TPLP **22** (5): 755–769, 2022.

doi:10.1017/S1471068422000230 First published online 27 June 2022

Making ProB Compatible with SWI-Prolog

DAVID GELEßUS and MICHAEL LEUSCHEL
Institut für Informatik, Universität Düsseldorf, Universitätsstr. 1, 40225 Düsseldorf, Germany
(*e-mails:* dagel101@hhu.de, michael.leuschel@hhu.de)

submitted 31 January 2022; revised 09 May 2022; accepted 08 June 2022

Abstract

Even though the core of the Prolog programming language has been standardized by ISO since 1995, it remains difficult to write complex Prolog programs that can run unmodified on multiple Prolog implementations. Indeed, implementations sometimes deviate from the ISO standard and the standard itself fails to cover many features that are essential in practice. Most Prolog applications thus have to rely on nonstandard features, often making them dependent on one particular Prolog implementation and incompatible with others. We examine one such Prolog application: ProB, which has been developed for over 20 years in SICStus Prolog. The article describes how we managed to refactor the codebase of ProB to also support SWI-Prolog, with the goal of verifying ProB's results using two independent toolchains. This required a multitude of adjustments, ranging from extending the SICStus emulation in SWI-Prolog on to better modularizing the monolithic ProB codebase. We also describe notable compatibility issues and other differences that we encountered in the process, and how we were able to deal with them with few major code changes.

KEYWORDS: Prolog, porting, compatibility, emulation, ProB

1 Introduction

For most of its existence, the Prolog programming language has been continuously developed and extended by different groups, often in parallel and somewhat independently from one another. New implementations of Prolog were (and still are) created regularly, either derived from other existing implementations or developed completely from scratch. Each of these implementations supports a different feature set than all others, often introducing new extensions to the language and libraries while changing or removing other features.

To establish a common basic definition of Prolog, the core of the language was formalized in an ISO standard (ISO/IEC 13211-1:1995 1995), which is followed by the vast majority of Prolog implementations nowadays – although some intentionally deviate from the standard in small or large ways. Furthermore, as this standard only specifies the *core* of the Prolog language, it does not cover the libraries and advanced language features of modern full-featured Prolog systems. Some of these nonstandard features have established themselves as *de facto* standards that are widely and consistently implemented across many modern Prolog systems. For many other features, no clear standard has

emerged though, with different systems often supporting the same feature with different interfaces. Some features are entirely specific to one system and not supported by other systems at all (Körner *et al.* 2022).

This situation complicates writing Prolog code that can be run on multiple different Prolog systems. The ISO standard core language alone is insufficient for many programs, as it does not provide important features like a module system, constraint programming, or even simple standard libraries like `library(lists)`. Most nontrivial Prolog code thus requires non-ISO features provided by individual systems. As Prolog applications are often only tested on the Prolog system for which they were originally developed, it is easy to accidentally rely on features and behavior not supported by other systems. Developers may also make the conscious choice to only support a single system, in order to reduce development effort or to make full use of a specific system's unique features.

As a case study, this paper examines the process of refactoring a large nonportable Prolog application to make it compatible with other Prolog systems. The application in question, ProB, is an animator, constraint solver, and model checker for formal specifications of safety-critical systems. It has been in continuous development since the early 2000s, based entirely on the SICStus Prolog system (Carlsson and Mildner 2012). As a result, the ProB codebase makes heavy use of SICStus-specific features and libraries, and compatibility with other Prolog systems was generally not considered in its past development.

Recently, we have begun refactoring the ProB codebase to make it compatible with more than just SICStus Prolog, with a specific focus on supporting SWI-Prolog (Wielemaker *et al.* 2012). Our goal is to support both Prolog systems in a single codebase – we are not planning to drop support for SICStus Prolog, but we also do not want to maintain separate branches for the two systems.

1.1 Motivation

Our effort to make ProB compatible with a Prolog system other than SICStus Prolog is motivated by two main factors.

1.1.1 Double toolchain and reliability

ProB is being used by several companies for safety-critical applications (Butler *et al.* 2020), for example, in the development of railway systems as a tool of class **T2** as defined by the European norm EN 50128.[1] However, we strive for ProB to be also used as a tool of class **T3**.[2] The use of a non-mainstream language like Prolog is one obstacle for T3 certification of ProB, among other aspects. Although ProB features an extensive test suite that will often detect Prolog system bugs (Bendisposto *et al.* 2014), this risk can be reduced even further by verifying the results using a second, independent Prolog implementation. One implementation acts as the primary toolchain, the other as a secondary

[1] A tool of class T2 "supports the test or verification of the design or executable code, where errors in the tool can fail to reveal defects but cannot directly create errors in the executable software" (EN50128 2011, Section 3.1.43).

