Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework

Edison Mera[1] Pedro Lopez-García[2,3] Manuel Hermenegildo[2,4]
edison@fdi.ucm.es pedro.lopez@imdea.org herme@fi.upm.es

[1] Complutense University of Madrid (UCM), Spain
[2] IMDEA Software, Spain
[3] Spanish Research Council (CSIC), Spain
[4] School of Computer Science, Technical University of Madrid (UPM), Spain.

Abstract. We present a framework that unifies unit testing and run-time verification (as well as static verification and static debugging). A key contribution of our overall approach is that we preserve the use of a unified assertion language for all of these tasks. We first describe a method for compiling run-time checks for (parts of) assertions which cannot be verified at compile-time via program transformation. This transformation allows checking preconditions and postconditions, including conditional postconditions, properties at arbitrary program points, and certain computational properties. Most importantly, we propose a minimal addition to the assertion language which allows defining unit tests to be run in order to detect possible violations of the (partial) specifications expressed by the assertions. We have implemented the framework within the Ciao/CiaoPP system and effectively applied it to the verification of ISO Prolog compliance and to the detection of different types of bugs in the Ciao system source code. Experimental results are presented that illustrate different trade-offs among program size, running time, or levels of verbosity of the messages shown to the user.

Key words: run-time verification, unit testing, static/dynamic debugging, assertions, program verification.

1 Introduction

We present an approach that unifies *unit testing* with *run-time verification* within an overall framework that also comprises *static verification* and *static debugging* [3, 7, 11, 12, 8]. This novel framework for program development is aimed at finding bugs in programs or validating them with respect to (partial) specifications given in terms of *assertions* (using the concept of *abstractions* as over-/under-approximations of program semantics). A novel and expressive language of assertions allows describing quite general program properties [10, 13, 4, 2].

The previous work in this context cited above has concentrated mostly on the static (i.e., compile-time) checking of such assertions as well as on techniques for reducing at compile-time the number of checks that have to be performed

dynamically (i.e., at run time): any assertions present in the program are verified (or falsified) to the extent possible during the compilation phase, since compile-time checking is always preferable to run-time checking –always incomplete as a means of verification. However the existence in all practical programs of data only known at run-time and the rich nature of the properties considered make a certain degree of run-time checking inevitable –a reasonable price to pay in return for property expressiveness.

In this paper we concentrate instead on the run-time portion of the model. Our aim is to a) develop effective implementation techniques for run-time checking that integrate seamlessly into our combined compile-time/run-time framework and b), based on this, to also develop *well-integrated* facilities for unit testing. To this end, we have first developed an implementation of run-time checks, as an evolution of the approach sketched in [12], based on transforming the program into a new one which preserves the semantics of the original program and at the same time checks during its execution the assertions. Such transformation allows checking preconditions and postconditions, including conditional postconditions, i.e., postconditions that must hold only when certain preconditions hold. It also allows checking properties at arbitrary program points (i.e., in literal positions in clause bodies) as well as certain computational properties (properties that are not specific to a program point but rather to whole computations, such as, for example, determinism, non-failure, or use of resources –steps, time, memory, etc.).

Our transformation also addresses to some extent one of the main drawbacks of run-time checking (in addition to incompleteness): the overhead introduced during execution of the program. The proposed transformation reduces run-time overhead by avoiding meta-interpretation whenever possible and by using special features of the low-level language when appropriate. Also, run-time checks can be compiled inline as opposed to calling a library, saving (meta-)call overhead. Another relevant issue addressed by our transformation is being able to provide messages to the user which are as informative as possible when a violation of the safety policy is found, i.e., when a run-time check fails. To this end, the transformation saves appropriate information at source code level in the transformed file. Depending on the level of code instrumentation selected, increasingly more accurate information about the assertions is saved, and, thus, presented, offering different trade-offs between information level and program size.

With respect to *testing*, we propose a minimal extension to the assertion language in order to be able to define *unit tests* [5]. The resulting language can express for example the input data for performing such unit tests, the expected output, the number of times that the unit tests should be repeated, etc. In contrast to previous work in this area (e.g., [1], [17], or the unit test framework recently included in SWI-Prolog [16]), a key contribution of our approach is that these unit tests blend in with the assertion language and reuse the overall framework. In particular, only *test drivers* need to be added because the assertions and their run-time tests act as the checkers for the cases defined by the unit tests. An advantage of our approach is that the unit test specifications can be

encapsulated in the same module that contains the predicates being tested, or placed in a separate file containing the tests for the module or modules of the application. This contrasts with, e.g., the `plunit` unit testing of SWI-Prolog, where unit test specifications are written in the source code of the module or in a dedicated file with the same name as the module being tested.

