, for operational properties). This can also be viewed as computing at compile-time the results of run-time checking of assertions for all possible executions. Compile-time checking of assertions also allows proving that some assertions are violated without having to run the program.

Compile-time checking of assertions is performed in our framework (see Figure 1) by a *program*

[4] As oppossed to *abstract* incorrectness symptoms, which are the ones detected by compile-time checking.

[5] It is out of the scope of this paper to discuss how program diagnosis should be performed. However, techniques such as declarative debugging [35, 6, 17, 18] or abstract debugging [12, 13] may be applied.

analyzer (the abstract interpreter) and an *assertion comparator*. Properties of the program are automatically derived by the analyzer module (the kind of analysis performed may be selected by the user or determined automatically based on the properties used in the current `checked` assertions). The derived properties are also expressed using assertions. They have the status "`true`," since they express properties which have been proved to hold. The `true` assertions are then compared against the given `check` assertions. The result might be that the assertion is validated (in which case the corresponding assertions are rewritten as "`checked`" assertions) or it is proved not to hold, in which case *abstract* symptoms are detected, the corresponding assertions are rewritten as "`false`" assertions, and error messages are presented to the user. Once again, diagnosis should be started (for example using abstract diagnosis [12, 13]) in order to detect the cause of the error. It is also possible that a (part of the) assertion cannot be proved nor disproved. In this case some assertions (or part of them) remain in `check` status (and possibly warning messages are presented to the user).

Note that it may also be interesting to implement analysis in a demand-driven way, so that information is inferred only for the program points which include assertions. The advantage of this approach is that it may be more efficient. However, three other considerations should be weighted against this. First, for many properties it is not possible to isolate the analysis of a given program point, and a global fixpoint has to be reached in any case, which requires analyzing at least the whole module involved. Also, the results of analysis are typically useful in other stages of compilation (e.g., to perform program specialization or other optimizations). Finally, in our experience bugs can often be detected by visual inspection of the assertions containing the information inferred by the analyzer, sometimes for program points which are "distant" from the user-provided `check` assertions. Compile-time checking is discussed further in Section 6.

3 The Assertion Language

Assertions may be used in different contexts and for different purposes. In run-time checking, assertions are traditionally used to express conditions which should hold at run-time. A usual example is to check that the value of a variable remains within a given range at a given program point. In declarative debugging [35], assertions have been used in order to replace the *oracle* by allowing the user expressing properties of the intended behaviour of the program [17, 18, 6]. Assertions can also be used to express properties about the program to be checked at compile-time. An example of this are type declarations (e.g., [24, 36], functional languages, etc.), which have been shown to be useful in debugging. Assertions have also been used to provide information to an optimizer in order to perform additional optimizations during code generation (e.g., [36], which also implements checking). Assertions have also been proposed as a means of providing additional information to the analyzer, which it can use both to increase the precision of the information it infers and/or to perform additional optimizations during code generation [39, 37, 26, 25]. Also, assertions can be used to represent analysis output in source form and for communication between different modules of the compiler which deal with analysis information [8].

The assertion language used in our framework has been designed with the aim of being useful in all the contexts mentioned above. With this objective in mind, we depart from previous proposals in allowing more general properties to be expressed. Each tool in each of the contexts will then use the properties which are relevant to it. Assertions are provided to specify the program points to which the properties are "attached." In this sense they work as schemas.

```
:- calls qsort(A,B) : list(A).                        % A1
:- success qsort(A,B) : list(A) => list(B).           % A2
:- comp qsort(A,B) : (list(A),var(B)) + does_not_fail. % A3

qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R1,[X|R2],R).
qsort([],[]).
```

Figure 2: An example predicate definition with assertions

Due to space limitations, we do not present here the complete assertion language, but rather we concentrate on a subset of it which suffices for illustrating the main concepts involved in compile-time and run-time checking of assertions. In particular, we will focus on *predicate assertions* rather than on *program point assertions*. A more detailed description of the assertion language can be found in [31, 30].

Predicate assertions relate properties to the invocations of a predicate. Three kinds of predicate assertions are provided; they relate properties to the execution states at the time of calling the predicate, at the time of its success, and to the whole of its computation. More than one predicate assertion (of the same or different kinds) may be given for the same predicate. In such a case, all of them should hold and composition of predicate assertions should be interpreted as their conjunction.

We first illustrate the use of this kind of assertions with an example. Figure 2 presents (part of) a CIAO [7] program which implements the *quicksort* algorithm for sorting lists in ascending order. The predicate `qsort` is annotated with predicate assertions which express properties which the user expects to hold for the program.[6] Three assertions are given for predicate `qsort` (`A1`, `A2`, and `A3`), the meaning of which is explained below.

3.1 Assertions on Success States

They are similar in nature to the *postconditions* used in program verification. They can be expressed in our assertion language using the assertion schema '`:- success` *Pred* `=>` *Postcond*.' It should be interpreted as "for any call of the form *Pred* which succeeds, on success *Postcond* should hold." For example, we can use the following assertion in order to require that the output of a procedure (`qsort`) for sorting lists be a list:

```
:- success qsort(A,B) => list(B).
```

Note that, in contrast to other programming paradigms, in (C)LP, calls to a predicate may

[6] Both for convenience, i.e., so that the assertions concerning a predicate appear near its definition in the program text, and for historical reasons, i.e., mode declarations in Prolog or `entry` and `trust` declarations in PLAI [8] we write predicate assertions as directives. Depending on the tool different choices could be implemented, including for example putting assertions in separate files or incremental addition of assertions in an interactive environment.

either succeed or fail. The postcondition stated in a `success` assertion only refers to successful executions.

3.2 Assertions Restricted to a Subset of the Calls

Sometimes we are interested in properties which refer not to all invocations of a predicate, but rather to a subset of them. With this aim we allow the addition of preconditions (*Precond*) to predicate assertions as follows: '*Pred* : *Precond*.' For example, `success` assertions can be restricted and we obtain an assertion of the form '`:- success` *Pred* : *Precond* => *Postcond*,' which should be interpreted as "for any call of the form *Pred* for which *Precond* holds, if the call succeeds then on success *Postcond* should also hold." Note that '`:- success` *Pred* => *Postcond*' is equivalent to '`:- success` *Pred* : `true` => *Postcond*.'