[2] That is, a tool that "generates outputs which can directly or indirectly contribute to the executable code (including data) of the safety related system" (EN50128 2011, Section 3.1.44).

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press

toolchain validating the output of the primary toolchain. Only if the two outputs match can the output of the tool be safely used.

1.1.2 Free and open-source toolchain and long-term support

ProB is free and open-source software. However, because of its dependency on SICStus Prolog, which is commercial and closed-source software, a SICStus Prolog license is required to use the ProB code. Users without such a license have so far been limited to using official pre-compiled builds of ProB. By supporting a free Prolog system such as SWI-Prolog, it becomes possible to build and run ProB from source without any commercial components, which makes it easier for outside users to work with the source code.

An open-source Prolog system also allows developers to debug and fix bugs in the Prolog implementation without assistance from its upstream developer. Some applications of ProB require a guarantee of long-term support (ranging up to 20 or 30 years), in which case it is important to not depend on a single upstream developer for support.

2 Prolog systems and standardization

The core Prolog language is formalized since 1995 in the ISO/IEC 13211-1:1995 standard (last revised in 2017), which specifies Prolog's syntax and semantics, including control structures and basic built-in predicates. Practically all Prolog implementations follow this standard to some degree – many aim for full compliance, but others intentionally deviate from the standard in small or large ways to support old nonstandard code or allow implementing new language features. However, even systems that fall into the latter category generally try to stay ISO-compatible unless there is a specific reason for deviating from the standard.

A second part to the standard was published in 2000 and describes a module system for Prolog. This part of the standard has been largely unsuccessful – as of 2022, aside from one modern implementation[3], most systems instead use some variation of the Quintus Prolog module system or a different nonstandard module system (Haemmerlé and Fages 2006). Due to this lack of popularity, discussion of "the ISO standard" usually only refers to part 1 of the standard unless part 2 is explicitly mentioned.

Work is ongoing on a technical specification that formalizes the syntax and semantics for definite clause grammars (DCGs), which are supported by most modern Prolog systems. The latest draft version of this TS was published in August 2021 (ISO/IEC WDTS 13211–3 2021).

Outside of the ISO standardization process, a group of Prolog system developers started the Prolog Commons initiative[4] in an effort to establish a common set of standard libraries across Prolog systems. As of 2022, this initiative appears to be inactive, with the reference manual being last updated in 2013. Some portability primitives developed as part of the initiative are now widely supported, notably conditional compilation and the Prolog flags `dialect` and `version_info` (see Section 5.1). On the other hand, the

[3] Amzi! Prolog + Logic Server – https://amzi.com/AmziPrologLogicServer/
[4] https://www.prolog-commons.org/

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press

Table 1. *ProB source code statistics*

	Files	Code lines	Comment lines
Core (Prolog)	165	85,680	15,651
Extensions (Prolog)	237	59,358	10,570
Extensions (C, C++)	84	78,784	1691
GUI (Tcl/Tk)	23	32,803	2753

(as of ProB 1.11.1, released 2021–12–29)

proposed libraries have not been adopted by any Prolog system as part of its standard library, although a few systems implement some Prolog Commons predicates in different modules.

No other aspects of Prolog have been formally standardized. This includes most standard library modules, as well as advanced language features like term expansion, coroutining, attributed variables, and mutable data. For some of these features, certain de facto standards have evolved, like the widely implemented coroutining predicates `dif/2`, `freeze/2`, and `when/2` (see Section 5.4.4). Many other features are not supported as consistently though, such as attributed variables, for which there are two similar but incompatible APIs (see Section 5.4.8).

In the remainder of this article, we will specifically focus on SICStus Prolog and SWI-Prolog, the two Prolog systems directly relevant to the ProB compatibility effort. Both are mature Prolog implementations that have been in development since the 1980s and continue to be actively updated and maintained. Although the two systems have been developed independently, both take influence from the Edinburgh/Quintus Prolog tradition and offer an overall similar set of language features, built-in predicates, and standard libraries.

SICStus Prolog offers full conformance to the core ISO Prolog standard. In contrast, SWI-Prolog no longer aims to be ISO-compliant, and in fact introduced certain changes that are incompatible with the standard and break long-standing tradition. Specifically, SWI-Prolog 7 changed the term format of lists and replaced the traditional meaning of double-quotes with a new string data type.

SICStus Prolog is commercial, closed-source software, developed by Mats Carlsson, Per Mildner, and others at RISE Research Institutes of Sweden. SWI-Prolog on the other hand is free and open-source, with development taking place in a public GitHub repository, primarily by the original author Jan Wielemaker, although outside contributions are also accepted.

3 ProB architecture

The core of ProB consists of a relatively large Prolog codebase, made up of about 400 files containing over 150k lines of code (see Table 1), along with a large test suite of almost 7000 unit tests and more than 2000 integration tests. Although development began in the 1990s, ProB is still actively developed and targets current versions of SICStus Prolog (4.6 and 4.7). The codebase is fully modularized, but especially its core modules are tightly interconnected.

