Improving the ISO Prolog standard by analyzing compliance test results

Péter Szabó[1] and Péter Szeredi[2]

[1] Budapest University of Technology and Economics,
Department of Computer Science and Information Theory,
H-1117 Hungary, Budapest, Magyar tudósok körútja 2,
pts@szit.bme.hu, WWW home page: http://www.inf.bme.hu/~pts/
[2] Budapest University of Technology and Economics,
Department of Computer Science and Information Theory,
H-1117 Hungary, Budapest, Magyar tudósok körútja 2,
szeredi@szit.bme.hu, WWW home page: http://www.cs.bme.hu/~szeredi/

Abstract. Part 1 of the ISO Prolog standard (ISO/IEC 13211) published in 1995 covers the core of Prolog, including syntax, operational semantics, streams and some built-in predicates. Libraries, DCGs, and global mutables are current standardization topics. Most Prolog implementations provide an ISO mode in which they adhere to the standard.

Our goal is to improve parts of the Prolog standard already published by finding and fixing ambiguities and missing details. To do so, we have compiled a suite of more than 1000 test cases covering part 1, and ran it on several free and commercial Prolog implementations. In this study we summarize the reasons of the test case failures, and discuss which of these indicate possible flaws in the standard.

We also discuss test framework and test case development issues specific to Prolog, as well as some portability issues encountered.

1 Introduction

This paper describes work on testing standards compliance of several Prolog implementations. Part 1 of the Prolog standard [9] and a related textbook [3] have been available for about ten years. By now all major implementations declare that they are standards compliant. In spite of this we have found that there are lots of minor and medium importance details that are interpreted differently by the implementations tested. In a few cases this is due to ambiguities in the standard, in lots of other cases there is a clear deviation from the standard. We believe that even minor discrepancies can make porting Prolog programs very difficult, and to advocate full adherence we have developed an appropriate test framework, described in this paper.

The main goal of this work is to improve interoperability of Prolog code, by clarifying standards compliance of major implementations and pinpointing sources of non-conformance. Even if the implementations remain unchanged,

these results can help in developing compatibility layers to support interoperability. In long term we hope that our test framework will contribute to the wider and more precise acceptance of the standard.

The paper is structured as follows. Section 2 gives an overview of the Prolog implementations used and tested in this work. Section 3 describes the architecture of the test framework. In section 4 we present the results of the testing, including some basic statistics, a brief analysis of common failures and proposals for standard improvement. Section 5 discusses related work, while in Section 6 we present some conclusions and plans for future work.

2 Implementations tested

It was our goal to test many different implementations, in order to identify as many problems as possible. Although we focused on free implementations, we included a commercial one (SICStus) as well. All the implementations are in active use today (and most of them are under active development), save aprolog. We did not want to test new or experimental implementations, the first versions of all of those we tested appeared at least 9 years ago.

These are the implementations we have run the tests on:

SICStus 3.12.3 The original test suite and framework was developed at SICS in an effort to make SICStus [8] ISO-compliant, so it was natural to keep it included as well. We have run the tests in the ISO mode of SICStus – the other, legacy mode is going to be dropped in version 4, anyway. Development of SICStus started in 1984, more than 10 years before the Prolog standard was published.

SWI-Prolog 5.4.7 SWI-Prolog [11, 12] is one of most popular, actively developed, free Prolog implementations. It also has an ISO and a legacy mode – the tests have been run in ISO mode. We had to turn garbage collection off, because otherwise the application aborted because of a failed assertion in the middle of the tests. SWI-Prolog exists since about 1990.

YAP 5.0.1 YAP [10] is also an actively developed free implementation. It also has an ISO mode, but we rather disabled it, because the application crashes when we try to consult a new source file in ISO mode. YAP has been available since 1989.

gprolog 1.2.16 GNU Prolog [5, 6] is also a free, WAM-based implementation with a native code compiler. Although the latest release is from September 2002, there is still activity on the mailing list. GNU Prolog appeared in 1996, one year after the standard.

