Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Máster Universitario en Métodos Formales en Ingeniería Informática

Trabajo Fin de Máster

A System for generating interactive tutorials for CiaoPP

Autor: Daniela Ferreiro de Aguiar
Director y colaboradores:
Manuel Hermenegildo Salinas,
José Francisco Morales

Madrid, 13 de febrero de 2023

Este Trabajo Fin de Máster se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Máster
Máster Universitario en Métodos Formales en Ingeniería Informática

Título: A System for generating interactive tutorials for CiaoPP

13 de febrero de 2023

Autor(a): Daniela Ferreiro de Aguiar
Tutor(a): Manuel Hermenegildo Salinas
Departamento de Inteligencia Artificial
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

Uno de los grandes retos para los métodos formales es resolver el problema de fiabilidad del software. El concepto de métodos formales hace referencia a técnicas y herramientas basadas en procedimientos matemáticos que se utilizan para especificar, validar y verificar sistemas de software. Actualmente, estas técnicas matemáticas no están integradas en la industria del desarrollo software y es debido principalmente a la falta de conocimiento sobre los métodos formales entre los profesionales. Es por ello que las universidades y centros de investigación tienen un papel esencial para el desarrollo de herramientas capaces de proporcionar el entrenamiento necesario para que estudiantes y profesionales puedan hacer uso de estos métodos.

Hemos escogido el sistema `Ciao` ya que incluye un conjunto único de herramientas para solucionar los retos planteados anteriormente. Entre estas se incluye `CiaoPP`, una herramienta automática de análisis estático que permite inferir propiedades del software y localizar errores combinando técnicas estáticas y dinámicas. En esta tesis de máster proponemos una arquitectura para la generación de tutoriales de la herramienta `CiaoPP` que proporcionen contenido tanto interactivo como estático. La generación de este contenido es posible gracias a la existencia de filtros aplicados por la herramienta `exfilter` para obtener ejemplos y ejercicios individualizados. También haremos uso de otras herramientas de `Ciao` como `Ciao` Playground y `LPdoc`.

Abstract

One of the grand challenges for formal methods is to solve the software reliability problem. The notion of formal methods refers to techniques and tools based on mathematical procedures that are used to specify, validate and verify software systems. Currently, these mathematical techniques are not integrated in the software development industry and it is mainly due to the lack of knowledge about formal methods among professionals. That is why universities and research institutes have an essential role to play in the development of tools capable of providing the necessary training so that students and professionals can make use of these methods.

We have opted for the `Ciao` system as it includes a unique of set of tools for facing the challenges mentioned above. These include `CiaoPP`, an automatic static analysis tool that infers software properties and locates errors combining static and dynamic techniques. In this master thesis we propose an architecture for the generation of tutorials of the `CiaoPP` tool that provides both interactive and static content. The generation of this content is possible thanks to the existence of filters applied by the `exfilter` tool to obtain individualized examples and exercises. We also make use of other tools such as `Ciao` Playground and `LPdoc`.

Contents

1. Introduction **1**
1.1. Structure of the document 2

2. The Ciao System **5**
2.1. CiaoPP: The Ciao Program Processor 5
2.2. Assertions 8
2.3. LPdoc 9
2.4. The Ciao Prolog Playground 10

3. An Interactive Documentation System for tutorials **13**
3.1. The exfilter tool 13
3.1.1. Filters 14
3.1.2. Algorithm 15
3.1.3. Incorporating automated feedback using exfilter 16
3.2. Architecture of the system 17
3.3. exfilter in the Ciao Playground 18

4. An interactive tutorial for CiaoPP **23**
4.1. Tutorial Content 24
4.1.1. Introduction 24
4.1.2. Notation 25
4.1.3. Using the CiaoPP Program Processor 26

5. Experimental Evaluation **37**
5.1. Static code generation 37
5.2. Interactive dynamic content 37

6. Conclusions **41**
6.1. Future work 42

Bibliography **47**

Chapter 1

Introduction

Modern software is growing in size and complexity in its demand for functionality and performance, making the *software reliability problem* more significant than ever [12]. The software industry is becoming more aware of this difficulty and is starting to pay attention to formal methods.

Formal methods address this problem by using rigorous mathematical techniques for describing, verifying, and analyzing software systems. Even though formal methods have been advocated as a means of increasing system reliability, use in the computing industry of these mathematical formalisms is limited in practice [2]. Several reasons have been identified [11, 3]: there are not enough programmers with the appropriate training to make use of them, although its relevance in education is gradually growing [18, 32]. Other reasons are the complex notation and the difficulty in mastering the support tools. We should also take into consideration the fact that there are some misconceptions about the use of formal methods [26].

The most frequently mentioned way of resolving this situation is a reform of education in order to provide the next generations of students with sufficient background and practical experience in formal methods [8, 47, 20]. However, teaching these techniques faces a number of challenges. On the one hand, despite the fact that the amount of literature on teaching formal methods has been growing, these texts are often aimed at researchers. Additionally, there is relatively little material covering certain topics, and in particular program analysis. In addition, the majority of the available material usually resides in conventional textbooks and it is not straightforward to convert it to more compelling channels such as interactive systems. The 2020 expert survey on formal methods [20] considers that universities and research institutes have a central role to play in the construction of such software tools and make sure that industry professionals and students can learn about formal methods.

Modern learning tools generally make use of advanced technology to produce content and interactive user feedback, so that they can be up to a certain degree described under the term intelligent tutoring systems (ITS) [5, 48]. Since these systems typically need to encode large amounts of knowledge, it is important to determine what to teach, as well as when and how to teach it. Most of the intelligent techniques used for this purpose can be classified into three groups [5, 4]:

1. **Curriculum sequencing.** Since every student is different, it is often interesting to have a non-linear curriculum depending on the abilities and capabilities of

each student. Such systems construct for each student an individual learning path.

2. **Intelligent analysis of student solutions.** A solution analyzer's objective is to decide whether the solution is correct or not, find the error made by the student, and identify possible causes in order to help the student correct it.
3. **Interactive problem-solving support.** This last technique provides students with personalized and intelligent help on each problem-solving step. The level of assistance can vary: from highlighting an error, to giving a hint, or to executing the next step for the user.

All these functionalities are intended to support the "intelligent" duties of the human teacher.

Then, if there is a solution, why hasn't it been solved before? A contributing factor is that it is only recently that the maturation of web solutions has given rise to the possibility of easily bringing platforms and environments to the web, such as interactive tutorials, making them more accessible and also scalable. With that, efficiency of code on the Web has become more important than ever. Thanks to recent innovations such as WebAssembly [25] users can write, compile, and run code in several languages directly from their browsers.

There are a few models currently working: `jsCoq` [19] is one of the first systems to embed a full theorem prover inside a browser. It is a platform and user environment for the `Coq` interactive proof assistant. For educational use, `jsCoq` allows the user to start interacting with proof scripts right away. This technology has been already used in different places: examples, tutorials, and courses[1]. Also `Lean` [14] has developed a playground accessible through the web. The intended uses of this infrastructure are web applications such as web IDEs[2], "live" tutorials and documentation[3], and online exercises. Moreover, it was used to develop course material for an interactive theorem proving course being offered in the spring of 2015 at CMU.

Our take on the problem is to devise a mechanism that produces code fragments automatically (content) and interactive user feedback by running the `exfilter` support tool in combination with the `Ciao` Prolog Playground [23, 24]. Our main goal is to help students and other users learn how to use advanced tools like `CiaoPP` (as well as to learn Prolog) without having to install and learn complex IDEs and environments, letting them try examples in a playground environment. More generally, our objective is to provide a tool that allows educators to create educational material. In addition to developing a specific tool, `exfilter`, we have also used this tool to create an interactive tutorial for `CiaoPP`. The tutorial introduces users to the approach of using static analysis as a program development tool and allows them to familiarize themselves with `CiaoPP` and other features of the `Ciao` system.

1.1. Structure of the document

The rest of the document is structured as follows: Chapter 2 describes the `Ciao` system, the framework that we have used to develop our tool. In Chapter 3 we introduce

[1] https://github.com/jscoq/jscoq#jsCoq-Users
[2] https://leanprover.github.io/tutorial/?live
[3] http://leanprover.github.io/tutorial

the proposed architecture for the generation of interactive tutorials, the `exfilter` tool, and a description of the available filters. Chapter 4 presents the tutorial created, which serves to evaluate the suitability of our architecture. Moreover, the tutorial illustrates the use of the combination of the `Ciao` tools on representative examples to verify their correctness. In Chapter 5 we provide some performance results by analyzing the examples of the tutorial. Finally, Chapter 6 summarizes the conclusions and suggests some possible new lines for future study.

Chapter 2

The Ciao System

Ciao [28] is a modern, multiparadigm programming language with an advanced programming environment. The main motivation behind the system is to offer a combination of programming paradigms and development tools that together help programmers produce better, more correct code in less time and with less effort. Regarding the objective of code correctness and the associated development tools, the two main approaches used currently are *verification*, which uses formal methods to prove specifications of the code, and *testing*.

The Ciao language introduces a development workflow [46, 30, 31] that unifies the above two approaches. In the Ciao model, program *assertions* (Section 2.2) are fully integrated into the language, acting as specifications for static analysis and as run-time check generators, unifying run-time verification and unit testing with static verification and static debugging. In Ciao the assertions are optional: if a program contains no user-provided assertions, Ciao can check the program against the assertions contained in the libraries used by the program.

The assertions used in Ciao are designed to serve many purposes: They can be processed by an autodocumenter (LPdoc, Section 2.3) in order to generate useful documentation while the system preprocessor (CiaoPP, Section 2.1) will use these assertions to report static analysis and verification results to the programmer.

Furthermore, the language is designed to be extensible in a simple and modular way using "packages" (syntactic and semantic extensions) and "bundles".

A high-level view of the Ciao System is shown in Figure 2.1. Blue-colored boxes represent user-written code; green boxes represent different tools within the system: the compiler, LPdoc and the CiaoPP Program Processor; and the red box represents the execution environment of the system, i.e., its run-time abstract machine and libraries.

2.1. CiaoPP: The Ciao Program Processor

CiaoPP[1] [30, 31, 46, 7] is the abstract interpretation-based program (pre)processor of Ciao. It is capable of statically finding non-trivial bugs, verifying that the program complies with specifications, and performing many types of program optimizations.

[1] https://cliplab.org/~clip/Software/Ciao/ciaopp-1.2.0.html/ciaopp_ref_man.html