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press

In addition to the main Prolog code, ProB includes a number of libraries implemented in C and C++, which are used to interface with external native libraries and to implement certain performance-critical code. ProB can also call various external tools, mostly to support additional specification languages. Most of these non-Prolog components are optional – with the notable exception of the Java-based B parser which is required for most ProB features.

For end users, ProB provides multiple different user interfaces: a command-line interface (`probcli`) for batch verification and data validation, and a set of GUIs (ProB Tcl/Tk, ProB 2 UI) for interactive animation, visualization, and verification. All of these user interfaces share the same Prolog core, but interact with it in different ways. `probcli` and ProB Tcl/Tk are also implemented in Prolog and can call the core directly. ProB 2 UI is instead implemented in JavaFX and runs as a separate process, communicating with the core of ProB via a network socket.

4 Timeline of the port

Before the compatibility effort began, the ProB code was only developed and used on SICStus Prolog and relied on various SICStus-specific language features. This made even basic compatibility with another Prolog system a challenge – for example, SWI-Prolog could not even parse many of the ProB files in their original form, because of SICStus syntax extensions and minor parser differences.

To evaluate the feasibility of the project, we began a proof-of-concept port in a separate branch of the ProB code. Expecting that the first porting attempt would be suboptimal and require later reworking, we did not focus much on maintainability and retaining SICStus Prolog compatibility. Rather, our main goal with this proof-of-concept port was to gain an understanding of the basic issues that needed to be solved and to decide what compatibility mechanism or library to use. After about 8 days of work (part-time by a single developer), we had adjusted the code sufficiently so that the ProB command-line REPL could be started on SWI-Prolog and simple expressions could be evaluated.

After this initial porting attempt, we started mostly from scratch to implement SWI-Prolog compatibility cleanly and remaining compatible with SICStus Prolog. At this point, we decided to use the SWI-Prolog dialect emulation mechanism (Section 5.2), which we were originally hesitant to use, because it is not natively supported by SICStus Prolog. This compatibility issue was easily worked around though, and the emulation proved to be very helpful with replacing much of the manual compatibility code that was needed in the first proof-of-concept port.

With our second port, it took roughly two months of part-time development to achieve basic REPL functionality on SWI-Prolog. After this, we focused on making all of ProB's basic unit tests pass on SWI-Prolog, which took about four months of part-time work in total. A significant portion of the development time was spent on developing the SICStus Prolog 4 emulation in SWI-Prolog (see Section 5.2) and reporting or fixing SWI-specific bugs and compatibility issues that we encountered. All of our additions and patches have been submitted upstream, and most of them are included in the SWI-Prolog 8.4 stable release from September 2021. We hope that this will reduce the effort needed for other developers looking to port SICStus 4 code to SWI-Prolog.

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press

5 Porting process

5.1 Conditional compilation

An important building block for dealing with Prolog system incompatibilities is the de facto standard *conditional compilation* mechanism based on the directives `if(...)`, `elif(...)`, `else`, and `endif`. It is commonly used to check whether a predicate, library, or other feature exists (see Section 5.4.1) and then choose an appropriate code path depending on the features and APIs supported by the running Prolog system. For example, the following code defines a predicate to initialize the random number generator, in a way that is compatible with both the SICStus Prolog and SWI-Prolog random number APIs:

```
:- if(predicate_property(set_random(_), _)). % SWI-Prolog
  set_new_random_seed :- set_random(seed(random)).
:- else. % SICStus Prolog
  :- use_module(library(random), [setrand/1]).
  set_new_random_seed :- now(TimeStamp), setrand(TimeStamp).
:- endif.
```

In fact, `if(...)` conditions can be arbitrary Prolog goals, allowing for more complex feature checks as well.

Using the Prolog flags `dialect` and `version_data`, which originated from the Prolog Commons initiative and are now widely supported, it is also possible to check the name and version of the running Prolog system. Many conditions can be written either using dialect/version conditions or using feature checks as described above. Feature checks are almost always preferable, as they better describe the intended meaning of the condition and do not need to be manually updated for new releases or other Prolog systems. Hard-coded version checks sometimes cannot be avoided though if there are behavior differences that a feature check cannot detect reliably or safely.

5.2 The SWI-Prolog/YAP emulation mechanism

Conditional compilation can be used to work around nearly all API differences and other compatibility problems. While this works well in cases where only a few incompatible features need to be used, it does not scale well for a large nonportable codebase like ProB, as every nonportable call in every source file would have to be wrapped in a conditional block. This can be solved to some extent by extracting the conditional code into a shared module, but this still requires manual changes to each source file that uses nonportable predicates.

