Coqoon

Alexander Faithfull, Jesper Bengtson, Enrico Tassi, Carst Tankink

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Coqoon

An IDE for interactive proof development in Coq

Alexander Faithfull[1], Jesper Bengtson[1], Enrico Tassi[2], Carst Tankink[2⋆]

[1] IT University of Copenhagen
[2] Inria - France

The date of receipt and acceptance will be inserted by the editor

Abstract User interfaces for interactive proof assistants have always lagged behind those for mainstream programming languages. Whereas integrated development environments (IDEs) have support for features like project management, version control, dependency analysis and incremental project compilation, "IDE"s for proof assistants typically only operate on files in isolation, relying on external tools to integrate those files into larger projects. In this paper we present Coqoon, an IDE for Coq projects integrated into Eclipse. Coqoon manages proofs as projects rather than isolated source files, and compiles these projects using the Eclipse common build system. Coqoon takes advantage of the latest features of Coq, including asynchronous and parallel processing of proofs, and—when used together with a third-party OCaml extension for Eclipse—can even be used to work on large developments containing Coq plugins.

1 Introduction

In the last decade, computer-aided proof development has been gaining momentum. Interactive proof assistants allow their users to state a theorem in a language that the system understands and then prove that theorem within the system. As long as the proof assistant's verification code is free from bugs, this guarantees that all proofs are actually correct, that no details have been overlooked, and that no mistakes were made. Mechanizing proofs in this way makes very large proofs feasible and protects against subtle and hard-to-notice human errors. Two recent milestones in computer science include the verification of an optimising C compiler [6] and of a micro-kernel [16]. Proof assistants have also been used to verify advanced results in mathematics, such as the Odd Order Theorem, using Coq [11], and the proof of the Kepler conjecture, using HOL-Light and Isabelle [13].

Meanwhile, on the other side of the great chasm between theory and practice, software developers too have come to appreciate computer assistance as they work. For a developer, however, that assistance comes not in the form of a proof assistant, but of an *integrated development environment* (IDE).

The IDE combines many important tools of the trade—such as editors, compilers, refactorers, profilers, debuggers, and project and release managers—into a single unified toolbox for working with code. At a glance, the developer has an overview of every aspect of a project, and the repercussions of changes in one area can be shown in every other affected area, allowing the developer to make any necessary corrections. Many IDEs can even abstract away the build process entirely, automatically inferring the relationships between source files and libraries and rebuilding them when necessary.

The workflows of interactive proof assistants are sufficiently similar to conventional programming languages that one might expect IDEs to exist for them as well, but this is not the case. Even though proof assistants are becoming more popular, there are still no real IDEs for them—none of them are truly *integrated*. Coq is one of the most widely-used proof assistants available, but its proofs are most often written using either Proof General or CoqIDE; these specialised text editors operate only on individual files, leaving project management entirely to developers. Projects are typically built using Makefiles, which require a POSIX-like environment; file dependencies are supported via command-line tools; and complex inter-project dependencies are not supported at all, leaving the work of building and linking projects together up to the user. Moreover, both Proof General and CoqIDE have a workflow, often referred to as the *waterfall* model, in which the editor is only aware of the state at one specific point: to view the state elsewhere,

⋆ Funded by the Paral-ITP ANR-11-INSE-001 project.

the user must either execute all commands up to the desired point or explicitly revert to an earlier point in the document, throwing away all the computations back to that point in the process. This workflow is not only alien to software developers, who are used to being able to edit their files at arbitrary points and receive immediate feedback from the IDE on what effect these changes had on the rest of their development, it is also very slow (although upcoming versions of CoqIDE improve this situation somewhat).

We argue that the lack of tool support for proof assistants is to the detriment of both theoreticians and software developers with an interest in verification. Requiring that developers learn an old-fashioned workflow in order to try out formal methods is unquestionably a deterrent, but that workflow is also a waste of time and effort for those who have grown accustomed to it. Integration and automation have made life easier for programmers: why should the same not also be true for proof authors?

In this paper we present Coqoon—an Eclipse-based IDE for proof development using the Coq proof assistant. Coqoon includes support for Coq projects, much like Eclipse's built-in support for Java projects: users can create Coq projects, structure these projects using folder hierarchies, and add Coq source files to these folders, and the Eclipse build system will automatically keep track of the project dependencies behind the scenes. Whenever a file is changed, moved, or renamed, everything that depends on it is automatically recompiled, and any errors are reported to the user.

Coqoon does away with the waterfall model, instead allowing the user to make changes anywhere in a file—and automatically and asynchronously reproving only the parts that are affected by that change. In this way, Coqoon behaves a lot more like an IDE that software developers are familiar with than the tools available to Coq developers today.

Coqoon is also an *integrated* development environment in the fullest sense of the term. Eclipse has a wide variety of plugins available, ranging from version control plugins like EGit to entire development environments like OcaIDE for OCaml, which can be used alongside Coqoon. The combination of Coqoon and OcaIDE is particularly useful, as it brings support for complex Coq developments that contain both proofs and OCaml plugins.

Coqoon depends on features added to Coq in version 8.5 allowing it to support Wenzel's PIDE library for asynchronous proof developments [24], which supports Coqoon's replacement for the waterfall model. Coq 8.5 also adds a two-step compilation process, known as the *quick compilation chain*, that can optionally produce `.vio` files in place of standard Coq `.vo` libraries; this new format produces larger, faster files whose proofs are unchecked, but which can later be efficiently compiled into the traditional format by checking the remaining proofs in parallel.

As a test case, we have imported the mechanised proof of the Odd Order Theorem into Coqoon, which is one of the largest Coq 8.5-compatible developments available. Previous versions of this project took over two hours to compile, but, using the quick compilation chain, the project can be compiled into `.vio` form in just seven minutes, and then into `.vo` form in a further twenty minutes. A user can resume working on the project directly after the first, shorter, compilation pass is completed. Coqoon is the first IDE to include native support for the quick compilation chain—indeed, no other IDE for Coq has an integrated build system—which makes it possible to work with even the most complex projects at speeds that were hitherto unimaginable.

We have also adapted Pierce's course on Software Foundations to be compatible with this version. This development contains plenty of exercises that demonstrate a wide variety of features of Coq, and can be used to try out Coqoon's capabilities in a smaller setting than the Odd-Order Theorem.

Download links and installation instructions for Coqoon, along with pre-packaged example projects, can be found at https://coqoon.github.io/.

