# Packaging proofs with Why3find

Loïc Correnson

# Packaging Proofs with Why3find

Loïc Correnson

Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

With the increasing maturity of proof assistants, diving into the development of large theories is appealing, but existing toolchains might not scale. Although standard software engineering methods can be applied to mechanized proof development, specific issues shall be addressed. In this article, we focus on the Why3 platform. We present why3find, an independent tool for supporting the development of large, trustworthy Why3 packages. Why3find is designed to address common issues encountered in real world industrial developments based on formal methods. It proposes Why3-based solutions for configuring projects, managing dependencies, proving and checking proofs, tracking axioms and possible inconsistencies, extracting code, generating documentation and distributing packages.

## 1 Introduction

Software engineering practices have been profoundly modified by the development of package repositories and package managers. Developers not only need to learn programming languages, they must learn their associated package ecosystems as well. Typically, the OCaml language is now supported by the so-called OCaml Platform[1]: the OCaml package repository, OPAM, and the build system, dune smoothly operate with each other. Similarly, the Rust[2] language comes with its Crates repository of packages and the Cargo package manager and build system; this observation generalizes to all main stream programming languages, such as Java, JavaScript, Python, *etc.*

Packages are shared by developers, so they can benefit from other's previous work and build their own new software upon. Quality of external packages can be an issue, and people are generally looking at some *metrics*, *e.g.* popularity, dependencies, activity, quality of documentation, etc., in order to choose the packages they will rely on.

What about developing with proof assistants? Following the principles of the Curry-Howard correspondence where propositions are types, and proofs are programs [Wad15], crafting theories shall be regarded as software engineering. Many mature programming languages are available for *programming* proofs. Let us simply mention Coq, Isabelle/HOL or Lean. Noticeably, Coq and Isabelle also come with their associated package repositories[3],[4]

Mechanized proofs are the Graal of Formal Methods practitioners: not only do you have a nice piece a software, not only do you have a proof of its correctness, but this proof has been completely machine checked. However, some questions come up: are there issues when using third-party proof packages? How far shall we trust the development of others? What is actually checked by the compilers? All proof assistants feature the declaration of *axioms*

[1] `https://ocaml.org/docs/platform`
[2] `https://www.rust-lang.org`
[3] `https://coq.inria.fr/coq-package-index`
[4] `https://www.isa-afp.org`

and unspecified *parameters*. How many of such *hypotheses* are you importing when linking your own proofs with external packages? Does the documentation provide hints regarding such concerns?

An illustration of how important those questions can be found in the guidelines from ANSSI[5] [ANS21, §4.1, p. 12], where security evaluators are asked to use the `Print Assumptions` command from Coq environment, and to review any missing definition by hand.

In this article, we present an open-source tool, named `why3find`, whose purpose is to provide some solutions to the aforementioned issues in the context of the Why3 platform. We have developed this tool to support our own industrial and academic developments, and we think that it can also benefit to other people in the Formal Method community. Briefly speaking, `why3find` is a package manager dedicated to Why3 development that smoothly integrates with the OCaml platform through OPAM and dune. Moreover, `why3find` is designed to be compliant with security evaluation and certification processes, by generating enhanced documentation decorated with proofs evidences and hypotheses reporting. It also eases proof replay and collaborative, incremental proofs development.

The article is structured as follows: in Section 2 we present Why3 development in general. Section 3 presents the features provided by `why3find` to ease the development of proofs. In Section 4, we introduce the notion of module consistency and how we implemented it in `why3find` to track axioms and logical inconsistencies. Then we illustrate the features proposed by `why3find` for generating documentation (Section 5) and for distributing packages (Section 6). We finally conclude and present some perspectives and future work directions.

## 2 Developing with Why3

Why3 [BFMP11, FP13] is a platform developed at INRIA and University of Paris-Saclay for Deductive Program Verification [Fil11]. It provides a rich language for writing logic specifications and programming, called WhyML, and relies on external theorem provers, both automated and interactive, to discharge verification conditions. The WhyML language is designed for the development of first-order logical theories. It also offers a programming language largely inspired from OCaml, featuring algebraic data-types, mutable records, exceptions, and contract-based specifications in Hoare logic style. Furthermore, Why3 can extract executable OCaml programs from WhyML code, hence providing a way to develop correct-by-construction OCaml programs.

WhyML sources are organized in files, each file defining collections of theories and modules. Each theory and module consists of logical declarations that can be abstract (assumed) or concrete (defined). The Why3 compiler uses *Weakest Precondition Calculus* to generate verification conditions (VCs) from logical declarations and program specifications. The generated VCs are then submitted to external provers. Why3 also offers a collection of *transformations* that can be applied interactively to decompose complex VCs into smaller, hopefully simpler ones. Interactive sessions can be stored on disk and replayed later from the command-line.

