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 have designed and implemented 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 first propose methods 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. The implemented transformation includes several optimizations to reduce run-time overhead. We also 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. This language can express for example the input data for performing the unit tests or the number of times that the unit tests should be repeated. 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. Several 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.

Keywords: dynamic verification, unit testing, static/dynamic debugging, assertions.

1 Introduction

We present a framework (and its implementation) that unifies *unit testing* and *run-time verification* (as well as *static verification* and *static debugging*). Our approach builds on [BDD+97,HPB99,PBH00a,HPBLG05], where an approach to program development has been designed and implemented whose objective is to on one hand validate and on the other find bugs in programs with respect to specifications that are given in terms of *assertions*. The approach is based on a novel and expressive language of assertions for describing safety policies and, in general, very general program properties [PBH97,PBH00b][CLI97,BCC+06]. We have also proposed strategies for static (i.e., compile-time) checking of such policies as well as techniques for reducing at compile-time, using information from static analysis, the number of checks that have to be done dynamically (i.e., at run time) [PBH98,PBH99,HPBLG05]. Using these techniques, any

assertions present in the program are falsified or verified as completely as possible during the compilation phase, since compile-time checking is always preferable to run-time, which is necessarily always incomplete as a means of verification. However the existence in all practical programs of parameters and data only known at run-time and the rich nature of the properties that we are interested in determine that a certain degree of run-time checking is inevitable. In return the approach allows using very expressive safety policies with reduced overhead.

While the static checking part of this model has been the subject of considerable work, in this paper we shift to the actual run-time checking of safety policies, which has received little previous attention. 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) to also develop integrated facilities for unit testing. To this end, we have first developed an implementation of run-time checks based on transforming the program into a new one which at the same time preserves the semantics of the original program and also checks during its execution the assertions present in it, and thus the safety policy. The 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., between any two literals in a body clause) as well as checking certain computational properties, i.e., 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, which introduces overhead due to additional (meta-)calls.

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 will be saved, and, thus, presented, offering different trade-offs between information level and program size.

With respect to the closely related subject of testing, we require only a minimal extension to the assertion language in order to be able to define *unit tests* [ER96]. 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., [BJ93], [ZGQC08], or the unit-test framework recently included in SWI-Prolog), a key contribution of our approach is that these unit tests blend in with and reuse the assertion language and 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.

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. The integration with the CiaoPP/Ciao compile-time checking allows reducing run-time overhead to checking only those aspects of the safety policy that could not be determined statically. I.e., only

the checks in assertions (including "tests") which cannot be verified at compile-time are converted into run-time checks. Note that since in our approach unit tests are also assertions, static analysis this also eliminates parts of or whole unit tests which may have been verified statically. 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. Finally, properties inferred by static analysis (e.g., types) can also be used for automatically generating input data for the unit tests (see [GZAP08] for a technique for this purpose).

2 The Ciao Assertion Language

Assertions are linguistic constructions which allow expressing properties of programs. In the Ciao assertion language, assertions are always instances of some *assertion schema*. Such schemas allow talking about preconditions, (conditional) postconditions, whole executions, program points, etc. Each schema in turn contains one or two logic formulae which are (intuitively) used to say things such as "X is a list of integers," "Y is ground," "$p(X)$ does not fail," etc. In this approach the user has a high degree of freedom for defining these logic formulae for the properties considered of interest.

For space considerations, we will focus on a subset of the Ciao assertion language (see [PBH00b] for a detailed description of the full language). In particular, although the language has assertions specifically designed for expressing properties related to the declarative semantics, in this paper we will focus on the *operational* semantics of programs. Also, although the assertion language incorporates significant syntactic sugar, we will use only the (unfortunately more verbose) raw forms.

The assertions refer to *execution states*. An execution state $\langle G \mid \theta \rangle$ consists of the current goal G and the current constraint store (or *store* for short) θ which contains information on the values of variables. The operational semantics is given in terms of *derivations*, which are sequences of reductions between such execution states. By *computation* we mean the (sorted) execution tree containing all possible derivations of a goal from a calling state. The rules for the grammar describing the assertion language considered (including the extensions that will be described later) are listed in Fig. 1.

