Porting and refactoring Prolog programs: the PROSYN® *case study*

EDISON MERA

Process Design Center (PDC), Breda, The Netherlands
(*e-mail:* mail@edisonm.com)

JAN WIELEMAKER

VU University Amsterdam, The Netherlands
(*e-mail:* J.Wielemaker@vu.nl)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Operational software systems need to be maintained. Prolog has strengths and weaknesses if it comes to software maintenance. Its reflexive capabilities and the fact that Prolog programs can be represented naturally as Prolog data are strengths when it comes to porting and refactoring. On the other side, its dynamic and untyped nature limit the possibilities for static analysis and safe refactoring. In this paper we evaluate the practical aspects of these processes based on a large case study. The system studied is PROSYN®, an expert system consisting of almost 1 million lines of Prolog code written in IF/Prolog IV and C++ based on Microsoft MFC. This system has been ported to SWI-Prolog, while the Ciao assertion language is being used to overcome the lack of a type system, to formalize specifications and to establish the necessary static analysis and run-time checking framework.

KEYWORDS: porting, run-time checking, refactoring

1 Introduction

Some legacy Prolog applications contain a vast amount of knowledge. At the same time, the world changes with respect to available Prolog infrastructure, operating systems of choice and expected interaction with users. This changing environment can require major refactoring of the code. Nowadays there exists vast information about refactoring for different languages (Fowler 2002) with Prolog being one of the tools used for refactoring of applications written in other languages (Kniesel et al. 2007; Pau and Kristinsson 1990). In contrast, refactoring Prolog *applications* did not receive much attention, being (Serebrenik et al. 2008) one of the few exceptions.

This article describes the refactoring process of a huge expert system called PROSYN® (Schembecker and Simmrock 1996). It was originally developed by a large number of (mainly) chemical engineers over a 20 year period. The system still runs, but only inside a virtual machine running Windows-2000. The system can still be compiled, using IF/Prolog V, VISUAL FORTRAN and MSVC 6, but neither of which is still on sale. The knowledge captured in PROSYN® is considered valuable enough to justify investing substantial resources to port it to a modern infrastructure and make it ready for future maintenance and extension of the knowledge base.

The contribution of this paper is to demonstrate that a combination of previously described techniques, in particular the YAP/SWI portability framework (Wielemaker and Costa 2011), the `Ciao` assertion framework (Hermenegildo et al. 2012) and its run-time checker (Mera et al. 2009), together with some relatively simple new tools can support the porting and refactoring process of a huge system at acceptable costs. This case is interesting because of the size and poor state of software maintenance at the start of the project and it is carried out in a commercial setting.

This article is organized as follow: first, we introduce the PROSYN® system and aspects that are relevant to porting and refactoring. In section 3, we demonstrate how we used the YAP/SWI portability framework to reach at an initial port. Next, we describe why we need additional annotations about the program and how a modified port of the `Ciao` assertion language fulfills these requirements. In section 5 we introduce our custom refactoring tools before we end with related work and the conclusions.

2 The PROSYN® system

The PROSYN® system is a huge expert system for the design of chemical processes. The software provides an interesting use case for porting and refactoring because the software is (1) very large (almost 1 million lines), (2) written over a long period of time by many people to whom we have no access, (3) many of the authors were not professional computer scientists and (4) the original system (IF/Prolog IV) differs substantially from the current target system (`SWI-Prolog`), notably its module primitives and built-in predicates. PROSYN® consists of the following components:

Expert modules to support a given task in chemical process design. The code of these modules is a mixture of pure Prolog code, calls to the PROSYN® UI infrastructure, calls to the PROSYN® data sources infrastructure and rules that are subject to meta interpretation.

User interface subsystem that was initially developed for X11 and later ported to MFC (Microsoft Foundation Classes). It consists of high level calls that present various styles of input widgets (select from list, text field, etc.) and feedback (text and graphics).

Data sources about properties of chemical substances. Such data are stored on text files in a variety of formats, databases or provided by dedicated external programs, and are accessed by a mixture of pure Prolog programs, calls to external executables and C interfaces.