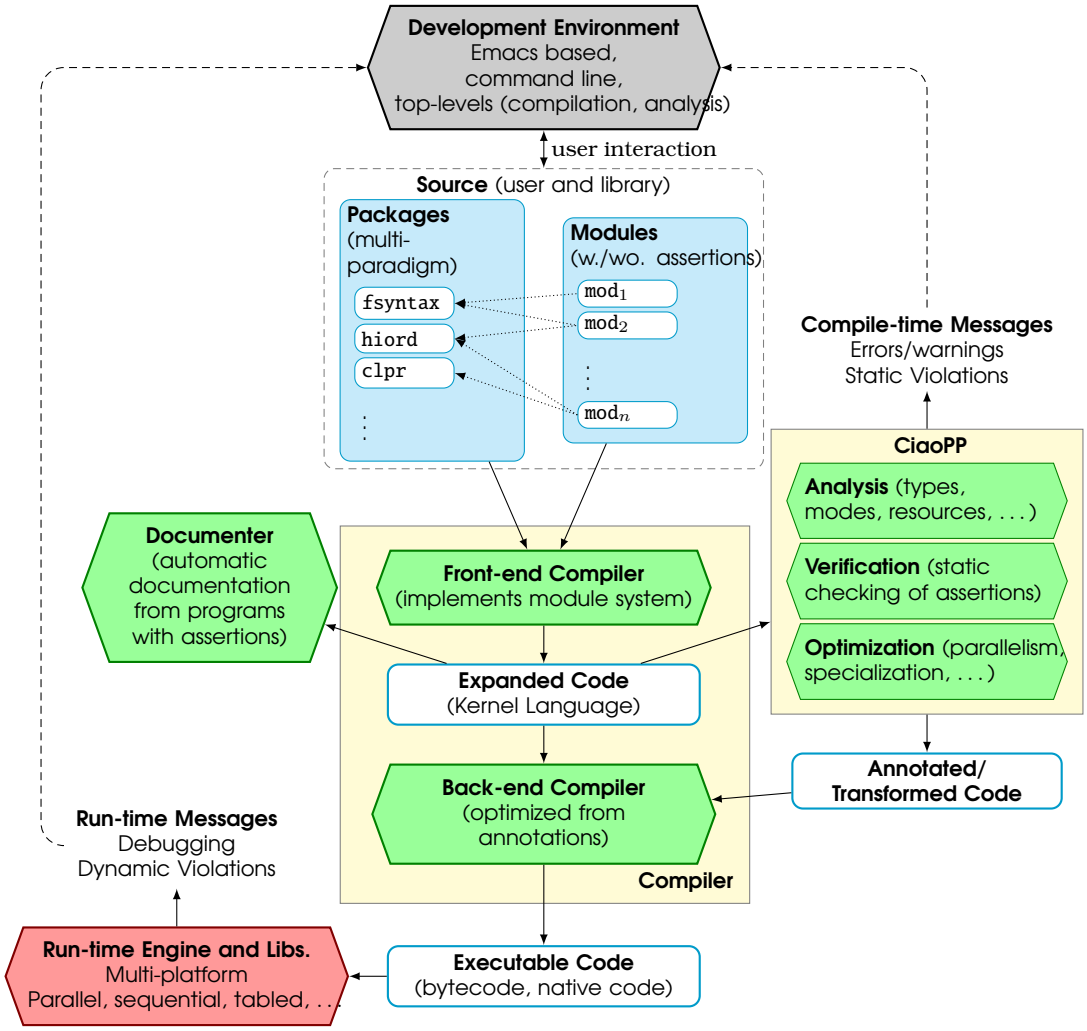

Figure 2.1: A high-level view of the Ciao system [28].

These can be applied to (Ciao) Prolog programs and also to many other high- and low-level languages, based on computing provably safe approximations of properties, using the technique of abstract interpretation [13]. The tasks performed by CiaoPP include:

- Inference of properties at the level of predicates and literals of the program, including types, modes and other variable instantiation properties, non-failure, determinacy, bounds on computational cost, bounds on sizes of terms in the program, etc.
- Static debugging and verification. This includes checking how programs call system library predicates and also checking the assertions present in the program or in other modules used by the program.
- Source to source program transformations such as program specialization, slicing, partial evaluation, and program parallelization (with granularity control). It also produces run-time test annotations for assertions that cannot be checked completely at compile-time, so that the program can be run safely by dynamically checking properties, and also generates test cases automatically from as-

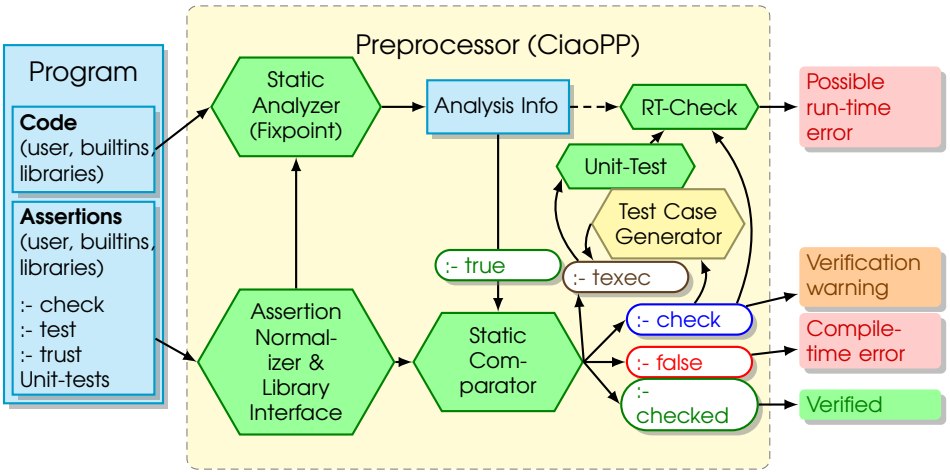

Figure 2.2: Architecture of the CiaoPP verification framework.

sertions.

- Producing abstract models of programs that act as certificates of the correctness of the code. The system is used to certify that code is safe with respect to the given policy (i.e., an abstraction-carrying code approach to mobile code safety [1]).

All the aforementioned features rely on the statically inferred properties based on fixpoint computation. Figure 2.2 provides an overview of the components of CiaoPP. The input to the process is the user program (possibly written in a different language) which can include a set of assertions (specification of the program). The *Static Analyzer* component has several fixpoint computation algorithms that are used to produce analysis information. This information (`true` assertions) is used to statically check the assertions in the *Static Comparator*. For each assertion originally with status `check`, the result of this process can be: that it is verified (the new status is `checked`), that a violation is detected (the new status is `false`), or that it is not possible to decide either way, in which case the assertion status remains as `check`, as detailed in the next section. In such cases, a warning and/or a run-time test generated by the *Run-time Check Annotator* component for (the part of) the assertion that could not be discharged at compile-time may be displayed.

Test cases can be also generated by the test generation module (CiaoTest [10, 9]). CiaoTest is integrated into CiaoPP and is able to generate goals for a predicate (`texec`'s) satisfying the assertion precondition. This module executes these goals to check that this assertion holds for those cases or, alternatively, find errors. CiaoTest also executes any user-defined unit tests relying on the run-time checking framework.

CiaoPP has been applied to the analysis, verification, and optimization of a number of languages (besides Ciao) such as Java, XC (C like) [38], Java bytecode [41, 40], ISA [37], LLVM IR [36], Michelson [43], ..., and resources ranging from execution time to energy consumption [39, 35].

2.2. Assertions

Assertions are linguistic constructions which allow expressing properties of programs. In this thesis we use the Ciao assertion language [46, 30, 31, 45]. These assertions can be used to express different properties of predicates in the source code, including functional properties, e.g., types, modes, sharing, aliasing, ..., as well as nonfunctional properties such as resource usage (energy, time, memory, ...), determinacy or non-failure. The user uses them in order to write specifications, express properties that some part of the code must fulfill, describe unknown code, etc.

One of the most commonly-used components of the Ciao assertion language is the pred assertion [30, 31, 6], which allows describing sets of *preconditions* and *conditional postconditions* on the state for a given predicate as well as global properties. A pred assertion is of the form:

```
:- [ Status ] pred Head [:  Pre ] [=> Post ] [+ Comp ].
```

where *Head* is a predicate descriptor that denotes the predicate that the assertion applies to, and *Pre* and *Post* are conjunctions of *property literals*. These properties are predicates which can also be used as run-time checks and can be inferred by some abstract domain in CiaoPP. *Pre* expresses properties that hold when *Head* is called, namely, at least one *Pre* must hold for each call to *Head*. *Post* states properties that hold if *Head* is called in a state compatible with *Pre* and the call succeeds. Finally, *Comp* can be used to describe properties of the computation such as cost, termination, determinism, non-failure, etc., and they apply to calls to the predicate that meet *Pre*.

Each assertion has a *Status* which is a keyword of the meaning of the assertion. The existing *Statuses* are the following:

- check: the assertion expresses properties that must hold at run-time, i.e., that the analyzer should prove or generate run-time checks for them. check is the *default* status, and can be omitted.
- trust: the assertion represents an actual behavior of the predicate that the analyzer assumes to be correct although it may not be able to infer it automatically.
- checked: the analyzer proved that the property holds in all executions.
- true: the analyzer inferred the assertion.
- false: the analyzer proved that the property does not hold in some execution.

As mentioned before, *parts* of assertions that cannot be discharged statically will remain in check status and run-time tests will be generated for them if necessary.

Example 1 *Consider the following* check *assertions of app/3:*

```
1  :- pred app(Xs, Ys, Zs) : ( list(Xs), list(Ys) ) => list(Zs).
2  :- pred app(Xs, Ys, Zs) : list(Zs) => ( list(Xs), list(Ys) ).
3
4
5  :- prop list/1.
6  list ([]) .
7  list ([_|T]) :- list(T) .
```

the first assertion expresses that calls to predicate app/3 with the first and second arguments bound to a list are admissible, and that if such calls succeed then the third argument should also be bound to a list. The second assertion captures another mode of use, stating that calls with just the third argument bound to a list are also admissible, and in that case on success the first and second arguments should be bound to a list.

Note the definition of the list/1 property (in this case a regular type) in lines 5-6. There are properties, such as num/1, list/1, or var/1, which are builtin properties, i.e., are defined in modules which are loaded by default, so there is no need to define them.

Definition 1 (Meaning of a Set of Assertions for a Predicate) *Given a predicate represented by a normalized atom Head, and a corresponding set of assertions* $\{a_1 \ldots a_n\}$*, with* $a_i =$ *":- pred Head : Pre$_i$ => Post$_i$." the set of* assertion conditions *for Head is* $\{C_0, C_1, \ldots, C_n\}$*, with:*

$$C_i = \begin{cases} \texttt{calls}(\mathit{Head}, \bigvee_{j=1}^{n} \mathit{Pre}_j) & i = 0 \\ \texttt{success}(\mathit{Head}, \mathit{Pre}_i, \mathit{Post}_i) & i = 1 \ldots n \end{cases}$$

where calls(*Head*, *Pre*)states conditions on all concrete calls to the predicate described by *Head*, and success(*Head*, *Pre*$_i$, *Post*$_i$) describes conditions on the success constraints produced by calls to *Head* if *Pre*$_i$ is satisfied. These allow representing behaviors for the same predicate for different call substitutions (multivariance). If the assertions a_i above, $i = 1, \ldots, n$, include a + $Comp$ field, then the set of *assertion conditions* also include conditions of the form comp(*Head*, *Pre*$_i$, $Comp_i$), for $i = 1, \ldots, n$, that express properties of the whole computation for calls to *Head* if *Pre*$_i$ is satisfied.

The assertion conditions for the assertions in the example above are:

$$\left\{ \begin{array}{llll} \texttt{calls}(& app(Xs, Ys, Xs), & ((list(Xs) \wedge list(Ys)) \vee (list(Zs)))), \\ \texttt{success}(& app(Xs, Ys, Xs), & (list(Xs) \wedge list(Ys)), \quad list(Zs))), \\ \texttt{success}(& app(Xs, Ys, Xs), & (list(Zs)), \quad (list(Xs) \wedge list(Ys))) \end{array} \right\}$$

2.3. LPdoc

LPdoc [27, 29] is a tool which generates documentation manuals automatically from one or more logic program source files, written in ISO-Prolog, the Ciao language extensions, and other (C)LP languages, as well as text files in several formats. In particular, LPdoc processes Prolog files adorned with assertions and machine-readable comments, which should be written in the Ciao assertion language. From these, it generates manuals in many formats including postscript, pdf, texinfo, info, HTML, man, etc., as well as on-line help. In particular, LPdoc can create and maintain fully automatically WWW and info sites containing on-line versions of the documents it produces. A simplified view of LPdoc's operation is illustrated in Figure 2.3.

One of the big advantages of this approach is that it is easier to keep the on-line and printed documentation synchronized with the source code. As a result, manuals change continually as the source code is modified.

As mentioned above, one of the most useful characteristics of the assertions used in Ciao is that they are designed to serve many purposes. Any assertions present in programs can be processed by LPdoc for the purpose of generating documentation. LPdoc is specially relevant in our context because it includes specific commands for

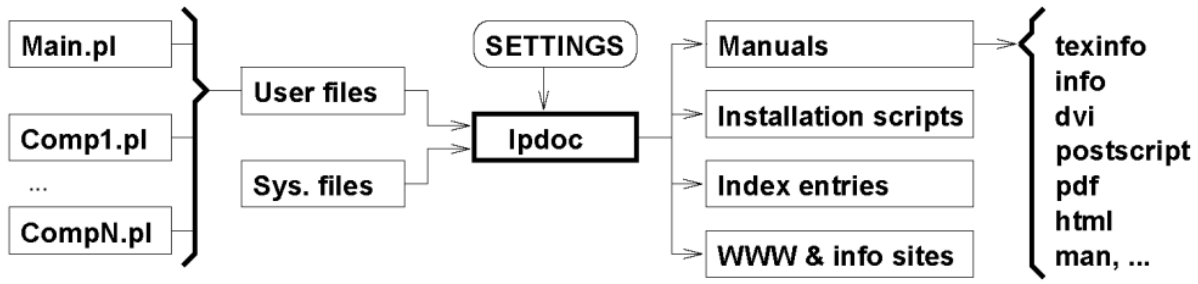

Figure 2.3: Overall operation of LPdoc

embedding editable and runnable code in place or in the Ciao Playground, presented in the following section.