Ciao-Prolog 1.10p6 Ciao [2] is a free implementation actively developed by researchers, with a lot of advanced static analysis tools (e.g. type checking, instantiation analysis and partial evaluation) integrated. Ciao-Prolog was released in 1995.

aprolog 1.22 aprolog is an unpublished, minimalistic, very slow standard Prolog implementation by Miklós Szeredi for educational purposes only. It lacks

a module system, a WAM implementation, a garbage collector, cyclic term unification, line number reporting for exceptions and unbounded integer arithmetics. The reason we use it is that it has been developed with standards in mind – only ISO Prolog, nothing more. And it is also a demonstration that even simple implementations can be standards compliant. Although aprolog was written in 1997, some of the built-in predicate implementations date back to as far as 1984 – 11 years before the standard came out.

When we tried to port the framework to XSB (version 2.7.1), we faced the *need too many registers* error when trying to assert a fact with too complicated arguments (i.e. try `asserta(t([a(b),a(b),...,a(b)]))` with a list of more than 260 elements). A major rewrite might thus be necessary for an XSB port.

We are not planning to port the framework to other implementations by hand, but we are designing a generic approach, that autodetects as much as possible about the new Prolog system, thus adding the correct port to a new implementation (or a new version) will be easier and faster.

3 Test suite and framework

This section describes the structure of the software. We first deal with the test cases and then give an outline of the framework.

3.1 Test suite design

Our test suite consists of 1012 test cases, each corresponding to a section in the standard. Each test case has the following properties:

meta-information identifier (series : number), author, relevant section number of the ISO standard, dangerousness indication, irrelevancy conditions;
goal to be executed;
environment in which the goal is executed: files to create, Prolog source files to load, Prolog flags to alter, cleanup code after running goal;
expectations which determine whether the test run is OK or not. The following outcomes can be expected: success (with possible constraints on variable substitutions of each solution), failure, exception (with possible constraints on the exception term). There is a macro expansion facility in the expectations, so they can be written to be more compact and consistent.

Here is an example test case:

```
test_case(strprop:2, iso_doc, %8.11.8.4         % meta-information
  stream_property(S, output),                              % goal
  {[S <-- FOut ], [S <-- COut], ...},             % expectations
  [ pre((open(bar, write, FOut),                      % environ-
    current_output(COut))), clean(bar)]).             %     ment
```

The meaning of test case *strprop:2* is the following. This is a test case whose goal is taken from the ISO standard section 8.11.8.4. The test case verifies that if we open an output stream (`FOut`), then *stream_property/2* has to enumerate it as well as the current output stream. The curly braces signify that the enumeration order is not significant, and `...` signifies that there might be more solutions. It is possible to prescribe an order using square brackets instead of curly braces.

Some of the tests are dangerous: they hang the process (i.e. they cause segmentation violation, infinite loop or infinite memory allocation) in some implementations, but work fine in others. The reason of the hanging is clear in some cases (e.g. the unification `X=f(X)` can hang a Prolog implementation if it does not support unification of STO ("subject to occurs-check") terms), but sometimes there is a bug in the Prolog implementation, and the process hangs for no apparent reason. (Testing *halt/1* is also dangerous, because *halt/1* aborts the whole Prolog process, including the framework.) When the standards allows a hang in a particular case, we call the test case *irrelevant*, otherwise we call the test case *dangerous*. The test case can also be *irrelevant* if it does not hang the process, but it can be skipped for other reasons; for example, integers are bounded in some implementations, but unbounded in others, and tests that check for an *int_overflow* error must be skipped in unbounded implementations.

The framework accepts *implementation aspect declarations*, based on which the framework can decide whether a test is *irrelevant* with respect to an implementation. And even when a test is relevant, its expectations can be made dependent on the declarations. Currently declarations are not used as much as they could be: for example, a lot of tests fail, because predicate indicators in exceptions are not module qualified the same way in all Prolog implementations – and the test suite expects the SICStus way. The solution would be to introduce a new declaration for this aspect, and make the corresponding test cases use it. Declarations could also be used to have *dangerous* tests skipped; currently they have to be commented out by hand in the test suite.