General infrastructure that takes care of saving and loading collected facts on the current session and meta-interpretation of rules. The core facilities also provide a subsystem that achieves demand loading of modules, to overcome memory limitations of the former hardware.

Portability framework PROSYN® was initially written in IF/Prolog IV on Unix/X11 and later ported to IF/Prolog V on Windows-NT/MFC. A large part of the Prolog portability issues was resolved by renaming predicates to *V4, e.g., `clauseV4/2` throughout the application and providing a module that emulates the version IV functionality in version V. The X11/MFC portability was realized using a module with generic GUI calls that are used throughout the application. Depending on a setting, dynamic predicates are generated that relay these calls to an X11 or MFC layer.

IF/Prolog V comes with a module system that is close to ISO/IEC 13211-2:2000 (Prolog – Part 2: Modules). Meta-calling is facilitated by a directive `meta/1`, which behaves similar to the deprecated `SWI-Prolog` `module_transparent/1` directive[1]. In addition, it uses `:/2` to qualify

[1] `http://www.swi-prolog.org/pldoc/doc_for?object=(module_transparent)/1`

predicates and `@/2` to provide a calling context for goals. For example, `m:p@c` executes the predicate `p/0` of the module `m` in the calling context `c`. Modules have a distinct declaration section that provides the module name, exports declared using `export/1` and additional declarations such as `dynamic/1`. In contrast, the `SWI-Prolog` module system is based on Quintus Prolog.

A disadvantage of the `meta/1` declaration is that it does not specify *which* argument is used as a module sensitive argument, nor whether arguments are used as calls or not. [2] The PROSYN® infrastructure defines a large collection of meta-predicates. Lacking meta-predicate information that allows for locating arguments that are goals, even simple program analysis such as dependency tracking becomes a challenge.

Many predicates in PROSYN® have meta declarations while their arguments are not module sensitive or have dynamic declarations while they are not asserted nor retracted. Some of the dynamic declarations seem to be related to the debugging facilities of IF/Prolog. It is unclear why the superfluous meta declarations appear in the program.

3 Porting PROSYN®

The minimal requirement to revive PROSYN® was to migrate it to currently available compilers running on a currently available operating system. In addition, the owner (PDC) wants to replace the UI with a web-based UI to be able to offer PROSYN® as a service to its customers and it wants to make it easier to inspect and extend the knowledge captured in the system. We considered two options: (1) restart from scratch after selecting an appropriate expert system shell or (2) realize a minimal port to a modern environment and perform gradual refactoring steps from there.

It was estimated that it would require too many resources and there would be a too long period without any usable result at all to make (1) a serious option. Instead, we decided on a pilot project of three weeks, carried out by one of the authors (Wielemaker) and Johan Romme, who had studied the existing code base and has basic knowledge about running PROSYN® as well as building PROSYN® using the original infrastructure. The purpose was to port the core of PROSYN® to `SWI-Prolog` following the emulation technique described in (Wielemaker and Costa 2011). Below we briefly introduce the compatibility framework and the results of this pilot.

3.1 The SWI/YAP portability framework

The SWI/YAP portability framework introduces a directive `:- expects_dialect(Dialect).` that states that the file is written for the given Prolog *Dialect*. If the target Prolog is *not Dialect*, it performs the following steps: (1) load the file library(dialect/*Dialect*), and (2) make the expected dialect available to macro expansion hooks through the predicate `prolog_load_context/2`.