Proof development is modular. When some module A depends on definitions from module B, properties of module B are *assumed* when proving properties of module A. This strategy makes proof development tractable, but makes external dependencies an issue for large development: one shall manually review that everything has been proved along *all* dependencies, which can be tedious.

The Why3 platform offers several command-line and GUI tools for supporting the development of Why3ML theories and programs:

[5] French National Cybersecurity Agency

| | |
|---|---|
| `why3 config` | for detecting and configuring provers; |
| `why3 prove` | for proving VCs (with a single prover); |
| `why3 ide` | for proving and transforming VCs interactively; |
| `why3 replay` | for checking and/or replaying saved interactive sessions; |
| `why3 doc` | for generating HTML documentation from sources; |
| `why3 extract` | for extracting OCaml programs from sources. |

A typical development with Why3 consists in repeatedly executing the following tasks: edit Why3ML source files; replay all proof sessions with `why3 replay`; interactively fix broken proofs under `why3 ide`; generate code and documentation with `why3 extract` and `why3 doc`. This process works well out-of-the-box for small or medium size projects, and one can easily automate some steps by using makefiles and shell scripts. However, when facing large projects, say thousands line of Why3 codes, software engineering problems arise.

We now describe the features proposed by `why3find` to leverage the capabilities of `why3` tools and address some issues related to large development and also common issues regarding security evaluation.

# 3 Automated Proving

Discharging verification conditions (VCs) is actually performed by running SMT solvers or other automated provers from `why3 ide`, possibly after applying transformations. This process is essentially manual although Why3 provides *proof strategies*, which consists of scripts that can automate this process up to a certain extent. Why3 offers predefined proof strategies and users can write their own ones.

Our experience on large academic and industrial projects — several thousands lines of Why3 code — is that proof strategies like `auto level 2` and `auto level 3` are generally able to tackle most proofs. The few residual proof tasks can still be tackled, provided we add some proof hints such as intermediate code assertions, lemma functions or by using (by) and (so) Why3 operators.

Hence, to simplify code base management, we decided to stick to automated proof strategies only. This method has many advantages from a software engineering perspective: (1) proof hints inside code is much more stable than looking for appropriate transformations and manually compose their parameters under why3 ide; (2) we have a simple criterion for whether a property is *provable* or not: it shall be discharged by applying the automated strategy; (3) proofs are easy to build: open `why3 ide`, select root node, click on `auto level 3`, wait for everything to be proved; (4) in case something goes wrong, apply transformations by hand under `why3 ide` to understand why the proof failed, add the necessary proof hints to the code, and replay Step (3).

Still, this iterative process suffers from some pitfalls: (a) it still requires user interaction through a graphical user interface; (b) when this process is replayed on a different machine, the resulting why3 proof session files differ, essentially because of different solver proof times; (c) checking the proof session files on a different machine is likely to fail because of inappropriate timeouts; (d) when debugging proofs, `why3 ide` still spends too much time on retrying proof branches that are likely to fail from previous runs; (e) listing all transformations actually used for a project is not that easy. Those issues make Why3 code maintenance and proof checking difficult in practice.

Alleviating those difficulties but still following our automating strategy was the primary objective of `why3find`. Our solution builds on several new ingredients: *proof certificates* which are simplified forms of Why3 proof sessions that serve many purposes, *prover calibration* to deal with machine-dependent proof times, and a *configurable* automated proof strategy. Additionally, `why3find` uses a local cache for proof results to speed up iterative proof development, and can manage a cloud of provers spread over multiple machines. We now describe in more details those new features.