2.4. The Ciao Prolog Playground

The Ciao Playground [23] is a web-based Prolog development tool that allows the execution of programs and queries through the browser. It is based on a web-based editor component and the wasm build grade of Ciao (using WebAssembly[2] and compiled with Emscripten[3]).

The Ciao Prolog Playground offers three main functionalities:

- A very easy way to run and share Prolog code, directly from any modern browser. The main advantage over other ways of using Ciao is that the playground does not require any installation or interaction with a server since everything runs within the browser.
- An easy way to embed runnable code examples in tutorials, manuals, slides, exercises, etc., and in general any kind of document. These documents can be developed with many tools, such as Google Docs, Jupyter notebooks, Word, Powerpoint, LaTeX, Pages, Keynote, web site generators, etc.

  The examples are stored in the documents themselves and do not need to be uploaded to (or edited in) any server.
- An easy way to create and distribute applications. The Playground can be specialized to create standalone web-based applications, with editor and top level, which also do not need installation, running fully within the user's browser. An example is the s(CASP) playground [24].

In our case we will be using mainly the second functionality above, i.e., embedding runnable code examples in documents. We will explain and illustrate the main features of this use of the Ciao Playground by means of the following exercise. The output generated for it by the LPdoc tool is shown in Figure 2.4:

Example 2.4 can be generated by including in the source file the following code:

[2] https://webassembly.org/
[3] https://emscripten.org/

```
% TASK 1 - Rewrite with Prolog arithmetic

factorial(0,s(0)).    % TODO: Replace s(0) by 1
factorial(M,F) :-     % TODO: Make sure that M > 0
    M = s(N),         % TODO: Compute N from M using is/2 (note that N is unbound! clear the equation)
    factorial(N,F1),
    times(M,F1,F).    % TODO: Replace times/3 by a call to is/2 (using *)

% When you are done, press the triangle (\"Run tests\") or the arrow (\"Load into playground\").
```

★ Show solution

Figure 2.4: Exercise: factorial using ISO-Prolog arithmetic

```
```ciao_runnable
:- module(_, _, [assertions]).

:- test factorial(A, B) : (A = 0) => (B = 1) + (not_fails, is_det).
:- test factorial(A, B) : (A = 1) => (B = 1) + (not_fails, is_det).
:- test factorial(A, B) : (A = 2) => (B = 2) + (not_fails, is_det).
:- test factorial(A, B) : (A = 3) => (B = 6) + (not_fails, is_det).
:- test factorial(A, B) : (A = 4) => (B = 24) + (not_fails, is_det).
:- test factorial(A, B) : (A = 5) => (B = 120) + (not_fails, is_det).
:- test factorial(A, B) : (A = 0, B = 0) + (fails, is_det).
:- test factorial(A, B) : (A = 5, B = 125) + (fails, is_det).
:- test factorial(A, B) : (A = -1) + (fails, is_det).

%! \begin{hint}
% TASK 1 - Rewrite with Prolog arithmetic

factorial(0,s(0)).    % TODO: Replace s(0) by 1
factorial(M,F) :-     % TODO: Make sure that M > 0
    M = s(N),         % TODO: Compute N from M using is/2 (note that N is unbound!
                      % clear the equation)
    factorial(N,F1),
    times(M,F1,F).    % TODO: Replace times/3 by a call to is/2 (using *)

% When you are done, press the triangle (\"Run tests\") or the arrow (\"Load into
% playground\").
%! \end{hint}
%! \begin{solution}
factorial(0,1).
factorial(N,F) :-
    N > 0,
    N1 is N-1,
    factorial(N1,F1),
    F is F1*N.
%! \end{solution}
```
```

As we can see, we have defined LPdoc-style commands which mark different parts of the program. The playground identifies the different parts and classifies them. For example, the playground will recognize a solution, a hint, and the parts of the code hidden to the user. In this case, the solution will be shown only when the "⋆ Show solution" button is clicked. The hint part is always visible.

When this exercise is opened, the page creates inside the browser the `CiaoWorker`, which imports the main bundles (i.e., `ciaowasm`, `core`, and `builder`). When a button is clicked, a certain function in the `CiaoWorker` is called to perform the corresponding Ciao-related action(s). In our case, one button loads the code in the top-level (the `load_code` function performs this task) and another button runs the tests (the `run_tests` function executes this other action). In order to correctly print the output on the top level, the solution of these queries is parsed and validated before finally being shown.

Another important point is that the `CiaoWorker` imports the minimum bundles and does it just when needed. So when a tutorial runs the tests using the `CiaoWorker`, the bundle required for the tests (`ciaodbg`) will be imported. Once a bundle is imported a first time, it remains in the browser's cache and does not need to be reloaded.

Chapter 3

An Interactive Documentation System for tutorials

3.1. The exfilter tool

The main purpose of exfilter is as an aid in the process of generating manuals and tutorials automatically, in order to keep Ciao's documentation synchronized with the system and to avoid having to keep track of what documentation needs to be changed when an update is made. The exfilter tool thus contributes to the process of learning by aiding in the generation of educational resources such as exercises, examples, etc.

As mentioned before, CiaoPP performs a number of program debugging, analysis, and source-to-source transformation tasks. The output produced by CiaoPP generally contains significant amounts of information, including transformations, static analysis information, assertion checking, or verification counterexamples. These results are typically presented as a new version of the source file annotated with (additional) assertions, which are generally (but not only) *Predicate Assertions*.

The full analysis results produced by CiaoPP can be quite large, and cover the whole file or program. However, most often, in a tutorial it is interesting to show only a small fraction of this information at a time, the particular part that helps to understand the topic or step being explained. In order to do so, we propose a mechanism which includes the use of filters. These filters make it possible to extract only selected parts of CiaoPP's output such as, for example, particular properties of a concrete predicate, particular types of assertions, etc. This thus allows exfilter to display only the relevant information of the analysis. We illustrate the proposed mechanism with a motivating example.

Let us revisit the append program:

```
1  :- module(_,app/3,[assertions]).
2
3  :- pred app(Xs, Ys, Zs) : ( list(Xs), list(Ys) ) => list(Zs) .
4
5  app([],     Ys, Ys).
6  app([X|Xs], Ys, [X|Zs]) :-
7    app(Xs, Ys, Zs).
```

where as before the pred assertion indicates that app/3 should be called in this pro-

gram with the first and second parameter being a list, and on success the third argument should also be bound to a list. Assume that we analyze it with the regular types domain where property `list(X)` is represented. Running the CiaoPP analyzer we obtain the result of the analysis as a new source file.

At this point, we would like to extract one or many fragments of the analysis. Thanks to the existence of filters, the only thing that we need to achieve this is to select the filters that match with the output we are looking for. Considering the previous example again, some of the different results that we can obtain are shown in Figure 3.1, where first listing shows the output from CiaoPP (type analysis, predicate level output) and the other two listings show two exfilter outputs where we have first selected the checked assertions and in the second one the true assertion.

Listing 3.1: CiaoPP compiler output (types)

```
1  :- module(_,app/3,[assertions]).
2
3  %% %% :- check pred app(Xs,Ys,Zs)
4  %% %%    : ( list(Xs), list(Ys) )
5  %% %%    => list(Zs).
6
7  :- checked calls app(Xs,Ys,Zs)
8    : ( list(Xs), list(Ys) ).
9
10 :- checked success app(Xs,Ys,Zs)
11   : ( list(Xs), list(Ys) )
12   => list(Zs).
13
14 :- true pred app(Xs,Ys,Zs)
15   : ( list(Xs), list(Ys), term(Zs) )
16   => ( list(Xs), list(Ys), list(Zs) ).
17
18 app([],Ys,Ys).
19 app([X|Xs],Ys,[X|Zs]) :-
20    app(Xs,Ys,Zs).
```

Listing 3.2: apply filter 1

```
1  %% %% :- check pred app(Xs,Ys,Zs)
2  %% %%    : ( list(Xs), list(Ys) )
3  %% %%    => list(Zs).
4
5  :- checked calls app(Xs,Ys,Zs)
6    : ( list(Xs), list(Ys) ).
7
8  :- checked success app(Xs,Ys,Zs)
9    : ( list(Xs), list(Ys) )
10   => list(Zs).
```

Listing 3.3: apply filter 2

```
1  :- true pred app(Xs,Ys,Zs)
2    : ( list(Xs), list(Ys), term(Zs) )
3    => ( list(Xs), list(Ys), list(Zs) ).
```

Figure 3.1: A simple example applying exfilter

3.1.1. Filters

A filter is a set of keywords defining the relations between input and output. These keywords are composed of words or phrases. For instance, if we run the analyzer on a program, the properties that hold if a predicate succeeds can be represented by true assertions. As mentioned before, a true predicate assertion is of the form:

:- true pred *Head* [: *Pre*] [=> *Post*] .

We can see that all true assertions follow the same syntax: they are preceded by ":-" and the keywords true and pred. Once we have the pattern that true assertions follow, we create a rule that searches for all predicates that start with ":- true pred". The rule is implemented using DCGs as follows:

```
1  truepred(Ys) --> ":- true pred ", tpkeep(Xs), {append(":- true pred ",Xs,XXs),
2                   append(XXs,"\n\n",Ys)}.
3
4  tpkeep(".") --> ".", !.
5  tpkeep([X|Xs]) --> [X], {X \= 0'. }, tpkeep(Xs).
```

The principal filters that exist are summarized in Table 3.1, with a brief description of the code fragments that the tool extracts.

Although these filters already extract a fraction of the analysis, it may happen that from that first extraction we want an even smaller part. For this purpose, supplementary filters have been created. Table 3.2 shows some of the other filters that the first set of filters can be combined with. They follow the same idea explained above.

Filter	Description
`all`	keep all data
`tpred`	all `true` (predicate) assertions
`tpred_plus`	all `true` assertions including *comp* properties
`tpred_regtype`	all `true` assertions and all *regtypes*
`regtype`	only all *regtype* definitions
`warnings`	all warnings
`error`	all errors
`check_pred`	all `check` assertions, `false` assertions and `checked` assertions
`warn_error`	all warnings and all errors
`test`	all tests

Table 3.1: Existing filters

		Filter									
Supplementary Filter	**Description**	`all`	`tpred`	`tpred_plus`	`tpred_regtype`	`regtype`	`warnings`	`error`	`check_pred`	`warn_error`	`test`
`name` = *Pred*	results of a specific predicate, with *Pred* being the predicate.	X	X	X	X	X	X	X	X	X	X
assertion = *[Terms]*	assertions that contain a series of terms, where *Terms* is the list of terms to be matched.	X	X	X	X	X	X	X	X	X	X
`comments` = on	If we add this option then the comments of the pred assertions will be added as well.								X		
`absdomain` = *AD*	*AD* can be *types* or can be *modes*. We have to add this option to obtain the assertions related to a particular domain, such as types or modes.		X	X	X						

Table 3.2: Supplementary filters

3.1.2. Algorithm

In this section, we will explain the pseudocode of the algorithm used for the prototype implementation. Algorithm 1 takes as input a program $File$, a domain $Domain$, a valid $Filter$, and one or more optional supplementary filters (Shown in Section 3.1.1). The algorithm starts by analyzing the program file $P\_analysis$. First, `exfilter` applies the $Filter$ for all predicates in $P\_analysis$, i.e., filters those predicates which meet a certain keywords requirement. Afterwards if the set of $Filters\_added$ is not empty, the

tool checks if the filtered predicates verify the conditions of each of the supplementary filters. Finally, the algorithm returns two new files: one containing all filtered predicates and another with the entire analysis.

We can see that it is a progressive process since each filter is applied successively and not all at once. Whenever an abstract domain is not specified, `CiaoPP` chooses a default set of analyses for the program such as, e.g., a types domain (the regular types domain `eterms`) and a modes domain (the sharing/freeness domain `shfr`). The algorithm only fails if a filter does not exist or is not written correctly.