The file library(dialect/*Dialect*) typically pushes a directory holding a (partial) library that is compatible to *Dialect* to the front of the `library` search path. In addition, it exports emulations for built-in predicates of the target system and rewrites goals and directives conditionally on the expected dialect using `term_expansion/2` and `goal_expansion/2`. For example, calls to conflicting built-in predicates are mapped like this:

[2] Most todays Prolog systems whose module system is derived from Quintus use the `meta_predicate/1` declaration where an integer in the head implies that the argument is *called* with *N* additional arguments and a `:` annotation implies that the argument is module-sensitive without specifying how.

```
goal_expansion(concat_atom(List,Delim,Atom),if_concat_atom(List,Delim,Atom)) :-
    prolog_load_context(dialect, ifprolog).
if_concat_atom(List,Delim,Atom) :-
    maplist(write_term_to_atom,List,AtomL),atomic_list_concat(AtomL,Delim,Atom).
write_term_to_atom(Term, Atom) :-
    ( atomic(Term)->Atom = Term ; with_output_to(string(Atom), write(Term)) ).
```

The power of the SWI/YAP compatible framework is amplified by the flexibility of `SWI-Prolog`. `SWI-Prolog` poses few restrictions on source code layout. For example, most directives may be executed as goals, often even after providing clauses for a predicate, system predicates may be redefined, it is possible to make cross-module calls to private predicates and definitions can in general be resolved lazily. While these features give great freedom to the programmer, they make program analysis from the source code nearly impossible. For this reason, analysis of `SWI-Prolog` programs is executed on the *loaded program* rather than the source.

The typical process to port a Prolog application that was originally written for a system for which no portability framework exists consists of the steps below, repeating steps 2 to 4 until the program is loaded without errors: (1) create an initial library(dialect/*Dialect*) and *Dialect* library directory, (2) try loading the main file, (3) if library files are reported missing, add a dummy library file to the dialect library, (4) try to resolve issues with conflicting built-in predicates, directives, operators and other syntax issues by adding rules to library(dialect/*Dialect*).

After these steps, missing predicates can be identified using `list_undefined/0` and can be added to library(dialect/*Dialect*) as well as the compatibility libraries. From this moment, (static) analysis and testing must be used to identify further issues. Section 4 elaborates on this process after the obvious issues were resolved by running and debugging the program.

3.2 Evaluation of porting PROSYN®

Porting PROSYN® turned out to be relatively complex. The main file provides only a minimal infrastructure. The interesting parts of the system are loaded on demand as explained in section 2. It took about a week with two persons to get enough of IF/Prolog emulated to get the first operational module *loaded*.

The aim with PROSYN® was to *migrate* rather than to *port* it, i.e., the result did not need to run on IF/Prolog anymore. This implies that the port can be realized as a mixture of emulation and rewriting, with a delicate tradeof between both. Emulation is generally more complicated, but has two advantages: (1) it leaves the code as-is, so it is guaranteed not to introduce new bugs into the existing code base and (2) it deals with all places where the construct is used together.

The disadvantage is that the code remains in its poor state using the *V4 (see section 2) emulation of IF/Prolog IV and the lack of proper meta-predicate declarations limit the usability of the `SWI-Prolog` IDE tools. Notably they prevent the editor to jump to source code and often make the source-level debugger revert to showing decompiled code rather than the original source because it fails to relate the compiled code to the detailed source layout.[3] In practice, we made a quick assessment for each predicate, considering the following factors: (a) the number of places it was used, (b) estimated time to create an emulation, (c) is a *complete* emulation easy and possible? (d) estimated time to rewrite the calling clause, (e) estimated risk to introduce errors with either approach, (f) estimated consequences of such errors, e.g., how easy is it to detect them?

[3] Just adding meta-predicate declarations instead of the module_transparent declarations does not work because it causes the run-time system to qualify the meta-argument with `:/2` in an implementation not prepared to handle such `:/2` term.

It took two weeks with two developers to reach at a state where we could execute some partial examples from the PROSYN® manual. During this period we implemented the IF/Prolog dialect emulation, some of which required low-level modifications to `SWI-Prolog` to make the emulation precise. Below are our most relevant conclusions.