Both the run-time check generation and the unit testing approaches proposed have been implemented within the CiaoPP/Ciao system. We provide some experimental results which illustrate the implementation trade-offs involved. As mentioned before, thanks to the CiaoPP/Ciao machinery only the (parts of) assertions which cannot be verified at compile-time are converted into run-time checks. Since in our approach unit tests are also assertions, static analysis can also eliminate parts of or whole unit tests. At the same time, the tight integration also allows using the unit test drivers to exercise run-time checks corresponding to those parts of assertions that could not be checked at compile-time, even if they were not conceived as tests.

2 The Ciao Assertion Language

Assertions are linguistic constructions which allow expressing properties of programs. They allow talking about preconditions, (conditional) postconditions, whole executions, program points, etc. For space considerations, we will focus on a subset of the Ciao assertion language: assertions referring to *execution states* and *computations* (see [13, 2] for a detailed description of the full language). Also, although the assertion language incorporates significant syntactic sugar, we will use only the (unfortunately more verbose) raw forms. An execution state $\langle G \mid \theta \rangle$ consists of the current goal G and the current constraint store θ which contains information on the values of variables. By *computation* we mean the (sorted) execution tree containing all possible sequences of reductions between execution states of a goal from a calling state.

Predicate Assertions: They refer to properties of a particular predicate. In the schemas below a concrete assertion will include concrete values in place of *Pred*, *Precond* and *Postcond*. In all schemas *Pred* is a *predicate descriptor*, i.e., a predicate symbol applied to distinct free variables, and *Precond* and *Postcond* are logic formulas about execution states, that we call *state-formulae*. An atomic *state-formula* is constructed with a *state property predicate* (e.g., `list(X)` or `X > 3`) which expresses properties about (the values) of variables. A *state-formula* can also be a conjunction or disjunction of *state-formulae*. Standard (C)LP syntax is used, with comma representing conjunction (e.g., "`( list(X), list(Y) )`") and semicolon disjunction (e.g., "`( list(X) ; int(X) )`").

– *Describing success states:* `:- success` *Pred* [`:` *Precond*] `=>` *Postcond*`.`

Interpretation: in any call to *Pred*, if *Precond* succeeds in the calling state and the computation of the call succeeds, then *Postcond* should also succeed in the success state.

Example 1. The following assertion expresses that for any call to predicate `qsort/2` with the first argument bound to a list of numbers, if the call succeeds, then the second argument should also be bound to a list of numbers:

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

If *Precond* is omitted, the assertion is equivalent to:
`:- success` *Pred* `: true =>` *Postcond*.
and it is interpreted as "for any call to *Pred* which succeeds, *Postcond* should succeed in the success state."

– *Describing admissible calls:* `:- calls` *Pred* `:` *Precond*.
Interpretation: in all calls to *Pred*, the formula *Precond* should succeed in the calling state.

Example 2. The following assertion expresses that in all calls to predicate `qsort/2`, the first argument should be bound to a list of numbers:
`:- calls qsort(L,R) : list(L,num).`

The set of all `call` assertions is considered *closed* in the sense that they must cover all valid calls.

– *Describing properties of the computation:* `:- comp` *Pred* [`:` *Precond*] `+` *comp-formula*.
Interpretation: for any call to *Pred*, if *Precond* succeeds in the calling state, then *comp-formula* should also succeed for the computation of *Pred*.

Example 3. `:- comp qsort(L,R) : ( list(L,num), var(R) ) + not_fails.`
where the atom `not_fails` is implicitly interpreted as `not_fails(qsort(L,R))`, i.e., it is as if it executed $\langle qsort(L,R) \mid \theta \rangle$ and checked that it does not fail.

In addition, other assertion schemas such as `entry` and `exit` assertions can be used to refer to external calls to the module.[5]

Program-point assertions: The program points considered are the places in a program in which a new literal may be added, i.e., before the first literal (if any) of a clause, between two literals, and after the last literal (if any) of a clause. Program-point assertions are literals appearing at the corresponding program point and which are of the form: `check(`*state-formula* `).` The resulting assertion should be interpreted as "whenever computation reaches a state originated at the program point in which the assertion is, *state-formula* should succeed."

Status: Independently of the schema used, each assertion has a flag (`check`, `trust`, `true`, etc.), the assertion "status," which determines whether the assertion is to be checked, to be trusted, has already been proved correct by analysis, etc. Again for simplicity we use only the `check` status herein (which is assumed by default when no flag is present).