Algorithm 1 Exfilter algorithm

```
1: procedure EXFILTER(File, Domain, Filter, Filters_added)
2:     P_analysis ← Analyze File with Domain
3:     for all Predicate p ∈ P_analysis do
4:         result ← Filter(p)
5:         for all opt ∈ Filters_added do
6:             if opt(result) ≠ ∅ then
7:                 result ← opt(result)
8:             end if
9:         end for
10:        return result, P_analysis
11:    end for
12: end procedure
```

3.1.3. Incorporating automated feedback using exfilter

The fact that questioning and feedback improves learning and comprehension of texts has long been supported by research [44]. Since one of the objectives of `exfilter` is to support learning, we have created a mechanism which gives an appropriate feedback. In order to concentrate all features in the same tool, we have added new filters for the inclusion of exercises in the documentation by developing three distinct modes of exercises:

- `equal`: Prints out the clauses from the user's answer that do not match with the file that contains the solution. It takes care of trivial differences such as different variable names and different formatting of the code in the files.

  This can be used for "fill in the blanks" exercises where people will not need to write full lines or blocks of code, but rather fill in some portions of it and without having to run the code.

  The disadvantage of using this approach is that `exfilter` can miss a lot of matches because the answers can be expressed in different ways. For example, the user writes an illegal extra space, misses periods or misplaces commas, etc. A solution would be to use an approach that focuses on querying for *semantic* characteristics of code [21].

- `errors`: Another task is to find bugs in a program and fix them. Users submit their solutions and `exfilter` checks them. To do this, we apply the above mentioned filter `warn_error` which looks for any error or warning. If there are none then the program has been corrected successfully.

Additionally, a `message` filter has been created to extract a message or messages by a certain term. For example, it can be used when trying to check whether or not a predicate has singleton variables, and we do not care about the other errors it may have. Using this filter, `exfilter` emits all messages containing the term "singleton" checking if the predicate still has singleton variables or not.

- `verify_assert`: Given a specification we can ask a user to build a program. Analysis information allows us to conclude that the program is incorrect or incomplete, i.e., that the program does not satisfy the requirements. The idea is that given the solution of the user, `exfilter` checks if the user's program verifies the assertions (specifications). In case of failure, the analyzer will provide hints to understand the origin of the errors.

  In order to do so, we have created two new filters: `checked_pred` and `notchecked_pred`. `checked_pred` filters the `checked` assertions. In contrast, `notchecked_pred` filters the `false` assertions and `check` assertions.

3.2. Architecture of the system

After introducing the `exfilter` tool, we provide an informal overview of the components and workflow of our approach to generating tutorials (in our case, all tutorials are generated with `LPdoc` using `Ciao`):

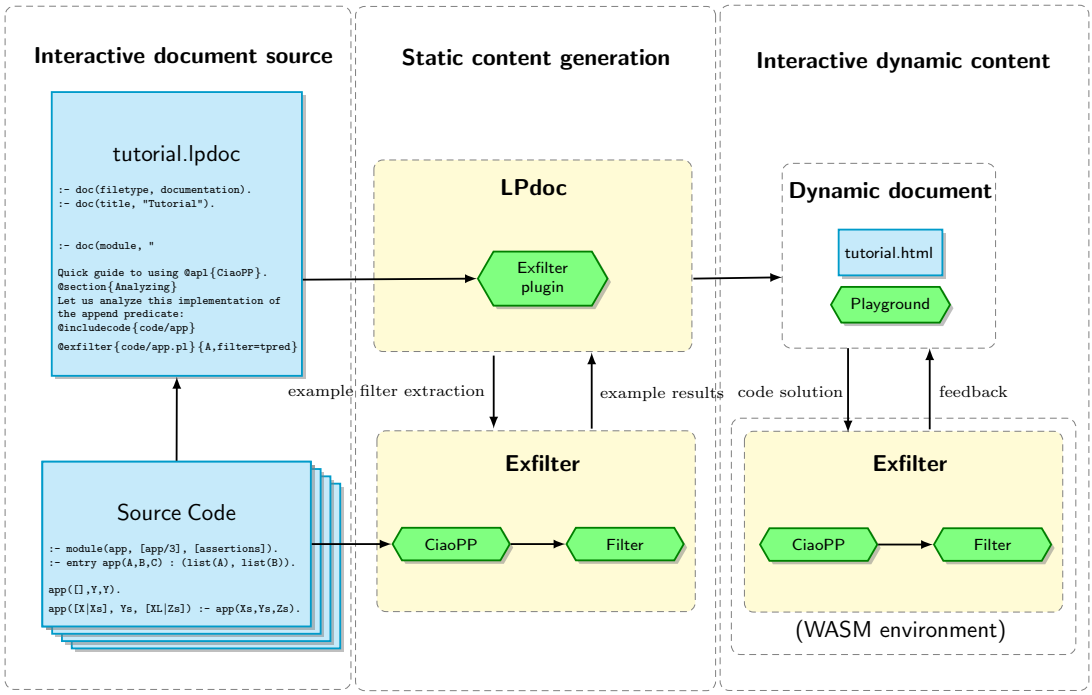

Figure 3.2: Architecture of the interactive documentation system.

The two core elements of our architecture are: the static content generation and the interactive dynamic content. Static content remains the same across pages. In order to include these, we have supplemented a plugin for `LPdoc` that recognizes a command with syntax `@exfilter{File}{Options}` which concatenates `File` with `Options`, finds the result file ensuring that this file exists, and incorporates it in the documentation.

The generation process follows these steps: While we are writing the tutorial, if we want to show an example (e.g., the result of analysis of a given predicate), we will have to use the aforementioned LPdoc plugin for exfilter. This is done by inserting the following command in the LPdoc document source:

```
@exfilter{app.pl}{A,filter=tpred}
```

This command is composed of the file to which we want to apply exfilter (app.pl) and the analysis and the filter options that we want applied (-A,filter=tpred). If this file has already been generated before, the contents of this file are simply included in the documentation. But if it does not exist, then LPdoc will create the file and we will have to run manually exfilter to fill it in[1]. Once the static content is generated, using LPdoc we will be able to generate the output of the tutorial in HTML.

On the other hand, dynamic content changes based on user interaction with the page. This enables users to do exercises and check their code directly from the tutorial. As mentioned before, LPdoc includes specific commands for embedding *editable* and *runnable* code in place. So when an exercise is included in the tutorial and the assignment is completed, users can check their solution by running exfilter from the browser. exfilter will "correct" that exercise and display the result.

3.3. exfilter in the Ciao Playground

In this section we explain the changes that we have made in the Ciao Playground in order to include exfilter. We will describe it with representative examples.

As mentioned before, the purpose of exfilter is to facilitate the task of generating appropriate examples and exercises to include in tutorials. We will start by explaining how interactive examples are generated. To include the example used above in a tutorial (as shown in Figure 3.3), we need to add the following code in the source file:

```
1  ```ciao_runnable
2  %! \begin{exfilter}
3  :- module(_,[app/3],[assertions]).
4
5  :- pred app(Xs , Ys , Zs) : ( list(Xs), list(Ys) ) => list(Zs) .
6
7  app([], Ys , Ys).
8  app([X|Xs], Ys , [X|Zs]) :-
9  app(Xs , Ys , Zs).
10 %! \end{exfilter}
11 %! \begin{opts}
12 comments=on,filter=check_pred
13 %! \end{opts}
14 ```
```

We can see that new commands have been added to mark regions to be processed by exfilter and options for it. The code between the exfilter directives is the code that will be shown in the tutorial and the user can modify. The part between the *opts* directives is the set of filters that we want to apply. In this case all check assertions, false assertions and checked assertions together with the comments will be shown.

CiaoPP is run in another process when generating static content. This cannot be done currently when the dynamic content is being generated due to current limitations in

[1] Although this may change, it is currently a deliberate choice, aimed at allowing a clean separation of the generation of examples from the generation of manuals.

WebAssembly. Thus, in order to implement this feature, `CiaoPP` is currently run in the same process using conditional compilation. This is activated when compiling the `exfilter` bundle with `LINUXwasm32`.

```
ciaopp_call_(Args,Result):-  get_arch(wasm32), !,
    io_once_port_reify(cmdrun_(Args), Port, OutString, ErrString),
    Result = append(OutString,ErrString),
    port_call(Port).

ciaopp_call_(Args,Out):-
    process_call(path(ciaopp),Args,[stderr(stdout), stdout(string(Out))]).
```

Furthermore, to run `exfilter` directly from the browser using `CiaoWorker` we have added an `exfilter` button and its functionality. When the "?" button is clicked, the `exfilter` and `CiaoPP` bundles are imported. Note that this only happens the first time since they remain in the browser's cache. Then, the `run_exfilter` function will run `exfilter` with the attributes that we have indicated. Finally, output is displayed to the user, as we can see in Figure 3.4.

```
async function run_exfilter(pg) {
  const mod = pg.curr_mod_path();
  const modbase = pg.curr_mod_base();
  const opts = pg.options_exfilter();

  await pg.toplevel.do_query("run_dynamic(\"" + mod + "\",\"" + opts +"\")",
  {msg:'Loading exfilter'});

  var str = await pg.cproc.w.readFile(modbase+'.txt');
  if (str !== null) {
    await show_text(pg, str);
  }
  playgroundCfg.auto_action = 'exfilter';
}
```

The code is also connected to the playground with a *Load in playground* button (i.e., the "↗" button) that opens the user's code in a new playground tab. This allows users to use the top level and run queries on their code.

On the other hand, to include exercises like the one shown in Figure 3.5, we need to add this code:

```
 ```ciao_runnable
 :- module(_,[app/3],[assertions]).
 %! \begin{exfilter}
 :- pred app(Xs , Ys , Zs) : ( list(Xs), list(Ys) ) => list(Zs) .

 app([], Ys , Yss).
 app([X|Xs], Ys , [X|Zs]) :- app(Xs , Ys , Zs).
 %! \end{exfilter}
 %! \begin{opts}
 solution=errors,message=singleton
 %! \end{opts}
 %! \begin{solution}
 :- pred app(Xs , Ys , Zs) : ( list(Xs), list(Ys) ) => list(Zs) .

 app([], Ys , Ys).
 app([X|Xs], Ys , [X|Zs]) :- app(Xs , Ys , Zs).
 %! \end{solution}
 ```
```

```
1  :- module(_,[app/3],[assertions]).
2
3  :- pred app(Xs , Ys , Zs) : ( list(Xs), list(Ys) ) => list(Zs) .
4
5  app([], Ys , Ys).
6  app([X|Xs], Ys , [X|Zs]) :-
7  app(Xs , Ys , Zs).
```

Figure 3.3: Example: predicate app/3

```
1  :- module(_,[app/3],[assertions]).
2
3  :- pred app(Xs , Ys , Zs) : ( list(Xs), list(Ys) ) => list(Zs) .
4
5  app([], Ys , Ys).
6  app([X|Xs], Ys , [X|Zs]) :-
7  app(Xs , Ys , Zs).
```

```
%% %% :- check pred app(Xs,Ys,Zs)
%% %%    : ( list(Xs), list(Ys) )
%% %%    => list(Zs).

:- checked calls app(Xs,Ys,Zs)
   : ( list(Xs), list(Ys) ).

:- checked success app(Xs,Ys,Zs)
   : ( list(Xs), list(Ys) )
   => list(Zs).
```

Figure 3.4: Result of applying exfilter to predicate app/3

Here the playground identifies the code that we want to show, the mode of the exercise and the solution. In our example, we want the user to correct the singleton variable written in the base case of predicate app/3. The process is similar to the previous one but in this case the run_exfilter_exercise function will return an appropriate feedback using the filter errors and extracting the messages related to the singleton variables. The result is shown in Figure 3.6.

```
async function run_exfilter_exercise(pg) {
  const mod = pg.curr_mod_path();
  const modbase = pg.curr_mod_base();
  const opts = pg.options_exfilter();
  const sol = pg.solution_exercise();

  await pg.toplevel.do_query("run_exercise(\"" + mod + "\",\"" + sol + "\",\""+ opts + "\")",
  {msg:'Loading exfilter'});

  var str = await pg.cproc.w.readFile(modbase+'.txt');
  if (str !== null) {
    await show_text(pg, str);
  }
  playgroundCfg.auto_action = 'exfilter_exercise';
}
```

```
1  :- pred app(Xs , Ys , Zs) : ( list(Xs), list(Ys) ) => list(Zs) .
2
3   app([], Ys , Yss).
4   app([X|Xs], Ys , [X|Zs]) :-
5   app(Xs , Ys , Zs).
```

★ Show solution

Figure 3.5: Exercise: Correct predicate `app/3`

```
1  :- pred app(Xs , Ys , Zs) : ( list(Xs), list(Ys) ) => list(Zs) .
2
3   app([], Ys , Yss).
4   app([X|Xs], Ys , [X|Zs]) :-
5   app(Xs , Ys , Zs).
```

```
WARNING: (lns 2-3) [Ys,Yss] - singleton variables in app/3
```

★ Show solution

Figure 3.6: Result of applying `exfilter` to the answer

Chapter 4

An interactive tutorial for CiaoPP

We have created a tutorial[1] with three main purposes:

- Showcase the capabilities of the tools developed.
- Exhibit the benefits of formal methods, and in particular of the CiaoPP approach.
- Serve as a didactic introduction to CiaoPP.