- The PROSYN® code frequently uses `include/1` to include files. This was poorly supported in `SWI-Prolog`. Notably it was not possible to have the module header in an included file and detailed source-code location information was not available for included code. Although it is rather easy to replace `include/1` by one of other `SWI-Prolog` loading predicates, it was eventually decided to improve the support for `include/1`.
- The `@/2` was added to the kernel, it calls a qualified predicate with a given calling context[4], being too critical in the original code that partial emulations caused too many subtle errors.
- Code that was calling external executables and MFC code was generally rewritten. Full emulation was hard or impossible due to its dependency on MFC, which is not available on Linux, or its dependencies on Windows path-names. Many cases allowed for simple replacements with `SWI-Prolog`'s operating system access primitives. In some places, external C-code to access specific data sources was replaced with a pure Prolog alternative.
- The X11/MFC user interface switch was extended with two alternatives, one with predicates to provide terminal/text-based interaction and other to provide a web-based interface.

4 Debugging the PROSYN® system

As described in section 3.2, two weeks of classical debugging achieved that only some basic functionality became operational. The code was unstable. Some of the errors had been introduced long ago while porting to IF/Prolog V and Windows, others by incomplete emulation of IF/Prolog and some by reimplementing the user and operating system interaction. This, combined with a lack of knowledge about the program, implementation and program comments that are largely in German and often outdated caused progress to be slow. With the first author of this article (Mera) taking over the project, the situation got even worse because he was less familiar with the tools and the system at that moment. There was need for a more automated process.

Static analysis is a candidate to reduce the need for the laborious run-time debugging route. Despite the progress in the field of static analysis, software validation and run-time verification, there is a lack of even basic static analysis tools for Prolog programs, targeted at large systems. Rigorous static analysis of PROSYN® is impossible due to the lack of machine readable annotations of the sources as well as the lack of a scalable effective global analysis framework.

The `Ciao` assertion language (Hermenegildo et al. 2005) is one of the strongest systems for describing program properties and doing both static and run-time verification of these properties. We established a subset of the `Ciao` framework that scales and is useful even if only a tiny fraction of the code is annotated with assertions: the `Ciao` run-time checker and a subset of static validation tools. This was realized with a mixture of porting using the YAP/SWI portability framework and reimplementation. In the subsequent section we provide more details on the simplified `Ciao` assertion language that we use. We provide examples on how the assertion system helped in making explicit program properties and debugging PROSYN®.

[4] `http://www.swi-prolog.org/pldoc/doc_for?object=(context_module)/1`

4.1 The Ciao Assertion Language

The Ciao assertion language (Hermenegildo et al. 2005) allows expressing computational properties of a program that can be run-time checked. Currently, we only use the class of pred *assertions*, which describes a particular predicate and, in general, follows the schema
`:- pred` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-Props*]`.`
where *Pred* is a predicate symbol applied to distinct free variables and *Precond* and *Postcond* are logic formulae about execution states. *Precond* is the precondition under which the pred assertion is applicable. *Postcond* expresses that in any call to *Pred*, if *Precond* holds in the calling state and the computation of the call succeeds, then *Postcond* also holds in the success state. Finally, the *Comp-Props* field is used to describe properties of the whole computation of the calls to predicate *Pred* that meet *Precond*. For example, the assertion:

```
:-pred p(A,B):(list(A,num),var(B))=>(list(A,num),list(B,num))+(not_fails,is_det).
```

states that for any call to predicate p/2 with the first argument bound to a list of numbers and the second one a free variable, if the call succeeds, then the second argument is also bound to a list of numbers. Additionally, not_fails and is_det express that the previous calls do not finitely fail (i.e., they produce at least one solution or do not terminate) and are deterministic.

An instrumental part of the implementation is the *assertion language reader*, which reads assertion declarations, normalizes them, and save them as a set of easy to handle Prolog facts. Differences in the module system as well as the fact that assertions only depend on the compiled file in Ciao, but on the the whole program in SWI-Prolog made us decide to implement a dedicated assertion reader whose syntax is more inspired by SWI-Prolog's PlDoc conventions. We also allow to define the same set of global properties for several predicates. The example below states that the predicates p/1, q/2 and r/2 are det (the composition of not_fails and is_det), and the arguments of r/2 are compatible with int/1 and atm/1:

```
:- pred [p/1, q/2, r(int, atm)] is det.
```