3 Run-Time Checking of Assertions

In this section we first focus on run-time checking of predicate assertions, and then we comment on the approach for program-point assertions. Our run-time checking system is composed of a set of transformations, to be performed by the

[5] Note that in CiaoPP the `pred` assertions of exported predicates can be used optionally instead of `entry` and `exit` assertions to define the module interface.

step one	step two
p :- *entry-checks*, *exit-preconditions-checks*, *ext-comp-checks*(**p1**), *exit-postconditions-checks*. % p renamed to p1 within module	**p1** :- *calls-checks*, *success-preconditions-checks*, *comp-checks*(*call_stack*(**p2**, *locator*)), *success-postconditions-checks*. **p2** :- body$_0$. ... **p2** :- body$_n$.

Fig. 1. The *transforming procedure definitions* scheme for run-time checking.

preprocessor, and a library containing a number of primitives that the transformed programs will call.

Applying the transformation that we call *transforming procedure definitions*,[6] the original predicate is rewritten so that it performs the run-time checks itself, each time it is called, and calls to it are left unchanged. Figure 1 illustrates this approach for a predicate `p`. In this transformation the original predicate `p` is renamed to `p2` and a new definition of `p`, which performs the run-time checks, is added by following two steps. "Step one" (first column of the figure) is used to add any run-time checks corresponding to, e.g., `entry` and `exit` assertions before and after a call to a new predicate `p1`. The objective of this first transformation is to separate external calls from internal ones. Then `p1` is defined so that it calls predicate `p2` and performs all run-time checks corresponding to each type of (kernel) predicate-level assertions, i.e., `calls`, `success`, or `comp` in the right place. In this kind of transformation, calls to `p` are left unchanged.

Transforming Single Predicate Assertions: We first consider the case where there is only one predicate assertion for a given predicate. We show schemes for transforming assertions into run-time checks for each type of (kernel) predicate assertion, i.e., `calls`, `success`, or `comp`. Other, higher-level assertions (such as `pred` assertions) and all additional syntactic sugar (such as modes or "star notation") are translated by the compiler into the kernel assertions before applying the transformation. These schemes express what run-time library predicates are called and where such calls are placed. Figure 2 shows the schemes, whereas the run-time library predicates are described below [7].

`checkc(`C`,`F`):` checks condition C and sets F to true or false depending on whether it succeeds or not. Defined as: `\+` C `->` F `= false ;` F `= true)`.

`rtcheck(`C`):` checks if condition C succeeds or not. If C fails, an exception is raised. This can be understood simply as `\+\+` C (so that bindings/constraints produced by the condition succeeding are removed –an *entailment* check).

`checkif(`F`,`P`):` postcondition P is checked if F is `true`. If P fails, an exception is raised. This can be defined as: `(`F `== true -> rtcheck(`P`) ; true)`.

[6] We refer the reader to [9] for a discussion of the trade-offs between the transformation described and an alternative one where the run-time checks are placed before and after any call to predicates affected by assertions.

[7] The schemas for `entry`/`exit` assertions are the same as the corresponding to `calls`/`success` assertions, and thus are not shown in the Figure.

Assertion:	The definition of *Pred* is transformed into:
`:- calls` *Pred* `:` *Cond*.	*Pred* `:- rtcheck(`*Cond*`),` *Pred'*. *Pred'* `:-` ... .
`:- success` *Pred* `:` *Precond* `=>` *Postcond*.	*Pred* `:- checkc(`*Precond*`,F),` *Pred'*, `checkif(F,`*Postcond*`).` *Pred'* `:-` ... .
`:- comp` *Pred* `+` *Comp*.	*Pred* `:- check_comp(`*Comp*`(G),G,`*Pred'*`).` *Pred'* `:-` ... .
`:- comp` *Pred* `:` *Precond* `+` *Comp*.	*Pred* `:- checkc(`*Precond*`,F),` `checkif_comp(F,`*Comp*`(G),G,`*Pred'*`).` *Pred'* `:-` ... .

Fig. 2. Translation schemes for different kinds of predicate assertions.

`checkif_comp(`F`,`$Comp$`(`G`),`G`,`$Pred'$`):` checks a computational property if F is *true*, for a given computational property $Comp(G)$, and a predicate *Pred'* to be checked. For example, if the property is `not_fails/1` and the predicate `qsort(A,B)`, then we call `checkif_comp(`F`,not_fails(`G`),`G`,qsort2(A,B))`. In turn, $Pred'$ is used to pass the direct call to the predicate (i.e., `qsort2(A,B)` in the example). If F is `false` then $Pred'$ is called, executing the procedure directly. If F is `true` then G is unified with $Pred'$ and $Comp(Pred')$ is called. This relies on the fact that `comp` properties are written assuming that the goal to be called is passed as an argument and that they take care of both running the procedure and checking whether the computational property holds. Again, if the (in this case, computational) property does not hold, an exception is raised. The predicate `checkif_comp/4` can be defined as:
`checkif_comp(fail, _, _,` *Pred*`):-  call(`*Pred*`).`
`checkif_comp(true,` *CompCall*`,` *Pred*`,` *Pred*`):-  call(`*CompCall*`).`

`check_comp(`$Comp(G)$`,`G`,`$Pred'$`):` a specialized version of `checkif_comp(true,` $Comp(G)$`,` G`,` $Pred'$`)`, where the first parameter is assumed to be true.