Predicate assertions: They refer to properties of a particular predicate. Given the schemas below, a concrete assertion will include concrete properties in place of the symbols *Pred*, *Precond* and *Postcond*. In all schemas *Pred* is a *predicate descriptor*, i.e., a predicate symbol as main functor and all arguments are distinct free variables (*pred-desc* in the grammar shown in Fig. 1), and *Precond* and *Postcond* are logic formulas about execution states, represented with the non-terminal symbol *state-formula* in the grammar. An atomic *state-formula* is a *State-prop* constructed with a *state property predicate* (e.g., `list(X)` or `X > 3`) which expresses properties about (the values) of the variables. A *state-formula* can also be a conjunction or disjunction of *state-formulae*. Standard (C)LP syntax is used, so that the comma should be interpreted as conjunction (e.g., "`( list(X), list(Y) )`"), and the semicolon as disjunction (e.g., "`( list(X) ; int(X) )`").

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

Interpretation: in any invocation of *Pred* if *Precond* holds in the calling state and the computation succeeds, then *Postcond* should also hold in the success state.

Example 1.

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

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

– *Describing admissible calls:* `:- calls` *Pred* `:` *Precond*.
Interpretation: in all activations of *Pred* the formula *Precond* should hold 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:
```
:- calls qsort(L,R) : list(L).
```

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: in any activation of *Pred* if *Precond* holds in the calling state then *comp-formula* should also hold 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, `entry` and `exit` assertions are identical to `pred` assertions, except that they refer to external calls to the module (or predicate). 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).

Program-point assertions: The program points that we will consider 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*). where *state-formula* is a logic formula about execution states (see the grammar in Fig. 1). The resulting assertion should be interpreted as "whenever execution reaches a state originated at the program point in which the assertion is, *state-formula* should hold."

The logic formulae: We allow conjunctions and disjunctions in the formulae, and choose to write them down, for simplicity, in the usual CLP syntax. Thus, logic formulae about execution states can be:

– An atom of the form $p(t_1, \ldots, t_n)$ with $n \geq 0$, where p/n is a *property predicate* (e.g., `list(X)` or `X > 3`).
– An expression of the form ($F1$, $F2$) where $F1$ and $F2$ are logic formulae about execution states and, as usual in CLP, the comma should be interpreted as conjunction (e.g., "`( list(X), list(Y) )`").
– An expression of the form ($F1$; $F2$) where $F1$ and $F2$ are logic formulae about execution states and, as usual in CLP, the semicolon should be interpreted as disjunction (e.g., "`( list(X) ; int(X) )`").

program-assert	::=	`:-` *predicate-assert* `.` | *prog-point-assert*
predicate-assert	::=	*pred-assert* | *status pred-assert* | `entry` *pred-cond* | `exit` *pred-cond*
		| `exec` *pred-cond* `+` *exec-formula*
pred-assert	::=	`calls` *pred-cond* | `success` *pred-cond* `=>` *state-formula*
		| `comp` *pred-cond* `+` *comp-formula*
pred-cond	::=	*pred-desc* | *pred-desc* : *state-formula*
pred-desc	::=	*Pred-name*`(`*args*`)`
args	::=	*Var* | *Var*`,`*args*
state-formula	::=	`(`*state-formula* `,` *state-formula*`)` | `(`*state-formula* `;` *state-formula*`)`
		| `compat(`*State-prop*`)` | *State-prop*
comp-formula	::=	`(`*comp-formula* `,` *comp-formula*`)` | `(`*comp-formula* `;` *comp-formula*`)`
		| *Comp-prop*
exec-formula	::=	`(`*exec-formula* `,` *exec-formula*`)` | *Exec-prop*
status	::=	`check` | `true` | `checked` | `trust` | `false`
prog-point-assert	::=	*status*`(`*state-formula*`)`

Fig. 1. Syntax of the assertion language.

3 Run-Time Checking of Predicate Assertions

We start by discussing two possible approaches regarding the source-to-source transformations to be performed in order to implement run-time checking schemes.

In the first kind of transformation the run-time checks are placed before and after any call to predicates which are affected by assertions; let `p/2` be one such predicate. We will call this kind of transformation "transforming calls". In the second kind of transformation the original predicate is rewritten so that it performs the run-time checks itself, each time it is called. In this case only the definition of the procedure is modified (in the example the original `p` predicate is renamed to `p'` and a new definition of `p` is added which performs the run-time checks; calls to `p` are left unchanged). We will call this kind of transformation "transforming procedure definitions."