The test contains tests from various sources:

iso_doc These test cases come directly from the standard [9]. The description of each built-in predicate contains an *Examples* section with goals and a natural language expectation for each goal, which describes what should happen when the goal is called. We kept the original goals, and formalized the expectations so the test case can be automatically validated. In addition to these, the informal examples and tables in the chapters about syntax and control constructs were also converted to test cases.

eddbali The executable specification of Prolog ([4], see below) has some test cases included. We have added them in our test suite, except for those having equivalent counterparts in the *iso_doc* part.

sics The test cases here were added by the authors during the development of the ISO mode of SICStus Prolog.

pts We added these recently, after we have run the test suite on 6 Prolog implementations.

3.2 Test framework architecture

The test framework, which runs in the same Prolog process as the tests themselves, considers test cases in the test suite in the order they are defined, and computes the test result for each of them. Possible test results are *dangerous, irrelevant, failed* and *OK*. Test results are logged, and the log file is later processed by a Perl script for statistics generation and validation of whether all test cases were considered. If some of the test cases were *missing,* this could be a sign that the Prolog process crashed in the middle.

Tests declared *dangerous* are ignored. If the irrelevancy condition of the test case with respect to the Prolog implementation is met, the test is considered *irrelevant* and it is skipped. Otherwise, the environment of the test is prepared, the test goal is run, the expectations are checked, the cleanup code is run. The test is considered *OK* if the expectations are fulfilled, otherwise it has *failed*. For *failed* tests, the outcome expected and the outcome received are also logged.

The test cases, the contents of all files and Prolog programs needed by the tests and the macro definitions are collected in a single Prolog source file as facts. The implementation language of the test framework is also Prolog: there is a big system-independent part (in standard Prolog), and there are helper files for each system: the *main* file, which loads all other files, contains the implementation aspect declarations, and provides some nonstandard functionality (such as *abolish_static/1*) using implementation-specific predicates; and the *utils* file, which implements a compatibility layer providing some utility predicates (such as *append/3* and *term_variables/2*). If possible, the utility predicates are loaded from a built-in module. Other software components are the Perl script that creates the statistics from the log files, and a Makefile that invokes the implementations with the appropriate arguments to run the tests.

The software, including our test framework and test suite can be obtained from `http://www.inf.bme.hu/~pts/stdprolog/` . Currently it needs a UNIX system with Perl, GNU Make and the Prolog implementations installed.

4 Test results

We have run the test suite on the implementations, getting a log file for each run. The log files contained detailed information about each test case, including the description of the failures. In this chapter we present the statistics resulting from the log file, and the conclusions we made from from the failures, including improvement possibilities for the both the standard and its implementations.

4.1 Statistics and evaluation concerns

The statistics depicted on Table 1 were generated by a script from the test results log file. The number of failed tests, however, should not be used as a quantitative measure for the standards conformance of the implementation. That is because multiple failures can be caused by a single reason. It is also obvious why SICStus

passes almost all the tests: the ISO mode of SICStus and the original version of the test suite have been developed by the same team. It is also not a surprise that aprolog fails in a few cases only: aprolog has been written from the ground up to be ISO compliant. The other implementations have quite a lot of failed test cases. That's quite reasonable, because they have not been designed with the standard in mind, but they have been patched after the standard came out.

Table 1. ISO compliance test statistics of 1012 test cases

system	version	#OK	#failed	#dangerous	#irrelevant
SICStus	3.12.3	1010	1	0	1
aprolog	1.22	996	7	0	9
gprolog	1.2.16	929	67	7	9
SWI-Prolog	5.4.7	816	158	8	30
YAP	5.0.1	632	363	7	10
Ciao-Prolog	1.10p6	541	454	7	10

Many implementations (SICStus, SWI-Prolog, YAP, gprolog and Ciao-Prolog) have two modes: an ISO mode and a backwards compatible mode. In the ISO mode, they try to follow the ISO standard as much as possible, while backwards compatible mode violates the ISO standard if necessary in order to run legacy Prolog programs. We ran our test suite with ISO mode enabled, if possible. gprolog has ISO mode only.

We do not know of any Prolog implementations that provide a strict ISO mode in which they disable non-ISO built-in predicates, and refuse all extensions, i.e. constructs forbidden by the standard. For example [1], the standard requires that an operator atom can be operand only if enclosed in parentheses, e.g. "`X= <`" should be changed to "`X=(<)`". Many Prolog implementations, including SICStus and aprolog allow both constructs. This can be considered a syntactical extension to the standard. We accept both in our test cases. Another example: section 6.3.4.3 of the standard forbids an infix and a postfix operator to have the same name. On the other hand, some Prolog implementations allow this, which can also be considered as an extension. However, when the standard explicitly says that a specific error must be thrown in a specific case, implementations must respect this, and our test cases validate each of these error conditions.