The tutorial demonstrates that there is no single method alone that guarantees correctness or finds all bugs, but we can be closer to this goal by using a combination of techniques. The techniques that will be showcased include:

- Specification: How to define (parts of) what the program is supposed to do.
- Static Analysis: What one can tell about the program without executing it.
- Verification: How to establish whether the program is correct or not based on the above.
- Testing: Can prove the presence of some bugs.
- Debugging: Trying to locate the cause of incorrectness.

These techniques and the CiaoPP approach to them are actually independent of language and architecture. However, and mentioned before, here we use the Ciao system which this thesis extends.

The approach of using static analysis as a program development tool was actually pioneered by the Ciao system. Ciao introduced a workflow that integrates static and dynamic verification and debugging to work cooperatively in a unified way. The combination of the Ciao tools covers all steps mentioned before.

The ability of Ciao to provide immediate feedback leads us to make of this a fundamental feature of the tutorial. Students can (a) submit their answers as many times as necessary, by running each exercise through the `exfilter` tool and getting immediate feedback, (b) get a helpful hint, (c) see the solution.

Figure 4.1 shows how the questions are presented to the student. A hint can be given and when the "⋆ Show solution" button is clicked the solution is shown. The

[1] https://ciao-lang.org/ciao/bndls/exfilter/examples/introtutorial.html/new-tutorial.html

question in Figure 4.1 is a training exercise to exhibit if the user can write the correct assertion.

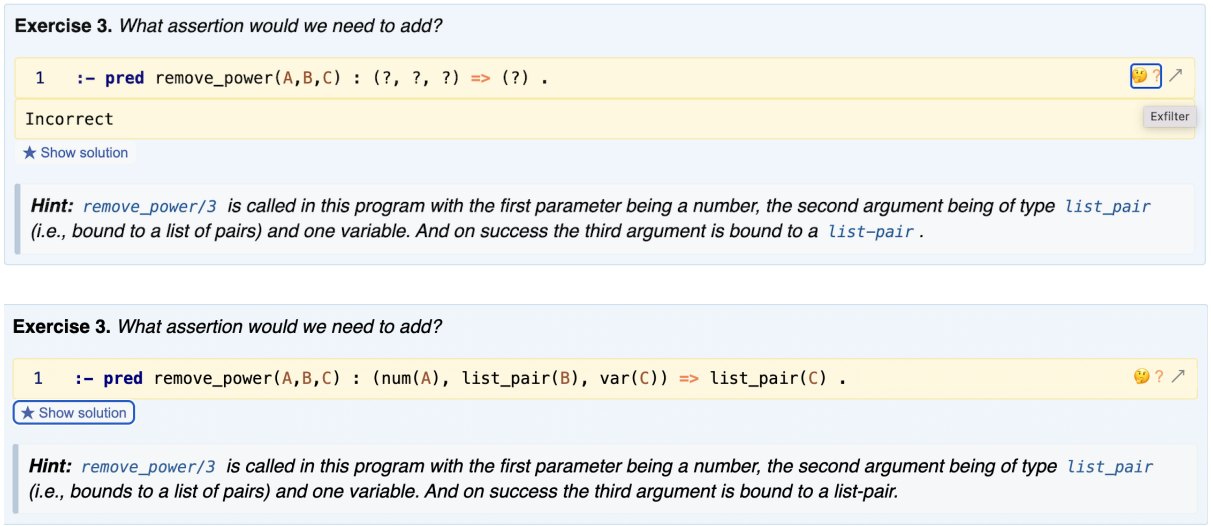

Figure 4.1: Web interface for Exercise 3.

4.1. Tutorial Content

Our methodology supports self-study and combines exercises, explanations, and examples. The content of this tutorial has the aim of illustrating step-by-step the use of the different `Ciao` tools on representative examples. In this section we will explain how the tutorial is organized, as well as how `exfilter` has been applied.

4.1.1. Introduction

In the introduction the example program is presented in detail. First a statement is referred to as the specification of the program, intended to provide a model of the problem. It is a narrative, supplied with one or more examples of the inputs and expected outputs.

Our example is taken from the Prolog programming contest at ICLP'95, Portland, USA [17] and the specification given is:

"Write a predicate `powers/3`, which is called with as first argument a list of non-negative numbers, as second argument a number N, and a free third argument. Such a call must succeed exactly once and unify the third argument with the list that contains the smallest N integers (in ascending order) that are a non-negative power of one of the elements of the first argument".

Some examples:

```
?- powers([3,5,4],17,Powers) .
Powers = [3,4,5,9,16,25,27,64,81,125,243,256,625,729,1024,2187,3125]
```

```
?- powers([2,9999999,9999998],20,Powers) .
Powers = [2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,65536,131072,262144,
          524288,1048576]
```

```
?- powers([2,4],6,Powers) .
Powers = [2,4,8,16,32,64]
```

The implementation of the program is already given since the tutorial aims to teach tools like CiaoPP rather than Prolog. However, it can be a good exercise to practice with both.

4.1.2. Notation

We introduce the Ciao notation, i.e., modules, assertions and regular types.

Modules and exports. We start by defining the module. The source of a Ciao module is typically contained in a single file. The predicates defined within a module are visible only if they are exported. The module declaration is helpful because during the analysis of a program, CiaoPP can assume that external calls are only to the exported predicates. This fact will give us more accurate information about the program since CiaoPP does not have to consider all the possible ways other predicates inside the module may be called, and only those that can actually occur in the module.

Regular types and other properties. Regular types can be used as properties to describe predicates. In CiaoPP we can define new regular types using regtype declarations. There are properties like num/1, var/1, or not_fails that are builtins, defined in libraries. For example, in our specification the first argument is a list of numbers. This property is available in the Ciao libraries, however the user can declare it using regtype declarations. So we show how to represent the set of all "lists of numbers" by the regular type list_num:

```
:- regtype list_num(X) # "@var{X} is a list of numbers." .
list_num([]).
list_num([X|T]) :-
    num(X),
    list_num(T).
```

Assertions. We describe the assertion schema. The assertions are the (partial) specification of the program. Writing assertions is optional but we recommend doing it since the more assertions are present in the program, the more likely errors will be detected automatically. To familiarize the user with the syntax we have proposed different exercises with the same pattern: first, we give a description of a particular predicate and then the user has to write the corresponding pred assertion. This is an example:

"In our example, we need that for any call to predicate powers/3 with the first argument bound to a list of numbers, the second argument bound to a number, and the third one unbound, if the call succeeds, then the third argument should also be bound to a list of numbers."

Exercise 2 (Solved). *What assertion would we need to add?*

```
:- pred powers(A,B,C) : (list_num(A), num(B), var(C)) => (list_num(C)).
```

In this kind of exercise, exfilter will compare syntactically the user's answer with the solution using the equal mode for exercises.

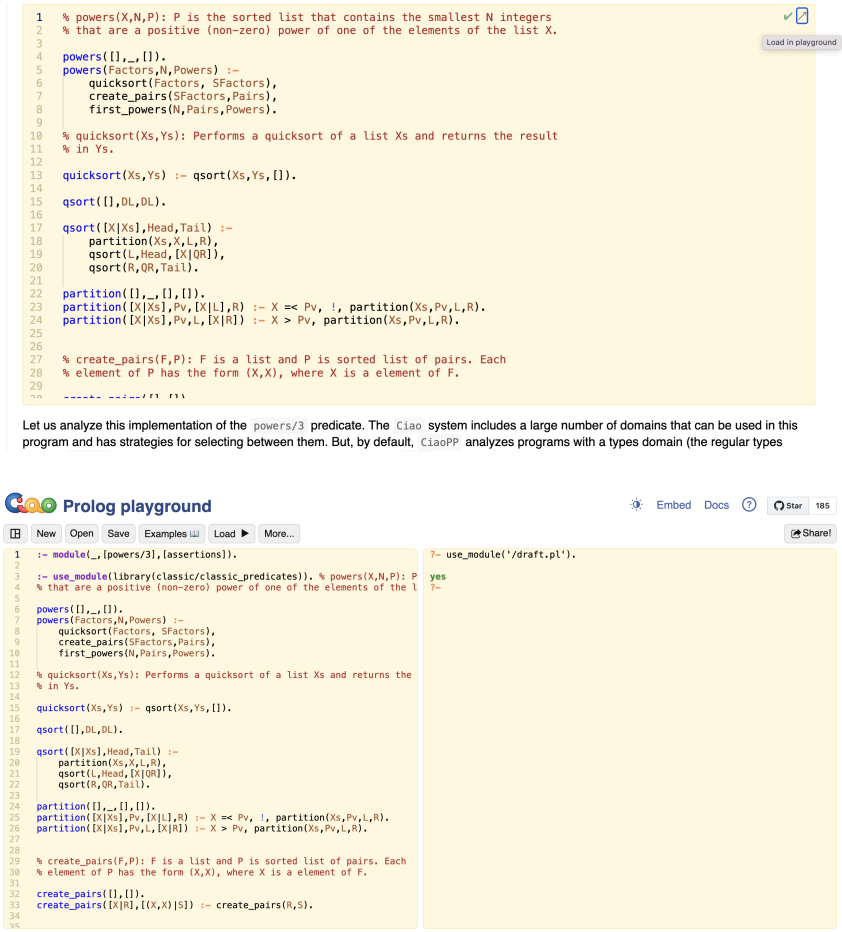

Figure 4.2: Loading `powers` to the `Ciao` Playground.

After the introduction of these standard contents, the implementation of the program is also presented. Figure 4.2 shows how the program can be loaded into the `Ciao` Playground letting the student experiment with it. From this point on, the tutorial illustrates the stages the programmer can follow to check the correctness of the program.

4.1.3. Using the `CiaoPP` Program Processor

Static Analysis. Static analysis consists in answering an implicit question of the form "What can you tell me about the collecting semantics of this program?". In order to answer this question `CiaoPP` computes safe approximations of the program semantics at different relevant points.

Given a program, a collection of abstract domains is automatically selected based on their relevance to the properties present in the programmer's assertions or in `Ciao` libraries. Domains are chosen by considering if they can abstract the properties in the assertions. If a domain understands a property in an assertion, the domain is run. A given property is often understood by more than one domain.

When the implementation of the program is shown we ask the users to run `CiaoPP`. If the program contains no assertions, by default, `CiaoPP` analyzes programs with

a types domain (the regular types domain `eterms`) and a modes domain (the sharing/freeness domain `shfr`). In the tutorial we will be working mainly with these two.