For example, the assertion `A2` in Figure 2 requires that if `qsort` is called with a list in the first argument position and the call succeeds, then on success the second argument position should also be a list.

3.3 Assertions on Call States

It is also possible to use assertions to describe properties about the calls for a predicate which may appear at run-time. This is useful for at least two reasons. If we perform *goal-dependent* analysis, (a variation of) `calls` assertions, namely `entry` assertions (see [8]), may be used for improving analysis information.[7] They can also be used to check at run-time whether any of the calls for the predicate is not in the expected set of calls (the "inadmissible" calls of [29]). An assertion of the kind '`:- calls` *Pred* : *Cond*' must be interpreted as "all calls of the form *Pred* should satisfy *Cond*." An example of this kind of assertion is `A1` in Figure 2 which expresses that in all calls to predicate `qsort` the first argument should be a list.

3.4 Assertions on the Computation of Predicates

Many properties which refer to the computation of the predicate (rather than the input-output behaviour) are not expressible with the assertions presented above. In particular, no property which refers to (a sequence of) intermediate states in the computation of the predicate can be (easily) expressed using `calls` and `success` predicate assertions only. Examples of properties of the computation which we may be interested in are: non-failure, termination, determinacy, non-suspension, etc. In our language this sort of properties are expressed by an assertion of the kind '`:- comp` *Pred* : *Precond* + *Comp-prop*,' which is interpreted as "for any call of the form *Pred* for which *Precond* holds, *Comp-prop* should also hold for the computation of *Pred*." Again, the field '`:` *Precond*' is optional. For example, `A3` in Figure 2 requires that all calls to predicate `qsort` with the first argument being a list and the second a variable do not fail.

[7] The `entry` (and `trust`) declarations are also instrumental in incremental modular analysis.

4 Defining Properties

Whereas each kind of assertion indicates *when*, i.e., in which states or sequences of states, to check the given properties, the properties themselves define *what* to check. As mentioned before, properties are used to say things such as "X is a list of integers," "Y is ground," "p(X) does not fail," etc. and in our framework they are logic predicates, in the sense that the evaluation of each property either succeeds or fails. The failure or success of properties typically needs to be determined at the time when the assertions in which they appear are checked. As also mentioned previously, assertions can be checked both at compile-time and at run-time. In order to simplify the discussion, in this section we will concentrate exclusively on run-time checking (the role of properties during compile-time checking will be discussed in Section 6).

In order to make it possible to check a property at run-time, some *code* must exist somewhere in the system that performs this check. If the set of properties were fixed, the code to be used when performing the run-time tests could be contained in a predefined library. However, we would like to allow the user to define new, quite general properties. Since our properties are predicates, and we have assumed that our source language is a logic and/or constraint programming language (in which it is natural to define predicates and which typically offers extended meta-programming facilities), we choose to allow the user to *write the definitions of properties in the source language.* Writing the definition of a property in the source language has the advantage that in principle no special run-time support is then needed for checking properties at run-time, since it suffices to compile the predicate that defines the property with the rest of the program and simply call it at run-time in the appropriate places.[8]

A property may be a built-in predicate or constraint (such as `integer(X)` or `X>5`, and including extra-logical properties such as `var(X)`), an expression built using conjunctions of properties,[9] or, in principle, any predicate defined by the user, using the full underlying CLP language. As an example consider defining the predicate `sorted(B)` and using it as a postcondition to check that a more involved sorting algorithm such as `qsort(A,B)` produces correct results.

However, while we would like to allow writing properties that are as general as allowed by the full source language syntax, some limitations are useful in practice. Essentially, we would not like the behaviour of the program to change in a fundamental way depending on whether the run-time tests are being performed or not. While we can tolerate a degradation in execution speed, turning on run-time checking should not introduce non-termination in a program which terminates without run-time checking. To this end, we require that the user ensure that the execution of properties terminate for any possible initial state. Also, checking a property should not change the answers computed by the program or produce new side-effects: we will require that the definition of the property does not further instantiate its input arguments and does not perform input/output, add/delete clauses, etc. The responsibility of determining that properties meet these conditions is given to the user, whom we require to identify in a special way the predicates which he or she have determined to be legal properties. This is done by means of a declaration of the form "`:- prop` *predicate/arity*."[10]

Given the classes of assertions presented previously, there are two fundamental classes of

[8] Also, this allows using the standard program optimization tools (e.g., the program specializer) to avoid the run-time overhead of checking properties when they can be proven statically to hold.

[9] Although disjunctions are also supported, we restrict our attention to only conjunctions in our presentation.

[10] Nevertheless the compiler performs some basic checks on properties and flags properties which can be detected with these checks to violate the required conditions.

properties. The properties used in the *Cond* of `calls` assertions, *Postcond* of `success` assertions, and *Precond* of `success` and `comp` assertions refer to a particular execution state and we refer to them as *properties of execution states*. The properties used in the *Comp-prop* part of `comp` assertions refer to a sequence of states and we refer to them as *properties of computations*.

4.1 Writing Properties of Execution States: Compatibility Vs. Instantiation Properties

Consider a definition of the predicate `string_concat` which concatenates two character strings (we assume that strings are represented as lists of ASCII codes):

```
string_concat([],L,L).
string_concat([X|Xs],L,[X|NL]):- string_concat(Xs,L,NL).
```

Assume that we would like to state in an assertion that each argument "is a list of integers." However, we must decide which one of the following two possibilities we mean exactly: "the argument is *instantiated* to a list of integers" (let us call this property `instantiated_to_intlist`), or "if any part of the argument is instantiated, this instantiation must be compatible with it being a list of integers" (we will call this property `compatible_with_intlist`). For example, `instantiated_to_intlist` should succeed for calls with argument `[]` and `[1,2]`, but should fail for `X`, `[a,2]`, and `[X,2]`. In turn, `compatible_with_intlist` should succeed for calls with argument `[]`, `X`, `[1,2]`, and `[X,2]`, but should fail for `[X|1]`, `[a,2]`, and `1`. We refer to properties such as `instantiated_to_intlist` above as *instantiation properties* and to those such as `compatible_with_intlist` as *compatibility properties* (corresponding to the traditional notions of "instantiation types" and "compatibility types").