There is an *Errors* section for each built-in predicate in the standard. The standard does not specify the order in which these are checked (see section 7.12), so our test cases accept any error if more than one is appropriate.

4.2 Errors and other flaws found in the standard

We will present typos and inconsistencies and ambiguity in the standard, revealed by the failed test cases.

Sometimes the standard itself is inconsistent. For example, the expression evaluation *Examples* in 9.1.7 have error terms which are missing from section 7.9.2c (which enumerates all possible errors during expression evaluation). Our test cases respect 7.9.2c and expect `type_error(evaluable, F/N)`.

The *Examples* in 9.1.7 contain several other errors and typos, for example they expect `0 is 7/35` to be true – which is presumably a typo, and the intended call is `0 is 7//35`. We have fixed those test cases in our test suite.

Section 7.8.3.4 has a bad example, too. It expects `call((write(3),3))` to emit a `3`, which contradicts the specification of *call/1*, which clearly states that an error should be thrown because `(write(3),3)` is not callable.

Section 8.14.2.4 is not clear enough whether `write_canonical(.)` and `write_canonical([1])` must put single quotes around their dots (currently some implementations do, others do not). The general rule states that the output must be able to be read back unambiguously, but in this case the ambiguity depends on the context, i.e. the the writing of layout characters after the dot in the future.

The standard does not specify what to do with non-ASCII characters in the Prolog code. For example, should it be possible to load a Prolog source file containing `X = á`? A straightforward solution would be to adopt Unicode, and to make the byte ↔ character transformation of a text stream specifiable in Prolog (e.g. using stream flags).

4.3 Suggested additions to the standard

Part 1 of the standard [9] leaves a lot of features open, while part 2 deals with modules. The standardization of libraries, DCGs, and global mutables is underway. There are other features in existing Prolog implementations to be considered, for example tabling and coroutines using call blocking/freezing. Most Prolog systems today, however, provide an implementation of these features, but since there is no definitive standard to do them in a uniform way, each system implements them differently. In this subsection we will present the nonstandard features we used in our test framework.

We believe that more power should be granted to the programmer when querying and manipulating the predicate database. This includes a decent *predicate_property/2* built-in, which can report whether a predicate is built-in, static, private etc. There should also be a way to unload a Prolog source file, including the ability to abolish the static predicates defined in it (*abolish_static/1*).

There should also be a set of standard modules (such as *lists* and *terms*). As of now, many Prolog systems already have these, but with a different syntax.

Although modules are documented in part 2 of the standard, most implementations ignore that. When a predicate indicator is reported in an error (e.g. *type_error*), its module qualification is not consistent. This should be clarified in the new standard.

The standard should also specify how to assert, query and retract predicates in modules other than the current one.

If the behaviour of the toplevel had been specified in the standard, testing would have been much easier and safer, because we could treat the Prolog implementation as a black box, and implement the testing framework in a different process, or we could have run the tests and the framework in a different Prolog implementation.

As of now, most Prolog systems provide an ISO mode in which they follow the ISO standard more precisely, but there is no standard way to activate this mode with a single call. Moreover, some implementations crash soon after ISO mode has been activated. Others just change the set of visible predicates, but they do not adjust the semantics of the predicates to make them ISO conforming. In some implementations, Prolog flags must be adjusted one-by-one to make the implementation more ISO-compatible. As a solution to this, a new Prolog flag should be defined which controls whether the system runs in ISO mode or not. For example, the *iso* flag with the `true` value, or the *language* flag with the `iso` value. If the standard introduced a strict ISO mode, a flag should be defined for this, too. For the Prolog implementations with a command line interface, the standard should prescribe a command line option (such as `--iso`) which enables ISO mode at startup. Activating ISO mode would have the following effects: all ISO predicates would be made visible, the semantics of some predicates would be changed to be ISO conforming, existing predicates would be forced to detect error conditions according to ISO, and to throw ISO conforming exception terms, Prolog flags would be reset to their ISO default values etc.