4.1.1 Run-time checking port to SWI-Prolog

The run-time checking (Mera et al. 2009) for Ciao was implemented as part of the CiaoPP unified framework for static-verification, run-time checking and unit-testing. We ported the run-time checker using the portability framework described in section 3.1. The existing minimal Ciao dialect support was extended with the following features:

Basic emulation of the Ciao package system (Gras and Hermenegildo 2000). This was realized by (1) mapping `:- use_package(Alias)` to `:- include(Alias)`, (2) emulation of `:- load_compilation_module(Alias)` by loading it into a unique context to avoid name conflicts with already imported predicates and (3) emulate `:- add_goal_trans/2` and `:- add_sentence_trans/2` on top of goal_expansion/2 and term_expansion/2.

Argument rearranging of meta-arguments in meta-predicates In Ciao the arguments of a meta-predicate are interpreted differently, by moving the first meta-argument to the first position to facilitate indexing, however it is not compatible with the ISO standard.[5] For example, `call(p(a,b),c,d,e)` must be converted to `p(c,a,b,d,e)`.

[5] ISO/IEC 13211-1:1995/Cor 2:2012., section 8.15.4 call/2..8

Message printing infrastructure The run-time checking system generates messages using Ciao's `io_aux:messages/1` predicate. Proper integration with the `SWI-Prolog` IDE tools requires usage of the Quintus-derived message infrastructure realized by `print_message/2`, and was implemented with conditional compilation.[6]

4.1.2 Using the assertion language for debugging PROSYN®

Often, we want to detect why a predicate is failing silently. The way to debug this is to add an assertion with the right types and global properties to inform the system about the expected behavior of the program. Consider the program below:

```
1   :- module(m, [p/1, q/1]).   3   p(a).
2   q(a) :- p(b).               4   p(1).
```

We would like to see why a call to `q/1` fails. Suppose we also know that `p/1` only accepts numbers and should not fail. We can now add the following assertions:

```
5   :- pred q/1 is multi.
6   :- pred p(+int) is multi.
```

By executing the program `q/1` we see:

```
?- call_rtc(q(A)).
ERROR: /tmp/m.pl:5: Run-Time failure in assertion for m:p(_1).
ERROR: In *calls*, unsatisfied properties: [int(_1)]. Because: ['_1'=b].
ERROR: /tmp/m.pl:3: Failed in m:p(_1).
ERROR: /tmp/m.pl:2: Failed during invocation of q(_)
ERROR: /tmp/m.pl:6: Run-Time failure in assertion for m:q(_1).
ERROR: In *comp*, unsatisfied properties: [multi]. Because: [fails(q(_1))].
ERROR: /tmp/m.pl:2: Failed in m:q(_1).
```

`call_rtc/1` is a meta-predicate used to report all the violations that where found. In this case two run-time errors are reported, one related with the `multi` property and other with the type of `p/1` argument (`int`). The `Ciao` assertions can also be used for static analysis, currently a small set of assertions that can be checked at compile time, by compiling the previous example we get:

```
?- use_module(m).
ERROR: /tmp/m.pl:6: In assertions for [m:p/1]
 /tmp/m.pl:2: Compile-Time failure in assertion for m:p(a).
 In *compat*, unsatisfied properties: [int(a)].
ERROR: /tmp/m.pl:6: In assertions for [m:p/1]
 /tmp/m.pl:4: Compile-Time failure in assertion for m:p(b).
 In *compat*, unsatisfied properties: [int(b)].
```

4.1.3 Improving the Knowledge Base with assertions

The assertion language also allows for specifying application specific properties about predicates, for example the rules as they appear in PROSYN®. We improved the consistency of the knowledge base by providing declarations about its configuration as assertions rather than the ad-hoc multi-file facts used in the original PROSYN® implementation. This formalization as assertions allows for checking the rule syntax statically and, in terms of performance is lighter. For example, the original code contained the following declarations (translated from German):