2 Coqoon, structured projects, and the build system

Coqoon is a family of plugins for the Eclipse framework that together implement an IDE for Coq developments. It has support for structuring these developments using Eclipse workspace projects, folders, and files, and for automatically managing the verification and build processes as dependencies change. To allow more interactive development of proof scripts, Coqoon processes them in the background, showing Coq feedback directly in the proof text editor using idioms familiar to programmers (e.g., by underlining errors in red).

As Coqoon is implemented on top of the Eclipse framework, it interoperates with other Eclipse components: version control plugins like EGit [9], for example, can be used with Coqoon projects.

2.1 Structured projects

Coqoon provides a more structured environment than Coq programmers are accustomed to. From the moment the user first creates a Coq project in Coqoon, it already has a complete build system; Coq source code must be placed in designated source folders, and when files start to depend on other files, they will automatically be marked for recompilation when their dependencies are moved or changed, even if those dependencies are in other projects. A progress bar—visible at the bottom

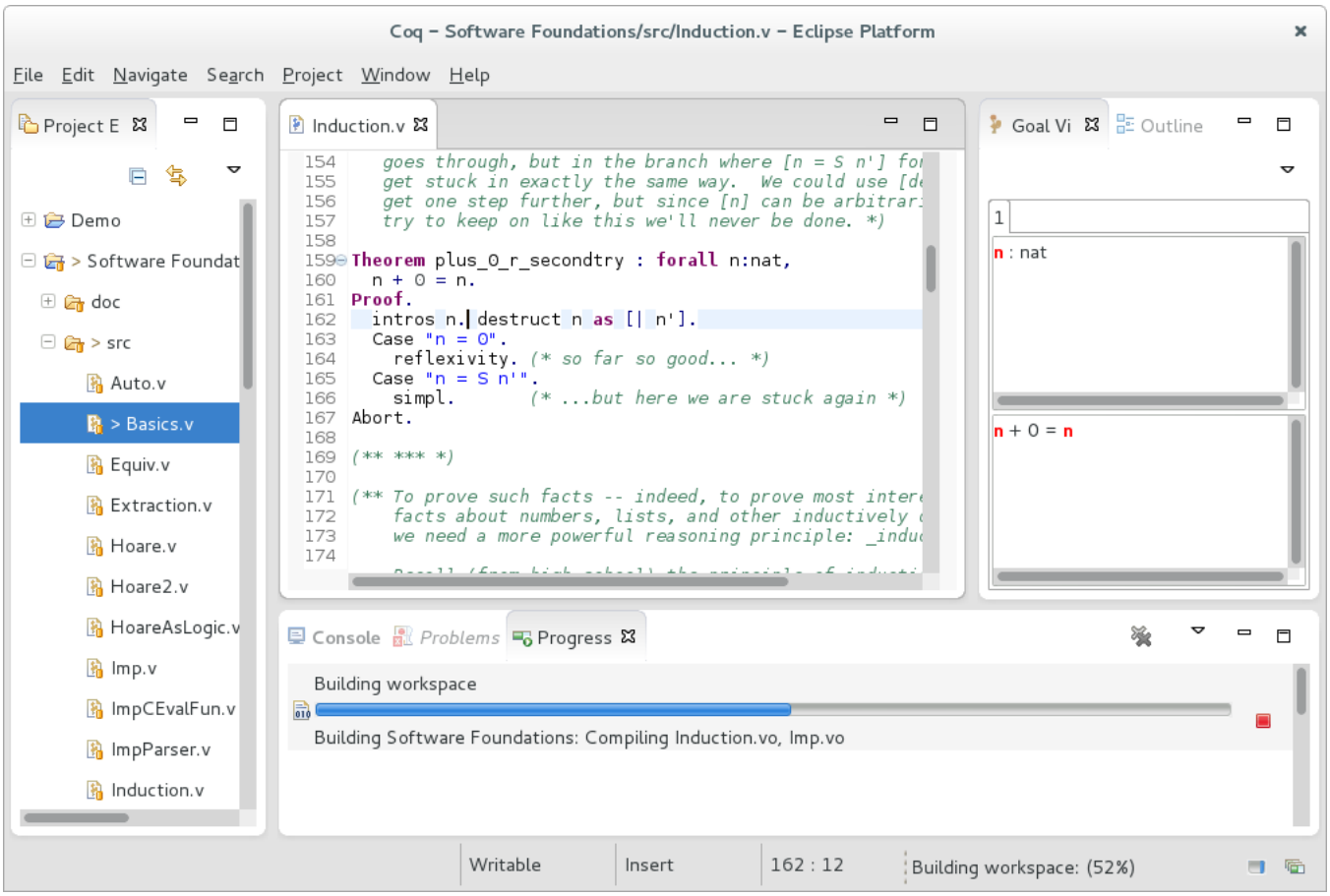

Figure 1: A screenshot of Coqoon, showing the project viewer, a Coq editor with syntax highlighting, and the goal viewer. The progress bar at the bottom of the screen shows the Coq project builder at work.

of Figure 1—displays the state of any build operations scheduled by the builder.

This need for structure is not just the IDE being difficult: it is precisely this structure that makes more sophisticated behaviour possible. The use of the Coqoon integrated build system allows Coq projects to support dependencies on other projects, or on external developments, whilst simultaneously freeing the developer from the need to think about the build system and making it work on all operating systems supported by Eclipse.

2.2 The Coq model

Replacing unstructured collections of files with structured projects is a start, but most IDEs go further. They transform source code files into a more structured representation (known as a *model*), providing a higher-level way of searching and manipulating code than simple operations on plain text. Eclipse's Java model, for example, presents Java documents as abstract syntax trees; nodes in the tree that represent identifiers can also be "resolved" through the model to see what they refer to in that particular context.

Coqoon provides a similar model for Coq code. Most Coq-specific operations on files begin by using the Coq model to convert an Eclipse file handle into a Coq model file handle, which presents an alternative view of the file as a sequence of parsed and tagged Coq sentences. The model also serves as a central place to cache these sequences, so files whose content has not changed do not need to be reparsed.

In general, the goal of this model is to provide a useful, Coq-centric view of the data and metadata of an Eclipse project. Coq project handles, for example, have methods for retrieving and modifying project configuration information, and projects can be traversed using the visitor pattern [10] to find named lemmas and definitions, making search algorithms easy to build. Errors produced by the compiler, or by an interactive session, are also stored in the model, which automatically maps them to Eclipse error markers.