An alternative solution that largely automates this process is the dialect emulation mechanism supported by SWI-Prolog and YAP (Wielemaker and Costa 2011). The basic concept of this mechanism is that Prolog code declares the "dialect", that is, Prolog implementation, for which it is written – for example, the directive `:- expects_dialect(yap).` indicates that the following code was written for YAP. If the declared dialect does not match the running Prolog system, a set of *emulation* libraries is loaded to replicate the behavior of the original system. This usually involves (re-)defining built-in predicates and

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press

operators, providing a different set of standard libraries, and translating unsupported language features using term/goal expansion.

This emulation mechanism is particularly useful for a codebase like ProB that was not originally written with portability in mind. Because the emulation mimics the original system's behavior as closely as possible, the original code often only needs few manual adjustments to run on the new system, compared to pure conditional compilation. Emulation cannot fully handle all differences though, and some complex cases still require manual fixes or conditional code.

SICStus Prolog itself does not support the `expects_dialect` mechanism, but this did not cause significant compatibility problems. As ProB supports SICStus natively, emulation is only needed on SWI-Prolog, so the `expects_dialect` directive can be simply skipped on SICStus.

When we began porting ProB, SWI-Prolog only provided emulation for SICStus Prolog 3 and not version 4. Although we were able to reuse some of the existing emulation, we had to implement many parts from scratch for the SICStus 4 emulation, because of significant differences in the standard libraries between the two versions. Internally, `expects_dialect` is based on plain Prolog modules, making it easy for Prolog programmers to extend it and enabling rapid prototyping of new emulation features.

5.3 Decoupling optional modules from the core of ProB

ProB includes a large number of features to support a diverse set of use cases. Beyond the core capabilities of animating and verifying B machines, ProB also for example supports visualizing various aspects of the machine, or working with other formalisms such as TLA$^+$ and Z. Not all of these features are needed at all times or for all use cases though.

Previously, ProB has always unconditionally loaded all of its modules, even when the code in question was not actually called. This did not cause problems running on SICStus Prolog, as almost all modules are pure Prolog code and thus always load successfully. For the few modules with problematic dependencies, ProB already supported compile-time flags or automatic detection to disable the module in question if the needed dependency is missing.

When we began supporting SWI-Prolog, this approach of unconditionally loading all features became a problem. Most of ProB's code initially did not load or even parse successfully on SWI-Prolog, usually because of SICStus-specific syntax and standard libraries, or C/C++ dependencies that were not made SWI-compatible yet. Fixing all of these errors at once would not have been realistic.

To address this, we began restructuring nonessential ProB code into separate *extensions* that can be selectively enabled or disabled. This is implemented using term expansion: ProB detects `use_module` directives that reference unavailable extensions and replaces them with stub definitions that safely display an error when called.

We have also introduced a global flag `prob_core_only`, which disables all optional extensions and leaves only required core modules enabled, to allow building/running a more minimal version of ProB. Most of our work to support SWI-Prolog was done with this flag set, so that we could first focus on solving compatibility issues in the core before gradually tackling the optional extensions.

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press

5.4 Compatibility issues between SICStus Prolog and SWI-Prolog

We will now present some specific notable differences and compatibility issues between SICStus Prolog and SWI-Prolog that we encountered while porting ProB, as well as the solutions or workarounds that we used.

5.4.1 Feature checks

Even though feature checks are commonly needed in compatibility code (Section 5.1), they are often tricky to perform in a way that works correctly on multiple Prolog systems.

The ISO core standard defines `current_predicate/1` for checking the existence of a given predicate, but according to the standard, it only succeeds for *user-defined* predicates. Although some implementations like SWI-Prolog also allow it to detect built-in predicates, others like SICStus Prolog strictly follow the standard and fail for built-ins, making this predicate unsuitable for feature checks. Instead, the nonstandard but widely supported `predicate_property/2` can be used. Although its intended purpose is to query information about a predicate, it can also be used to check whether a predicate exists at all, as it fails without error if asked about a nonexistant predicate.

Existence of libraries (and, more generally, source files) can be checked using the `exists_source/1` predicate on SWI-Prolog and YAP. This predicate is not supported by SICStus Prolog and many other systems, but can be emulated using the widely supported `absolute_file_name/3`[5].

Existence of arithmetic functions, such as `log(Base,X)`, can be checked using `current_arithmetic_function/1` on SWI-Prolog. No other Prolog system, including SICStus, provides a similar predicate, but a check such as `if(catch(_ is log(2,4), _, fail))` can be used instead.

5.4.2 Term output format

SWI-Prolog has full Unicode support in Prolog source code, whereas SICStus Prolog is limited to ISO 8859-1 characters outside of quoted literals and requires non-ASCII whitespace (e.g. nonbreaking space) to be escaped even inside quotes. This causes issues when terms written by SWI-Prolog are read by SICStus, as SWI will not quote or escape some characters as needed by SICStus.