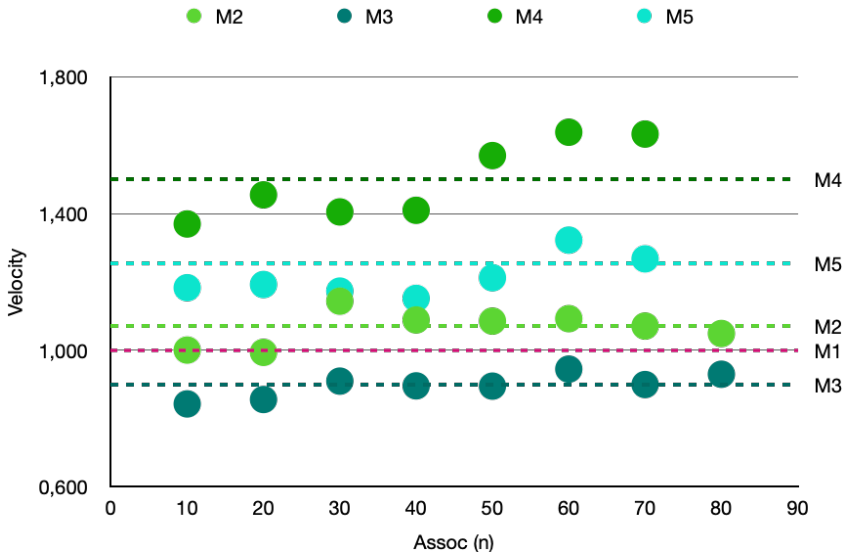


**Figure 1.** Experimenting the Velocity Law.

## 3.1 Prover Calibration

As mentioned above, replaying proofs on different machines is difficult because execution times are likely to be different from one computer to another. Of course, one could decide to multiply timeouts by a large factor, say 5 or 10, to guarantee that proofs will succeed on every machine. But when debugging proofs, this is *not* an option: waiting for all provers to reach their timeouts is far too long, and developers really want incomplete proofs to fail *fast*.

After facing too many times this issue, we decided to conduct a short study in order to measure how proof times are different from one machine to another. Our intuition is that, for given formula $\varphi$, prover $\pi$ and machine $M$, the proof time $T(\varphi, \pi, M)$ will roughly obey a linear law that only depends on the prover $\pi$ and the machine $M$. We introduce *velocity*, denoted by $\alpha^{\pi}_{M/M'}(\varphi)$, to be the ratio between proof time on machine $M$ and machine $M'$, for problem $\varphi$ with prover $\pi$. Our intuition is that velocity $\alpha$ (approximately) does not depend on $\varphi$:

$$\forall \varphi, \quad \frac{T(\varphi, \pi, M)}{T(\varphi, \pi, M')} \equiv \alpha^{\pi}_{M/M'}(\varphi) \simeq \alpha^{\pi}_{M/M'} \tag{1}$$

To test this law, we have set up a test bench with a collection of predefined, automatically generated formulas, and we have measured the associated proof times on various machines and various SMT solvers, namely Z3, CVC4 and Alt-Ergo. Choosing an appropriated family of reference proof problems was not so easy. For now, after several tries, we stick to a parameterized problem $\mathsf{Assoc}(n)$ that is formulated as follows: let $\oplus$ be an associative operator, we ask the solvers to prove that left and right parenthesized forms of $x_1 \oplus \ldots \oplus x_n$ are equal:

$$\frac{\forall xyz.\ x \oplus (y \oplus z) = (x \oplus y) \oplus z}{\forall x_1 \ldots x_n.\ x_1 \oplus (x_2 \oplus (\ldots \oplus x_n)) = ((x_1 \oplus x_2) \oplus \ldots) \oplus x_n}\mathsf{Assoc}(n)$$

Our intuition seems to be roughly correct, as illustrated in Figure 1: the diagram shows measured values of velocity $\alpha^{\pi}_{M_i/M_1}(\varphi(n))$ on different machines $M_{i\leq 5}$ for a range of formulas $\varphi(n) = \mathsf{Assoc}(n)$ with $10 \leq n \leq 80$, and prover $\pi = \mathsf{CVC4}$. For each machine $M_i$, the measured proof times are roughly dispatched around a median line that we define to be the value of $\alpha^{\pi}_{M_i/M_1}$. Similar results can be observed with other provers and other families of formulas.

The simple linear law of Equation (1) enjoys nice algebraic rules that allow us to convert proof times between different machines. For instance, we have the following rules:

$$\alpha^{\pi}_{M/M'} = 1/\alpha^{\pi}_{M'/M} \qquad \alpha^{\pi}_{M_1/M_2} = \alpha^{\pi}_{M_1/M_0} \times \alpha^{\pi}_{M_0/M_2} \tag{2}$$

We introduce the notion of *local proof time*, which is the proof time measured on a given machine, and the notion of *normalized proof time*, which is the proof time measured on one distinguished reference machine.

Each prover $\pi$ and median time $t$ used by `why3find prove` or configured by `why3find config` will be *calibrated* by looking for a parameter $n$ such that $\mathsf{Assoc}(n)$ is relevant for the configured median time.[6]

The first time a prover $\pi$ is calibrated on a machine $M$, this machine is designated to be the reference for this prover in the project. The proof time $t = T(\mathsf{Assoc}(n), \pi, M)$ is measured and the pair $(n, t)$ is recorded and versioned in the project configuration for prover $\pi$. Then, when using the same prover $\pi$ on a different machine $M'$, the *velocity* $\alpha^{\pi}_{M'/M}$ is locally measured and stored in the *local* cache of machine $M'$. It is then always possible to convert *local proof times* with prover $\pi$ on machine $M'$ to *normalized proof times* for reference machine $M$, and conversely, by using Equations (2).