The idea is that the user is reading the tutorial and working at the same time with the example in the playground. In case the playground is not being used we show the analysis result in the tutorial using `exfilter` and the filter `warn_error` (which filters only the warning and error messages):

```
WARNING (ctchecks_pp_messages): (lns 22-22) At literal 1 could not verify assertion:
:- check calls B=<A
   : ( nonvar(B), nonvar(A), arithexpression(B), arithexpression(A) ).
because on call arithmetic:=<(A,B) :

[eterms] basic_props:term(B),basic_props:term(A),basic_props:term(A),basic_props:term(B),
         basic_props:term(C)

[shfr]   native_props:mshare([[B],[B,A],[B,A,B],[B,B],[A],[A],[A,B],[B],[C]]),
         term_typing:var(A),term_typing:var(C)



WARNING (ctchecks_pp_messages): (lns 22-23) At literal 1 could not verify assertion:
:- check calls B>A
   : ( nonvar(B), nonvar(A), arithexpression(B), arithexpression(A) ).
because on call arithmetic:>(A,B) :

[eterms] basic_props:term(B),basic_props:term(A),basic_props:term(A),
         basic_props:term(B),basic_props:term(C)

[shfr]   native_props:mshare([[B],[B,A],[B,A,B],[B,B],[A],[A],[A,B],[B],[C]]),
         term_typing:var(A),term_typing:var(C)



WARNING (ctchecks_pp_messages): (lns 27-39) At literal 1 could not verify assertion:
:- check calls A>B
   : ( nonvar(A), nonvar(B), arithexpression(A), arithexpression(B) ).
because on call arithmetic:>(A,B) :

[eterms] basic_props:term(A),rt188(B)
with:

:- regtype rt188/1.
rt188(0).
[shfr]   native_props:mshare([[A]]),term_typing:ground([B])



WARNING (ctchecks_pp_messages): (lns 27-39) At literal 4 could not verify assertion:
:- check calls A is B
   : ( ( var(A), nonvar(B), var(A), arithexpression(B) ); ( var(A), nonvar(B), var(A),
   intexpression(B) ); ( nonvar(A), nonvar(B), num(A), arithexpression(B) );
   ( nonvar(A), nonvar(B), int(A), intexpression(B) ) ).
because on call arithmetic:is(A,B) :

[eterms] basic_props:term(A),rt201(B)
with:


:- regtype rt201/1.
rt201(A*B) :-
    term(A),
    term(B).
[shfr]   native_props:mshare([[A],[B]]),term_typing:var(A)
```

```
WARNING (ctchecks_pp_messages): (lns 47-48) At literal 1 could not verify assertion:
:- check calls A=\=B
   : ( nonvar(A), nonvar(B), arithexpression(A), arithexpression(B) ).
because on call arithmetic:=\=(A,B) :

[eterms] basic_props:term(A),rt112(A),basic_props:term(B),basic_props:term(B)

[shfr]   native_props:mshare([[A],[A,A],[A,A,B],[A,A,B,B],[A,A,B],[A,B],
         [A,B,B],[A,B],[A],[A,B],[A,B,B],[A,B],[B],[B,B],[B]])



WARNING (ctchecks_pp_messages): (lns 56-56) At literal 1 could not verify assertion:
:- check calls B=<A
   : ( nonvar(B), nonvar(A), arithexpression(B), arithexpression(A) ).
because on call arithmetic:=<(A,B) :

[eterms] rt112(A),basic_props:term(B),basic_props:term(B),basic_props:num(A),
         arithmetic:arithexpression(C)

[shfr]   native_props:mshare([[A],[A,B],[A,B,B],[A,B],[B],[B,B],[B]]),
         term_typing:ground([A,C])
```

These warnings are stating that there are a number of assertions that cannot be shown to hold. In particular, the analysis is saying that it is not possible to ensure that the calls that the program makes to predicates such as >=/2, </2, >/2, is/2, and =</2 respect the corresponding preconditions or calling modes, which generally require the arguments to be bound to arithmetic expressions when called. The interesting thing to note here is that any assertions have been included in the code. The warning messages stem from the assertions (specifications) that provide the pre-conditions and post-conditions for such library predicates in the Ciao system libraries. Thus, a first observation is that it is possible to identify potential bugs even without actually adding assertions to programs.

In particular, in view of the warnings above, it seems useful to be able to ensure that all these library predicates will always be called properly within our module and thus be more confident about the program. From here on, our objective is teaching the student how to gradually add assertions to provide information to the analyzer in order to increase its precision.

We start by asking the user to add a pred assertion. From the specification of the program (the problem statement) we know that the second argument is a number. We write this as follows:

```
:- pred powers(A,B,C) : num(A) .
```

In particular, this assertion is stating that the predicate powers/3 will be called using a number as a second argument. When the user proceeds to run again CiaoPP, CiaoPP will continue producing some warning messages but this time the following warning message will not appear:

```
WARNING (ctchecks_pp_messages): (lns 22-34) At literal 1 could not verify assertion:
:- check calls A>B
   : ( nonvar(A), nonvar(B), arithexpression(A), arithexpyression(B) ).
because on call arithmetic:>(A,B) :

[eterms] basic_props:term(A),rt122(B)
with:
```

```
:- regtype rt122/1.
rt122(0).
[shfr]   native_props:mshare([[A]]),term_typing:ground([B])
```

Another important point of this section is how to interpret the analysis output. As the programmer adds assertions, the analysis with the selected domains is performed and the resulting safe approximations are compared directly with the assertions that have been added. For each assertion originally with status check, the result of this process can be: that it is verified (the new status is checked), that a violation is detected (the new status is false), or that it is not possible to decide either way, in which case the assertion status remains as check.

By adding the assertion of *Exercise 2* of the tutorial, the user will be able to check that the file generated by CiaoPP will have the following assertion:

```
%% %% :- check pred powers(A,B,C)
%% %%    : ( list_num(A), num(B), var(C) )
%% %%    => list_num(C).

:- checked calls powers(A,B,C)
   : ( list_num(A), num(B), var(C) ).

:- checked success powers(A,B,C)
   : ( list_num(A), num(B), var(C) )
   => list_num(C).
```

This means that the assertion that has been included has been marked as checked, i.e., it has been validated. But this other information will also be emitted:

```
:- true pred sorted_insert(_A,X,_B)
   : ( list(^(('basic_props:num','basic_props:num')),_A), rt96(X), term(_B) )
   => ( list(^(('basic_props:num','basic_props:num')),_A), rt96(X), list1(^((num,num)),_B) ).

:- regtype rt96/1.
rt96((A,B)) :-
    num(A),
    num(B).
```

The assertion above, with a true prefix, expresses that the compiler has proved that procedure sorted_insert/3 produces as output a list of pairs, a new inferred type and a non-empty list of pairs. The students can see that the analyzer displays the information they have to add, so we ask them to write it into the specification as defining a new regular type. In our case, this regular type is defining the pairs, so users can copy it and change the name to make it clearer. This can be done as follows:

```
:- regtype num_pair(P) .
num_pair((X, Y)):-
        num(X),
        num(Y).

:- regtype list_pair(L) .
list_pair([]).
list_pair([X|Xs]):-
    num_pair(X),
    list_pair(Xs).
```

```
:- regtype list_pair1(L) .
list_pair1([X|Xs]):-
   num_pair(X),
   list_pair(Xs).
```

The three fragments above are taken directly from the CiaoPP output by exfilter, that also generates the code to be inserted in the tutorial. In the first one, we apply the filter check_pred (i.e., filter all check assertions, false assertions, and checked assertions). In the second one, we apply the filter tpred with the supplementary filter name=sorted_insert and absdomain=types (i.e., filters all true assertion related to the types of the predicate powers/3). And in the third one the filter regtype is used.

Program Debugging. In the sections above we ask the user to include assertions to describe some properties that are required to hold on. But we also mentioned that CiaoPP can identify errors without these assertions.

We start by giving a buggy implementation of predicate remove_power/3:

```
:- module(_,[remove_power/3],[assertions]).

remove_power(Power,[(Power1,Factor)|RestOut],[(Power1,Factor)|RestOut]) :-
    Power =\= power1, !.
remove_power(Power,[_|RestPFsIn],PFsOut) :-
    remove_power(Power,RestPFsIn,PFsOut1).
```

And we show the CiaoPP analysis output:

```
WARNING: (lns 5-6) [PFsOut,PFsOut1] - singleton variables in remove_power/3


WARNING (preproc_errors): (lns 2-4) goal arithmetic:=\=(Power,power1) at literal 1 does not
    ↪ succeed!


WARNING (preproc_errors): (lns 5-6) goal
    ↪ remove_power_bug1:remove_power(Power,RestPFsIn,PFsOut1) at literal 1 does not succeed!


ERROR (ctchecks_pp_messages): (lns 2-4) At literal 1 false assertion:
:- check calls A=\=B
   : ( nonvar(A), nonvar(B), arithexpression(A), arithexpression(B) ).
because on call arithmetic:=\=(A,B) :

[eterms] basic_props:term(A),rt2(B)
with:

:- regtype rt2/1.
rt2(power1).
```

The main challenge in debugging a verification failure is to obtain enough information about the failed verification attempt to debug the error. To help the programmer with this task, Ciao analysis displays information, warnings or makes suggestions. The idea of this section is to assist the user in diagnosing the output of the analysis and determining how to take corrective action. In the following we give an example of this approach.

Singleton variable: The first message is a warning message which indicates that there are singleton variables. We know that the singleton variables are those which appear

only once in a clause. As mistyping a variable is a common mistake, CiaoPP outputs the standard warning message indicating if a variable is used only once (these messages would also be produced by the compiler).

Exercise 5 (Detecting Bugs) *What variable do you need to change? (Only change the incorrect variable)*

```
remove_power(Power,[(Power1,Factor)|RestOut],[(Power1,Factor)|RestOut]) :-
    Power =\= power1, !.
remove_power(Power,[_|RestPFsIn],PFsOut) :-
    remove_power(Power,RestPFsIn,PFsOut1).
```

The student only has to correct the error explained. To do this, when the user submits the answer, we ask `exfilter` to run CiaoPP and filter those error/warning messages related to singleton variables (using the mode of exercises `errors`), if there is no message then the answer is correct. The same procedure is performed for each of the error/warning messages.

Although in the tutorial we have only worked so far with the two most used abstract domains: `shfr` and `eterms`, as mentioned before CiaoPP has a wide variety of abstract domains to perform analysis with. For example, knowing which predicates are deterministic for a particular class of calls has several interesting uses in debugging and verification.

In order to include this in the tutorial, we ask the user to analyze the example with the `nfdet` analysis. The `nfdet` combined domain carries nonfailure (`nf`) and determinism (`det`) info, i.e., the analysis will be able to detect procedures that can be guaranteed not to fail (produce at least one solution) and to detect predicates which are deterministic (produce at most one solution). It is also relevant to find predicates whose clause tests are mutually exclusive (meaning that only one of their clauses will pass), even if they are not deterministic, because they are called by other predicates that can produce more than one solution. We explain these concepts through an example.

Imagine that the predicate `sorted_insert/3` is defined without the cut:

```
sorted_insert([], X, [X]).
sorted_insert([(X1,F1)|L1], (X,F), [(X,F), (X1,F1)|L1]):- X =< X1 .
sorted_insert([P|L1], X, [P|L]):- sorted_insert(L1, X, L).
```

If we ask the user to analyze the program with this modification then CiaoPP's output will include the following assertions:

```
%% %% :- check pred sorted_insert(A,B,C)
%% %%    : ( list_pair(A), num_pair(B), var(C) )
%% %%    => list_pair1(C).

:- checked calls sorted_insert(A,B,C)
   : ( list_pair(A), num_pair(B), var(C) ).

:- checked success sorted_insert(A,B,C)
   : ( list_pair(A), num_pair(B), var(C) )
   => list_pair1(C).
```

The analyzer did verify the assertion, and, in addition to that, CiaoPP will display this other information:

```
:- true pred sorted_insert(A,B,C)
   : ( mshare([[C]]),
       var(C), ground([A,B]), list_pair(A), num_pair(B), term(C) )
   => ( ground([A,B,C]), list_pair(A), num_pair(B), list_pair1(C) )
   + ( multi, covered, possibly_not_mut_exclusive ).
```

As we mentioned before, the + field in pred assertions describes properties of the whole computation of the predicate (such as determinism or non-failure). multi states that there is at least one solution but it may have more. covered means that for any input there is at least one clause which succeeds. possibly_not_mut_exclusive denotes that mutual exclusion is not ensured. The output shown above is due to the fact that when the first argument is a non-empty list both the second and third clauses succeed. When reasoning about determinacy, it is a necessary condition (but not sufficient) that clauses of the predicate be pairwise mutually exclusive, i.e., that only one clause will produce solutions.