SWI-Prolog `write_canonical` outputs list syntax to avoid exposing the changed list term format. This can cause issues with restrictive parsers that do not support list syntax, such as the one used internally by ProB 2 UI.

Both issues can be worked around by setting appropriate `write_term` options on SWI-Prolog.

Traditionally and on SICStus Prolog, term output predicates like `print` will never quote atoms and functors. SWI-Prolog will add quotes as needed so that the term can be correctly read back. This causes problems when using `print` to display human-readable text from atoms. `expects_dialect` automatically reconfigures SWI-Prolog to the SICStus-compatible behavior.

[5] `absolute_file_name(Source, _, [access(exist), file_type(source), file_errors(fail)])`

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press

5.4.3 Term hashing

Beyond the basic `term_hash/2`, both SICStus Prolog and SWI-Prolog support options to customize hashing, but with largely incompatible APIs. Overall, the SICStus Prolog `term_hash/3` is more flexible than the predicates provided by SWI-Prolog.

In the case of ProB, we mainly used conditional compilation to select the best available hashing predicate. As of writing, some of ProB's hashing calls are still more optimized for SICStus behavior and options, leading to hash collisions on SWI-Prolog in some cases.

A further problem is that although each system guarantees stable hashes across releases, the hashes are not interchangeable between the two systems, and a hash from one system cannot be reproduced on the other. This is problematic as ProB can save hashes for later verification, expecting them to be reproducible. We have not found a good solution for this yet – we may need to use a non-built-in, but portable hash function implemented in Prolog or C that produces consistent results on both Prolog systems.

5.4.4 Coroutining

Both systems support the common `when/2`, `dif/2`, `freeze/2`, and `frozen/2` predicates. Additionally, SICStus Prolog supports `block` declarations for coroutines, which are not natively supported by SWI-Prolog. However, `expects_dialect` provides an emulation of `block`, which we enhanced to meet the needs of ProB, specifically to allow querying blocked goals using `frozen/2`.

SWI-Prolog's `frozen/2` was recently made fully compatible with SICStus Prolog, but its implementation at first had serious bugs – calling `frozen` on a variable for example, could cause internal errors in CLP(FD) or silently make `when/2` coroutines not execute[6]. These bugs were detected by the ProB test suite and have since been fixed upstream.

5.4.5 SICStus Prolog implementation details

ProB relies on some undocumented SICStus APIs and implementation details, which naturally causes issues on other systems. This was often easily solved by switching to an equivalent documented API, either always or only conditionally when not on SICStus.

In an extreme case, ProB heavily uses implementation details of SICStus `library(avl)` – it often directly manipulates the undocumented term structure of AVL trees, for performance gains compared to the documented API. Replacing all dependencies on this internal term structure would have been difficult and error-prone. Instead, we decided to build a custom `library(avl)` implementation that uses the same internal term format as the SICStus version. Our custom implementation is written in pure Prolog, based on `library(assoc)` of SWI-Prolog. Although the custom implementation is only needed on SWI-Prolog, it is also fully compatible with SICStus, where it has similar performance as the built-in `library(avl)`.

[6] See https://github.com/SWI-Prolog/swipl-devel/issues/828.

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press

5.4.6 CLP(FD)

SICStus Prolog `library(clpfd)` provides an extensive API with many features not found in other implementations. SWI-Prolog `library(clpfd)` also has a number of advanced features, but not as many as SICStus Prolog. In particular, ProB frequently uses the SICStus "FD set" API, which was not supported on SWI-Prolog, so we contributed a compatible implementation of the API to SWI-Prolog.

There is a notable technical difference between the two `library(clpfd)` implementations: the SWI-Prolog version is implemented in pure Prolog and supports arbitrarily large integers. In contrast, the SICStus version is implemented in native code, resulting in better performance at the cost of being limited to the tagged integer range. When this range is exceeded, SICStus `library(clpfd)` throws an overflow error – sometimes in hard to predict places, due to coroutining, making it difficult to catch overflow errors reliably. The SWI-Prolog implementation does not suffer from this problem, but for some use cases it is noticeably slower than SICStus.

5.4.7 Command-line syntax

SICStus and SWI-Prolog use different command-line option names and sometimes require a different argument order. To run ProB easily on both systems, we wrote a wrapper shell script that supports the most important options and translates or reorders them as needed for each Prolog system.

SICStus Prolog allows passing *system properties* at the command-line using the syntax `-Dname=value`. ProB uses this feature to implement compile-time flags, for example, to enable debugging/profiling features or disable optional extensions. SWI-Prolog has no equivalent feature, so we are currently using regular environment variables as an alternative.