Section 7.8.9.4 could clarify what to do when an uncaught exception is encountered and thus a "*system_error*" happens. We suggest printing the original exception to *user_error*, and then returning control to the toplevel, or – if no interactive toplevel exists – exiting from the Prolog process.

Since the primary goal of Prolog standardization is to make Prolog programs more portable across implementations, a standard library should be defined. There are many features (such as the lists and terms modules, predicates for manipulating the file system, predicates for TCP/IP communication, capturing stream output to a string, timeouts, blackboard/flag predicates, predicate mode and/or type declarations, suspended execution with coroutines and CLP(FD)) already present in multiple Prolog implementations, but because of the small differences in the load and invocation syntaxes and in the semantics, it not possible to write a portable Prolog program using these widely available features, without inserting implementation-specific code for each supported implementation.

In order to help the work of adding these common features to the standard (and also to help Prolog programmers write portable code), a compatibility library could be developed, which provides an implementation-independent interface for these features, and maps all its publicly available predicates to implementation-specific calls.

4.4 Common failures in implementations

We will discuss the typical reasons that made multiple test cases fail. The discussion covers the areas in which implementations should be improved, without

going into details about which test case failed in which implementation. These details can be found in the test results log available from the web page of our software.

Some Prolog implementations still use the old, Edinburgh semantics of the caret, i.e., they look for the caret in the middle of the 2nd argument of *setof/3* and *bagof/3*.

There are problems with stream properties returned by *stream_property/2*. It is common that standard properties are missing or the default values for `user_output` etc. are not compliant.

Some implementations do not make a proper distinction between the end-of-stream and past-end-of-stream states. Some of them even allow reading the EOF indicator after the past-end-of-stream.

There are also some typos in error terms, e.g. `type_error(atom,...)` is reported instead of `type_error(atomic,...)`.

It is a common mistake in some implementations that they mix throwing *existence_error* and *domain_error* when an invalid stream term is passed to them. Sometimes *stream* and *stream_or_alias* are confused in the error term. The standard is always clear, however, about what and when should be thrown.

In some implementations it is possible to open inherently unrepositionable files (such as UNIX character devices) with `reposition(true)`.

In some implementations *stream_property/2* returns the alias *user_input*, which is not a stream-term, in its first argument. A similar problem is that *current_predicate/1* returns built-in predicates.

When a stream error is reported, the stream term associated with the call is not copied properly, e.g. `42` is indicated instead of `'$stream'(42)`.

Some implementations confuse character code lists (codes) and character atom lists (chars), e.g. *atom_codes/2* returns an atom list instead of a code list.

The standard is always clear about what culprit should be reported in *type_error* messages. For example, `call((true,3))` must report `type_error(callable, (true,3))` instead of `type_error(callable, 3)`. But some implementations prefer to report the latter, more specific culprit. Our test suite has an implementation aspect declaration for this. There is a similar problem of reporting a *type_error* when the tail of the 2nd argument of *write_term/2* is not a list. Another similar case is that `myop` should not be part of the error triggered by `op(100,xfx,[myop,','])`.

Some Prolog flags or flag values are missing in some implementations, e.g. the flag *unknown* cannot have value *warning*. Sometimes the default value of the flag does not match the defaults prescribed in section 7.11 of the standard. Sometimes they have *false* instead of *off* etc.

There are a lot of problems in arithmetic error reporting. Some implementations make `+inf is 1/0`, `nan is sqrt(-1)` or `-inf is log(0)` true – all of which must have thrown an *evaluation_error* according to the standard. Sometimes an implementation fails to report an *int_overflow*, and the addition of two large integers succeed.

The result of `X is 7//-3` depends on the rounding function used (see in section 9.1.3.1) and thus it is implementation-specific: X becomes either -2 or -4. On the contrary, the semantics of the operations *mod* and *rem* are unambiguous, but some implementations get the sign of the result wrong.

All integer operations must throw an error if they do not receive integer arguments (e.g. `X is 1.0>>2`). However, sometimes they just convert the float they receive to an integer.