## 3.2 Proof Certificates

Why3 proof sessions created by why3 ide store very precise and complete results for all provers and transformations resulting from user interactions. Despite interesting Why3 developments for reusing past sessions on modified code [BFM+], proof sessions are not really robust in practice, especially when replaying a proof on different computers, when debugging failed proofs and when propagating changes along module dependencies.

We introduce a simpler proof format that only records *one single* way to discharge proof objectives, if any, using *normalized proof times* only (See §3.1). These proof certificates are stored and versioned along with other source files of the project, so that they are shared between different machines. This is different from cached proof results and Why3 session files that are both *local* to one machine.

Proof certificates are simple trees with the following abstract syntax, where $\pi$ identifies a prover, $t$ a *normalized* proof time and $\tau$ identifies a Why3 transformation (without parameters):

$$c ::= \bot \mid \pi : t \mid \tau(c_1, \ldots, c_n)$$

Certificate $\bot$ designates an unproved goal, $\pi : t$ a proof goal that can be discharged by prover $\pi$ in normalized time $t$, and $\tau(c_1 \ldots c_n)$ a proof goal that can be transformed by applying transformation $\tau$, resulting in $n$ sub goals respectively associated with proof certificates $c_{i \in 1..n}$. A proof certificate is *complete* when it contains no $\bot$.

Proof certificates serve many purposes in `why3find`: when complete, they witness successful proofs that can be concisely described in documentation (See §5); they ease the replay of successful proofs on *different* computers, but they also ease proof updates and proof debugging under the standard `why3 ide` when incomplete (See §3.4).

Technically, proof certificates are stored with prover calibration and velocity data with the granularity of Why3 modules and shall be versioned with other files of the project and distributed with proof packages (See §6). On the contrary, Why3 proof sessions are local data that shall *not* be versioned for the project.

## 3.3 Proof Strategy

Our `why3find prove` tool implements an automated proof strategy similar in spirit to Why3 built-in proof strategies `auto level 2-3` with several improvements. In this section, we describe this proof strategy and its integration with *proof certificates* introduced above.

[6] We use a dichotomic search based on some initial hard-coded guess for a family of SMT solvers.

Consider a new proof goal for which no proof certificate is available yet. The `why3find` proof strategy consists of several steps that are tried in sequence until success:

1. all *configured* provers are first tried sequentially with a low timeout (1/5 of the *local, median* time);
2. then all provers are tried in parallel with the normal timeout (*local, median* time), and interrupted as soon as one succeeds;
3. if no prover succeed, *configured* transformations are tried in sequence; we stick to the first transformation that can be applied and sub-goals are queued so that the proof strategy can be recursively applied on them;
4. finally, if no transformation is applicable, all provers are lastly tried in parallel with twice the normal timeout, like in step 2.

The key differences with built-in Why3 proof strategies are: (a) our proof strategy is easily and globally configurable (provers, median time, strategies); (b) we introduce sub-second timeouts (using 1-2s median time on modern computers *is* definitely an option); (c) we introduce a novel fail-fast prove-fast heuristic in Step 1. Actually, in many projects, we observed that there is *one* distinguished solver which is faster than the other ones at solving *most* of the generated proof goals. In such situations, we configure this particular solver to be the first one to be tried during Step 1 of our proof strategy. This choice is quite efficient: it limits the number of provers scheduled in parallel for most goals, and finally saves a lot of CPU load and time.

In Step 3, we do *not* backtrack on proof transformations for which the generated sub-goals would be incomplete in the end. This is important in order to fail as fast as possible on *incorrect* proofs. We experiment that it is more efficient in practice to spend time on debugging proofs and introducing proof hints inside Why3 code, compared to exploring *all* the possible proof trees that can be generated by applying every transformation on every proof node.

When all proof tasks have been dequeued, proof certificates (complete or not) are generated and stored on disk.

## 3.4 Incremental Proofs

Consider now a proof goal for which a proof certificate has been stored from a previous run. We have no guarantee that this proof certificate is still applicable: maybe the code for this goal or any of its module dependencies have been modified. Proof certificates are associated to proof goals by names (using module path, module name and declaration name), hence previous proof certificates can still be used as an initial *guess* on how to prove our possibly updated proof goal. At least, possibly some prefix of the proof certificate will still apply to the new goal.

Compared to re-exploring the entire proof tree from scratch, using *some* proof certificate as an initial guess, initially complete or incomplete, is very efficient in practice. We observed that proof certificates are quite stable during incremental proof development.