The design of Coqoon's Coq model is heavily based on that of Eclipse's own Java model, which has led to the internal use of some Java concepts in areas where Coq lacks any particular convention: the Coqoon model considers projects to consist of Java-style package roots

(top-level source and output directories) containing package fragments (those subdirectories of a root which contain source and output files), for example, although the concept of a package is not one native to Coq.

2.3 Coq interaction

Coqoon's integrated Coq editor communicates using the PIDE library. Originally developed for the Isabelle proof assistant, PIDE frees the user from having to explicitly direct the prover to make progress through a source file: proofs handled by PIDE are evaluated in parallel and out-of-order, and Coq's state after the evaluation of each sentence is saved, making it quick and easy to see how tactics affect the state of a proof. Section 3 explains the operation of the PIDE protocol in more detail.

When using PIDE, the operations of the prover in the background are transparent to the user. Any status messages and goal information associated with a command will automatically be displayed when the user moves the text cursor onto it, and errors are highlighted when the user makes a mistake.

This mode of interaction is much more convenient for the user, and is customary for IDEs for traditional programming languages, but is not widely supported by theorem prover interfaces.

2.4 The Coq build process

When a Coq file is added to, modified in, or removed from a Coq project, the integrated Coq builder is activated. The builder is responsible for compiling all of a project's Coq proofs into library files.

Whenever the Coqoon builder is activated, Eclipse provides it with a summary of the changes to the project since its last activation. The builder then uses the Coq model to extract the new dependency information from the changed files; it then rebuilds all the changed files and their dependents in an appropriate order, postponing the compilation of a file until its dependencies are also up-to-date.

This behaviour is common to virtually all IDEs, and—although it is not supported by existing Coq interfaces—many projects have built ad-hoc emulations of it for themselves. At the time of writing, for example, the CompCert project contains a pair of shell scripts—one for use with CoqIDE, and one for use with Proof General—which compiles the dependencies of a file, opens it in the appropriate editor, and recompiles that file when the editor is subsequently closed.

Projects and hybrid projects

All Eclipse projects are collections of files coupled with Eclipse build system metadata which expresses that project's specific requirements. A Coqoon project consists of Coq source code, information about the project's internal structure and its dependencies, and an instruction to the Eclipse build system explaining that the project is a Coq development under the control of the Coqoon builder.

This mechanism is sufficiently general that a project's metadata can have multiple instructions for the Eclipse build system—for example, a Coq project might declare that a bundled plugin is to be built with an OCaml builder. Indeed, a copy of Eclipse equipped with Coqoon and OcaIDE, an OCaml IDE for Eclipse[7], serves as a complete development environment for Coq projects with OCaml plugins; section 6.1 describes such a scenario in more detail.

2.5 Project dependencies

The Coqoon builder uses a project's load path to resolve the dependencies present in that project's source code, and also provides a user interface for manipulating those dependencies. This functionality is loosely inspired by the Java class path management found in Eclipse Java projects.

From Coq's perspective, the load path is simply a list of filesystem paths and their corresponding abstract Coq paths. To make this more portable, Coqoon represents a load path as a series of portable *providers*, each of which expands to a machine-specific fragment of the Coq load path.

There are five different kinds of load path providers:

- folders in Eclipse projects that contain source code files;
- folders in Eclipse projects that contain compiled source code;
- other Coqoon projects in the Eclipse workspace;
- folders in the local file system containing projects neither built with nor managed by Coqoon; and
- "abstract" entries which are likely to be available everywhere but whose location cannot be known in advance, like the Coq standard library.

The builder calculates how each of these should be represented in the Coq load path, and uses this information to resolve the dependencies of each file in the project.

The dependency resolution process is sophisticated enough to recognize when to prefer files that have yet to be compiled to files that are already available: if a project contains a file called `List.v`, for example, then other files in that project can safely depend on its compiled form by depending on `List`, even though the Coq standard library contains an identically named file which could potentially satisfy that dependency.

2.5.1 Abstract dependencies

When Coqoon encounters an "abstract" entry, it looks at that entry's identifier and searches an internal registry for a class that knows how to handle that identifier. These classes can then run arbitrary code to resolve the identifier; for example, the handler for the Coq standard library finds it by running a `coqtop` process with the `-where` option.

As this internal registry is also exposed through the standard Eclipse extension mechanism, it can also be used to add new kinds of dependencies to Coqoon; see section 4.4 for an example.

Aggressive rebuilding

Coqoon's internal dependency analysis behaves like that of `make`: when a file is older than one of its dependencies, it becomes a candidate for recompilation. As a result, making changes to a file with many transitive dependencies will trigger the recompilation of many other files.

As Coq proofs do not have a clean separation between their externally-visible interface and the internal implementation, this is the only safe way of ensuring that changes to a fundamental proof are appropriately reflected throughout a project. Coqoon offers two different mitigations to make this more palatable for large developments: the builder can be configured to recompile projects only when the user explicitly requests it, and it can also use the Coq 8.5 quick compilation chain, speeding up compilation drastically by postponing the evaluation of proofs.

Neatness and namespaces

Coq developments do not typically have a clean separation between source and output folders. In a simpler setting, the resulting clutter is merely annoying; in an IDE, however, compiled libraries and other derived files are normally entirely hidden from the user, which is a much harder task when these files are not systematically separated from source code.

The Coqoon builder emulates the behaviour of the Java builder to provide this separation: the Coq source file `src/SoftwareFoundations/Basics.v`, for example, is compiled into the library `bin/SoftwareFoundations/Basics.vo`, with the fully-qualified name `SoftwareFoundations.Basics`. Source folders can also be given a prefix to be added to the fully-qualified names of the files they produce.

Although there are as yet no conventions for managing the Coq library namespace, this approach is flexible enough to support any feasible convention that might be chosen in a future version of Coq.

3 PIDE: Coqoon's interaction with Coq

PIDE is a middleware layer originally developed by Wenzel [24] to bridge the gap between the Isabelle system, implemented in PolyML, and its user interface, written in Java.

For both historical and technical reasons, many proof assistants are written in a programming language that is a descendant of ML, a language conceived with that particular application in mind. IDEs, on the other hand, are more usually built atop platforms like Java or .NET; a layer like PIDE is thus necessary to bridge the gap between these environments.

PIDE also provides an asynchronous protocol to exchange data between the prover and the user interface. In contrast to the protocols developed with the waterfall model in mind, PIDE decouples execution and feedback: instead of sending commands one-by-one and waiting after each command for its results, it sends the document as a whole and expects the output of the prover to arrive asynchronously whenever a part of the document has been processed.