It turns out that both of these notions are quite useful in practice. In the example above, we probably would like to use `compatible_with_intlist` to state:

```
:- success string_concat(A,B,C) => ( compatible_with_intlist(A),
                                     compatible_with_intlist(B),
                                     compatible_with_intlist(C) ).
```

With this assertion, no error will be flagged for a call to `string_concat` such as `string_concat([20],L,R)`, which produces `string_concat([20],L,[20|L])` on success, but a call `string_concat([],a,R)` would indeed flag an error.

On the other hand, and assuming that we are running on a Prolog system, we would probably like to use `instantiated_to_intlist` for `sumlist` as follows:

```
:- calls sumlist(L,N) : instantiated_to_intlist(L).

sumlist([],0).
sumlist([X|R],S) :- sumlist(R,PS), S is PS+X.
```

to describe the type of calls for which the program has been designed.

The property `instantiated_to_intlist` might be written as follows (again, for running on a Prolog system):

```
:- prop instantiated_to_intlist/1.
```

```
instantiated_to_intlist(X) :-
       nonvar(X), instantiated_to_intlist_aux(X).

instantiated_to_intlist_aux([]).
instantiated_to_intlist_aux([X|T]) :-
       integer(X), instantiated_to_intlist(T).
```

(Recall that the Prolog builtin `integer` itself implements an instantiation check, failing if called with a variable as the argument.)

The property `compatible_with_intlist` might in turn be written as follows (also in Prolog):

```
:- prop compatible_with_intlist/1.

compatible_with_intlist(X) :-
       var(X).
compatible_with_intlist(X) :-
       nonvar(X), compatible_with_intlist_aux(X).

compatible_with_intlist_aux([]).
compatible_with_intlist_aux([X|T]) :-
       int(X), compatible_with_intlist(T).

int(X) :- var(X).
int(X) :- nonvar(X), integer(X).
```

Note that these predicates meet the criteria for being properties and thus the `prop` declaration is correct.

Ensuring that a property meets the criteria for "not affecting the computation" can sometimes make its coding somewhat tedious. In some ways, one would like to be able to write simply:

```
intlist([]).
intlist([X|R]) :- int(X), intlist(R).
```

but note that (independently of the definition of `int`) the definition above is not the correct instantiation check, since it would succeed for a call such as `intlist(X)`. In fact, it is not correct as a compatibility property either, because, while it would fail or succeed as expected, it would perform instantiations (e.g., if called with `intlist(X)` it would bind `X` to `[]`). In practice, it is convenient to provide some run-time support to aid in this task.

As we will see in Section 5, the run-time support of the framework ensures that the execution of properties is performed in such a way that properties written as above can be used directly as instantiation checks. Thus, writing:

```
:- calls sumlist(L,N) : intlist(L).
```

has the desired effect. Also, the same properties can often be used as compatibility checks by writing them in the assertions as `compat(`*Property*`)` (which should be interpreted as "*Property* holds in the current execution state or it can be made to hold by adding bindings (or constraints) to the current execution state"). Thus, writing:

```
:- success string_concat(A,B,C) => ( compat(intlist(A)),
                                     compat(intlist(B)),
                                     compat(intlist(C)) ).
```

also has the desired effect. As a general rule, the properties that can be used directly for checking for compatibility should be *downwards closed*, i.e., once they hold they will keep in holding in every state accessible in forwards execution. There are certain predicates which are inherently *instantiation* checks and should not be used as *compatibility* properties nor appear in the definition of a property that is to be used with `compat`. Examples of such predicates (for Prolog) are `==`, `ground`, `nonvar`, `integer`, `atom`, `>`, etc. as they require a certain instantiation degree of their arguments in order to succeed.

4.2 Writing Properties of Computations

Properties which appear in `comp` assertions refer to the entire execution of the predicate that the assertion refers to. It is therefore assumed that one of its arguments (the first one) is precisely the given call to which the property refers. For example, in assertion `A3` of Figure 2 for `qsort(A,B)`, the property `does_not_fail` (with 0 parameters) really means `does_not_fail(qsort(A,B))`. For this property, which should be interpreted as "execution of the predicate either succeeds at least once or loops," we can use the following predicate `does_not_fail` of arity 1 for run-time checking:

```
does_not_fail(Goal):-
      if( call(Goal),
          true,            %% then
          warning(Goal) ). %% else
```

where the `warning` predicate simply prints a warning message.

In this simple case, implementation of the predicate is not very difficult using the `if` builtin predicate present in many Prolog systems. However, it is not so easy to code predicates which check other properties of the computation and we may in general need to program a meta-interpreter for this purpose.

5 A Simple Run-time Checking Scheme

In this section we provide a possible scheme for translation of a program with assertions into code which will perform run-time checking. Our aim herein is not to provide the best possible transformation (nor the best definition of auxiliary predicates used by it), but rather to present simple examples with the objective of showing the feasibility of the implementation and hopefully clarifying the approach further.