`call_stack(`C`,` L`):` adds the current source code locator L to the locator stack S allowing to show the call stack on run-time errors. This can be understood as: `intercept(`C`, rtc_error(`S`,`T`), throw(rtc_error([`L`|`S`],`T`)))`.

The previous library predicates are implemented in such a way that they perform the checks without modifying the program state, introducing side effects, errors, etc. In other words, if all run-time errors are intercepted, the semantics of the program must be preserved.

Combining Several Predicate Assertions: We now consider the case where there are several assertions for a given predicate. Translating several `calls` or `success` assertions is relatively straightforward: the corresponding `rtcheck/1` and `checkc/2` are placed before the call to *Pred'*, and any calls to `checkif/2` are gathered after it. In the case of `calls` assertions run-time check exceptions for the unsatisfied assertions are thrown only if *all* such checks fail.

Combining computational properties is somewhat more involved. First we consider the case of a single `comp` assertion with several properties, such as, e.g.:

`:- comp qsort(A,B) : (list(A, int), var(B)) + ( is_det, not_fails ).`
In this case the properties will simply be nested in the *Comp* field as follows: *prop1*`(`*prop2*`(` *... propN*`(`*Pred'*`)` *...* `))` (the *Pred'* field stays obviously the same). For example, for the assertion above the *Comp* field will be `not_fails(is_det(qsort_1(A,B)))`. If the `comp` property has a precondition, it will be checked only once and then either the *Comp* field or $Pred'$ will be called.

The situation is more complex when several `comp` assertions have to be combined. Consider for example the following two `comp` assertions:

```
:- comp qsort(A,B) : (ground(A), var(B)) + is_det.
:- comp qsort(A,B) : (list(A,int), var(B)) + not_fails.
```

Assuming that `F1` and `F2` are the flags resulting from checking the conditions `ground(A), var(B)` and `list(A,int), var(B)` respectively, the composition of the two assertions above would be:

```
checkif_comp(F2, not_fails(G2), G2,
    checkif_comp(F1,is_det(G1), G1, qsort2(A,B))).
```

After all the transformations explained above have been made, an invocation of `call_stack/2` is instrumented in order to save the locator in the stack.

Program-Point Assertions: This is a comparatively simpler task than transforming predicate-level assertions: only one program point needs to be transformed for each assertion; only the `rtcheck/1` and `check_comp/1` primitives are required; and in the case of computational properties; their definitions are called directly. Clauses are transformed as follows:

Program-point assertion:	The clause is transformed into:
Pred `:-` ..., `check(`*Cond*`),` ...	*Pred* `:-` ..., `rtcheck(`*Cond*`),` ...
Pred `:-` ..., `check(`*CompProp*`(`*Goal*`)),` ...	*Pred* `:-` ..., `check_comp(`*CompProp*`(`*Goal*`)),` ...

4 Defining Unit Tests

In order to define a unit test we have to express on one hand *what to execute* and on the other hand *what to check* (at run-time). A key characteristic of our approach is that we use the assertion language described in Section-2 for expressing what to check. This way, the same properties that can be expressed for static or run-time checking can also be checked in unit testing. However, we have added a minimal number of elements to the assertion language for expressing *what to execute*. In particular, we have added a new assertion schema:

`:- texec` *Pred* [`:` *Precond*] [`+`*Exec-Formula*]`.`

which states that we want to execute (as a test) a call to *Pred* with its variables instantiated to values that satisfy *Precond*. *Exec-Formula* is a conjunction of properties describing how to drive this execution. In our approach many of the properties usable in *Precond* (e.g., types) can be run as value generators for these variables, so that input data can be automatically generated for the unit tests (see the technique described in [6]). However, we have defined some specific properties, such as random value generators.

Example 4. The assertion:

```
:- texec append(A, B, C) : (A=[1,2],B=[3],var(C)).
```

expresses that a call to `append/3` with the first and second arguments bound to `[1,2]` and `[3]` respectively and the third one unbound should be executed.