3.1 PIDE in a nutshell

PIDE consists of a relatively prover-agnostic frontend library, implemented in Scala, and a prover-specific backend in the prover's own implementation language (i.e., PolyML for Isabelle, or OCaml for Coq). This architecture is depicted in Figure 2.

PIDE's frontend and backend share a distributed data structure: the *PIDE document*. The document's content is represented as a list of *commands* for the proof assistant: in the case of Coq, these are full sentences terminated by a full stop. Apart from the raw source text of the command, the document also stores the output of Coq as it interprets each command. The frontend reacts to user edits to the text by updating the content of the proof document. Conversely, the backend triggers computation by the proof assistant, gathers the results in the proof document and reports them to the frontend.

In this model, the backend has a complete view of the document, and is free to evaluate its commands in any order it sees fit; the backend then relays the resulting—potentially out of order—status and feedback messages back to the frontend (and ultimately to the user). The frontend can also interrupt the backend with an update to the document, or direct the backend to focus its attention on a different region.

To bring PIDE support to Coq, Tankink wrote an OCaml implementation of the PIDE backend for use with Coq 8.5, also making some minor changes to the Scala library in the process [4]. Although Coqoon has benefited greatly from this work, it was not carried out with Coqoon in mind—it was originally intended for use with jEdit, a more limited text editor used as the main interface for Isabelle.

```
define_command(
  "Theorem a : True.",1)
define_command("Proof.",2)
define_command("trivial.",3)
define_command("Qed.",4)
update([1;2;3;4])
```

```
Theorem a : True. (1)
Proof. (2)
  trivial. (3)
Qed. (4)
```

(a) PIDE messages defining version 1 of the document.

```
define_command("auto.",5)
update([1;2;5;4])
```

```
Theorem a : True. (1)
Proof. (2)
  auto. (5)
Qed. (4)
```

(b) PIDE messages defining version 2 of the document.

Figure 3: PIDE messages and resulting document.

Even though jEdit and Eclipse are two very different environments, adding PIDE support to Coqoon has required only minor changes to PIDE, which shows that the library is not tied to one particular style of frontend.

3.2 The Coqoon side of PIDE: the frontend

The main function exposed to a client in the Scala library of PIDE is `update`. This function takes text insertion and deletion operations, and uses them to create a new version of the document that represents the text after applying this operation.

To reduce overhead, the Scala library transforms raw textual updates into *command updates*, which describe insertions and deletions of entire spans, rather than changes on the character level. When the user changes the document, PIDE first assigns identifiers to any new spans and then describes these changes in terms of deleting and inserting identifiers from the document. PIDE notifies the prover of these changes using two kinds of protocol messages. First, it sends a number of `define_command` messages that contain the identifier and the text of a command. Second, it sends a single `update` message, which contains the deletion and insertion operations. The Scala library also batches together text edits that take place in a very short span of time.

For example, Figure 3a shows a trivial proof, along with PIDE's view of it: a proof document consisting of four commands, each labelled with its unique identifier. Figure 3b shows the same proof after the user has made a minor change: from PIDE's perspective, command (3) has been deleted from the document, and command (5) has replaced it.

This approach requires the frontend to be able to quickly split the text into commands, i.e. to be, at least to some extent, aware of the syntax of the interactive prover.

To support Coq in PIDE, the code in the frontend that recognises commands was generalised, making it use an abstract parser instead of requiring the syntax of Isabelle. A parser for Coq commands was then developed.

Commands in Coq are always terminated by a single period (`.`), followed by whitespace or the end of the file. Recently Coq also introduced a class of "focus" commands, which are certain characters occurring on their own at the beginning of a line; these are also supported. Finally, the parser needs to recognize a number of cases where a phrase is not a command, for example when it occurs inside a comment or a string. The whole parser is implemented in less than 120 lines of Scala code, using Scala's parser combinators and a few simple regular expressions.

It is worth mentioning that, as of today, Coq does not provide a facility to quickly parse a document and split into commands. As a consequence, each user interface for Coq implements its own recognizer of commands.

3.3 The Coq side of PIDE: the backend

The Coq part of PIDE is PIDEtop — a plugin for Coq that communicates using PIDE's protocol. It responds to the two kinds of messages coming from the Scala library by updating its internal representation of the shared document: it stores the defined commands in a lookup table, and it keeps a list of identifiers to represent the document structure in response to an update.

After the document has been updated, PIDEtop schedules, then starts, the interpretation of any changed commands using the State Transaction Machine library [4] (STM). During scheduling, STM classifies the commands, and builds a structured representation of the proof document. This structure allows STM to make more efficient use of available resources. The best example of this is parallelization: most proofs in Coq can be calculated independent of the correctness of other proofs, which means that interpretation and computation of proof state can be carried out in parallel, using multiple processor cores. This allows for significant increases in speed compared to the waterfall model; in that model, Coq would be made aware of the document one command at a time, and would have to produce output immediately, leaving the prover no choice as to when to process each command.

When Coq is processing the document, it will asynchronously report information to PIDEtop, accompanied by the identifier for the command the information pertains to. The information is not restricted to the goal state, but includes other information that can be inferred by the proof assistant, such as abstract syntax trees of the commands, or the types of subexpressions.

PIDEtop reports the feedback it obtains as XML messages using the protocol's `print` message. This feedback is stored in the Scala library's representation of the document, where the client can access it and use it to highlight parts of the document or to show status information for each command. Historically the output of interactive provers, for example the goal state, is plain text. (Indeed, PIDE has a dedicated feedback message