5.1 Properties

In general, run-time testing assertions implies checking whether the properties contained in them hold or not. If they hold, computation should continue as usual. If they do not hold, an error message should usually be issued to the user. The following definition of predicate `check` can be used for this purpose (we assume that conjunctions and sets are implemented by means of lists):

```
check([]).
check([Cond|Conds]):-
        not(inst_prop(Cond)),!, error(Cond), check(Conds).
check([_Cond|Conds]):-  check(Conds).
```

where the `error` predicate simply prints a message informing about an assertion which does not hold. Thus, unless otherwise stated by the user (by enclosing a property in a `compat` meta-call) the checking of each individual property is performed by means of the predicate `inst_prop`, which represents the *instantiation check* introduced in Section 4.1. As an example, a possible implementation (for Prolog) of the `inst_prop` check is:

```
inst_prop(Cond):-
        copy_term(Cond,NCond), call(NCond), variant(NCond,Cond).
```

where `variant` checks that its arguments are identical up to variable renaming. This guarantees that `NCond` has not been further instantiated during run-time checking, i.e., that `Cond` is not only compatible, but also *implied* by the calling substitution. In a CLP setting, the `inst_prop` check needs to test this implication (i.e., it is an *entailment* test).

Alternatively, if the property is to be checked for compatibility (i.e., it is enclosed in a `compat` meta-call), the corresponding test may be done by simply calling the property, allowing that the variables be further instantiated, i.e., that additional constraints be placed on the store. However, we do not want this possible further instantiation to be "propagated" to the rest of the execution. This can be ensured for example by using backtracking to undo things, e.g. (recalling that compatibility properties are wrapped around a `compat` meta-call):

```
compat(Cond):- not(not(Cond)).
```

5.2 Success Assertions

A possible translation scheme for `success` assertions into run-time tests is the following. Let A(`p/n`) represent the set of current assertions for predicate `p` of arity `n`. Let S be the set $\{Postcond$ s.t. '`:- success` $p(X1, ..., Xn)$ `=>` $Postcond$' $\in A(p/n)\}$. Then the translation is:

```
p(X1,...,Xn):- p_int(X1,...,Xn), check(S).
```

A possible translation scheme for `success` assertions with a precondition follows. Let RS be the set $\{(Precond, Postcond)$ s.t. '`:- success` $p(X1, ..., Xn)$ `:` $Precond$ `=>` $Postcond$' $\in A(p/n)\}$. Then the translation is:

```
p(X1,...,Xn):-
        collect_valid_postc(RS,S),
        p_int(X1,...,Xn), check(S).
```

The predicate `collect_valid_postc/2` collects the postconditions of all pairs in `RS` s.t. the precondition holds. Note that those assertions whose precondition does not hold are directly discarded. A possible implementation of such predicate is given below:

```
collect_valid_postc([],[]).
collect_valid_postc([(Pre,Post)|Cs],PC):-
        not(not(inst_prop(Pre))),!,
        PC = [Post|PCs], collect_valid_postc(Cs,PCs).
collect_valid_postc([_|Cs],PC):-
        collect_valid_postc(Cs,PC).
```

The double negation by failure around `inst_prop(Pre)` is not extrictly required. However it is introduced for reducing the memory-usage overhead introduced by run-time checking.

5.3 Calls Assertions

Let C be the set $\{Cond$ s.t. '`:- calls` $p(X1,...,Xn)$ `:` $Cond$' $\in A(p/n)\}$. A possible translation scheme for `calls` assertions into run-time tests is:

```
p(X1,...,Xn):-  check(C), p_int(X1,...,Xn).
```

5.4 Comp Assertions

A possible translation scheme of `comp` assertions into run-time tests is the following. Let RC be the set $\{(Prec, Comp\_prop)$ s.t. '`:- comp` $p(X1,...,Xn)$ `:` $Prec$ `+` $Comp\_prop$' $\in A(p/n)\}$. Then the translation is:

```
p(X1,...,Xn):-
        collect_valid_postc(RC,C),
        add_arg(C,p_int(X1,...,Xn),C1),
        ( C1 == [] ->
          call(p_int(X1,...,Xn)) %% then
        ; call_list(C1) ).       %% else

call_list([]).
call_list([C|Cs]):- call(C), call_list(Cs).
```

where the predicate `add_arg` adds the goal `p_int(X1,...,Xn)` as the first argument to any property of the computation, i.e.:

```
add_arg([],_,[]).
add_arg([C|Cs],Goal,[NC|NCs]):-
        C=..[F|Args], NC=..[F,Goal|Args], add_arg(Cs,Goal,NCs).
```

Note that both `success` and `calls` assertions are in a sense special cases of `comp` assertions, since properties of call and success states can also be formalized as properties of the computation. For example consider the following predicates which could be used for checking `calls` and `success` properties at run-time:

```
calls(Goal,Prop):-                success(Goal,Prop):-
      ( call(Prop) ->                   call(Goal),
          true                          ( call(Prop) ->
      ;   error(Prop) ),                    true
      call(Goal).                       ;   error(Prop) ).
```

the assertion '`:- calls p(X) : ground(X)`' could be written '`:- comp p(X) + calls(ground(X))`.' Thus, an assertion language with only the `comp` predicate assertion would suffice. However, `calls` and `success` assertions appear very often in program debugging and their treatment (at least for run-time checking) is much simpler than that of the more general `comp` assertion. As a result, it is interesting to have a dedicated predicate assertion for them and only use `comp` assertions when the specification is not expressible as `calls` or `success` assertions.

6 Compile-Time Checking

We now turn our attention to compile-time checking of assertions. As mentioned before, and motivated by the availability of practical global static analyzers supporting a number of abstract domains, our approach is to compare the information generated during global analysis with the `check` assertions present in the program. Because we typically support properties which are statically undecidable, the information available at compile-time will not always allow determining whether a given assertion will hold at run-time or not. This case may also arise because the analysis itself is not accurate enough. We accept the fact that the approach will be weaker *in general* than that offered by, e.g., *strong* type systems. On the other hand, the same results obtained with a strong type system can be achieved by selecting an analysis that uses the same type system as abstract domain and providing sufficient (type) assertions in the program.