5.4.8 Minor incompatibilities

Finally, we want to mention a few incompatibilities that are important, but not complex, and were easily solved or worked around.

Operator declarations. Global on SICStus Prolog, but on SWI-Prolog they are module-local and have to be imported/exported or declared globally. `expects_dialect` solves this automatically. A few cases in ProB needed manual adjustment, but those were already questionable – for example, using an operator from a module that was never imported.

SWI-Prolog list term format. Required few changes overall, as almost all code uses list syntax and does not rely on the `'.'` list term functor. The difference was only noticeable in code that inspects free-form terms, such as term printing and type checking utilities.

Double-quoted literals. Represents a list of character codes traditionally, including on SICStus Prolog, but is a "true" string on SWI-Prolog by default. The meaning can be changed using the ISO standard Prolog flag `double_quotes` – setting it to `codes` forces the traditional behavior regardless of the default of the Prolog system.

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press

Mutable terms. SICStus Prolog has a specialized mutable term data type. SWI-Prolog allows mutating any compound term's arguments. `expects_dialect` can easily emulate the SICStus mutable term API.

Attributed variables. SICStus Prolog and SWI-Prolog implement two similar, but incompatible attributed variable interfaces. ProB handles these differences using conditional compilation. Although the SWI-Prolog attvar interface is less powerful than the SICStus Prolog one, this has so far not impacted ProB's use of attvars.

Simple naming differences. In many cases (too many to list), SICStus Prolog and SWI-Prolog provide identical predicates under different names, or with the same name in a different library module. In rare cases, both systems have identically named predicates with different behavior, such as `copy_term/2` (does not copy attributes on SICStus) or `lists:prefix/2` (arguments are reversed on SICStus). All of these differences are handled automatically by `expects_dialect`.

6 Empirical results

6.1 Current state of compatibility

At the time of writing, ProB's command-line interface is usable on SWI-Prolog, supporting interactive REPL usage, batch operations, and running ProB's test suite. However, because most noncore parts of ProB are currently disabled on SWI-Prolog (as explained in Section 5.3), a number of ProB features are not available yet, such as support for formalisms other than B and Event-B.

To estimate the completeness and correctness of ProB running on SWI-Prolog, we used ProB's integration test suite, which contains a total of 2129 tests. Of these tests, 954 require features that are disabled in core-only mode, and 47 further tests are currently disabled for other reasons.

This leaves 1128 tests that can currently be executed on both SICStus Prolog and SWI-Prolog. Most of these tests run successfully on SWI-Prolog, but 72 tests (6.4 %) still fail. Many of these test failures are because ProB on SWI-Prolog is unable to find a solution that it can find on SICStus Prolog. This makes ProB on SWI-Prolog less powerful than on SICStus Prolog, but does not affect correctness.

However, some test failures on SWI-Prolog are caused by ProB reporting false positives, for example, invariant violations even though the invariant is true. We were also able to construct a case that produces a false *negative*, where ProB on SWI-Prolog does *not* find a known error in the specification that is detected when running on SICStus. We are still investigating the underlying causes of these incorrect results. As ProB has previously encountered implementation bugs in SWI-Prolog related to coroutines[7], we suspect that a similar SWI-Prolog bug might be causing these failures, but they may also be due to remaining compatibility issues in the ProB code.

Until these correctness problems are fixed, ProB on SWI-Prolog on its own cannot be relied on yet. This is mitigated in a double toolchain setup, where a mismatch with SICStus would be detected and prompt further investigation of the results.

[7] Such as https://github.com/SWI-Prolog/issues/issues/105 (which has since been fixed).

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press

Table 2. *Performance benchmark tests of ProB*

	Walltime (ms)			Runtime (ms)				
Test ID	SICS	SWI	Factor	SICS	SWI	Factor	Cat.	Description
65	46	36	0.78	44	27	0.61	micro	Sequence solving
23	559	1015	1.82	416	571	1.37	micro	Sequence concatenation
800	3366	6110	1.82	1364	4446	3.26	cbc	Bosch cruise controller
1748	29,443	75,056	2.55	28,069	69,060	2.46	mc	MCCountToMAXINT50000
56	67	202	3.01	67	190	2.84	cs	Graph coloring
1635	860	3212	3.73	736	2949	4.01	dv	ClearSy/Alstom DTVT
1963	1374	5203	3.79	1328	4968	3.74	mc	Volvo cruise controller
778	6500	24,960	3.84	5523	22,992	4.16	dv	Siemens data validation
1336	2187	8947	4.09	2097	8505	4.06	cs	SlotSolver
1195	82	372	4.54	61	355	5.82	cs	Graph isomorphism large
255	392	1905	4.86	369	1679	4.55	mc	Bepi Colombo mode protocol
383	713	4185	5.87	694	4097	5.90	cs	Sudoku hex puzzle
414	56	329	5.88	43	285	6.63	cbc	BPEL2B PurchaseOrder
2015	4355	27,568	6.33	3425	26,606	7.77	dv	ClearSy Caval
55	238	1910	8.03	226	1868	8.27	cs	Sort by permutation
378	1691	14,160	8.37	1609	13,634	8.47	cs	NQueens as events
221	232	2000	8.62	228	1948	8.54	cs	Crew allocation puzzle
1920	643	6464	10.05	561	6224	11.09	mc	Performance of while loop
1745	15	180	12.00	15	165	11.00	cs	EulerWay
1394	833	11,996	14.40	824	10,952	13.29	dv	Alstom IXL Lausanne
49	2396	37,307	15.57	2249	32,460	14.43	cs	SAT test (flat200-90)
387	1467	25,982	17.71	1443	24,976	17.31	cs	TwoQueensSevenKnights
1715	627	15,790	25.18	620	14809	23.89	dv	Alstom while loop test
1746	14	581	41.50	13	528	40.62	cbc	JavaCard model
40	808	150,964	186.84	608	148,040	243.49	micro	Direct product operator