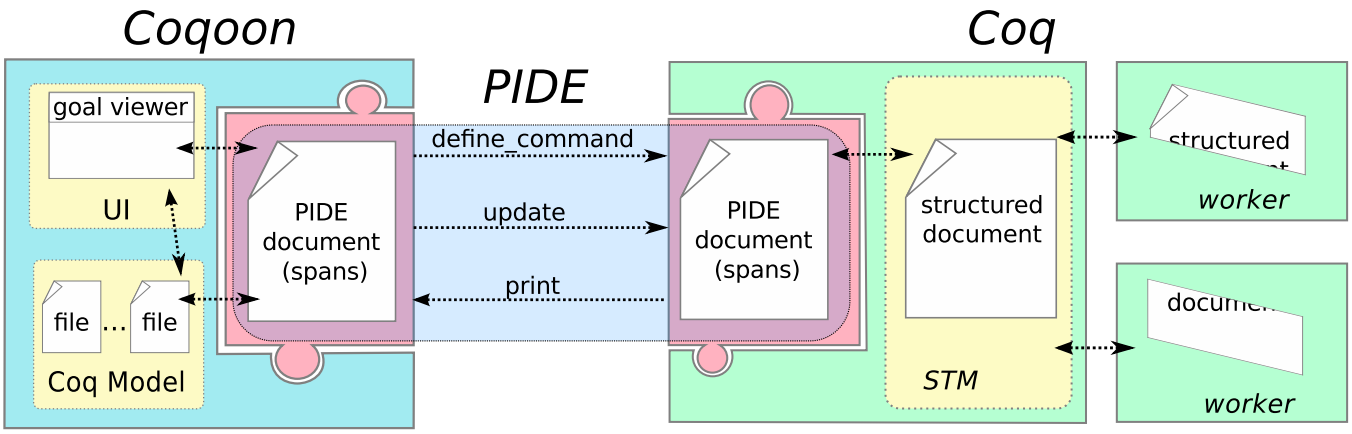

Figure 2: An overview of the architecture of Coqoon, PIDE and Coq.

type for pre-printed goals of this form.) Nevertheless, a frontend capable of presenting goals in a more sophisticated way can take advantage of the structured XML data that feedback messages carry. For example, Coqoon renders each goal in a separate tab, and shows the (potentially large) goal context in a widget with dedicated scroll bars. A data structure in which goals are distinct and their context is represented as a list is already part of Coq; exposing that structure to a PIDE client is a simple matter of encoding it in XML.

3.4 The PIDE document and Coqoon's Coq model

The Coq model aggregates data coming from both PIDE and the Coq builder. Both sources provide error messages and error locations. PIDE may provide errors contained in the text buffer the user is editing, while the builder may provide errors contained in other files that are part of the project. Storing all of this data in the model makes it easy to report errors in a uniform way.

Some kinds of data generated by Coq, like the goal state, are not yet stored in the model, but are instead retrieved from the PIDE document on demand. For example, when the text cursor is placed on a command, the goal viewer fetches the goal state to be displayed from the active editor's PIDE document.

4 Package management with OPAM

Coqoon is distributed in a normal Eclipse package repository, but for it to work properly, Coq and the PIDE backend have to be installed too. Bundling these tools in Java plugins is not practical, so they cannot be included in the repository; users of older versions of Coqoon had to set up Coq by hand.

Recent versions of Coqoon improve this situation by including a plug-in that provides support for OPAM, a young but promising package manager for software written in OCaml, like Coq.

4.1 What OPAM is for

The OCaml programming language specifies a portable byte code representation, but unlike the Java one, it has no efficient virtual machine for executing it. Hence OCaml software is compiled to native code. This native code is not portable, and – unlike a C object or a Java class file – its interface is not specified or guaranteed. As a consequence, only object files compiled with the same version of the OCaml compiler can be linked together.

This peculiar aspect of OCaml makes building software like Coq and its extensions like the PIDE backend quite a challenge, since all the pieces of the puzzle need to be compiled consistently.

Updating OCaml software can be equally cumbersome. Linking in OCaml is static, and the compiler performs a cross-module inlining optimisation. This means that, when an OCaml library is updated, all software that was linked with it needs to be recompiled. For the Coq user, this means that, in order to install a bugfix for Coq, they must also rebuild all Coq extensions, including the PIDE backend.

The OCaml community has recently started to use the OPAM package manager to distribute software. This package manager understands the peculiarities of OCaml: it installs all software in *roots* (directories) that are tied to a specific version of the OCaml compiler. Software is installed by compiling the sources in a particular root, and software packages come with enough metadata to (re)trigger their (re)compilation in the appropriate order.

Although OPAM itself is a command line utility, Coqoon's OPAM plug-in provides a user interface for managing OPAM roots.

4.2 A programmatic interface to OPAM

Coqoon's OPAM plug-in provides a Scala interface to OPAM's functionality: each root is represented by an

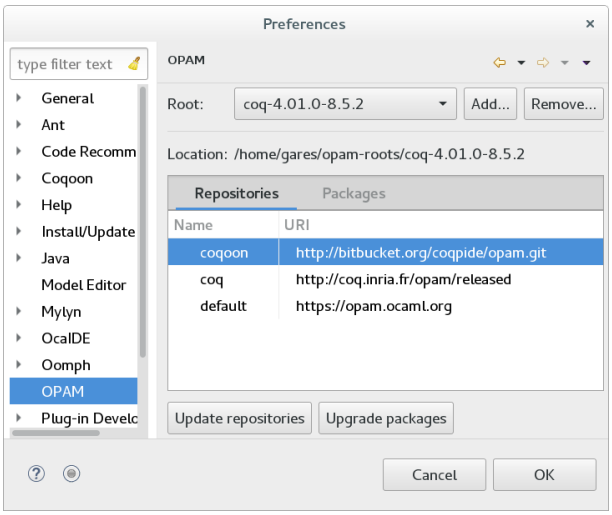

Figure 4: Package repositories of the OPAM root.

object containing a number of packages, each of which in turn contains a number of versioned packages that can be installed into their parent root. The root itself is a directory on the file system, placed outside the Eclipse workspace.

All of these objects wrap calls to the command line utility, managing its execution and parsing its output. It also provides a logging facility, to let users follow the execution of a long running task, and implements a cache to minimize the number of external process invocations.

Although this interface was developed for Coqoon, it does not depend on it, and could be used by other Eclipse plug-ins that require OPAM support.

4.3 Working with OPAM roots

The OPAM plug-in extends the Eclipse preference window with a panel for managing OPAM roots. Each root fetches software from a particular set of repositories; Fig. 4 highlights one that contains Coq-related software.

The "Add..." button lets the user register an already existing OPAM root or create a new one. In this last case, the plug-in provides a dedicated window, shown in Fig. 5, to select certain parameters for the new root: its location on the filesystem and the desired versions of Coq and OCaml.

4.4 OPAM for Coq

Coq inherits the peculiarities of OCaml: all Coq extensions have to be compiled with the same OCaml version, and all Coq theories have to be checked with the same version of Coq.

To make this process less cumbersome, the Coq community has recently started to distribute Coq extensions and Coq theories with OPAM. The OPAM integration

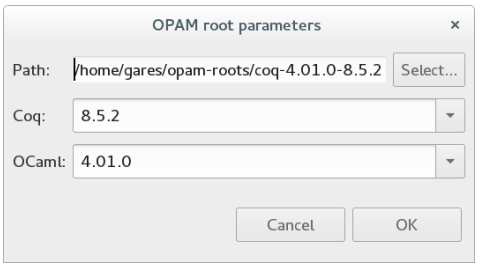

Figure 5: Parameters for the OPAM root to be created.