Informally, the actual checking of the assertions at compile-time is performed as follows (precise details on how to reduce assertions at compile-time can be found in [32]). The properties which appear in the user-provided `check` assertions are compared one by one with the properties inferred by the analysis. An assertion is validated if all its properties are *implied* by the analysis results (preconditions require special consideration in this process). On the other hand, errors are detected if any property specified is *incompatible* with the analysis results. If it is not possible to prove nor to disprove an assertion, then such assertion is left as a `check` assertion, for which run-time checks might be generated. However, if some properties are implied but others cannot be proved nor disproved, the assertion as a whole can be *simplified*, in the sense of reducing the number of properties which have to be checked at run-time.

For example, assume that we have the following user-provided assertions:

```
:- check success p(X,Y) =>  (intlist(X),ground(Y)).
:- check comp    p(X,Y) +   (does_not_fail,terminates).
```

and that we are running a mode and non-failure analysis which has inferred the following information:

```
:- true success  p(X,Y) =>  (var(X),ground(Y)).
:- true comp     p(X,Y) +   does_not_fail.
```

Then, the user-provided assertions could be transformed into:

```
:- check success p(X,Y) => intlist(X).
:- check comp    p(X,Y) +  terminates.
```

With this compile-time simplification process in mind, we discuss further the nature of the properties which may appear in assertions and their treatment. In traditional systems which focus on compile-time checking (e.g., type systems), the properties allowed are usually restricted to those for which the available analyzer (e.g., the type checker) can decide whether they hold or not at compile-time. Conversely, in traditional systems which focus on run-time checking, usually only properties which are executable are considered. While most systems using assertions focus on either run-time or compile-time checking, in the framework we propose both techniques are combined. As a result, compile-time checking must be able to deal (at least safely) with properties that have perhaps been written with run-time checking in mind or for which no specific analysis is available. Conversely, the run-time checking machinery must also be able to deal correctly with properties that are primarily meant for compile-time checking.

Let us divide properties into classes, *from the point of view of a given analysis.* First, we will call *native* properties those which are directly "understood" (abstracted) by this analysis. This is the case for example of properties like `ground` or `var` for a mode analysis, `does_not_fail` for a non-failure analysis, `terminates` for a termination analysis, or a predicate defining a (regular) type for a regular type analysis, etc. These native properties can be recognized when appearing in an assertion either by name (as with `ground`, `var`, etc.) or by syntax (e.g., for regular types [40, 15], which in our case are defined by a regular (pure) logic program, and this can be recognized at compile-time).

If a property appearing in an assertion is native of an analysis then it is often possible to either prove it or disprove it, provided that the analysis is accurate enough and the "direction" of approximation performed by the analysis is the appropriate one [32, 10] (this is the case of the properties `var` and `does_not_fail` in the example above). Also, if the analysis is `precise` (in the sense that the abstract operations do not lose information beyond the abstraction implied by the abstraction function used [14]), then whether such properties hold or not will be decidable in all cases. However, since there may in general be cases in which properties remain for run-time checking (and because in our framework the definition properties can be called from user programs) we require that there be an executable definition of all properties available in the system.

On the other hand, note that there are properties which can be proved (or disproved) at compile-time by a given analyzer but for which no accurate definition can be written in the underlying language. An extreme example of this is the property `terminates`, for which it is obviously not possible to define a run-time test which will give a warning if it does not hold. For these properties, an *approximate* definition may be given, and this approximation should be correct in the usual sense that all errors flagged should be errors, but there may be errors that go unchecked. For example `terminates` may simply be over-approximated as `terminates(_)`, and the user should obviously not expect non-termination problems to be detected at run-time with this definition. In summary, it is not necessary that the executable definition of all properties be an exact implementation of a given property, but the user must provide (or import) some code for each property and understand and take into account the impact of approximation being performed in the property definition when using these properties in assertions.

Conversely, and again for a given analysis, there may properties which are defined precisely and are perfectly executable at run-time, but which may not be *native* for that analysis. For them, the analysis may not be capable of obtaining an exact representation (abstraction). However, a useful approximation (usually an over-approximation) of such property can be obtained by *directly analyzing the code which defines the property.* As an example, consider the code for the property `intlist` (as defined in Section 4). By simply analyzing this code the mode analyzer can abstract it (in its use as an *instantiation* property) as `ground`.

The fact that the resulting internal representation in the analyzer of a non-native property is itself an approximation must be taken into account. For example, if an over-approximation of the property definition is performed (as in the example above), and the analysis is itself an over-approximating analysis, an occurrence of the property in an assertion cannot be proved to hold. However, it can be proved that the such property occurrence does `not` hold, if the information inferred by the analysis is *incompatible*, thus detecting an incorrectness symptom. In fact, in the example above it would be detected that the `intlist(X)` (i.e., `ground(X)`) requirement on success is incompatible with the inferred information `var(X)`, statically detecting the presence of an error.

In general, typical analyzers obtain over-approximations of properties, i.e., they succeed for a superset of the cases in which the exact property would succeed. However, for the case of properties in preconditions of `success` or `comp` assertions, under-approximations (i.e., the approximation succeeds for a subset of the cases in which the exact property would succeed) rather than over-approximations should be considered. Otherwise, the preconditions cannot be guaranteed to hold, and therefore it would not be possible (in general, although there are some exceptions) to guarantee that the preconditioned assertion is applicable (since the exact precondition could possibly not be applicable, even though its approximation is implied). More details on the use of approximations for program debugging can be found in [10, 32].

7 A Sample Debugging Session with the CIAO System

We now illustrate some uses of the proposed framework by means of a sample session with `ciaopp`, the CIAO system preprocessor, which is currently a part of the programming environment, and which is directly based on the proposed approach.[11] `ciaopp` uses as analyzers both the LP and CLP versions of the PLAI abstract interpreter [28, 9, 21] and adaptations of Gallagher's type analysis [20], and works on a large number of abstract domains including moded types, definiteness, freeness, and grounding dependencies (as well as more complex properties, such as bounds on cost, determinacy, non-failure, etc., for Prolog programs).