Example 5. We can define a unit test using the assertion in Example 4 together with the following two assertions expressing *what to check at run-time*:

```
:- check success append(A,B,C):(A=[1,2],B=[3],var(C)) => C=[1,2,3].
:- check comp    append(A,B,C):(A=[1,2],B=[3],var(C)) + not_fails.
```

The success assertion states that if a call to `append/3` with the first and second arguments bound to `[1,2]` and `[3]` respectively and the third one unbound terminates with success, then the third argument should be bound to `[1,2,3]`. The comp assertion says that such a call will not fail. □

The advantage of the integrated framework that we propose is that the execution expressed by a `texec` assertion for unit testing can also be used for checking parts of other assertions that could not have been checked at compile-time and thus remain as run-time checks. This way, a single set of run-time checking machinery is used for both run-time checks and unit testing. In addition, static checking of assertions can safely avoid (possibly parts of) unit test execution.

We now introduce another predicate assertion schema, the `test` schema, which can be seen as syntactic sugar for a set of predicate assertions:

`:- test` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-Exec-Props*]`.`

This assertion is interpreted as the combination of three assertions:[8]

`:- texec` *Pred* [`:` *Precond*] [`+` *Exec-Props*]`.`
`:- check success` *Pred* [`:` *Precond*] [`=>` *Postcond*]`.`
`:- check comp` *Pred* [`:` *Precond*] [`+`*Comp-Props*]`.`

For example, the assertion:

```
:- test append(A,B,C) : (A=[1,2],B=[3],var(C)) => C=[1,2,3]
                      + (not_fails,times(5)).
```

is conceptually equivalent to that in Example 4, plus the two in Example 5.

These are examples of predefined properties that can be used in *Exec-Formula*:

`try_sols(N):` Expresses an upper bound `N` on the number of solutions to be checked. For example, the assertion:

```
:- texec append(A, B, C): (A=X, B=Y, C=Z) + try_sols(7).
```

expresses that the call to `append(X, Y, Z)` should be executed to get at most the first 7 solutions through backtracking.

`times(N):` Expresses that the execution should be repeated `N` times. For example, while checking ISO prolog compliance, a test for the `retract`/1 predicate failed rarely, so that the test was modified adding the primitive `times`/1:

```
:- test  retract_test7(A) + times(50).
retract_test7(A) :- retract((foo(A) :- A,call(A))).
```

in order to repeat the test fifty times to increase the chances of test failure.

[8] In fact, a completeness assertion –using "`<=`", see [13]– could also be generated.

`exception(Excep)`: Expresses that a test execution should throw the exception `Excep`. For example, consider the predicate `p`/1 defined as follows:

```
p(a).
p(b) :- fail.
p(c) :- throw(error(c, "error c")).
```

The following tests succeed:

```
:- test p(A) : (A = a) + not_fails.
:- test p(A) : (A = b) + fails.
:- test p(A) : (A = c) + exception(error(c,_)).
```

The first one states that the call `p(a)` should not fail, the second one that `p(b)` should fail, and the third one that `p(c)` should raise an exception.

`user_output(String)`: Expresses that a predicate should write the string `String` into the current output stream. For example, the following test involving the library predicate `display`/ 1 succeeds:

```
:- test display(A) : (A = hello) + user_output("hello").
```

However, the following tests report an error:

```
:- test display(A) : (A = hello) + user_output("bye").
:- test display(A) : (A = hello) + user_output("hello!").
```

Other properties are provided for example to express that a predicate should write the string `Str` into the current error stream (`user_error(Str)`), to express a time-out `T` for a test execution (`resource(ub, time, T)`), or to generate random input data with a given probability distribution (e.g., for floating point numbers, including special cases like *infinite*, *not-a-number* or *zero* with sign).

5 Generating User-friendly Messages

Whenever a run-time check fails, an exception is raised. An exception handler will then catch the exception and report the error. However, with the transformations presented so far little information can be provided to the user beyond the precondition or postcondition that is producing the violation, since this is the only parameter passed to most of the checking predicates. In contrast, during compile-time checking, when an assertion is proved not to hold, both the assertion and the program point where the assertion was violated are reported, in a format designed so that the graphical program development environment can locate these points in the source code and highlight them automatically.

In order to also provide precise information when reporting violated assertions when performing run-time checks, we have added an extra argument to the checking predicates through which certain information is passed, such as the location of the corresponding assertion(s) and the calling program point in the source code. This information can then be passed to the exception handler when the exception occurs, which prints it in a format that is compatible with that used when reporting compile-time checking errors. Thus, run-time errors can also be easily traced back to the sources automatically by the development

environment. The transformation instruments the transformed code to include the necessary information.

There is a clear trade-off between the size of and the overhead introduced in the instrumented program and the quality of the messages issued. Different levels of information may be appropriate for different contexts. The current implementation of the run-time check transformations offers several optional levels of instrumentation. For brevity we report on two levels in our experiments, explained below:

Low: information is saved to report the actual assertion being violated and the property or properties that caused such violation.