Clearly, each scheme has advantages and disadvantages, specially when considering a program consisting of several modules. When transforming calls, additional run-time checking code will be introduced in all modules that call the predicate which contains a given assertion. This will likely result in a larger code size than in the transforming procedure definitions approach, since a program can easily see a large number of assertions from, e.g., libraries. Also, if a given file containing an assertion is modified, all the modules using it will have to be recompiled. The big advantage of the transforming calls approach is that if no run-time assertion checking is required in a given module, only that module needs to be recompiled, whereas in the transforming procedure definitions approach all the modules containing procedures with run-time checks and which are used by the given module need recompilation. Thus, for libraries, in the transforming calls approach only one version of each file is compiled whereas in the transforming procedure definitions approach typically two versions of the libraries are kept in the system, one with run-time checks and the other one without. Both approaches allow mixing modules with and without run-time checks. Another potential advantage of the transforming calls approach is that it makes it easier for certain kinds of analysis and specialization algorithms (specially those which are not multivariant) to analyze and optimize programs annotated with run-time checks. On the other hand, if the analysis and specialization system is multivariant (as in the case of CiaoPP) this is less of an issue.

In view of all the advantages and disadvantages discussed in this work, we currently use the transforming procedure definitions approach. Figure 3 illustrates this approach

Step One	Step Two
pred :- *entry-checks*, *exit-checks (preconditions)*, **pred'**, *exit-checks (postconditions)*. %% rename pred by pred' %% inside the module	**pred'** :- *calls-checks*, *success-checks (preconditions)*, *comp-checks(call_stack(* **pred"**, *locator))*, *success-checks (postconditions)*. **pred"** :- $body_0$. ... **pred"** :- $body_n$.

Fig. 2. Transformation scheme for a predicate *pred*, predicate assertions.

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*`,`*Pred'*`)`. *Pred'* :- ... .
:- `comp` *Pred* : *Precond* `+` *Comp*.	*Pred* :- `checkc(`*Precond*`,F)`, `checkif_comp(F,`*Comp*`,`*Pred'*`)`. *Pred'* :- ... .

Fig. 3. Translation schemes for different kinds of predicate assertions.

for any assertion. Our run-time checking system is composed of a set of transformations, to be performed by the preprocessor, and a library containing a number of primitives that the transformed programs will call. Figure 3 presents schemes of how procedures are transformed in order to incorporate run-time checking, for each type of (kernel), predicate level assertions, 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) is translated by the compiler into the kernel assertions before applying the transformation. In the case of `entry` and `exit` assertions, a renaming technique is used inside the module to avoid checks in internal calls, as shown in the "Step One" column.

The run-time library includes the following predicates. These predicates can actually be used for both the transforming calls and transforming procedure definitions approaches.

`checkc(`C`,`F`)`: checks condition C and sets F to true or false depending on whether it succeeds or not. From a logical point of view this can be understood as:

$$(\backslash + C \texttt{ -> } F \texttt{ = false ; } F \texttt{ = true})$$

`rtcheck(`C`)`: checks if condition C succeeds or not. If C fails an exception is raised. From a logical point of view this can be understood simply as $\backslash + \backslash + C$.

`checkif(`F`,`P`)`: postcondition P is checked iff F is `true`. If P fails an exception is raised. From a logical point of view this can be understood as:

$$(F \texttt{ == true -> rtcheck(}P\texttt{) ; true}).$$

`rtcheck(`C`)` is a specialized version of `checkif(true,`C`)`.

`check_compif(`F`,`$Comp$`,`$Pred'$`)`: checks a computational property iff F is *true*. For a given computational property `P/1`, and a predicate *Pred'* to be checked, a term *P(Pred')* is built and passed as *Comp*. For example, if the property is `not_fails/1` and the predicate `qsort_1(A,B)`, then *Comp* = `not_fails(qsort_1(A,B))`. In turn, $Pred'$ is used to pass the direct call to the predicate (i.e., `qsort_1(A,B)` in the example). If F is `false` then $Pred'$ is called, executing the procedure directly. If F is `true` then *Comp* 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. From a logical point of view this can be understood as:

(*F* `==` `true` `->` *Comp* `;` *Pred'*)`.`

`check_comp(`*Comp*`,`*Pred'*`):` a specialized version of `check_comp(true,` *Comp*`,`*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.