We consider the program in Figure 3. Note that the program is a module (which helps performing precise global analysis). Also, note that properties can be defined in the module itself (e.g., `sorted_num_list`) or imported from other modules (e.g., `list`), and they can also

[11] We have implemented the schema of Figure 1 as a *generic framework*. This genericity means that different instances of the tools involved in the schema can be incorporated in a straightforward way. Currently, two different experimental debugging environments have been developed using this framework: `ciaopp`, the CIAO system preprocessor, developed by UPM, and `fdtypes`, an assertion-based type inferencing and checking tool developed by Pawel Pietrzak at the U. of Linköping, in collaboration with UPM. Also, an assertion-based preprocessor for PrologIV has been developed by Claude Lai of PrologIA extending the work of [38], which is based on the same overall design, but separately coded and using simpler analysis techniques. These three environments share the same source language (ISO-Prolog + finite domain constraints) and the same assertion language [31], so that source and output programs (annotated with assertions and/or run-time tests) can be easily exchanged. `fdtypes` has been interfaced by Cosytec with the CHIP system (adding a graphical user interface) and is currently under industrial evaluation.

```
:- module(qsort, [qsort/2], [assertions]).
:- use_module(library(lists),[list/2]).

:- entry qsort(A,B) : (list(A, num), var(B)).

:- calls qsort(A,B) : list(A, num).                        % A1
:- success qsort(A,B)  => (ground(B), sorted_num_list(B)). % A2

qsort([X|L],R) :-
        partition(L,L1,X,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R2,[X|R1],R).
qsort([],[]).

:- calls partition(A,B,C,D) : (ground(A), ground(B)).      % A3
:- success partition(A,B,C,D) => (list(C, num),ground(D)). % A4

partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right):-
        E < C, !, partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        E >= C,   partition(R,C,Left,Right1).

append([],X,X).
append([H|X],Y,[H|Z]):- append(X,Y,Z).

:- prop sorted_num_list/1.

sorted_num_list([]).
sorted_num_list([X]):- number(X).
sorted_num_list([X,Y|Z]):-
        number(X), number(Y), X<Y, sorted_num_list([Y|Z]).
```

Figure 3: A tentative qsort program

be “builtins” (i.e., in modules loaded by default, such as `var`, `ground`, `num`, etc.). The `entry` declaration informs the analyzer that in all calls to `qsort`, the first argument will be a list of numbers and the second a free variable. This will aid goal-dependent analysis in order to obtain more accurate information (and will also allow program optimizations). `A1` uses the parametric type `list(A, num)` which means that `A` is (or should be) a list of numbers. `A2` combines a mode property (`ground`) with a user-defined property (`sorted_num_list`). The code defining such property is included in the program in Figure 3. `A3` only contains mode properties while `A4` contains a combination of type and mode properties. Note that none of the assertions included in the program is compulsory and that properties natively understood by different analysis domains may be combined in the same assertion.

Using type and mode analysis, the assertions `A1` to `A4` are simplified at compile-time into:

```
:- checked calls qsort(A,B) : list(A,num).                      % A1
:- check success qsort(A,B) => sorted_num_list(B).              % A2
:- false calls partition(A,B,C,D) : ground(A),ground(B).        % A3
:- checked success partition(A,B,C,D) => (list(C,num), ground(D)). % A4
```

Assertion A3 has been detected to be false. This is a (compile-time, or *abstract*) incorrectness symptom, indicating that the program does not satisfy the (partial) specification given because the predicate `partition` will not be called in the right way. At this point, (possibly abstract) diagnosis should start in order to detect the cause of the error. The obvious thing to do is to check the calls to `partition` and inspect its arguments. By doing this, the user could easily detect that in the definition of `qsort`, `partition` is called with the second and third arguments reversed. By correcting this bug we obtain the following definition of `qsort`:

```
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R2,[X|R1],R).
qsort([],[]).
```

With this new version of the program, we proceed to perform compile-time checking of the assertions once more. This time, the obtained status of assertions is as follows:

```
:- checked calls qsort(A,B) : list(A,num).                        % A1
:- check success qsort(A,B)  => sorted_num_list(B).               % A2
:- checked calls partition(A,B,C,D) : ground(A),ground(B).        % A3
:- checked success partition(A,B,C,D) => ( list(C, num), ground(D) ).% A4
```

No assertion is now detected to be false. Thus, we cannot conclude that the specification does not hold. Moreover, assertions A1, A3, and A4 have been detected to hold in the program. However, A2 has not been statically proved. We can see that it has been simplified, and this is because the mode analysis has determined that on success the second argument of `qsort` is ground, and thus this does not have to be checked at run-time. On the other hand the analyses used in our session (types and modes) do not provide enough information to prove that the output of `qsort` is a *sorted* list of numbers. While this property could be captured by including a more refined domain such as constrained types, it is interesting to see what happens with the analyses selected.[12]

Assuming that we stay with the analyses selected previously, the following step in the development process is to compile the program obtained above with the "generate run-time checks" option. In the current implementation of `ciaopp` we obtain the following code for predicate `qsort` (the code for `partition` and `append` remain the same as there is no other assertion left to check):

```
qsort(A,B) :-
```

[12] Whether the property `sorted_num_list` holds in A2 is undecidable with only (over approximations) of mode and regular type information. However, it may be possible to prove that it does *not* hold (another example of how properties which are not natively understood by the analysis can also be useful for detecting bugs at compile-time): while the regular type analysis cannot capture perfectly the property `sorted_num_list`, it can still approximate it (by analyzing the definition) as `list(B, num)`. If type analysis for the program were to generate a type for B not compatible with `list(B, num)`, then a definite error symptom could be detected.

```
        'qsort/int'(A,B),
        postc([ qsort(C,D) : true => sorted(D) ], qsort(A,B)).

'qsort/int'([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R2,[X|R1],R).
'qsort/int'([],[]).

sorted_num_list([]).
sorted_num_list([_]).
sorted_num_list([X,Y|Z]):-
        X<Y, sorted_num_list([Y|Z]).
```

where `postc` is the built-in predicate in charge of checking postconditions of predicates – it accepts a list of assertions whose postcondition must be checked. The reason for using this predicate instead of `check` (which only receives the postcondition as argument), introduced in Section 5, is that if an error is detected, a more informative message can be printed than if only the postcondition responsible for the error is available.