[6] Due to `Ciao` and `SWI-Prolog` have different message infrastructures, a more elegant integration was difficult.

```
explainable_rules(_, [rule1(_,_),rule2(_,_),rule3(_,_,_)]).
target_mask(rule1(A,_),rule1(A,_)). target_mask(rule3(A,B,_),rule3(A,B,_)).
hidden_rules([rule1(_,_),rule2(_,_)]).
```

These declarations were translated into the following assertions:[7]

```
:- true pred [[rule1/2 + kbmask([+,-]), rule2/2] is kbhidden,
              rule3/3 + kbmask([+,+,-])] is kbrule.
```

For PROSYN®, 200 assertions where generated by refactorizing the knowledge base, and some inconsistencies in the facts where made evident and fixed. The number of modified lines of code was approximately 3000 and the number of lines removed 1000.

5 The Refactoring Tool

We have three key requirements for our refactoring tool. First, it must be capable of dealing with the fact that the program state cannot be derived easily from the sources but needs to be analyzed in the materialized (loaded) program as explained in section 3.1. Second, it must be capable of dealing with meta-calling and third, it must scale well. Our strategy is to provide generic primitives that match Prolog terms in the compiled code and provides rules similar to `term_expansion/2` and `goal_expansion/2`, where we can use the full power of Prolog to construct the target term based on the context and reflexive capabilities of Prolog. This tool uses the source-location capabilities of `SWI-Prolog` to replace the matching term in the source files[8] The primitives are described informally below. They operate on sentences (i.e., declarations and clauses) or partial sentences in the source and therefore cannot easily split or join sentences.

expand_term(*Module:Sentence, Pattern, Replacement, :Expander, +Action*) *[det]*
In all modules that match *Module*, in all sentences that subsume *Sentence*, replace each subterm that subsumes *Pattern* with the term *Replacement*, provided that the goal *Expander* succeeds. *Expander* can be used to finalize the shape of *Replacement* as well as to veto the expansion (see section 5.2 for an example). The *Action* argument is one of `show` to show the changes that *would* be made or `save` to actually apply the changes.

expand_goal(*Module:Caller, M:Goal, Replacement, :Expander, +Action*) *[det]*
Similar to `expand_term/4`. Replace in each clause whose head subsumes *Caller* each call to *Goal* that resolves to a predicated defined in the module *M*. It uses the code walker to find goals and resolve the predicate called by *Goal*.

expand_sentence(*Module:Pattern, Replacement, :Expander, +Action*) *[det]*
Similar to `expand_term/4`, but only applies to the entire sentence.

Prolog is a powerful term rewriting language, that enables us to define both complex generic and application-specific transformations in a few lines of code, as we will see further, for example, some possible replacements are shown in table 1, for the given program:

5.1 Scenarios

This section demonstrates the power of our primitives and how these predicates suffice for most refactorings, first on general refactoring tasks that can also be found in e.g., (Serebrenik et al. 2008) and then in section 5.2 for application-specific transformations.

[7] The `Ciao` assertion framework allows for the definition of new properties using the `:- prop` declaration.
[8] `http://www.swi-prolog.org/pldoc/doc_for?object=(prolog_walk_code)/1`

Program	Command	Replacements

`:- module(p1, []).`
`e(A,Z):-f(t(A/*1*/),t(A/*2*/)),`
`p(Z).`
`f(A,B):-p(A),p(B).`
	`expand_term(p1:(e(_,_):-_),`
	`f(X,Y),f(Y,X),true,show).`
		`-e(A,Z):-f(t(A/*1*/),t(A/*2*/)),`
		`+e(A,Z):-f(t(A/*2*/),t(A/*1*/)),`

	`expand_term(p1:_,`
	`f(X,Y),f(Y,X),true,show).`
		`-e(A,Z):-f(t(A/*1*/),t(A/*2*/)),`
		`+e(A,Z):-f(t(A/*2*/),t(A/*1*/)),`
		`-f(A,B):-p(A),p(B).`
		`+f(B,A):-p(A),p(B).`

Table 1. Possible replacements made by `expand_term/4` depending on the specified scope.