6.2 Performance comparison

Although our SWI-Prolog support is far from finished, it is complete enough that we can successfully run some complex test cases. This allows us to compare the performance of a real-world application running on SICStus Prolog and SWI-Prolog. However, the measurements have to be taken with a grain of salt: ProB has been developed and optimized for SICStus Prolog for over almost 20 years. As such, the comparison is somewhat unfair to SWI-Prolog. We expect that performance will improve in the future as we continue to test and optimize ProB on SWI-Prolog.

To evaluate the performance for individual applications of ProB, we selected a few of ProB's tests from the `codespeed` test category. This test category is used to detect performance regressions in ProB. The tests in Table 2 are further refined into five different categories: micro (micro benchmarks), cs (constraint solving), cbc (constraint-based symbolic verification checks), mc (model checking), dv (data validation on large data).

The experiments were run from source, using ProB's test runner. Each experiment is a test in ProB's test database (file `src/testcases.pl` in the ProB source code[8]). The test number can be found in Table 2. The tests were run using SICStus Prolog 4.7.0, SWI-Prolog 8.5.6, and ProB 1.12.0-nightly (`8ad8e874`) on macOS 12.1 on a MacBook Pro

[8] The ProB source code and example machine files used in the tests are available at: https://www3.hhu.de/stups/downloads/prob/source/

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press

(13" 2019, 2.8 GHz Quad-Core Intel Core i7). Table 2 contains both walltime and runtime (time excluding garbage collection or time spent in non-Prolog code).

As one can see in the above table there is a marked difference in performance in favour of SICStus Prolog. For two-thirds of the tests, ProB on SWI-Prolog is 3×–15× slower, and there is one outlier with even a 186× slowdown. On the positive side, the memory consumption (not in Table 2) remains similar. Furthermore, for applications where a double toolchain (i.e. the dv category; see Section 1.1.1) is required, runtimes are reasonable for a secondary validation toolchain: 3.84 slower for Siemens data validation or 6.33 slower for ClearSy data validation example.

As to the reasons for the current performance difference, we can only provide a few guesses. One reason is certainly that ProB makes heavy use of `block` coroutines in its constraint solver; these directives have efficient built-in support in SICStus Prolog, but are translated to regular Prolog code in SWI-Prolog. One might also think of the JIT in SICStus Prolog, but the first JIT version of SICStus Prolog actually made ProB run slower for some benchmarks and did not result in big gains. Performance of SWI-Prolog can vary depending on the C compiler and use of profile-guided optimization – we have not yet experimented which configuration yields best performance for ProB.

7 Related work

Wielemaker and Costa (2011), which introduced the `expects_dialect` mechanism, also presents a case study in which the Alpino parser suite (van Noord 2006) was ported from SICStus Prolog 3 to SWI-Prolog 6. The current version of Alpino[9] remains compatible with both Prolog systems, although it has not been updated to support recent versions of either system (SICStus Prolog 4 and SWI-Prolog 7/8).

Many of the issues described in the Alpino case study are similar to those that we encountered with ProB, such as library differences, lack of `block` coroutines in SWI-Prolog, and incompatible GUI libraries. When porting ProB, we were able to build on many parts of SWI-Prolog's SICStus Prolog emulation that were originally developed for Alpino. However, because of version differences, many parts of the emulation had to be adjusted or rewritten to support SICStus Prolog 4.

Logtalk (Moura 2013), an object-oriented Prolog derivative, is itself implemented in Prolog and highly portable, supporting 15 different "backend" Prolog systems. Programs written in Logtalk automatically benefit from this portability, as most Logtalk features are independent of the backend system. Not all libraries are abstracted this way though – for example, CLP(FD) is exposed largely unmodified from the backend. Although Logtalk provides much broader portability than other solutions like `expects_dialect`, it is less suitable for porting large amounts of previously nonportable Prolog code, as Logtalk is a significantly different language from regular Prolog.