In case of failure, the `why3find prove -i` option can be used to interactively fix the proof: it generates on-the-fly a local Why3 proof session and launches a pre-configured `why3 ide` so that the user can debug its proof. Moreover, compared with a full Why3 proof session generated by `why3 ide`, every proved sub-goals is already registered with a complete proof tree generated from the certificate, so that the user can quickly focus on the failed goals and `why3 ide` will no longer waste time to re-try proof attempts that are doomed to fail. Finally, `why3 ide` is pre-configured with custom Why3 proof strategies that mimics the different steps of `why3find` proof strategy, with respect to the current project configuration.

This process reveals to be user-friendly and very efficient in practice. Still, we optimized it a bit further. First, each proof replay indeed generates a new proof certificate with different

$$\bot \simeq \bot \qquad \frac{t/2 \leq t' \leq 2.t}{\pi : t \simeq \pi : t'} \qquad \frac{c_i \simeq c'_i}{\tau(\overline{c}) \simeq \tau(\overline{c}')}$$

**Figure 2.** Similar Proof Certificates.

proof times, even if we use calibration. Second, when a proof fails, we generally obtain a proof tree with only few leaves to be completed. Once fixed, probably some intermediate nodes of the tree can now be solved by provers without requiring a transformation to be applied. However, since we still use the previous proof certificate as an initial guess for the updated goal, all previous intermediate transformations will still be applied.

To avoid those two issues, we proceed as follows. First, we define a relation of *similarity* between two certificates: two certificates are *similar* if they only differ by proof time on their leaves, and all the proof times are within half-to-twice from each others (see Figure 2). After proof replay or completion for a given proof goal, the newly computed proof certificate is stored on disk only when it is *not* similar to the previously existing one for the same goal. This strategy greatly improves the stability of proof certificates when synchronizing projects between different machines.

Second, when storing an incomplete proof certificate, every transformation node with only incomplete children is removed and replaced by $\bot$. This avoids keeping doomed proof trees that do not really simplify the initial proof goal.

Last, but not least, we implement `why3find prove -m` option that tries to *minimize* proof certificates: when reusing a complete proof certificate as a guess, we try to replace transformation proof nodes $\tau(c_1, \ldots, c_n)$ with a single prover call by applying Step (2) of our automated proof strategy instead. This operation can be costly, although it is optional: large proof certificates are still correct and can still be used to replay proofs automatically.

The `why3find prove` command provides options to further tune the proof strategy and how proof certificates are reused and/or updated.

## 3.5 Speeding Up Proofs

Whatever proof task is performed, for a proof goal or for calibration, all prover results are stored and cached locally by default. We use the hash of proof tasks computed by Why3 to efficiently associate prover results to proof goals. Proof results are automatically upgraded or downgraded with respect to timeouts (*e.g.* a success with proof time 2.5s becomes a timeout for asked time 1s).

Using the cache during development is necessary in practice. It saves a significant amount of time during the process of incrementally developing and fixing proofs. Moreover, it also makes proof minimization not that costly, since proof attempts that are known to fail cost nothing but a disk access. However, the `why3find prove` command provides dedicated options to further tune or disable cache management.

A key principle of `why3find` proof strategy is to *fail fast* on incorrect proofs. For this purpose, when new proof tasks are dequeued, we give priority to tasks with *incomplete* certificates, and `why3find prove` can be asked to stop on the first incomplete proof. This strategy, combined with cache management and launching pre-configured `why3 ide` on-demand, makes proof development quite user-friendly and efficient.

## 3.6 Proof Server

Both Why3 and `why3find` schedule proof tasks in parallel to benefit from all available cores of the local machine. However, many proof tasks must wait for an available slot from `why3` proof scheduler. Moreover, when several developers want to collaborate on a given

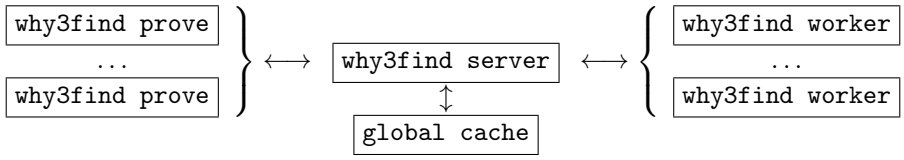


**Figure 3.** Proofs in the Cloud.

project, they all need to install the same provers, with the same versions, on their respective machines.

To alleviate those issues, `why3find` can establish a proof server and run a cloud of workers that can dynamically offer their cores and their solvers to proof developers. The proposed architecture is illustrated in Figure 3.

Servers and workers are implemented with the Zmq library[7] which offers built-in support for robust server-worker architecture with automatic load-balancing facilities. The `why3find server` command establishes a server with a centralized on-disk proof cache, and waits for developers and workers to connect.

Proof tasks emitted by developers when running `why3find prove` are automatically submitted to the Server, which will immediately reply when the proof task is already in the centralized proof cache. This first query is fast since only the hash of the proof task is exchanged with the server. In case the proof task is unknown, the Server asks for the proof task input file and stores it in its global cache.