4 Combining Several Predicate Assertions

The schemes presented previously have illustrated how a single assertion is translated into run-time checks. 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. But note that in the case of `calls` assertions, run-time check exceptions for the unsatisfied assertions are thrown only if all such checks failed.

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(checkif_comp(F1, is_det(qsort_1(A,B)),qsort_1(A,B))),
                 checkif_comp(F1,is_det(qsort_1(A, B)),qsort_1(A,B))).
```

5 Run-Time Checking of Program-Point Assertions

Clauses are transformed as follows for run-time checking at program-points:

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

This is a comparatively simpler task than implementing predicate-level assertions: the natural transformation is a similar one to the "transforming calls" approach, but with the advantage that only one program point needs to be transformed for each assertion. Also, only the `rtcheck/1` and `check_comp/1` primitives are required. In the case of computational properties its definition is called directly.

6 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 supported by the Ciao/CiaoPP system 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 grammar for expressing *what to execute*. They appear underlined in Fig. 1. In particular, we have added a new assertion schema for expressing *what to execute*: `:- exec` *Pred* [`:` *Precond*] [`+`*exec-formula*]`.`

This assertion 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 about how to drive this execution. In uur approach many of the properties usable in *Precond* (e.g., types) can be run as value generators in order to generate values for these variables. We also have specific generator properties such as, for example, for generating random values for the variables (e.g., for floating point numbers) including special cases like *infinite*, *not-a-number* or *zero* with sign. Properties typically inferred by static analysis (e.g., types) can also be used for automatically generating input data for the unit tests (see [GZAP08] for a technique for this purpose).

Regarding the atomic formulas appearing *exec-formula* (*Exec-prop* in the grammar) the following are two (currently defined) useful properties:

`try_sols(N)`: Expresses an upper bound `N` on the number of solutions to be checked.
`times(N)`: Expresses that a the execution should be repeated `N` times. This increases the chances of test failure, for intermittent failures.

Example 4. The assertion:

```
:- exec append(A, B, C) : (A=[1,2,3],B=[4],var(C)) + times(5).
```

expresses that a call to `append/3` with the first and second arguments bound to `[1,2,3]` and `[4]` respectively and the third one unbound should be executed five times.

Example 5. The assertion:

```
:- exec 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.

Example 6. We can define a unit test with the previous 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 the first assertion for unit testing is also 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 run-time checking machinery is used for run-time checks and unit testing. In addition, static checking of assertions can safely avoid (parts of) unit tests execution.

7 Compound Assertions for Unit Tests

In order to simplify the process of writing tests we introduce another predicate assertion schema, the `test` schema, which can be seen as syntactic sugar for a set of predicate assertions, and has the form:[2]

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

This assertion is interpreted as the combination of three assertions, one assertion expressing *what to execute*:

`:- exec` *Pred* [`:` *Precond*] [`+` *Exec-Props*].

and two assertions expressing *what to check*:

`:- 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 translated into the assertion in Example 4, plus the two in Example 6.

We now give some more (non-exhaustive) examples of unit test definition using compound assertions.

Example 7. **Testing Failures and Exceptions:** In this example we illustrate the use of some computational properties, namely, the property `fails` (respectively `not_fails`), which expresses that the whole computation described by the test should fail (respectively should not fail), and the property `exception(Excep)`, which is used for expressing that a test execution should throw the exception `Excep`. Consider the predicate `p`/2 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 says that the call `p(a)` should not fail; the second one says that the call `p(b)` should fail; and the third one that the call `p(c)` should raise an exception. However, the following test reports an error, i.e., fails:

```
:- test p(A) : (A = c) + not_fails.
```

Example 8. **Testing the Written Output:** For this purpose we use the (computational) property `user_output(String)`, which expresses that a predicate should write the string `String` into the current output stream.