Note also that the definition of predicate `sorted_num_list` has been optimized by the *abstract specializer* [33, 34] by eliminating the `number` tests. This is possible by taking advantage of type analysis which tells us that on success the second argument of `qsort` is a list of numbers.[13]

If we run the program with run-time checks in order to sort, say, list `[1,2]`, the CIAO system generates the following error message:

```
?- qsort([1,2],L).
ERROR: false success assertion for Goal qsort([1,2],[2,1])
Precondition: true  holds, but
Postcondition: sorted_num_list([2,1]) does not hold.

L = [2,1] ?
```

By observing this error message one can easily realize that there is some problem with `qsort`, as `[2,1]` is not the result of ordering `[1,2]` in ascending order. This is a (now, run-time) incorrectness symptom, which can be used as the starting point of diagnosis, using the previously mentioned (abstract) declarative diagnosis techniques, standard debugging, etc. The result of such diagnosis should indicate that the call to `append` is the cause of the error and that the right definition of predicate `qsort` is the one in Figure 2.

[13]Note that the availability of the abstract specializer allows an alternative implementation of the whole framework (also using both compile-time and run-time checking of assertions) by first generating in a naive way a program which performs run-time checking of all assertions and then applying the abstract specializer to this program. The resulting code would be similar to that obtained with the previous approach (first simplifying the assertions in a specialized module and then generating code for those which cannot be statically proved). `checked` assertions will result in run-time tests that are optimized away, false assertions will result in run-time tests that are transformed to `error`, etc. However, we have opted for the first alternative because we have found that it is easier for the user to understand things in terms of simplified assertions rather by looking at the run-time tests which remain in transformed code.

		Types			*Modes*			*Aliasing*		
Prog	**Ps**	**Props**	**Infer**	**Simp**	**Props**	**Infer**	**Simp**	**Props**	**Infer**	**Simp**
ann	66	514	9.64	0.55	265	1.60	1.22	419	2.22	6.57
palin	6	28	0.56	0.19	15	0.18	0.02	22	0.21	0.02
progeom	10	58	0.70	0.65	56	0.08	0.06	57	0.06	0.06
queen	6	28	0.23	0.09	26	0.05	0.03	28	0.04	0.04
warplan	31	132	8.33	0.12	71	1.83	0.07	98	2.35	0.10

Figure 4: Analysis/Checking Performance

8 A Preliminary Experimental Evaluation

The actual evaluation of the practical benefits of these tools is beyond the scope of this paper, but we are encouraged by our own experiences with the system (and the significant industrial interest in the prototype shown). It has certainly been observed during use by the system developers and a few early users that the environment can indeed detect some bugs much earlier in the program development process than with any previously available tools. Interestingly, this has been observed even when *no specifications (no assertions) are present in the user program.* This is due to the fact that the library modules (such as those defining the usual system built-ins and standard libraries) often do include a rich set of assertions for the (exported) predicates defined in these modules. As a result, symptoms in user programs are often flagged during compilation simply because the analyzer/comparator pair detects that assertions for the system library predicates are violated by program predicates.

It is also not our current purpose to perform a detailed evaluation of the *performance* of the system. However, preliminary results also show that performance is reasonable. Figure 4 presents results for `ciaopp`, inferring *types* (using Gallagher's type analyzer [20]), *modes* (using a variant of the Sharing+Freeness domain [27]), and *variable aliasing* (using the standard Sharing+Freeness). Analysis times are relatively well understood for these domains. The assertion processing time (normalization, simplification, etc.) obviously depends on the number of assertions in the input program. Given the lack at this point of a standardized set of benchmarks including assertions, for our preliminary evaluation we have opted for a simple (and with obvious drawbacks, but at least repeatable) method of generating programs with assertions automatically: previous to our measurements, we have run the analyzer on each program, producing a program annotated with `true` assertions (which express the analysis results) for each predicate. We have then rewritten such assertions into `check` assertions, and used the resulting program again as input to the system. **Prog** is the program being debugged and **Ps** the number of predicates, and, thus, of assertions (analysis variants were collapsed into one per predicate) in the program. **Props** is the number of properties which appear in the program assertions. **Infer** the analysis time, and **Simp** the time taken by the comparator to simplify the input assertions. These times are relative to the time taken by the a standard Prolog compiler (the SICStus compiler, in this case) to compile the program without assertions. For example, a 2 for **Infer** means that analysis time is twice a normal Prolog compiler time for the benchmark.

Clearly, in our case all assertions should be proven to be checked statically (and, indeed `ciaopp` does so). Figure 5 provides some data on the run-time cost of the assertions eliminated. It shows the slowdowns incurred when running the programs with the assertions relative to the running times of the original programs without assertions. **Prog** and **Props** are as before. Obviously, in our stylized case, when running the programs with assertions through `ciaopp` no

	With Run-time Checks					
	Types		*Modes*		*Aliasing*	
Prog	**Props**	**Slowdown**	**Props**	**Slowdown**	**Props**	**Slowdown**
ann	514	2.95	265	3.55	419	3.50
palin	28	15.0	15	6.00	22	9.00
progeom	58	104	56	65.0	57	66.0
queen	28	6.10	26	6.10	28	6.10
warplan	132	190	71	151	98	177

Figure 5: Run-Time Checking Cost

slowdowns occur, since all run-time checks are eliminated.

Again, the purpose of presenting these results is just to give a flavor for the behavior of the system. Clearly, the results should be contrasted with those obtained in an exhaustive evaluation, using more realistic, user provided assertions, which is left as future work.