On the other side, workers can be created with command `why3find worker` to offer their cores and provers to the Server. Pending proof tasks received by the Server are submitted to available workers. Proof results are sent back to the Server, which stores them in the cache and sends them back to the emitters, provided they are still connected and waiting for them.

Developer processes can also play the role of workers. Actually, when `why3find prove` emits a proof task, it is sent to the Server *and* locally scheduled on its own cores. Depending on the respective machine loads, any of the working processes may reply first. The mission of the Server is to orchestrate and synchronize all these tasks.

The Server's global cache is more sophisticated than the `why3find prove` local caches. Where local caches on each developer's computer can be easily erased to save space, we want a more clever cache management on Server side. We currently use a multi-layered cache. New entries are always inserted in top-layer 0. When some entry is not found in layer $n$ it is looked up in layer $n+1$ until it is found. Once an entry is retrieved from layer $n > 0$, it is immediately promoted to layer 0. Periodically, Server maintainers might ask `why3find server` to prune the global cache up to layer $N$: each layer $k < N$ is renamed into layer $k+1$ and layers $k \geq N$ are simply erased. This simple heuristic makes live queries to remain in layer 0, and layers $n > 0$ to store all queries that were not claimed since the last $n$ maintenance periods. The `why3find server` command can be used to obtain statistics on cache layers in order to ease cache maintenance.

# 4 Checking Proofs Consistency

The Why3 platform implements and promotes a modular proof strategy: when asked to prove a property from module $A$, all other results from module $B$ that $A$ depends on are assumed as hypotheses. This is a mandatory feature to have in order to scale and to develop large theories, otherwise proof development would be exponential in the number of proved properties. However, this modular proof strategy implies that proofs for module $A$ are

[7] `https://zeromq.org`

complete if, and only if, all proofs from module $B$ that $A$ depends on are also complete, and so on. Off the shelf, Why3 provides no tool to check such a completion.

Moreover, the Why3 language allows the introduction of declarations with *no* associated definition: one can introduce abstract types, undefined logic functions and predicates, *axioms*, and undefined functions with *assumed* post-conditions and effects. It is even possible to *admit* a code annotation within a function definition.

To fully trust a Why3 development, we should be able to answer the following questions:

(a) Given a module $A$, are all of its properties proved?

(b) If module $A$ depends on module $B$, are all properties of $B$ also proved?

(c) Does module $A$ have assumed hypotheses? Is it possible to fulfill or justify them in some way?

(d) What about $B$ hypotheses when $A$ depends on $B$?

There are some problematic code patterns that an evaluator shall be aware of. For instance, the following abstract declaration introduces an inconsistency whenever function `f` is invoked, since it is admitted that its returned value would be strictly both positive and negative:

```
val f () : int
  ensures { result > 0 }
  ensures { result < 0 }
```

To avoid such inconsistencies in admitted properties, our idea is to ask developers to witness *some* possible definition for abstract declarations and to prove them to be valid. Hence, one can be definitely sure that abstract definitions are free of inconsistencies.

Fortunately, Why3 provides a feature for refining modules, called module cloning, that we will investigate in more details soon.

But even *defined* and *proved* declarations might drive us into dire straits. Consider for instance the following definition for abstract function `f` above:

```
let f () : int
  ensures { result > 0 }
  ensures { result < 0 }
  = assume { false } ; any int
```

This function definition is trivially proved by Why3, although the logical inconsistency is still there. The problem comes from the assumed clause *inside* the definition of the value, which is not required by Why3 to be proved, contrarily to regular `assert` clauses. Unfortunately, as far as we know, Why3 does not offer any way to refine further such an assumed clause. Hence, we shall pay special attention to them.

As a matter of fact, providing evidences of *proofs completeness* — Questions (a) and (b) — and *proofs consistency* — Questions (c) and (d) — is not so easy when dealing with large code bases. Moreover, for developers to rely on properties provided by third party libraries, they will expect convincing and concise answers to such questions.

Since the main purpose of `why3find` is to provide packaging and evaluation-oriented features for Why3 developments, we shall address those issues. Our proposal is to systematically look for proof completeness and track possible sources of inconsistencies and to report on them, typically when generating documentation.

The remaining of the section briefly introduce Why3 module cloning and how it can be used to provide witness of absence of logical inconsistencies.

## 4.1 Why3 Module Cloning

From the definition of the WhyML language, a Why3 module consists of a collection of types, logic constants, functions and predicates, axioms and lemmas, and functions with contracts.