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!").
```

Example 9. **Testing Multiple Solutions:** Assume now that want to check all possible solutions to a call to `append`/3 with the first two arguments uninstantiated. We can write the following assertion for this purpose:

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

There are also other properties that can be used 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 values with a given probability distribution.

[2] Note that the syntax grammar presented previously does not include this extension.

8 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.

Reporting simply that some condition failed is less informative than saying where it did, to what assertion it corresponds, or what was the last call mode of the predicate that violated it. In the case of a `comp` assertion the actual call could also be printed.

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, an in a particular format so that the graphical program development environment can locate these points in the source code and highlight them automatically.

Our goal is to provide precise information when reporting violated assertions also when performing run-time checks. This requires adding an extra argument to the checking predicates through which certain information is passed, such as the location of the corresponding assertion(s) and the call program point in the source code. This information can then be passed to the exception handler when the exception occurs, and the handler can print it out in a suitable way. In particular, messages are generated in a format that is compatible with that used when reporting compile-time checking errors, and thus run-time errors can also be easily traced back to the sources by the program development environment. The transformation is responsible for instrumenting the transformed code to include the necessary information.

On the other hand, while having rich information available when a run-time check fails is crucial to being able to locate bugs in programs, there is a clear trade-off between the size of the program and the overhead introduced in it and the quality of the messages issued. Different levels of information may be appropriate for different contexts. For example, programs can be compiled with a setting that implies lower overhead and, if an exception is raised, the program can be recompiled with a higher level of instrumentation and rerun until the exception is raised again, this time obtaining more precise information for location of the error in the sources. Also, in systems that are resource constrained, such as many pervasive and embedded systems, lower levels of instrumentation would be appropriate and perhaps even load and use of the pretty printer library can be avoided, since the error messages can be interpreted in a different host.

There are several levels of instrumentation in the current implementation of the run-time check transformations that can be configured. However, to keep this discussion shorter, we report on 2 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.[3]

To illustrate the different code instrumentation levels, consider the following assertion and property definitions, in addition to a definition of `qsort/2` such as that of Figure 4:

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

```
:- success qsort(A,B)  => (ground(B),sorted_num_list(B)).
:- 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]).
```

which ensures that `qsort/2` always returns a ground, *sorted* list. Assume also that the program has been written in a buggy way (about which we will discover later). If we select *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]).
       Unsatisfied <<success>> property:
       sorted_num_list([2,1]).
ERROR: (lns 16-21) Check failed in qsort/2.
}
```

Note that 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 be able to determine the literal in which the predicate was called causing the violation. If we perform instead the transformation with the *high* instrumentation level 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]).
       Unsatisfied <<success>> property:
       sorted_num_list([2,1]).
ERROR: (lns 16-21) Check failed in qsort/2.
ERROR: (lns 16-21) Check failed when invocation of
       qsort([3,1,2],_1)
       called 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]).
       Unsatisfied <<success>> property:
       sorted_num_list([3,2,1]).
ERROR: (lns 16-21) Check failed in qsort/2.
}
```

In this example we have used the `call_rtc`/1 meta-predicate, which intercepts the run-time error, shows the related message and lets the execution program continue as if the program where not being checked. With this new output it is easier to detect the error. Looking at the call stack dump, we can see the list of predicates being checked up to the call of the buggy code. 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.

```
:- 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. 4. A quick-sort program with assertions.

Qsort	Low				High			
Obj Size:	Inline		Library		Inline		Library	
7625 (bytes)	Modes	Types	Modes	Types	Modes	Types	Modes	Types
Entry	1.37	1.04	1.19	1.22	1.81	2.11	1.32	1.35
Exit	1.50	1.78	1.11	1.15	1.94	2.22	1.23	1.28
Comp*	1.69	1.93	5.43	5.45	2.60	2.85	5.53	5.55
E/E/C	2.27	2.62	5.72	5.77	3.20	3.55	5.83	5.88
Calls	1.32	1.62	1.14	1.17	1.72	2.01	1.23	1.26
Success	1.45	1.73	1.07	1.11	1.85	2.13	1.14	1.19
Comp	1.64	1.88	5.35	5.38	2.57	2.81	5.45	5.48
C/S/C	2.07	2.41	5.53	5.58	3.01	3.35	5.64	5.69

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

9 Implementation and Experimental Results

We have implemented the framework within the Ciao/CiaoPP system.

The call stack dump was implemented by reusing the exception handling mechanism which is native in Ciao. Each time an exception is cached in a predicate with run-time checks enabled, a locator is added to the exception. This way, a more informative message of the form "Failed when ... called ..." can be generated. However, such exception handling mechanism was implemented using meta-calls, assert and retracts, causing a negative impact in the benchmarks that use it.