9 Discussion

Software development is a difficult and error-prone task. Automatic tools for aiding in validation and debugging of programs are of great importance, especially those which allow finding problems at compile-time. Type checking is without a doubt one of the most successful techniques for compile-time bug detection. Type systems can be regarded as simple assertion-based frameworks with a limited property language. These properties (i.e., the types) are defined using a restricted syntax which (in our terms) guarantees that the resulting expression is natively understood by the analyzer (generally just a checker, see below). In traditional strongly typed languages, type declarations must exist for each procedure and each declaration must be as accurate as possible. Then, an efficient type checking algorithm is used. If type checking succeeds, then the program is guaranteed to be type-correct. This avoids the need for run-time checking. The type checking algorithm is typically (quasi) decidable in the sense that if the program is type-correct then the algorithm is able to prove it. Thus, the traditional approach is to reject those programs which do not pass the type check as they are (almost surely) incorrect w.r.t. the given type declarations.

In spite of the above mentioned benefits of strongly typed systems, there are many situations in which such a framework is too restrictive. Examples of this are when we do not wish to impose having assertions (e.g., type declarations) for all predicates (which would be unnatural for untyped languages), when the assertions are not as accurate as possible (for example, only some arguments are described), or, even more importantly, when we are interested in properties which are more general than types but for which we may not have a complete algorithm for checking them at compile-time. Nowadays, more and more powerful static analyzers are available which are capable of inferring non-trivial properties about programs, but which fall in the above category in that, unlike (traditional) types, these properties are in general not completely decidable at compile-time. Thus, such analyzers can only perform a safe approximation, i.e., if analysis concludes that the property holds, then it actually holds. However, analysis may not be able to conclude that certain property holds when it indeed holds, even if it understands this property "natively."

One of the main motivations for the framework we propose is to help automate as much as possible the validation and debugging of programs w.r.t. properties which lay out of traditional type-systems. Unless we do so, we cannot use in an automatic way the results offered of the large number of existing and very powerful analyzers which "only" approximate properties. Also, we believe that the approach we propose is arguably more suitable as an extension to untyped languages, such as Prolog and many instantiations of the CLP scheme.

Once we lift the requirement that properties be statically decidable we open up a different design space beyond that of classical type systems which offers much more flexibility than traditional strong type systems: assertions are optional, the user can define new properties, and the approach can deal with properties which type systems simply cannot handle. In order to achieve this, the framework has to correctly deal throughout with approximations. This extension is done knowingly at the expense of completeness, in the sense that there may be cases in which the program is correct w.r.t. the (partial) specification but we may not be able to prove it statically. However, this loss of completeness only occurs for the more general cases, since the traditional "complete" cases (such as decidable type systems) also fall within the framework (in the form of a particular abstract domain).

References

1. A. Aggoun, F. Benhamou, F. Bueno, M. Carro, P. Deransart, W. Drabent, G. Ferrand, F. Goualard, M. Hermenegildo, C. Lai, J.Lloyd, J. Maluszynski, G. Puebla, and A. Tessier. CP Debugging Tools: Clarification of Functionalities and Selection of the Tools. Technical Report D.WP1.1.M1.1-2, DISCIPL Project, June 1997.
2. K. Apt, editor. *From Logic Programming to Prolog*. Prentice-Hall, Hemel Hempstead, Hertfordshire, England, 1997.
3. 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.
4. K. R. Apt and D. Pedreschi. Reasoning about termination of pure PROLOG programs. *Information and Computation*, 1(106):109–157, 1993.
5. F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.
6. J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linkoping, Sweden, May 1997. U. of Linkoping Press.
7. F. Bueno. The CIAO Multiparadigm Compiler: A User's Manual. Technical Report CLIP8/95.0, Facultad de Informática, UPM, June 1995.
8. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.
9. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 1998. In Press.
10. F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linkoping, Sweden, May 1997. U. of Linkoping Press.
11. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
12. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science, pages 22–50. Springer-Verlag, 1996.
13. M. Comini, G. Levi, and G. Vitiello. Abstract debugging of logic programs. In L. Fribourg and F. Turini, editors, *Proc. Logic Program Synthesis and Transformation and Metaprogramming in Logic 1994*, volume 883 of *Lecture Notes in Computer Science*, pages 440–450, Berlin, 1994. Springer-Verlag.
14. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
15. P.W. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
16. P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.

17. W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581, 1988.

18. 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.

19. G. Ferrand. Error diagnosis in logic programming. *J. Logic Programming*, 4:177–198, 1987.

20. J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

21. M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, 1996.

22. M. Hermenegildo and The CLIP Group. Programming with Global Analysis. In *Proceedings of ILPS'97*, pages 49–52, Cambridge, MA, October 1997. MIT Press. (abstract of invited talk).

23. M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

24. P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.

25. A. Kelly, A. Macdonald, K. Marriott, P. Stuckey, and R. Yap. Effectiveness of optimizing compilation for CLP(R). In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pages 37–51. MIT Press, 1996.

26. K. Marriott and P. Stuckey. The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *19th. Annual ACM Conf. on Principles of Programming Languages*. ACM, 1992.

27. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

28. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992. Originally published as Technical Report FIM 59.1/IA/90, Computer Science Dept, Universidad Politecnica de Madrid, Spain, August 1990.

29. L. Naish. A three-valued declarative debugging scheme. In *8th Workshop on Logic Programming Environments*, July 1997. ICLP Post-Conference Workshop.

30. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997.

31. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. Technical Report CLIP2/97.1, Facultad de Informática, UPM, July 1997.

32. G. Puebla, F. Bueno, and M. Hermenegildo. A Framework for Assertion-based Debugging in Constraint Logic Programming. In *Proceedings of the JICSLP'98 Workshop on Types for CLP*, Manchester, UK, June 1998.

33. G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.

34. G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *Journal of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 1999. To appear.

35. E. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.

36. 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.

37. P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

38. E. Vetillard. *Utilisation de Declarations en Programmation Logique avec Constraintes*. PhD thesis, U. of Aix-Marseilles II, 1994.

39. R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

40. E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Concurrent Prolog: Collected Papers*, pages 211–244, 1987.