Declarations can be abstract, which means that the module is generic and that it can be refined by assigning concrete definitions to abstract declarations. Refinement is introduced by using module cloning declarations, with the following syntax:

```
module M
  ...
  clone A with x_1 = a_1, ..., x_n = a_n
  ...
end
```

Such a declaration inserts all the declarations from module $A$ into module $M$ after substituting abstract declarations $A.x_i$ with $a_i$. Each $a_i$ can be either abstract or concrete depending on the context, and unless specified, axioms from $A$ are turned into lemmas when inserted in $M$. On the contrary, lemmas from module $A$ are treated as axioms from $M$'s point of view, since Why3 already issued appropriate proof obligations for them when proving module $A$, like module dependencies. When refining abstract functions with contracts, the new target function contract is proved to be consistent with the abstract one.

## 4.2 Proof Consistency

The problem with abstract module declarations is that they can introduce inconsistencies, as illustrated in the introduction. Without diving here into the complete semantics of the WhyML language [Fil13], logical inconsistencies might occur when there is no way to refine an abstract declaration with a concrete definition such that associated hypotheses (axioms and contracts) can be fulfilled.

Hence, if module $A$ has been cloned inside some module $M$ in such a way that all abstract definitions from $A$ are assigned to concrete definitions, then proving $M$ entails that $A$ is free from logical inconsistencies.

Of course, if module $M$ or module $A$ have dependencies on other modules, they can also be sources of inconsistencies. Partially cloning a module also introduces abstract declarations that shall be taken into accounts.

To avoid this problem, we introduce the notion of module consistency. Module $A$ is defined to be consistent if and only if there exists a module $M$ that clones $A$ such that after taking cloning into account:

(a) Module $M$ has no residual axiom and no residual abstract function contract.

(b) The definitions of $M$ — and $A$ — are free of assumed clauses.

(c) All dependencies of $M$ — and $A$ — are consistent modules.

Notice that we do not require abstract types, constants, pure logical functions and predicates to be assigned a concrete definition. Indeed, since every type is *inhabited* by construction in Why3, provided module $M$ has no more axioms, any residual abstract declarations may be assigned any concrete value of the expected type without introducing any logical inconsistency.

## 4.3 Checking Module Consistency

Proof consistency is checked by `why3find` when proving a module and when generating package documentation. For every module $A$ with abstract declarations, it checks for local

consistency inside the modules of the package only. This allows for reporting consistency at the package level, without taking into account future module instances from package clients.

Moreover, since modules of Why3 standard library are fully realized in Coq, they can be considered to be fully consistent as well, even if their Why3 declarations contain bundles of axioms and abstract values.

Local proof completeness and consistency is reported by the `why3find doc` tool inside the generated documentation (See §5), and by `why3find prove` on demand.

For proof developers, witnessing evidence of soundness for a module with abstract definitions is relatively easy: one "just" needs to create, somewhere in the package, a clone instance with enough concrete definitions to be fully proved. In practice, we always succeed in providing a concrete witness of consistency for every module of our largest projects. Moreover, during this process, we have sometimes observed that some hypotheses were missing from our initial design in order to make the abstract module realistically implementable.

Computing and consolidating statistics from proof certificates and local module consistency actually provides a concise and relevant feedback regarding the trust base of Why3 developments. Moreover, it can be smoothly integrated to the generated documentation, see Section 5 for more details.

# 5 Providing Documentation

Generating a good documentation for formal developments is not an easy task. As an illustration, for documenting the Coq development of CompCert [Ler09], X. Leroy developed a dedicated tool, coq2html[8] to overcome the limitations of Coq built-in documentation generator.

The built-in documentation generator distributed with Why3 also has its own limitations. In particular, it does not offer simple typographic facilities, and it provides no feedback on proved goals and hypotheses. Moreover, cross-referencing documentation generated from different Why3 developments is not possible (but for the standard library) and this is an issue when distributing package documentation.

Hence, we provide an enhanced documentation generator, `why3find doc` that addresses those issues. The main additional features of our documentation generator are: (a) limited support for Markdown styling; (b) foldable code parts, paragraphs or entire sections (*e.g.* to hide proofs); (c) proof certificates and feedback regarding completeness and consistency; (d) inlined clone definitions (useful for nested and exported clones); (e) inter-package cross-references; (f) additional documentation-only chapters.

Regarding proof completeness, the documentation generated by `why3find doc` outputs, for each Why3 declaration, the associated proof certificate and its completeness. A summary is also consolidated by module, by source file and by package: each identifier in the generated HTML documentation is accompanied by small colored icons and detailed tooltips to summarize whether the proof certificates are complete or not.

Similarly, local proof consistency is reported in the generated documentation with small colored icons, and tooltips with detailed statistics on proof consistency. Additional information is provided for consistency, namely the list of sound instances found (when consistent) or the list of incomplete instances (when not fully consistent).