Most arithmetic operations can return either an integer or a float – but rounding operations such as `float` and `truncate` have a specific return type (defined in section 9.1.6). Some implementations, however, do not respect this, and have `7.0 is floor(7.4)` instead of `7 is floor(7.4)`.

Although this is not a validity but a reliability issue, we have to mention the dangerous tests here: those that hang particular Prolog implementations. We have found segmentation violations, infinite loops and infinite memory allocations, none of which should ever happen when running Prolog code. Sometimes we could not even identify a specific test case or set of test cases which created the danger, for example when the Prolog process died with a failed assertion in the middle of the garbage collection. But most of the time, the problem was caused by a single test case, which we found and declared dangerous in that implementation. For example, `atom_concat(A,A,AA)` triggered a bug in the native-language code of *atom_concat/3* in one of the implementations when A was an atom already of maximum length, and the bug caused the process to emit a segmentation violation. In another case, we could identify that testing *abolish/1* causes instability, but none of the individual test cases made the process crash.

We have found many similar fundamental flaws (dangerous or not) in the error handling code of the built-in predicates in many implementations. Section 7.12.1 is perfectly clear about what to do when an error happens: the call has to be replaced by an appropriate call to *throw/1*. On the contrary implementations tend to throw the wrong error, just fail, print an error message and then fail, or even crash – none of which is conforming.

Although full stop (`.`) marks the end of a term only if it is followed by a layout character, the layout character must not be consumed by *read/1* (see section 8.14.1.4). Some implementations do consume it, however.

There are many problems with *read/1*, e.g. some implementations cannot read `write(0'\')` or quoted atoms as structure names such as `'is'(1,2)`. Some implementations have problems reading terms when *char_conversion/2* is in effect.

The standard expects *read/1* to throw a *representation_error* when a limit (such as maximum atom length) is exceeded. However, some implementations throw *syntax_error* instead.

Some implementations allow an atom to contain zero-coded characters (e.g. `'\000\'`), and they can even read those from text streams.

number_chars/2 in some implementations has problems ignoring whitespace in the beginning of the number.

clause/2 in some implementations returns a preprocessed predicate body for dynamic predicates. Not only calls get module qualification, but calls to built-in predicates are also replaced to calls to implementation-specific, hidden predicates.

5 Related work

We have gathered tests from various sources when compiling our test suite, see Subsection 3.1 for the details. The ISO working group X3J17 dedicated to improve the Prolog standard has also published a test suite [7] of 570 test cases. Most of them are directly copied from the standard, similarly as our 675 *iso_doc* test cases.

[1] suggested that the ISO Prolog standard was not taken seriously, and most implementations were not compliant. We believe that a test suite like ours, which works in multiple Prolog implementations can reveal many specific problems, which implementors can focus on if they strive for standards compliance. We are planning to publish a detailed technical report and notify the implementors about the failures we have found.

The standard has a formal semantics in its appendix. This semantics is formalized such a way that the specification can be executed – thus we can get a completely compliant implementation. The text of the appendix, the executable specification and an executor implementation in Prolog is available separately [4]. The executable specification, as it is now, is very inefficient, and it also has some limitations and bugs, see section 2.5 of [4].

6 Conclusion and future work

A good programming language standard makes writing portable programs possible – provided that the standard covers all the features used by the program, and that implementations conform to the standard. Validation tests can reveal weak spots of the standard and also problems in the implementations. We have written a test framework and compiled a test suite in order to explore the areas in which the Prolog standard and its implementations can be improved. We have analyzed the test results, classified the reasons why some cases failed, identified common problems and even some weaknesses of the Prolog standard. It was not our goal to fix the problems, but to attract attention of the implementors and the creators of the standard to them.

Our test suite covers most other test collections available ([9, 4, 7]), and our test framework has been ported to several Prolog implementations, thus it can be considered general.

It is our primary goal to cooperate with the standard designers and Prolog implementors. We are planning to improve and extend the test suite, and port the test framework to as many implementations as possible. But in order to get others involved into our work, first we have to polish the framework and document it properly. First we are planning to publish a technical report for the

implementors, with the failed tests documented in detail, so they can start fixing the problems.