High: in addition, predicates with assertions are further instrumented so that when a run-time check fails a call stack dump is also shown up to the exact program point where the violation occurs, showing for each predicate the literal in its body that caused such violation.[9]

To illustrate these levels, consider the following assertion and property definitions, in addition to a definition of `qsort/2` such as that of Figure 3:

```
:- success qsort(A,B)  => (ground(B),sorted_num_list(B)).
:- prop sorted_num_list/1.
sorted_num_list([]).
sorted_num_list([X]):- num(X).
sorted_num_list([X,Y|Z]):- num(X),num(Y),X=<Y,sorted_num_list([Y|Z]).
```

which ensures that `qsort/2` always returns a ground, *sorted* list. Assume also that the program has been written in a buggy way (to be discovered later). With *low* instrumentation level the output during execution would be similar to:

```
?- qsort([1,2],X).
{In /tmp/qsort.pl
ERROR: (lns 8-9) Run-time check failure in assertion for:
       qsort:qsort([1,2],[2,1]).
In *success*, unsatisfied property:
       sorted_num_list([2,1]).
ERROR: (lns 16-21) Check failed in qsort:qsort/2.}
```

Two errors are reported for a single run-time check failure: the first error shows the actual assertion being violated and the second marks the first clause of the predicate which violates the assertion. However, not enough information is provided to determine which literal made the erroneous call. For the *high* instrumentation level transformation the output is:

```
?- call_rtc(qsort([3,1,2],B)).
{In /tmp/qsort.pl
ERROR: (lns 8-9) Run-time check failure in assertion for:
       qsort:qsort([1,2],[2,1]).
In *success*, unsatisfied property:
       sorted_num_list([2,1]).
ERROR: (lns 16-21) Check failed in qosrt:qsort/2.
```

[9] This can also be done at a lower level, via engine primitives, but we are interested herein in measuring only the cost of source level transformations.

```
:- calls   qsort(A,B) :  list(A,num).
:- success qsort(A,B) :  list(A,num) => list(B,num).
:- comp    qsort(A,B) : (list(A,num), var(B)) + not_fails.

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

:- calls   partition(A,B,C,D) : (list(A), num(B)).
:- success partition(A,B,C,D) : (list(A), num(B)) => (list(C), list(D)).
:- comp    partition(A,B,C,D) : (list(A), num(B)) +  (not_fails,is_det).

partition([],B,[],[]).
partition([E|R],C,[E|Left1],Right):- E < C, !, partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):- partition(R,C,Left,Right1).
```

Fig. 3. A quick-sort program with assertions.

```
ERROR: (lns 16-21) Check failed when invocation of
       qsort:qsort([3,1,2],_1)
       called qsort:qsort([1,2],_2) in its body.}
{In /tmp/qsort.pl
ERROR: (lns 8-9) Run-time check failure in assertion for:
       qsort:qsort([3,1,2],[3,2,1]).
In *success*, unsatisfied property:
       sorted_num_list([3,2,1]).
ERROR: (lns 16-21) Check failed in qsort:qsort/2.}
```

This example uses the `call_rtc`/1 meta-predicate to intercept the run-time error, show the related message, and continue execution as if the program where not being checked. The output makes it easier to locate the error since the call stack dump provides the list of calling predicates being checked.

Note that the first part of the assertion is not violated, since `B` is ground. However, on success the output of `qsort/2` is a sorted list but in reverse order, which gives us a hint: the arguments in the call to `append/3` are mistakenly swapped.

6 Experimental Results

We now report on some experimental results from our implementation within the Ciao/CiaoPP system of the testing and run-time checking approach proposed. Both have been integrated fully into the development environment allowing easy execution of tests and run-time checking of assertions present in modules. The system is available in the latest Ciao betas (1.13.x) at `http://www.ciaohome.org`. The experiments measure both program size and time overhead due to run-time checks. We first used the qsort program in Figure 3, with an input list of size 600 to run several experiments for different settings:

– **Library or inlined run-time checks:** we have implemented the transformation first as described in the previous sections, where the `check` predicates

Qsort	Low						High					
Obj Size:	Inline			Library			Inline			Library		
7467 (bytes)	M	T	M+T	M	T	M+T	M	T	M+T	M	T	M+T
Entry	1.41	1.69	1.77	1.34	1.38	1.44	1.66	1.94	2.02	1.57	1.61	1.68
Exit	1.55	1.82	1.97	1.28	1.33	1.44	1.78	2.06	2.21	1.50	1.55	1.65
Comp*	1.67	1.89	1.93	5.46	5.49	5.54	2.05	2.28	2.31	5.64	5.68	5.73
E/E/C	2.32	2.67	2.88	5.88	5.95	6.11	2.88	3.23	3.44	6.25	6.31	6.48
Calls	1.42	1.64	1.75	1.32	1.33	1.43	1.62	1.84	1.95	1.50	1.51	1.61
Success	1.55	1.77	1.92	1.26	1.29	1.39	1.74	1.97	2.12	1.42	1.44	1.55
Comp	1.63	1.85	1.88	5.38	5.41	5.46	2.01	2.24	2.28	5.57	5.60	5.65
C/S/C	2.10	2.46	2.65	5.66	5.73	5.88	2.63	3.00	3.20	5.98	6.11	6.26

Table 1. Qsort size increment with several configurations of run-time checks.