Such facilities ease the work of evaluators when they examine the trust base of a development with Why3. Since all packages and modules have concise summaries of their dependencies, abstract parameters and axioms, proof completeness and proof consistency, an evaluator can easily navigate to the residual missing parts and forge its opinion on robustness with respect to the documentation provided by the proof developers.

[8] `https://github.com/xavierleroy/coq2html`

# 6 Packaging & Distributing

The development of a (large) project with Why3 requires managing some configuration data, including source files, dependencies, proof strategies, *etc.* All those data are generally materialized in shell scripts or in Makefiles together with additional data and commands to package the development and make it available for distribution.

With `why3find`, we propose a collection of command line tools to manage the project configuration data and ease its packaging and installation process.

More precisely, `why3find config` will manage the following data for a given project: (a) the other packages the project depends on; (b) the provers and transformations to be used (See 3.3); (c) the *median time* used to define normalized timeouts (See 3.1); (d) the number of parallel proof jobs; (e) the proof server to connect, if any (See §3.6). The `why3find config` provides many options to print, update or reset the global project configuration. Configuration data is stored in a configuration file that shall be versioned with the source files of the project.

When a Why3 development has been properly proved and documented with `why3find`, one may want to distribute it and make it available for other projects. To distribute a Why3 package, we decide to rely on the OPAM and dune infrastructures of OCaml Platform. Actually, recent versions of dune offer a nice feature, named *installation sites* for centralizing a local repository of files from different packages. Hence, `why3find` has its own dune installation site and the `why3find install` tool will automatically generate a dedicated `dune` file to install all the required artifacts for a Why3 package: configuration, sources files, proof certificates, prover calibration *and* documentation.

Moreover, `why3find` also provides features to ease the extraction of OCaml code from Why3 sources. In particular, `why3find extract` produces a `dune` file for compiling the extracted code, so that it can also be linked or installed with other OCaml source files in the project.

All tools provided by `why3find` actually know how to deal with installed packages, for proving, extracting and generating documentation. Altogether, we hope that those integrated features would bring the packaging and distribution of Why3 packages to real life.

# 7 Conclusion

We have presented a prototype tool, `why3find`, that provides facilities upon the standard Why3 tool box. From our experience on large academic and industrial projects, `why3find` really facilitates the work of both proof developers and proof evaluators and offers an infrastructure based on the OCaml Platform for distributing Why3 packages. We plan to release `why3find` in open source via OPAM within a very near future.

Although already functional, `why3find` can still be improved in many directions. Proof automation can still be improved, and our experimental model for prover calibration shall be further tested and explored: the level of automation currently provided by the tool opens the route for experimenting new heuristics and systematically evaluating their benefits. The documentation generator can also be improved in many ways. Finally, proof consistency would deserve to be refined and formalized in a proof assistant.

We also expect feedback from the community and possible integration with other proof platforms such as Coq, Isabelle or PVS. For now, we have only proposed our own, opinionated, proposals to overcome some common issues in large development with formal methods and Why3 in particular, although we are opened to external contributions. Our intention is really to contribute to make Why3 a full-featured platform for the collaborative development of mechanized proofs and proved libraries.

**Acknowledgments.** I would like to warmly thank the JFLA reviewers for their kind suggestions and insights. I would also like to thank Benjamin Jorge for its fruitful contributions regarding the proof strategy of Why3find. Finally, I would like to thank all the developers and security evaluators I've met so far for their instructive feedback regarding the usage and the deployment of formal method tools in their respective settings.

# References

[ANS21] Requirements on the use of coq in the context of common criteria evaluations. Rapport technique v1.1, ANSSI & INRIA, décembre 2021.

[BFM+] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond et Andrei Paskevich : Preserving user proofs across specification changes. pages 191–201.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché et Andrei Paskevich : Why3: Shepherd your herd of provers. *In Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. `https://hal.inria.fr/hal-00790310`.

[Fil11] Jean-Christophe Filliâtre : Deductive Software Verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397–403, 2011.

[Fil13] Jean-Christophe Filliâtre : One logic to use them all. *In* Maria Paola Bonacina, éditeur : *Automated Deduction – CADE-24*, pages 1–20, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[FP13] Jean-Christophe Filliâtre et Andrei Paskevich : Why3 — where programs meet provers. *In* Matthias Felleisen et Philippa Gardner, éditeurs : *Proceedings of the 22nd European Symposium on Programming*, volume 7792 de *Lecture Notes in Computer Science*, pages 125–128. Springer, mars 2013.

[Ler09] Xavier Leroy : Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[Wad15] Philip Wadler : Propositions as types. *Commun. ACM*, 58(12):75–84, nov 2015.