Currently it is hard to port the framework to a new Prolog implementation: the programmer has to adjust a lot of settings and write helper predicates after a careful study of the Prolog implementation – and they do not get proper feedback if they do it wrong. To help this, we are working on an autodetection mechanism similar to GNU autoconf, and we're also writing a porting tutorial.

More *implementation aspect declarations* should be added instead of commenting out test cases. It is important to have as many declarations as possible, because they can eliminate a lot of false non-compliance messages at once, and they can also reduce the risk of dangerous test cases. Most declarations should be auto-detected instead of being specified manually.

The handling of dangerous tests should be improved. It should be possible to declare an implementation-specific danger level for each test case, and the framework would skip that test or run it in a separated process time- and memory-constrained if necessary. (This would also make the effect of an uncaught exception testable.) Semi-automatic tools should be developed to help the programmer find the dangerous test cases.

The test framework ignores module qualification when it validates dynamic clause bodies or error culprits. However, in some cases this is not precise enough, for example in the example of section 8.9.3.4, the call to `retract((foo(A) :- A,call(A)))` might have to be replaced by `retract((foo(A) :- A,call(user:A)))` in order to succeed when the clause `foo(X) :- call(X), call(X)` is asserted. That's because asserting involves module qualification, which transforms the clause to `foo(X) :- call(user:X), call(user:X)` .

The effects of some tests are hard to describe in the current framework (e.g. when both cyclic terms and module qualification is involved). This should be improved.

Some Prolog implementations have problems parsing the test suite, even when it is in functional notation, e.g. some of them cannot parse `0'\'` as an integer constant. To help this, the framework should have its own, standard compliant reimplementation of `read/2`, and it should use this to read the test case clauses.

As the standard develops, and possibly new parts get added, the test suite has to be extended correspondingly. It would be useful if the standard itself had a formal test suite in its appendix, in addition to the informal *Examples* for each built-in predicate.

References

1. Roberto Bagnara. Is the ISO Prolog standard taken seriously? *The Association for Logic Programming Newsletter*, 12(1):10–12, February 1999. URL `http://www.cs.unipr.it/~bagnara/Papers/Abstracts/ALPN99a`.
2. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. *The Ciao System. Reference Manual (V1.10).* School of Computer Science, Tech-

nical University of Madrid (UPM), 2004. URL `http://clip.dia.fi.upm.es/Software/Ciao/`.

3. Laurent Cervoni, Abdelali Ed-Dbali, and Pierre Deransart. *Prolog: Reference Manual*. Springer, 1996.
4. Pierre Deransart and AbdelAli Ed-Dbali. Executable specification for Standard Prolog. URL `http://www.uc.pt/logtalk/links.html`, Download URL `ftp://ftp-lifo.univ-orleans.fr/pub/Users/eddbali/SdProlog`, 5 July 1996.
5. Daniel Diaz and Philippe Codognet. GNU Prolog: Beyond compiling Prolog to C. *Lecture Notes in Computer Science*, 1753:81–92, 2000.
6. Daniel Diaz and Philippe Codognet. The GNU Prolog system and its implementation. In *ACM Symposium on Applied Computing (2)*, volume 1, pages 728–732, 19–21 March 2000.
7. Jonathan Hodgson. Validation test suite for ISO standard conformance. URL `http://www.sju.edu/~jhodgson/x3j17.html`, 2 October 1998.
8. Intelligent Systems Laboratory, SICS, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog User's Manual (for version 3.12.3)*, October 2005. URL `http://www.sics.se/sicstus/docs/3.12.3/html/sicstus.html/`.
9. ISO. *ISO/IEC 13211-1. International Standard, Invormation technology – Programming languages – Prolog – Part 1: General core*, 1 edition, 1995.
10. V. Santos-Costa, L. Damas, R. Reis, and R. Azevedo. *The Yap Prolog User's Manual*. Universidade do Porto and COPPE Sistemas, 2006. URL `http://www.ncc.up.pt/~vsc/Yap/`.
11. Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, December 2003. Katholieke Universiteit Leuven. CW 371.
12. Jan Wielemaker. *SWI-Prolog 5.6.4 Reference Manual*. Human-Computer Studies, 2006. URL `http://gollem.science.uva.nl/SWI-Prolog/Manual/`.