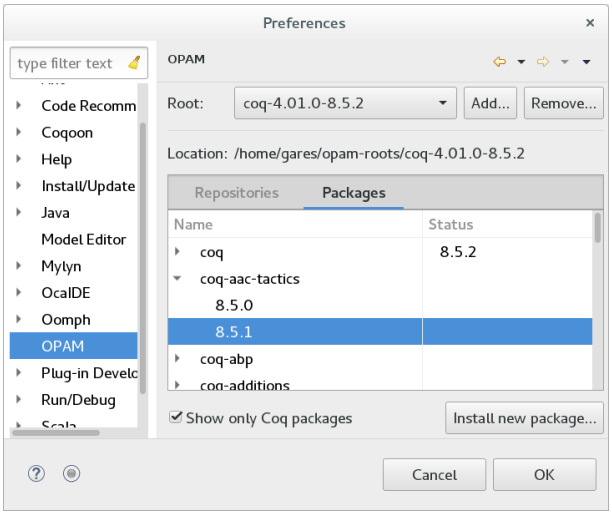

Figure 6: Packages status / installation of new packages.

plug-in covers this aspect too: the packages tab, shown in Fig. 6, lets the user browse the package list and install, remove or update each package.

These Coq packages are installed in the selected OPAM root and constitute what we call the *Coq system installation*. These packages are also contributed to Coqoon's load path resolution process: an installed OPAM package can satisfy an abstract dependency of the same name. The relationship between these packages and the Eclipse workspace is explained in the next section.

The user can set up many OPAM roots, possibly based on different Coq versions and containing different sets of Coq packages. Switching between them requires a single click; the OPAM integration plug-in automatically reconfigures Coqoon to use the version of Coq provided by the active OPAM root.

5 External collaboration

Coq has a large and vibrant community and it is important that Coqoon can be introduced into existing developers' workflows without disrupting their productivity. To this end, we have ensured that it is possible for users of both Proof General and CoqIDE to not only import

but to also co-develop Coqoon projects. Coqoon also has some limited support for working on developments that were not initially designed with Coqoon in mind without changing the internal structure of these projects.

Coq itself provides some support for building basic projects. The `coq_makefile` utility generates a standard Makefile out of a configuration file, typically called `_CoqProject`. This file contains a list of source files and a mapping from source directories to abstract Coq paths. Java projects do not typically require this mapping to be made explicit, as the directory hierarchy specifies the fully-qualified name of a Java class. Coqoon encourages Coq projects to respect the Java convention, but, as it allows projects to map source folders to abstract Coq paths, it interoperates well with projects that do not follow this convention.

5.1 External development of Coqoon projects

Coqoon projects separate their source files from their binary compiled files. This is standard practice for most software development IDEs but atypical behavior for Coq theory developers.

To make this separation work outside of Eclipse, Coqoon includes a Python script that mimics the behavior of its internal build system. When first invoked, this script will try to find all the dependencies of the project, prompting the user for more information if necessary. Once all the dependencies are satisfied, the script generates a `Makefile` and a `_CoqProject` file, allowing the user to build the project and to load it in any development environment supporting the `_CoqProject` format (which both Proof General and CoqIDE do). The dependency information that the user provides is cached; subsequent calls to the script will only scan the source folders for new, deleted, or moved files or folders and update the two output files accordingly. The user will only be asked to re-supply the path to a dependency if the script, for some reason, cannot find that dependency at its old location.

The variant of the Coqoon build system implemented in the Python build script allows Coqoon and non-Coqoon users to work on a project using much the same workflow. In particular, dependencies are automatically recalculated, and the portability of the load path is preserved. Keeping the same workflow also makes it much less likely that a change made outside of Coqoon will be incompatible with Coqoon's way of managing projects.

5.2 Coqoon projects and abstract dependencies

Large Coq developments are increasingly common, and these are often built on top of existing projects. Tools like Coqoon are expected to make this kind of dependency management easy.

Ideally, a new project would only depend on off-the-shelf components, i.e. whatever is provided by the Coq system installation (with OPAM packages). But many Coq projects are moving targets, and users often need to modify or improve them in order to be able to use them effectively.

To make this easier, Coqoon projects can also declare that they provide a named abstract dependency. These declarations take priority over all others: for example, if a project provided the dependency `coq-sample-lib`, which was also available in OPAM, then the project's implementation would be preferred.

This makes it easy to temporarily fork part of the Coq system installation without having to reconfigure all the projects that depend on it. Once the user's changes have been incorporated into the OPAM package, they can revert to using it by simply deleting their Coqoon project.

5.3 Importing projects into Coqoon

There are two ways to import a project into Coqoon. The first is to create an empty Coq project, and then to copy the files from the external project into it. Although this approach works, it has a downside: as the files are simply copied into the Eclipse workspace, any existing connections to version control systems like Git are lost.

Alternately, the user can add Coqoon project metadata to the external project; Eclipse will then be able to import this project and to treat it as if it were a normal Coqoon project. By using this approach, the files can remain outside the Eclipse workspace, allowing them to remain connected to an external version control system. This also allows Coqoon's build system to coexist with the project's native build system; as Coqoon's output files are placed in different locations, the two are unlikely to interfere. This approach is the best way to keep a project compatible with both Coqoon and with other tools. (Coqoon has no special support for this process yet, although we intend to add a special import wizard that takes care of adding metadata to existing projects.)

6 Test cases

To assess the maturity of the tool we apply it to two Coq developments: the *Odd Order Theorem*, a large formalization that comprises both Coq theories and a Coq extension, and the widely used teaching course in *Software Foundations* by Pierce.

6.1 The Odd Order Theorem and the Math.Comp. library

The Odd Order Theorem by Feit and Thompson is a masterpiece of modern mathematics for which its last author received the Fields medal in 1970 and the Abel prize in 2008. This result was not only famous because of

its profound influence on the last fifty years of research in group theory, but also for its length, weighing in at more than two hundred and fifty pages. Indeed, its length and intricacy caused many to raise concerns about the correctness of its entire argument.

In 2012 a team of fifteen people, led by Gonthier [11], completed a formal verification of the proof, and of the mathematical theories it builds upon, using the Coq system. The project took six years to complete (including three years of work on the part of this paper's third author). The resulting body of formalized mathematics is divided into two main parts: the so called Mathematical Components library (Math.Comp. for short), that covers many general purpose mathematical theories (group theory, linear algebra, character theory ...) and the main proof which builds upon them.