Removal of superfluous exports A method to remove the export declaration in predicates that are not being used in any place follows:

```
remove_useless_exports(M,Action) :-
  expand_sentence(Module:(:-module(M0,L)),
    (:-module(M0,N)),include(is_used,L,N),
    Action),
  expand_sentence(M:(:-export(K)), Exp,
    (list_sequence(L,K),include(is_used,L,N),
     (N==[]              ->Exp='$RM'
     ;list_sequence(N,S),Exp=(:-export(S)))),
    Action).
```

```
list_sequence([E|L],S) :-
  l_sequence_2(L,E,S).
l_sequence_2([E|L],E0,(E0,S)):-
  l_sequence_2(L,E,S).
l_sequence_2([],E,E).
is_used(F/A,M):-functor(H,F,A),
  predicate_property(_:H,
    imported_from(M)),
  M\==user.
```

`'$RM'` is an escape term that stands for removal of the whole term.

Variable renaming The renaming requires that there are no other variable with the same name in the sentence being transformed. The predicate `refactor_context/2` provides access to additional context information. The context `variable_names` provides the variable name dictionary as returned by the `variable_names` option of `read_term/3`.

```
rename_variable(Sentence,Name0,Name,Action) :-
    expand_term(Sentence,Var,'$VAR'(Name),
      ( refactor_context(variable_names, Dict), \+ memberchk(Name =_,Dict),
        var(Var), memberchk(Name0=V,Dict),V==Var), Action).
```

Conjunction replacement To perform replacements in a conjunction of goals we have to consider two cases: one if the target is at the end of the body and other if it is in the middle.

```
replace_conjunction(Sent,Conj,Repl,Act):-
  expand_term(Sent,Conj,Repl,true,Act),
  extend_conj(Conj,Rest,Conj2),
  extend_conj(Repl,Rest,Repl2),
  expand_term(Sent,Conj2,Repl2,true,Act).
```

```
extend_conj(Var,Rest,(Var,Rest)):-
  var(Var),!.
extend_conj((A,C0),Rest,(A,C)):-
  !,extend_conj(C0, Rest, C).
extend_conj(Last,Rest,(Last,Rest)).
```

5.2 An application specific refactoring in PROSYN®

PROSYN® contains the predicate `clauseV4/2`, which is part of the IF/Prolog IV to V migration. It acts as `clause/2` if the argument is qualified and operates on the module `xpsfact` otherwise. It is defined as:

```
clauseV4(M:Head,B):-!,clause(M:Head,B).|clauseV4(Head,B):-clause(xpsfact:Head,B).
```

Below are the rules that unfolds calls to this predicate if it is safe to do so, and the table with the replacements performed in `clauseV4/2` and similar predicates.

```
expand_goal(M:_,xif4:clauseV4(H0,B),clause(H,B)resolve_head(H0,M,H),save).

resolve_head(H,    _, _) :- var(H),!,fail.      % unknown head
resolve_head(M:_,  _, _) :- var(M),!,fail.      % unknown module
resolve_head(M:H,  M, H).                       % current context
resolve_head(H,    _, H) :- H = (_:_).          % qualified
resolve_head(H,  xpsfact, H).                   % context is xpsfact
resolve_head(H,  _, xpsfact:H).                 % otherwise
```

Predicate	Replacement	Amount	Predicate	Replacement	Amount
clauseV4/_	clause/2	7680	retractallV4/_	retractall/1	3090
assertaV4/_	asserta/1	1370	assertzV4/_	assertz/1	3300
atomic_lengthV4/2	write_length/3	460	list_length/2	length/2	1240
executeV4/1	*meta-call unfolded*	260	**Total**		17400

6 Related work

The most relevant work is described in "Improving Prolog programs: Refactoring for Prolog" (Serebrenik et al. 2008). Such article describes ViPReSS, a refactoring tool written in SICStus Prolog and evaluation of this tool in a 53,000 lines in-house application. The article provides an overview of refactoring operations and pointers to more literature with more comprehensive refactoring operations. ViPReSS system analyses the source code and creates a series of vi/ed edit commands to modify the source. In contrast, our discovery and source code location is based on the loaded executable to overcome the difficulties that result from program transformation to support the compatibility framework. Where ViPReSS cannot deal with meta-predicates, our infrastructure must be able to take care of this to support refactoring in PROSYN®. Our set of available refactoring operations has evolved on 'as needed' basis and is less comprehensive than what is available in ViPReSS.