Qsort	Low						High					
exec time:	Inline			Library			Inline			Library		
675 (us)	M	T	M+T	M	T	M+T	M	T	M+T	M	T	M+T
Entry	1.00	1.86	1.87	1.05	1.89	1.90	1.01	1.89	1.87	1.03	1.91	1.91
Exit	1.02	2.73	2.73	1.03	2.76	2.78	1.02	2.74	2.75	1.03	2.79	2.80
Comp*	1.01	1.87	1.87	1.02	1.93	1.92	1.02	1.88	1.90	1.05	1.91	1.92
E/E/C	1.01	3.60	3.60	1.04	3.67	3.68	1.02	3.62	3.65	1.05	3.69	3.69
Calls	3.52	165	162	76	243	321	42	207	205	135	301	382
Success	5.62	329	333	164	515	667	42	380	383	229	595	746
Comp	6.39	166	167	106	272	343	82	254	254	264	447	512
C/S/C	9.77	352	353	194	578	761	91	450	453	379	776	948

Table 2. Slowdown of `qsort/2` with several configurations of run-time checks.

are assumed to be in a library (columns labeled **Library**). Ratios shown are w.r.t. the execution time of the program with no run-time checks. In addition, an alternative approach has been implemented in which the definitions of the run-time check library predicates are actually *inlined* in the calling program. This often achieves better performance but sometimes at the cost of increased code size. Note, however, that code size does not increase in all cases because such inlining is, in fact, a restricted kind of partial evaluation that tries to solve as many unifications as possible at compilation time, and sometimes terms become smaller after such optimization.

– **Use of types or modes properties:** since checking complex types, such as in the `list(int)` check, which needs to traverse lists of integers over and over again,[10] is more expensive than checking modes (which in our case is handled through a call to the `var/1` ISO Prolog builtin) we have separated these cases in the experiments. In columns labeled T and M only types or

[10] This overhead can be significantly reduced via multiple specialization [15, 14]. However, that optimization has not been applied in this case in order to measure the overhead of fully checking the assertion.

App Name	Source Metrics				Compiled		Run-Time Checked (ratio)			
	Size		Assertions		Binary		Low		High	
	Lines		Modules		Object		Inline	Library	Inline	Library
Ciao	S	4018	A	3062	B	2881	1.34	1.39	1.47	1.48
	L	121305	M	610	O	6660	2.78	2.73	2.93	2.85
CiaoPP	S	4819	A	1131	B	13073	1.15	1.17	1.20	1.21
	L	152536	M	517	O	12868	1.28	1.28	1.33	1.32
LPdoc	S	316	A	105	B	5052	1.22	1.23	1.33	1.29
	L	8810	M	8	O	736	1.18	1.07	1.23	1.12

Table 3. Size (in kilobytes) of binary and object files using several instrumentation levels of run-time checks, for large benchmarks.

modes are checked respectively, whereas in columns labeled $M{+}T$ both types and modes are checked.

- **Low or high instrumentation:** as defined in Section 5.
- **Using several kinds of assertions:** several combinations of different kinds of assertions have been tested (first column).

Tables 1 and 2 present the overhead, in size and time respectively, for the experiments expressed as a ratio w.r.t. the execution of the program with run-time checks disabled. Execution was on a MacBook Pro, Intel Core 2 Duo at 2.4Ghz, 2GB of RAM, Ubuntu Linux 8.10 and Ciao version 1.13. The columns in the tables present combinations of the configurations explained above. The rows show results for different kinds of assertions. For `comp` assertions we have that in **Comp*** the check is performed only at the entry point of the module, but not for the internal calls that occur inside.

The results show that the *high* level of instrumentation is quite expensive while the overhead implied by the *low* level is better, specially in the case of inlining. This confirms our expectations. The high overhead implied by the *high* level of instrumentation is due in part to the simplistic way in which this type of instrumentation is implemented for these experiments. Note also that the values of the `Library` column are quite large when compared with the ones of the `Inline` column because the inline transformation avoids metacalls.