In order to solve this, the user can add either the complementary test $X > X1$ in the third clause, or a cut in the second clause. Obviously, for any particular call only one of the clauses with $X =< X1$ or $X > X1$ will succeed. Adding one of these two options and analyzing the program again the programmer can see that the predicate is now deterministic:

```
:- true pred sorted_insert(A,B,C)
   : ( mshare([[C]]),
       var(C), ground([A,B]), list_pair(A), num_pair(B), term(C) )
   => ( ground([A,B,C]), list_pair(A), num_pair(B), list_pair1(C) )
   + ( det, covered, mut_exclusive ).
```

In the tutorial, in order to show the assertions related with the determinism of predicate sorted_insert/3, we ask exfilter to run CiaoPP and analyze the example with the nfdet abstract domain. Then the filter tpred_plus and the supplementary filter name=sorted_insert are applied (i.e., filter all true assertions including *comp* properties of predicate sorted_insert).

Testing. The specification throughout the program is that predicate powers/3 is called with a list of numbers as first argument, a number N as second argument, and a free third argument. But, in the original specification instead of general numbers non-negative integers are used. Thus, we now ask the user to define a new regular type (we use nnegint from the Ciao libraries) and include it in the program:

```
:- prop list_nnegint(X) + regtype
# "Verifies that @var{X} is list of non-negative integers." .
list_nnegint([]).
list_nnegint([X|T]) :-
     nnegint(X),
     list_nnegint(T).
```

(Note that here we could have also used parametric types, i.e., a parametric list type, but we have used plain predicates for simplicity.) Then this assertion can be added:

```
:- pred powers(A,B,C) : (list_nnegint(A), nnegint(B), var(C)) => list_nnegint(C) + not_fails.
```

In addition, the problem statement also establishes that the numbers in the list must be in ascending order. For this purpose, we add an additional property that defines a sorted list:

```
:- prop sorted/1.
sorted([]).
sorted([_]).
sorted([X,Y|Ys]) :-
    X=<Y,
    sorted([Y|Ys]).
```

and add it also to the assertion:

```
:- pred powers(A,B,C) : (list_nnegint(A), nnegint(B), var(C))
                      => (list_nnegint(C), sorted(C))
                      + not_fails .
```

However, when running CiaoPP, the user will see that this assertion cannot be proven nor disproven statically with the standard CiaoPP domains.

```
:- check success powers(A,B,C)
   : ( list_nnegint(A), nnegint(B), var(C) )
   => ( list_nnegint(C), sorted(C) ).

:- check comp powers(A,B,C)
   : ( list_nnegint(A), nnegint(B), var(C) )
   + not_fails.

:- checked calls powers(A,B,C)
   : ( list_nnegint(A), nnegint(B), var(C) ).
```

This is because there is no abstract domain that covers properly the `sorted/1` property. This is something that can occur specially with user-defined properties. In order to deal with this problem there are different possible approaches:

1. The student can generate test cases automatically from the call field of the assertions to try to find a counterexample, i.e., an error.
2. A new abstract domain can be implemented for a specific property, i.e., in this case a new domain that infers if a list is sorted or not.
3. The property can be proven by hand or with an automatic theorem prover.

While the second and third solutions can potentially verify that there are no errors in the program, the first one can find bugs but cannot verify that there are none. However, the second and third solutions are beyond the scope of this tutorial, so we ask the user to continue with the first approach. When CiaoPP cannot verify (parts of) the assertions statically, *runtime-checking* instrumentation can be added to the program. *Runtime checks* ensure that execution paths that violate the assertions are captured during execution, serving as test oracles. However, since run-time checks can become expensive if used indiscriminately, they are most often used before deployment in combination with unit testing. Static analysis and dynamic testing are complementary approaches to software validation. In CiaoPP, assertions that have not been verified, the user can use CiaoTest [10, 9] (which is integrated into CiaoPP) to check them and to find errors. This can be done in two ways: the student can include some unit tests or CiaoPP/CiaoTest can also generate random tests.

In order to show to the user how CiaoTest works, we make a modification to the quicksort/2 predicate (this predicate is used in the implementation of powers/3):

```
% quicksort with a slight mistake: it may fail when there are repeated numbers in the list
quicksort(Xs,Ys) :- qsort(Xs,Ys,[]).

qsort([],DL,DL).

qsort([X|Xs],Head,Tail) :-
    partition(Xs,X,L,R),
    qsort(L,Head,[X|QR]),
    qsort(R,QR,Tail).

partition([],_,[],[]).
partition([X|Xs],Pv,[X|L],R) :- X < Pv, !, partition(Xs,Pv,L,R). % (1) should be >= (or =<
    ↪ below)
partition([X|Xs],Pv,L,[X|R]) :- X > Pv, partition(Xs,Pv,L,R).
```

This predicate sorts a given list of integers from lowest to highest. However, we have introduced an intentional bug (1 in the listing) that causes the program to fail when a list with repeated elements is given.

The user can also add these three assertions to check the behavior of the predicate. They cover the examples given in the problem statement.

```
:- test powers(A,B,C) : (A = [3,4,5], B = 17)
                      =>(C = [3,4,5,9,16,25,27,64,81,125,243,256,625,729,1024,2187,3125])
                      + not_fails.
:- test powers(A,B,C) : (A = [2,9999999,9999998], B = 20)
                      => (C = [2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,
                              65536,131072,262144,524288,1048576])
                      + not_fails.
:- test powers(A,B,C) : (A = [2,4], B = 6)
                      => (C = [2,4,8,16,32,64])
                      + not_fails.
```

When these (unit) tests are added, powers/3 will be run with for example $[3,4,5]$ as input A and 17 for input B. The output generated in C will then be checked to be instantiated to $[3,4,5,9,16,25,27,64,81,125,243,256,625,729,1024,2187,3125]$.

As mentioned before, while we can find errors with unit tests, hand-written test cases are tedious to write and they may not cover some cases. So random test generation can be performed. If at least one unit test fails, then random test generation is skipped. However, if all unit tests pass, test generation is performed as a last step to try to find test cases that make the assertions fail, hence revealing faults in the code.

CiaoTest will generate by default 100 cases for each assertion, or will stop before if it finds one case that does not meet the assertion post-condition. As mentioned before, this generation is random, so in order to increase the number of cases, we can also ask the user to run CiaoTest a couple of times.

If at this moment in the tutorial the students take a look into CiaoPP's output file (and also the messages issue by CiaoPP), they will see that some of the assertions left to be checked after static analysis, have been proven false by counterexamples found via test generation:

```
:- checked calls powers(A,B,C)
   : ( list_nnegint(A), nnegint(B), var(C) ).

:- false success powers(A,B,C)
   : ( list_nnegint(A), nnegint(B), var(C) )
   => ( list_nnegint(C), sorted(C) )
   + by((texec 3)).

:- texec powers(A,B,_C)
   : ( (A=[49,8,49]), (B=27) )
   + id(3).

:- false comp powers(A,B,C)
   : ( list_nnegint(A), nnegint(B), var(C) )
   + ( not_fails, by((texec 4)) ).

:- texec powers(A,B,_C)
   : ( (A=[54,54,37]), (B=97) )
   + id(4).

:- checked test powers(A,B,C)
   : ( (A=[3,4,5]), (B=17) )
   => (C=[3,4,5,9,16,25,27,64,81,125,243,256,625,729,1024,2187,3125])
   + not_fails.

:- checked test powers(A,B,C)
   : ( (A=[2,9999999,9999998]), (B=20) )
   => (C=[2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,65536,131072,262144,
     524288,1048576])
   + not_fails.

:- checked test powers(A,B,C)
   : ( (A=[2,4]), (B=6) )
   => (C=[2,4,8,16,32,64])
   + not_fails.
```

In the computation properties field of the assertions that have been marked as `false`, they can see a new property `by/1` that indicates the test case that made it fail. If they look at the lists generated, it is not too difficult to realize that there are repeated elements in them, and that dealing with this may be the source of our problems.

Again, one should keep in mind that even if `CiaoTest` does not find any cases that violate the assertion, this does not assure that the assertion is true.

The last two code fragments of this explanation have been generated by `exfilter` using the filters `check_pred` and `test`.

Chapter 5

Experimental Evaluation

In this chapter we present the experimental evaluation of the architecture proposed for the generation of interactive tutorials. We evaluate the two main parts of the architecture: the static code generation and the interactive dynamic content. We ran all experiments in a MacBook Air with Apple M1 processor and 16 GB of RAM.

5.1. Static code generation

First of all, we will summarise the static generation process scheme discussed before: first, we indicate to `LPdoc` the source files and we include in these files the calls to `exfilter` with the examples that we want to show, and the appropriate options. When we run `LPdoc` it creates output files with placeholders for `exfilter` to fill in, and warns us that we need to run `exfilter` to fill them. At this point `exfilter` is run which in turn runs `CiaoPP` and produces the filtered outputs that are to be included in the final versions of the `LPdoc` documents. Once this process has finished, `LPdoc` is run again, now incorporating the filtered outputs in the appropriate places in the document being generated.

We performed experiments with the static examples of the tutorial presented in the previous chapter, measuring the times required for each part of the process. The time `LPdoc` takes to generate the documentation is 3.1 seconds on average and the time it takes for `exfilter` to run all the analyses and fill in all the slots in all the files is 17.6 seconds on average. These steps can be unified into one (this feature is implemented and can be activated by the user if desired); in this case `LPdoc` calls `exfilter` directly. With this option activated the tutorial generation takes 30.0 seconds on average.

In any case we have decided to activate by default the division of the process into separate phases because then, when there is an error in the analysis or there is a complex example which takes longer, it is easier to control and identify which part of the process is causing the error or is taking too long.

5.2. Interactive dynamic content

In this case `Ciao` runs natively on the browser and, thanks to the absence of server calls and the relatively high performance of WebAssembly, the process is quite fast and smooth, once the system components have been loaded for the first time. The

No	Program	Filter	Supplementary filters	Execution time, ms	
				first exec	*other exec*
1	remove_power_bug	errors		1631.00	447.00
2	remove_power_bug	warn_error		1665.00	449.00
3	powers	verify_assert		2056.00	870.00
4	powers	all	name=powers	1777.00	629.00
5	powers	check_pred	comments=on	1771.00	578.00
6	sorted_insert	tpred_plus		1675.00	443.00
7	qsort_bug	errors	message=arity	1630.00	432.00
8	create_pairs	tpred_regtypes	absdomain=types	1622.00	420.00
9	qsort	test		1526.00	331.00

Table 5.1: Execution times running exfilter the first time and after the first execution.

Ciao system is modular and composed of several independent bundles. The most important bundles are loaded when starting the playground, but not all bundles are loaded by default. We have measured experimentally the performance of the playground when the bundles necessary to execute exfilter are loaded and when they are not. We have considered some of the examples shown in the tutorial presented in the section 4. The selection is shown in table 5.1. We have tried to cover a reasonable range of different filters. Column **Program** lists the name of the program analyzed. Columns **Filter** and **Supplementary filters** are the set of filters applied. Finally, **Execution time** shows the execution time of an exfilter query the first time and the execution time after the first exfilter query in our experiment. Execution times are given in milliseconds.

Table 5.1 shows the differences between the executing the first time (*first exec*) and the rest of the executions, once everything is loaded (*other exec*). Both exfilter and the CiaoPP-related bundles are only loaded in the first analysis. Although the main goal is to have a correctly working exfilter system on the browser, in order to be generate the results, the experimental evaluation also suggests that these results can be obtained for these examples in reasonable times.

As we mentioned before, WebAssembly offers performance that is reasonably competitive with native Ciao. The playground simulates the operation of the Ciao system Emacs environment. In order to get an overall idea of the cost in terms of the time taken to run exfilter in the playground when compared with running natively, on different browsers, we present Table 5.2. Code compiled to WebAssembly runs on average $1.77\times$ slower in Chrome and $1.71\times$ slower in Firefox than native code. We can see that the performance achieved is sufficient for a large range of interactive and complex tasks and to run exfilter effectively.