The entire development sums up to 125 Coq modules for a total of 161,000 lines of code: 93 modules and approximately 121,000 lines for the Math.Comp. library, and 32 modules and 40,000 lines for the main proof. All Coq modules are written in a custom language, called SSReflect, that is provided by a plugin for Coq. The plugin, itself a 7,500-line OCaml program, is also part of the Math.Comp. library. The entire source code amounts to 7.4 megabytes.

This code base constitutes one of the largest developments for Coq, and pushes the system close to its limits; as a consequence, building it and working on it has never been a pleasant experience for the user. The dependency graph of its components, for example, is too large to be printed in this paper,[1] and building the entire project takes around two hours. This time is how long one needs to wait in order to build on top of the Math.Comp. library, or browse it comfortably, or simply to be able to go back to work after having made a minor change to one of the core modules. Despite that, other formalization projects have started depending on (parts of) the Math.Comp. library, inheriting along with it the complexity and time consumption of its build process. In particular, building the SSReflect plugin by hand has always been a source of trouble for its users, and the long time required to build the entire library eventually pushed the authors of the library to provide reduced versions of it for those users who did not need all of its power.

Importing this gargantuan project into Coqoon revealed a few deficiencies in our implementation. Coqoon's dependency resolver, for example, was overwhelmed by the size of the dependency graph, in some cases taking more than ten seconds to work out a file's dependencies. Luckily, this was easily remedied by the addition of a simple cache.

To spare the user from a prolonged compilation process, support for the quick compilation [4] chain, a new feature provided by Coq 8.5, was also added to Coqoon. This process separates Coq compilation into two phases: the first is very quick, checking only definitions and statements, while the second, slower, phase completes the compilation by checking the proofs. As the first phase produces intermediate files that can be used in place of traditional Coq libraries, it only takes around seven minutes of computation on an ordinary laptop computer before the entire set of 125 modules is usable.

As the user does not need to wait for the second phase in order to work with the development, it can typically be run as a background task. Unlike the first phase, it can take great advantage of parallel hardware, because each proof can be checked independently of the others: on a computer with a dozen cores, the proofs for the whole development can be completed in as few as 15 minutes.

In addition to that, we added support to OCaml modules to the Coq builder. Combined with the OCaml builder provided by OcaIDE, this has made it possible to build both the SSReflect plugin and the Coq modules that depend on it in a single integrated build process.

Finally, the PIDE backend for Coq was made more responsive and robust when dealing with long modules. Most of the files in the Mathematical Components library are more than a thousand lines of code in length, and some are more than four thousand lines. For comparison, in the CompCert compiler,[6] another Coq flagship project, composed of 5.2 megabytes of sources, more than 30% of the modules are longer than a thousand lines.

As a result of these changes, we believe Coqoon represents the best platform for working on such large developments. In particular, at the time of writing, no other IDE for Coq can handle projects that contain both OCaml and Coq code, and Coqoon is the only one to incorporate the quick compilation chain as an integrated part of an automatic build system.

6.2 Software Foundations

This is a relatively small Coq development that complements the Software Foundations book by Pierce et al. It is a widely adopted course that touches on topics like logic, functional programming, interactive theorem provers, and techniques for software verification. Universities in the United States, Japan and Europe use it in their curricula.

Coqoon has been used at the IT University of Copenhagen in conjunction with the Software Foundations teaching material for three years. We have found that the use of a more familiar development environment makes Coq much more user-friendly for students, showing that building on top of an IDE brings advantages for beginners and experts alike.

[1] The interested reader can browse it online: http://math-comp.github.io/math-comp/htmldoc/libgraph.html

7 Related work

Over the last thirty years, there have been multiple attempts to make the interaction with proof assistants easier. Initially, all interaction was through a Read-Eval-Print loop (REPL), a command-line interface that interprets each command typed by the user and prints out the resulting goal state (or an error) before requesting new commands. Some proof assistants, such as HOL [12] and HOL Light [14], still use this as their primary mode of interaction.

7.1 Waterfall interaction

Proof General [1], based on Emacs, was the one of the first interfaces that offered more than just a REPL, and is the only one of the early interfaces that endures until today, going so far as to define the *de facto* standard method of interaction with Coq: the waterfall model. Although this still required the user to direct proof processing manually, it was nevertheless a significant improvement over the bare REPL.

The Proof General model of interaction has been duplicated by several other Coq tools, including CoqIDE, which is a GTK+-based interface bundled with Coq [22], and three Eclipse plugins. The first was created by Aspinall as an attempt to port Proof General itself to Eclipse [2]; the other by Charles and Kiniry who, as part of the Moebius project, built the plugin ProverEditor for Coq in Eclipse [8]; the third, Kopitiam [19], by Mehnert, is Coqoon's immediate predecessor.

CoqIDE is a custom cross-platform text editor. It does not add any truly new interaction features, beyond some Coq-specific code templates and the ability to invoke `make` and the Coq verifier from the interface. While it allows the user to have multiple buffers open, there is no relation between the contents of the buffers. The version of CoqIDE shipped with Coq 8.5 was improved by Tassi to support processing the waterfall in parallel. However, the fundamental interaction with Coq will not change: the user still needs to manually direct Coq to process parts of the active document.

The Proof General plugin for Eclipse was only available for Isabelle, and has not been under active development since 2010, based on its Eclipse update site[2]. It offered interaction based on the waterfall model, and a high-level overview of individual proofs, but did not provide any support for structured projects.

Conversely, the ProverEditor plugin for Eclipse was only available for Coq. Its project support consisted of automatic Makefile generation and support for invoking `make`—unlike Coqoon, it did not integrate into Eclipse's build system. ProverEditor was discontinued in 2009, when the last update to their GitHub page was made.

[2] http://proofgeneral.inf.ed.ac.uk/eclipse/products/

Kopitiam targeted the 8.3 series of Coq, which had no structured way of sending and receiving messages: Coq 8.4 introduced an XML-based protocol for executing commands, and Coq 8.5 allows developers to add support for entirely new protocols such as PIDE, but Coq 8.3 only supported interaction through the standard Coq REPL. This was extremely brittle, and required constant polling to read responses from Coq. Kopitiam had no support for Coq projects.