7 Conclusions

This paper describes our experience with large-scale refactoring of a huge legacy Prolog program. We evaluated the YAP/SWI portability infrastructure two times for this project. First, it was applied by the lead developer (Wielemaker) of SWI-Prolog to port PROSYN® from IF/Prolog V and next, it was applied by Mera to port the Ciao assertion language. We can safely conclude that a dynamic Prolog implementation such as SWI-Prolog can accommodate code written for other systems at modest costs, although the process does require experienced Prolog programmers.

A test framework was realized using a combination of SWI-Prolog's unit test framework and Ciao assertion language based framework. Ciao assertions are being added to the code base incrementally to help debugging modules of PROSYN®. Finally, a flexible and lightweight infrastructure for performing refactoring tasks was developed on top of the SWI-Prolog source information primitives and its call-graph analysis. The refactoring process can be expressed naturally using Prolog rules, leaving the responsibility for minimal and syntactically correct transformations of the source to the tool. It perform fast changes in huge amounts of code, for example, in a desktop PC (i7 3.4GHz), a simple term replacement over all PROSYN® sources that modified 156 files in 7743 places took 33 seconds.

The presented work demonstrates that the integration of existing tools from SWI-Prolog and Ciao, together with a new refactoring tool, provides a good basis for porting and refactoring of large and poorly structured programs. The new PROSYN® is well on track with implementing proper quality measures and is being tested by in-house users.

The Ciao assertion port to SWI-Prolog, its run-time checker, and the refactoring tools are LGPL SWI-Prolog Packages, availables as repositories at: https://github.com/edisonm/

References

FOWLER, M. 2002. Refactoring: Improving the design of existing code. In *XP/Agile Universe*, D. Wells and L. A. Williams, Eds. Lecture Notes in Computer Science, vol. 2418. Springer, 256.

GRAS, D. C. AND HERMENEGILDO, M. V. 2000. A new module system for prolog. In *Computational Logic*, J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, Eds. Lecture Notes in Computer Science, vol. 1861. Springer, 131–148.

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. *TPLP 12,* 1-2, 219–252.

HERMENEGILDO, M. V., PUEBLA, G., BUENO, F., AND LÓPEZ-GARCÍA, P. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and the ciao system preprocessor). *Sci. Comput. Program. 58,* 1-2, 115–140.

KNIESEL, G., HANNEMANN, J., AND RHO, T. 2007. A comparison of logic-based infrastructures for concern detection and extraction. In *Proceedings of the 3rd workshop on Linking aspect technology and evolution*. LATE '07. ACM, New York, NY, USA.

MERA, E., LÓPEZ-GARCÍA, P., AND HERMENEGILDO, M. V. 2009. Integrating software testing and run-time checking in an assertion verification framework. In *ICLP*, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer, 281–295.

PAU, L. F. AND KRISTINSSON, J. B. 1990. Softm: A software maintenance expert system in prolog. *Journal of Software Maintenance: Research and Practice 2,* 2, 87–111.

SCHEMBECKER, G. AND SIMMROCK, K. H. 1996. Heuristic-numeric process synthesis with prosyn. In *AIChE*. Symposium series, vol. 92. American Institute of Chemical Engineers, New York, NY, 275–278.

SEREBRENIK, A., SCHRIJVERS, T., AND DEMOEN, B. 2008. Improving prolog programs: Refactoring for prolog. *TPLP 8,* 2, 201–215.

WIELEMAKER, J. AND COSTA, V. S. 2011. On the portability of prolog applications. In *PADL*, R. Rocha and J. Launchbury, Eds. Lecture Notes in Computer Science, vol. 6539. Springer, 69–83.