Table 3 shows experimental results for larger programs, namely, the Ciao, CiaoPP, and LPdoc systems (including the libraries they use), all of which contain numerous assertions in their code. It shows the size (in kilobytes) of binary and object files using several instrumentation levels of run-time checks. The binary refers to the statically-linked executable of the main module of such systems which corresponds to the command-line executable. The object files include all the libraries used by such systems. Note that in all cases the sizes of the files depend on the number of assertions instrumented for run-time checking. Interestingly, the impact of run-time tests on execution time in these much larger benchmarks is much smaller than for qsort. For example, the overhead introduced in the execution of LPdoc, which includes a good number of assertions in its source, is in practice below the measurement noise level.

Regarding unit tests, we have added at the time of writing 220 unit tests to the Ciao/CiaoPP system (in addition to the other traditional system tests which did not use the unit test framework). These tests have been effective in detecting some errors introduced in those modules during later code changes. The execution time of such tests is approximately 90 seconds in the computer described before. We also have applied the implemented framework to the verification of ISO Prolog compliance of Ciao. We have coded 976 unit tests for this purpose. These allowed the detection of a large number of previously unknown limitations and errors: 262 issues related to non-compliance with the standard, 90 related to missing predicates or functionality, and 39 related to bugs in the functionality. While a large number of these were repetitions of a few individual errors they have been nevertheless very useful. These tests currently run in under 15 seconds. This time is much less than the other tests for Ciao because they are concentrated in only one file and the driver does not need to scan all the source code. Note that in these experiments we are not doing any compile-time checking, which would in fact eliminate many of the unit tests.

7 Conclusions

We have described our design and implementation of a framework that unifies unit testing and run-time verification (as well as static verification and static debugging). A key contribution of our approach is that a unified assertion language is used for all of these tasks. We have proposed methods for compiling run-time checks for (parts of) assertions which cannot be verified at compile-time via program transformation. We have also proposed a minimal addition to the assertion language which allows defining unit tests to be run in order to detect possible violations of the (partial) specifications expressed by the assertions. We have implemented the framework within the Ciao/CiaoPP system and presented some experimental results to illustrate different trade-offs among program size, running time, or levels of verbosity of the messages shown to the user. The experimental results confirm our expectations regarding these trade-offs: run-time checks do not pose an excessive amount of overhead when low levels of instrumentation are introduced and the calls to library predicates are inlined. The tests and run-time checks are proving quite useful in practice for detecting bugs.

Acknowledgments: This work was funded in part by EU projects FET IST-15905 *MOBIUS*, IST-215483 *SCUBE*, FET IST-231620 *HATS*, and 06042-ESPASS, Ministry of Science projects TIN-2008-05624 *DOVES*, Ministry of Industry project FIT-340005-2007-14, and CAM project S-0505/TIC/0407 *PROMESAS*.

References

1. F. Belli and O. Jack. Implementation-based Analysis and Testing of Prolog Programs. In *ISSTA '93: Proc. of the ACM SIGSOFT Int'l. Symp. on Software Testing and Analysis*, pages 70–80, New York, NY, USA, 1993. ACM.

2. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.
3. 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 WS on Automated Debugging–AADEBUG*, pages 155–170. U. Linköping Press, May 1997.
4. The CLIP Group. Program Assertions. The Ciao System Documentation Series – TR CLIP4/97.1, Facultad de Informática, UPM, August 1997.
5. N. S. Eickelmann and D. J. Richardson. An Evaluation of Software Test Environment Architectures. In *ICSE '96: Proc. of the Int'l. Conf. on Software Engineering*, pages 353–364. IEEE Computer Society, 1996.
6. M. Gómez-Zamalloa, E. Albert, and G. Puebla. On the Generation of Test Data for Prolog by Partial Evaluation. In *Workshop on Logic-based methods in Programming Environments (WLPE'08)*, volume WLPE/2008/06, pages 26–43, 2008.
7. M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, 1999.
8. M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Comp. Progr.*, 58(1–2), 2005.
9. E. Mera, P. López-García, and M. Hermenegildo. Towards Integrating Run-Time Checking and Software Testing in a Verification Framework. Technical Report CLIP1/2009.0, T. U. Madrid (UPM), March 2009.
10. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *ILPS'97 WS on Tools and Environments for (C)LP*, October 1997. `ftp://clip.dia.fi.upm.es/pub/papers-/assert_lang_tr_discipldeliv.ps.gz`.
11. G. Puebla, F. Bueno, and M. Hermenegildo. A Framework for Assertion-based Debugging in Constraint Logic Programming. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, Venezia, Italy, September 1999.
12. G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer, September 2000.
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 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.
15. G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *JLP*, 41(2&3):279–316, November 1999.
16. J. Wielemaker. SWI Prolog Unit Tests. `http://www.swi-prolog.org/pldoc/package/plunit.html`.
17. L. Zhao, T. Gu, J. Qian, and G. Cai. Test Frame Updating in CPM Testing of Prolog Programs. *Software Quality Control*, 16(2):277–298, 2008.