Within the ASAP EU project, the SICStus Prolog-based tools ECCE and LOGEN (Leuschel *et al.* 2006) were made compatible with Ciao. At the time, conditional compilation was not available yet, so manual compatibility modules for each system were used instead.

[9] https://github.com/rug-compling/Alpino

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press

8 Conclusion and future work

We presented the process of making ProB, which was originally developed only for SICStus Prolog, also compatible with SWI-Prolog. Our motivations were, among others, a double toolchain for certification and ensuring long-term support.

Through the use of the `expects_dialect` mechanism supported by SWI-Prolog, we were able to do so *without* forking the codebase while keeping support for SICStus Prolog; many differences between the two systems were bridged automatically without large manual code changes. SWI-Prolog initially did not support SICStus Prolog 4 emulation, but we were able to contribute this support without much difficulty. Our additions to the emulation have been released as part of SWI-Prolog 8.4 and can be used by other developers looking to port software from SICStus 4.

Yet `expects_dialect` is not a one-line solution. There remain many differences between Prolog systems that cannot be handled automatically or are better handled by manual refactoring, and optimizations made for one Prolog system do not always translate well to other systems. Furthermore, porting a complex application to a new Prolog system is likely to uncover bugs in the targeted system – even with a popular and actively maintained one like SWI-Prolog.

We are planning to further improve SWI-Prolog compatibility and performance in ProB, with the eventual goal of fully supporting it as a second toolchain in addition to SICStus Prolog. In the future, we may also investigate supporting other Prolog systems, such as YAP (Costa *et al.* 2012) or Ciao (Hermenegildo *et al.* 2012).

Acknowledgements

We would like to thank Jan Wielemaker for his exceptionally quick responses on the SWI-Prolog forum and bug tracker and for being so open to improving SICStus Prolog compatibility in SWI-Prolog. We also thank Mats Carlsson and Per Mildner for the excellent support provided for SICStus Prolog over many years. Finally, we are grateful to ClearSy, Alstom and Thales for pushing and helping ProB to move towards T2 and T3 certification.

Competing interests.

The authors declare none.

References

Bendisposto, J., Krings, S., and Leuschel, M. 2014. Who watches the watchers: Validating the ProB validation tool. In *Proceedings of the 1st Workshop on Formal-IDE*. EPTCS 149.

Butler, M. J. et al. 2020. The first twenty-five years of industrial use of the B-method. In *Proceedings FMICS*. LNCS 12327. 189–209.

Carlsson, M. and Mildner, P. 2012. SICStus Prolog — the first 25 years. In *Theory and Practice of Logic Programming*. Vol. 12. 35–66.

Costa, V. S., Rocha, R., and Damas, L. 2012. The YAP prolog system. In *Theory and Practice of Logic Programming*. Vol. 12. 5–34.

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press

EN50128 2011. Railway applications – communication, signalling and processing systems – software for railway control and protection systems. European standard, CENELEC.

Haemmerlé, R. and Fages, F. 2006. Modules for Prolog revisited. In *Logic Programming*, S. Etalle and M. Truszczyński, Eds. Springer, Berlin, Heidelberg, 41–55.

Hermenegildo, M. V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J. F., and Puebla, G. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming 12,* 1-2, 219–252.

ISO/IEC 13211-1:1995 1995. Information technology — programming languages — Prolog — part 1: General core. Standard, ISO. 06.

ISO/IEC WDTS 13211–3 2021. Definite clause grammar rules. Working Draft TS N279, ISO. 08.

Körner, P. et al. 2022. 50 years of Prolog and beyond. arXiv:2201.10816, to appear in *Theory and Practice of Logic Programming*.

Leuschel, M., Elphick, D., Varea, M., Craig, S., and Fontaine, M. 2006. The Ecce and Logen partial evaluators and their web interfaces. In *Proceedings PEPM*. IBM Press, 88–94.

Moura, P. 2013. A portable and efficient implementation of coinductive logic programming. In *Practical Aspects of Declarative Languages*. LNCS, vol. 7752. Springer, 77–92.

van Noord, G. 2006. **A**t **L**ast **P**arsing **I**s **N**ow **O**perational. In *Proceedings TALN*. 20–42.

Wielemaker, J. and Costa, V. S. 2011. On the portability of Prolog applications. In *Practical Aspects of Declarative Languages*. Springer, Berlin, Heidelberg, 69–83.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. 2012. SWI-Prolog. In *Theory and Practice of Logic Programming*. Vol. 12. 67–96.

https://doi.org/10.1017/S1471068422000230 Published online by Cambridge University Press