We now report on some experimental results from our implementation within the Ciao/CiaoPP system of the testing and run-time checking transformations proposed. The experiments report both size and time overhead due to run-time checks. We have used the qsort program in Figure 4, with an input list of size 600 to run several experiments for different variations of the following parameters:

- **Library or inlined run-time checks:** we have implemented the transformation first as described in the previous sections, where the `check` predicates are assumed to be in a library. The results are provided in the columns labeled **Library**. The ratios shown are with respect to 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

Qsort	Low				High			
exec time:	Inline		Library		Inline		Library	
661 (us)	Modes	Types	Modes	Types	Modes	Types	Modes	Types
Entry	1.00	1.75	1.00	1.77	1.01	1.76	1.01	1.77
Exit	1.00	2.51	1.01	2.64	1.01	2.52	1.02	2.54
Comp*	1.00	1.76	1.01	1.77	1.05	1.81	1.06	1.82
E/E/C	1.00	3.26	1.03	3.29	1.05	3.31	1.08	3.35
Calls	3.36	50.89	65.30	121.37	36.58	86.15	112.65	169.62
Success	4.58	101.00	151.54	265.54	38.98	141.96	209.01	325.77
Comp	6.25	53.59	95.86	152.66	118.47	164.30	223.53	281.82
C/S/C	10.20	117.84	192.01	323.90	120.74	238.56	386.44	547.87

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

program. This often achieves allows better performance but sometimes at the cost of the increased code size. Note however that not in all cases the code is increased, 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,[4] 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 the columns labeled *Types* only types are checked, whereas in the columns labeled *Modes* only the modes are checked.
- **Low or high instrumentation:** as defined in Section 8.
- **Using several kinds of assertions:** several combinations of different kinds of assertions have been tested (first column).

Table 1 and 2 present the overhead, in size and time respectively for the experiments, expressed as the 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, patch 13.

Note that the columns in the tables have been organized with several combinations of the configurations explained above. In the rows of the tables we have tested the different kind of assertions. For assertions about computational properties 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 also due in part to the lack of optimization in the exception handling mechanism of Ciao.

Table 3 shows experimental results for larger programs, namely, the systems Ciao, CiaoPP and LPdoc (and 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

[4] This overhead can be significantly reduced via multiple specialization [PH99,PH95]. 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		Objects		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.

several instrumentation levels of run-time checks. The binary refers to the statically linked executable of the main program of such systems and in all of them, it is the command line tool provided. 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 below the measurement noise level.

In order to facilitate the execution of tests, the unit testing framework has been integrated in the development environment allowing executing the tests present in a module easily. The execution of the tests is done as follows:

1. The user selects the module or the directory that contains the modules with tests to be executed.
2. The assertions are read and each time a test is found, a method is added to the main procedure of an auto generated program that invoques such method. The goal of such method is to call the predicate being tested in the way specified by the unit test commands.
3. The modules being tested are compiled with run-time checking enabled.
4. The main procedure that invoques the tests is called by the unittest driver in a separate process, to prevent undesirable side effects or failures if the program being checked aborts due to an unexpected error. This program writes a log file containing the results of the execution (such as, for example, exit or failure of the predicate, unhandled exceptions and so on), that is further analyzed by the unittest driver in order to take actions depending on the observed behavior.
5. If a test causes the failure of the main program, the control is returned to the driver, and the aborted test is recorded to be processed. After that, the driver (optionally) tries to execute the remaining tests. This process continues until all the tests are executed.
6. The generated log file is processed by the driver and, depending on the verbosity level, different information about the execution is presented, such as for example, the tests passed, failed, aborted and in each one the cause of such behavior. At this point, the run-time check exceptions saved in the log file are processed in order to show the related message.

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), which have helped us to check whether some errors have been introduced in the

development process. 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 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, that would in fact eliminate many of the unit tests.

10 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. This has allowed us to propose and implement unit testing via a minimal addition to the assertion language. We have proposed methods 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. We also have 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 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. Several experimental results have been presented 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, except with high levels of instrumentation (e.g., gathering information on the call stack). However, this is due to the simplistic way in which this type of instrumentation is implemented, which can be optimized using lower-level primitives. For example, it prevents the compiler from performing some classical optimizations like tail recursion. We also plan to further extend the assertion language with more primitives such as `time_out(T)`, which can be used to express that a test should finish in less than `T` milliseconds, `user_error(Str)` which expresses that a predicate should write the string `Str` into the current error stream, or to add more properties for generating random input data values with a given probability distribution. We plan to study how the multiple specialization present in CiaoPP can further reduce run-time overhead. Finally, we are also working on an improved and more compositional strategy to defining computational properties.