No Program	Execution time, ms		
	Firefox	Chrome	`Ciao`
1	$1,513$	$1,631$	952
2	$1,533$	$1,665$	960
3	$1,923$	$2,056$	$1,164$
4	$1,663$	$1,777$	$1,049$
5	$1,933$	$1,771$	$1,041$
6	$1,471$	$1,675$	969
7	$1,592$	$1,630$	784
8	$1,532$	$1,622$	965
9	$1,589$	$1,526$	778
Slowdown: geomean	$\times 1.71$	$\times 1.77$	$-$

Table 5.2: Performance comparison of running `exfilter` on the `Ciao` playground (on different browsers) vs. running on a native `Ciao` on the same machine.

Chapter 6

Conclusions

Formal methods are becoming more popular in the software industry, and accordingly more relevant in education. Motivated by this and the development of some innovative tools such as `CiaoPP`, WebAssembly, and the the `Ciao` playground we have designed an architecture to generate interactive tutorials for the `Ciao` system. The overall architecture consists of two phases: the static content generation and the interactive dynamic content generation. The idea behind dividing the architecture into two phases and not generating the whole content interactively is to save trouble for the user. The static content is only generated once and then it is not modified until there is a change in the program or in the system.

We now enumerate the main contributions of this thesis:

- First, we presented a tool capable of filtering analysis results. Without any filtering, the full analysis result can be very large and showing all parts of it in a tutorial can be tedious for the reader. In order to obtain the fragments desired `exfilter` calls `CiaoPP` and once the program is processed by `CiaoPP` the set of filters extract code fragments automatically from the analysis results. Although we have shown the available filters, these are not fixed, i.e, they may be modified or new filters can be created.

  Our tool is also useful for the ease of recording results. When the static code generation is done, results are displayed in files and these are stored. This situation enables exfilter to determinate when a bug was discovered or when it was fixed since it is possible to review the results of analysis of previous versions.

- Second, we have included `exfilter` in the `Ciao` playground. `exfilter` also includes filters that let users obtain useful feedback when trying, for example, to understand and remedy verification failures. Therefore, this allows both examples and interactive exercises to be generated.

- Finally, we tested the architecture proposed creating a new tutorial for CiaoPP. CiaoPP was selected since is a collection of static analysis and debugging tools designed to optimise and verify Ciao programs using formal methods. This new tutorial differs from those already created for the Ciao system in being capable of providing inmmediate feedback and examples allowing self-study. So far, the results shows that our approach works effectively and can be applied to other systems.

6.1. Future work

We end this thesis with some lines of future work.

Incremental analysis. Although our tool is fully functional, there are some improvements that can be made in terms of performance. `exfilter` currently performs a complete re-analysis for each set of changes, which is sometimes too costly. Incremental analysis reduces the cost of re-analysis by reusing previous information. `CiaoPP` performs incremental analysis [22] in the following way: When a module has already been analyzed, `CiaoPP` takes advantage of it and becomes more efficient and precise, since it will use the information obtained during analysis instead of (re-)analyzing the module from scratch. Integrating this feature into `exfilter` could be very useful specially when a modification is made in an example or exercise.

Semantic code search. When using the `equal` filter, `exfilter` compares syntactically the user's answer with the solution of the exercise. Even though it avoids textual differences such as variable names it still is not the ideal way to do this comparison. The solution would be to not rely in the syntaxis and instead use semantic characteristics of the program, i.e., use a technique for code search based on abstract interpretation. The process consists of taking the answer of the user and the solution and using assertions to see if the answer of the user meets the same specifications as the solution. This is done by checking the assertions statically against the preanalysis results.

Other applications. We would like to explore other applications such as using `exfilter` as an oracle to correct individual programming assignments. Custom interactive proof tools can generate a positive effect on student engagement [33, 34, 42]. This would enable students to verify immediatly their solutions against hidden "oracle" specifications. For example, an assignment can ask the student to write a linear algorithm, if it is not linear then `exfilter` gives sufficient information to debug it. Since `CiaoPP` allows stating assertions about the efficiency and complexity of the program [15, 16], adding the following computational property to the assertion: `+ steps_o(length(A))` could check if the user's solutions comply the specification.

Finally, although the framework has been presented in the context of `Ciao` programs, the same architecture will work directly in other programming languages. As we have mentioned `CiaoPP` has been applied to the analysis and verification of a number of languages (e.g Java, java bytecode, ISA, Michelson, etc) and it would be interesting also to apply `exfilter` to develop tutorials for the application of `CiaoPP` to these languages.

Bibliography

[1] E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

[2] Robin Bloomfield, Dan Craigen, Frank Koob, Markus Ullmann, and Stefan Wittmann. Formal methods diffusion: Past lessons and future prospects. volume 1943, pages 211–226, 01 2000.

[3] Jonathan Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8:189–209, 07 1993.

[4] Peter Brusilovsky. Adaptive and intelligent technologies for web-based eduction. *KI*, 13:19–25, 01 1999.

[5] Peter Brusilovsky, Elmar Schwarz, and Gerhard Weber. Elm-art: An intelligent tutoring system on world wide web. In Claude Frasson, Gilles Gauthier, and Alan Lesgold, editors, *Intelligent Tutoring Systems*, pages 261–269, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[6] F. Bueno, D. Cabeza, M. V. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[7] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. V. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd Int'l. Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[8] Holloway C. Michael and Butler Ricky W. Impediments to industrial use of formal methods. Technical report, 1996.

[9] I. Casso, J. F. Morales, P. Lopez-Garcia, and M. V. Hermenegildo. An Integrated Approach to Assertion-Based Random Testing in Prolog. In Maurizio Gabbrielli, editor, *Post-Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19)*, volume 12042 of *LNCS*, pages 159–176. Springer-Verlag, April 2020.

[10] I. Casso, J. F. Morales, P. Lopez-Garcia, and M. V. Hermenegildo. Testing Your (Static Analysis) Truths. In *Pre-proceedings of the 30th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'20)*, September 2020.

[11] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, dec 1996.

[12] Patrick Cousot. *Abstract Interpretation Based Formal Methods and Future Challenges*, volume 2000, pages 138–156. 03 2001.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[14] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Doorn, and Jakob Raumer. The lean theorem prover (system description). volume 9195, pages 378–388, 08 2015.

[15] S.K. Debray, P. Lopez-Garcia, M. V. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[16] S.K. Debray, P. Lopez-Garcia, M. V. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. Technical Report TR CLIP20/95.0, T.U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, December 1995.

[17] Bart Demoen, Phuong-Lan Nguyen, Tom Schrijvers, and Remko Tronçon. The first 10 prolog programming contests. 2005.

[18] B. Dongol, L. Petre, and G. Smith. *Formal Methods Teaching: Third International Workshop and Tutorial, FMTea 2019, Held as Part of the Third World Congress on Formal Methods, FM 2019, Porto, Portugal, October 7, 2019, Proceedings*. Lecture Notes in Computer Science. Springer International Publishing, 2019.

[19] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jsCoq: Towards hybrid theorem proving interfaces. In Serge Autexier and Pedro Quaresma, editors, *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, Coimbra, Portugal, 2nd July 2016*, volume 239 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–27. Open Publishing Association, 2017.

[20] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. The 2020 expert survey on formal methods. In Maurice H. ter Beek and Dejan Ničković, editors, *Formal Methods for Industrial Critical Systems*, pages 3–69, Cham, 2020. Springer International Publishing.

[21] I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo. Semantic Code Browsing. *Theory and Practice of Logic Programming, 32nd Int'l. Conference on Logic Programming (ICLP'16) Special Issue*, 16(5-6):721–737, September 2016.

[22] I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo. Incremental and Modular Context-sensitive Analysis. In *Workshop on Horn Clauses for Verification and Synthesis (HCVS 2021)*, March 2021. (Talk).

[23] G. Garcia-Pradales, J.F. Morales, and M. V. Hermenegildo. The Ciao Playground. Technical report, Technical University of Madrid (UPM) and IMDEA Soft-

ware Institute, 2021. Available at http://ciao-lang.org/ciao/build/doc/ciao_playground.html/ciao_playground_manual.html.

[24] G. Garcia-Pradales, J.F. Morales, M. V. Hermenegildo, J. Arias, and M. Carro. An s(CASP) In-Browser Playground based on Ciao Prolog. In *ICLP'22 Workshop on Goal-directed Execution of Answer Set Programs*, August 2022.

[25] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. 52(6):185–200, jun 2017.

[26] J.A Hall. Seven myths of formal method. *IEEE Software*, 46:11–19, 09 1990.

[27] M. V. Hermenegildo. A System for Automatically Generating Documentation for (C)LP Programs. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30 of *Electronic Notes in Theoretical Computer Science*, March 2000.

[28] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012.

[29] M. V. Hermenegildo and J.F. Morales. The LPdoc Documentation Generator. Ref. Manual (v3.0). Technical report, UPM, July 2011. Available at `http://ciao-lang.org`.

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

[31] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[32] Claudio Menghi João F. Ferreira, Alexandra Mendes. *Formal Methods Teaching: 4th International Workshop and Tutorial, FMTea 2021, Virtual Event, November 21, 2021, Proceedings*. Lecture Notes in Computer Science. Springer, 2021.

[33] Eduard Kamburjan and Lukas Grätz. Increasing engagement with interactive visualization: Formal methods as serious games. In João F. Ferreira, Alexandra Mendes, and Claudio Menghi, editors, *Formal Methods Teaching*, pages 43–59, Cham, 2021. Springer International Publishing.

[34] Philipp Körner and Sebastian Krings. Increasing student self-reliance and engagement in model-checking courses. In João F. Ferreira, Alexandra Mendes, and Claudio Menghi, editors, *Formal Methods Teaching*, pages 60–74, Cham, 2021. Springer International Publishing.

[35] U. Liqat, Z. Banković, P. Lopez-Garcia, and M. V. Hermenegildo. Inferring Energy Bounds via Static Program Analysis and Evolutionary Modeling of Basic Blocks. In *Logic-Based Program Synthesis and Transformation - 27th International Symposium*, volume 10855 of *LNCS*. Springer, 2018.

[36] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR. In *Proc. of FOPARA*, volume 9964 of *LNCS*, pages 81–100. Springer, 2016.

[37] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Proceedings of LOPSTR'13*, volume 8901 of *LNCS*, pages 72–90. Springer, 2014.

[38] P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo. Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption. *Theory and Practice of Logic Programming, Special Issue on Computational Logic for Verification*, 18(2):167–223, March 2018.

[39] E. Mera, P. Lopez-Garcia, M. Carro, and M. V. Hermenegildo. Towards Execution Time Estimation in Abstract Machine-Based Languages. In *PPDP'08*, pages 174–184. ACM Press, July 2008.

[40] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, pages 29–32, April 2008. Extended Abstract.

[41] J. Navas, M. Méndez-Lojo, and M. V. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *BYTECODE'09*, volume 253 of *ENTCS*, pages 6–86. Elsevier, March 2009.

[42] James Noble, David Streader, Isaac Oscar Gariano, and Miniruwani Samarakoon. More programming than programming: Teaching formal methods in a software engineering programme. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods*, pages 431–450, Cham, 2022. Springer International Publishing.

[43] V. Perez-Carrasco, M. Klemen, P. Lopez-Garcia, J.F. Morales, and M. V. Hermenegildo. Cost Analysis of Smart Contracts via Parametric Resource Analysis. In *Static Aanalysis Simposium (SAS'20)*, volume 12389 of *LNCS*, pages 7–31. Springer, 2020.

[44] Stephen Peverly and Rhea Wood. The effects of adjunct questions and feedback on improving the reading comprehension skills of learning-disabled adolescents. *Contemporary educational psychology*, 26:25–43, 01 2001.

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

[46] G. Puebla, F. Bueno, and M. V. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.

[47] Thierry Scheurer. Formal methods: The problem is education. In Floor Koornneef and Meine van der Meulen, editors, *Computer Safety, Reliability and Security*, pages 198 – 210, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[48] Mia K. Stern and Beverly Park Woolf. Curriculum sequencing in a web-based tutor. In Barry P. Goettl, Henry M. Halff, Carol L. Redfield, and Valerie J. Shute, editors, *Intelligent Tutoring Systems*, pages 574–583, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

Appendix A