References

BCC+06. 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`.

BDD+97. 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, Linköping, Sweden, May 1997. U. of Linköping Press.

BJ93. B. Belli and O. Jack. Implementation-based Analysis and Testing of Prolog Programs. In *ISSTA '93: Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 70–80, New York, NY, USA, 1993. ACM.

CLI97. The CLIP Group. Program Assertions. The Ciao System Documentation Series – TR CLIP4/97.1, Facultad de Informática, UPM, August 1997.

ER96. Nancy S. Eickelmann and Debra J. Richardson. An Evaluation of Software Test Environment Architectures. In *ICSE '96: Proceedings of the 18th International Conference on Software Engineering*, pages 353–364, Washington, DC, USA, 1996. IEEE Computer Society.

GZAP08. 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)*, 2008. To appear.

HPB99. M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

HPBLG05. 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 Computer Programming*, 58(1–2):115–140, October 2005.

PBH97. 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. Available from `ftp://clip.dia.fi.upm.es/pub/papers/assert_lang_tr_discipldeliv.ps.gz` as technical report CLIP2/97.1.

PBH98. G. Puebla, F. Bueno, and M. Hermenegildo. A Framework for Assertion-based Debugging in Constraint Logic Programming (abstract). In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP'98)*, number 1520 in LNCS, pages 472–473, Pisa, Italy, October 1998. Springer-Verlag.

PBH99. 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)*, pages 31–39, Venezia, Italy, September 1999. U. Ca' Foscari di Venezia.

PBH00a. G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.

PBH00b. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

PH95. 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.

PH99. G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

ZGQC08. Lingzhong Zhao, Tianlong Gu, Junyan Qian, and Guoyong Cai. Test Frame Updating in CPM Testing of Prolog Programs. *Software Quality Control*, 16(2):277–298, 2008.

Appendix A: Transformation Schemes

We start by discussing two possible approaches regarding the source-to-source transformations to be performed in order to implement run-time checking schemes. We concentrate first on the predicate (`calls`, `success`, and `comp`) assertions. These two approaches are illustrated in Figure 5.

In the first kind of transformation (Figure 5-b) the run-time checks are placed before and after any call to predicates which are affected by assertions (`p/2` in the example). We will call this kind of transformation "transforming calls." In the second kind of transformation (Figure 5-c) the original predicate is rewritten so that it performs the run-time checks itself, each time it is called. In this case only the definition of the procedure is modified (in the example the original `p` predicate is renamed to `p'` and a new definition of `p` is added which performs the run-time checks; calls to `p` are left unchanged). We will call this kind of transformation "transforming procedure definitions."

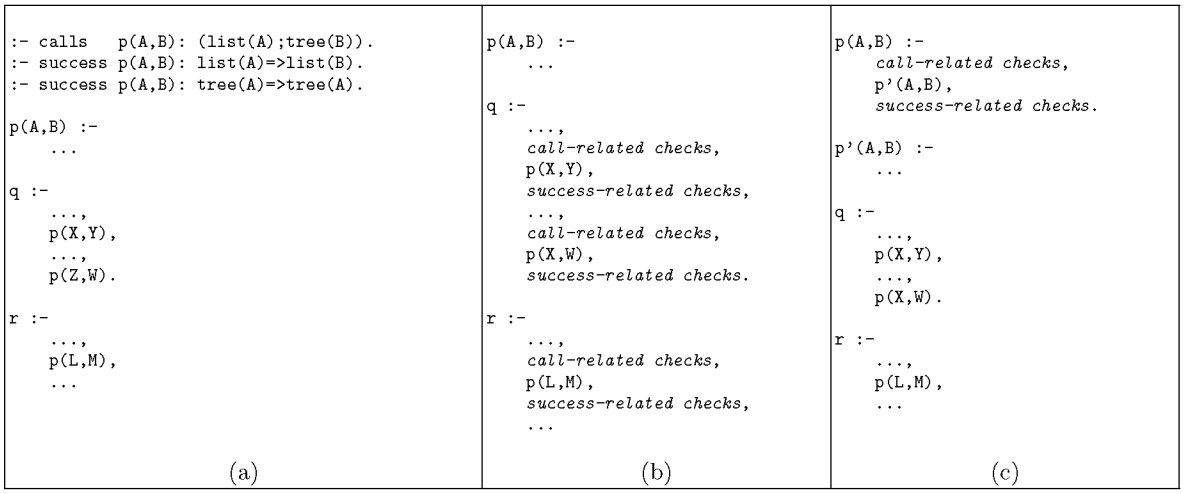

Fig. 5. Two possible transformation schemes (b and c) for predicate assertions.

Appendix B: Examples of unit test Definitions

Other examples:

```
:- test pred display_fail + (user_output("hello"), fails) # "Test OK".

display_fail :- display(hello), fail.
```

Characteristics	Transforming Calls	Transforming Predicates
Code size increase	higher	lower
Number of files to recompile if an assertion changes	many	one
Two versions of each file needed in order to compile with and without run-time checks	no	yes
Modules with and without run-time checks can be mixed	yes	yes

Fig. 6. Advantages and disadvantages of transformation schemes.

Assertion:	The definition of *Pred* is transformed into:

`:- calls` *Pred* `:` *Cond*.	*Pred* `:-`
	`check(`*Cond*`),`
	Pred'.
	
	Pred' `:-`
	`... ,`

`:- success` *Pred* `:` *Precond* `=>` *Postcond*.	*Pred* `:-`
	`checkc(`*Precond*`,F),`
	Pred',
	`checkif(F,`*Postcond*`).`
	
	Pred' `:-`
	`... ,`

`:- comp` *Pred* `+` *Comp*.	*Pred* `:-`
	`check_comp(`*Comp*`,`*Pred'*`).`
	
	Pred' `:-`
	`... ,`

`:- comp` *Pred* `:` *Precond* `+` *Comp*.	*Pred* `:-`
	`checkc(`*Precond*`,F),`
	`checkif_comp(F,`*Comp*`,`*Pred'*`).`
	
	Pred' `:-`
	`... ,`

Fig. 7. Translation schemes for different kinds of predicate assertions.

```
:- test pred call_test10(X) : (X=(write(3), call(1)))
   + (user_output("output"),
      exception(error(type_error(callable, 1), 'in metacall')))
   # "Wrong test".

:- meta_predicate call_test10(goal).

call_test10(X) :- call(X).

:- test pred cut_test5 + (user_output("Cut disjunction"), fails) # "Test OK".

cut_test5 :- (! ; write('No')), write('Cut disjunction'), fail.
```

Appendix C: Verifying ISO-prolog Compliance of Ciao

In this section we describe how the implemented the framework within the Ciao/CiaoPP system has effectively been applied to the verification of ISO-prolog compliance.

	Tests failed	
i	Incompatible format of syntax error exception	10
i	Incompatible format of type error exception	9
i	Incompatible format of permission error exception	28
i	Incompatible format of Domain error exception	2
i	An error is expected, but ciao just fails	138
i	Ciao throws an error different than the specified in the standard	15
i	The predicate in Ciao Fails, but in ISO, it should succeed	22
i	The execution of a predicate should raise an error, but it succeed	19
m	Predicates with missing functionality	24
i	Ciao adds more information to a predicate (module expansion)	6
i	More solutions than the expected	1
f	stream manipulation related errors	14
f	unexpected abort of the test being executed	14
i	non-ascii characters (not iso, but SICSTUS-EDDBALI-like behavior)	7
f	aborted tests	1
m	Tests changed because currently we can't deal with several errors	7
m	stream options unimplemented:	2
m	alias for streams unimplemented:	32
m	stream option eof_action unimplemented:	12
m	stream option past_end_of_stream unimplemented:	2
m	unimplemented options for close:	5
f	char handling related errors:	2
i	Malformed body (negation of cut):	1
f	current output related:	1
i	predicate that succeeds:	1
m	failed test because time_out(_) property is not implemented:	1
f	Tests with side effects:	7
i	Arity mismatch issues:	3
m	Not relevant tests in ciao, due to unimplemented arithmetic behavior	5
i	Incompatibilities	262
m	Missing predicates or functionality	90
f	Failures and errors	39
	Total number of failed tests	391
	Total number of executed tests	976
	Percentage of passed tests	60 %

Fig. 8. Summary of the application of unit tests for ISO-prolog compliance