Kopitiam offered one unconventional extension to the waterfall model: it allowed Coq proofs to be interleaved with Java source code. Using aspect-oriented programming to hook into the internals of the Java editor, it added Coq-like controls to step through decorated Java programs: stepping over a Java command would cause it to be 'executed' in an environment based on a separation logic framework built by Bengtson *et al.* [5]. As the waterfall does not map cleanly onto any Java concepts, this was fragile and difficult to use, but it was an interesting extension—and one which we intend to reintroduce in the future using PIDE.

7.2 PIDE and asynchronous editors

With the introduction of PIDE, Wenzel ushered in the third generation of proof assistant interaction: instead of requiring the user to micromanage the system's execution, it allows asynchronous interfaces, such as Coqoon. The flagship application of the PIDE approach for Isabelle is Isabelle/jEdit [25], which is now the standard frontend to the Isabelle system.

Because PIDE and Isabelle/jEdit have been developed in tandem, the editor makes full use of the features we have described in Section 3: the editor allows asynchronous interaction with Isabelle, and marks up the proof document using information obtained during interpretation. Isabelle/jEdit has been partially adapted to support Coq by Tankink [4]—the resulting combination being called Coq/jEdit—but this adaptation does not have the full power of an IDE.

jEdit is an extensible text editor, not an IDE, and the way it was extended by Wenzel in order to support entire developments is Isabelle specific. Following the original design of provers of the HOL family, the Isabelle system does not provide a notion of separate compilation: files are just loaded by a single prover instance one after the other, with an option of using concurrent threads to speed up the process. The PIDE protocol is even able to multiplex multiple text buffers to the same prover instance, and expects the prover to sort that out.
The way Coq works is closer to how traditional programming languages work. The Coq compiler can deal with one file at a time, and unrelated files can be processed by different instances of the compiler, possibly in parallel. As a result Coq/jEdit can only work with a single file and relies on the user to provide their own build system

for larger projects. Coqoon is able to take care of the entire build process of large developments, even when they include custom Coq plugins, as described in section 2.4.

Another limitation of Coq/jEdit is that, while Isabelle/jEdit maintains a model of proof documents using PIDE, Coq/jEdit does not. The design of Isabelle's language makes it much easier to integrate that model with jEdit's syntax-and-text oriented views. Isabelle's proof language, Isar, is a two-tiered language, that consists of an outer syntax that gives structure to proof documents, the Isar language proper, and numerous inner syntaxes used for specification and proof methods, the most notable being the Higher Order Logic; HOL. The transition between these languages is syntactically indicated using quotation marks. The outer syntax has a simple structure and can easily be parsed by the Scala library of PIDE. The inner syntaxes are more versatile, and the parsing and processing is handled by the Isabelle side of PIDE. It is this outer syntax that is exposed to jEdit. In Coq's language, there is no syntactical separation between the different languages used, making it difficult to implement a Scala-side parser that exposes the structure. (Coqoon's model takes some steps in this direction, but it is necessarily full of special cases and heuristics.) This lack of a Scala-implemented parser for Coq means that jEdit plugins that rely on such a parser do not work.

Finally, because jEdit is not under active development, its plugins have also grown stale, not being updated to new models and tools. For Isabelle/jEdit, Wenzel already had to change the core of jEdit to allow the PIDE plugin to paint text when semantic information comes in. This means that Isabelle/jEdit is a small fork of jEdit itself, and that it requires its users to install the entire client, instead of just a plugin. Coqoon works on standard Eclipse distributions.

A second client in the PIDE ecosystem is Isabelle/Eclipse [23]. The development of this Eclipse plugin is on hiatus at the moment, but the version that is available, emulates the Isabelle/jEdit interaction model in Eclipse: it does not provide any 'Eclipse-specific' features like project management or compilation of single files. In its current state, it behaves much like Isabelle/jEdit, but using Eclipse to provide the visual elements for the interface.

Clide [21] is another system that builds upon the PIDE architecture for Isabelle. It is a web interface that is mainly aimed at real time collaboration on proof documents. In a similar fashion to Google Docs, several users can work on the same document, seeing each other's modifications and the responses from Isabelle. It supports projects, but only as a way of grouping together collaborations with others; as such, files in a project are not verified when one file changes, and errors in a proof document are not shown until it is opened.

7.3 Another approach: the ALF tradition

The ALF proof assistant [18] and its modern-day descendants–chief amongst them Agda [20]—take a rather different approach to the construction of proofs. Whereas Coq proofs consist of a sequence of invocations of tactics, each of which manipulates Coq's internal representation of a proof term, a proof in the ALF tradition consists simply of the finished proof term: the indirect manipulations performed by tactics in the Coq world are replaced by direct modifications of potentially incomplete terms in source files. Compared to ALF-style proofs, Coq proofs are thus somewhat akin to an edit script: they enumerate the steps taken by the user to arrive at a complete proof term, which are analogous to the steps that the user would perform directly in ALF.

Using this approach in practice requires a more intelligent interface than a simple text editor. Agda proofs, for example, are typically written using an advanced Emacs mode equipped with the ability to rewrite regions of the document according to the transformations supported by the prover. However, this mode shares many of the other drawbacks of tools built on extensible editors: in particular, it has no support for project management.

8 Conclusion

This paper presents Coqoon, an IDE for the interactive proof assistant Coq in Eclipse. Coqoon moves away from traditional synchronous proof development and towards an asynchronous model that allows any part of a proof document to be modified and rechecked without having to retract unrelated proofs. It also supports Coq projects that are fully integrated into the Eclipse build system: files can be added, deleted, and moved at will, and Coqoon will track these changes and rebuild affected files whenever necessary. Coqoon can also make use of the large number of plugins already available for Eclipse, such as the OCaml plugin OcaIDE, turning Coqoon into a complete development environment for even the most complex Coq projects, or the version control plugin EGit.

Coqoon also brings support for Coq projects to other Eclipse projects and plugins, paving the way for complete IDEs for software verification where programs and proofs of their correctness can be maintained within the same project—or even in the same file.

Together, these features represent a significant advance: a truly integrated and comprehensive proof assistant IDE, bringing to the world of proof assistants a workflow that software developers have enjoyed for decades.

8.1 Future work: embedding Coq

Our work with OcaIDE shows that Coqoon can already interoperate with other development environments built on the Eclipse platform. The next step in our work is to provide even tighter integration between the Coq and Java development environments.

Java projects can already be turned into "hybrid" projects containing both Java programs and Coq proofs about those programs, but this is only a start. There are already several tools that embed assertions and proofs directly into the source code that they describe, like Dafny [17], Spec# [3], and VeriFast [15]. The IDE seems like an obvious home for this functionality: that is, it should be possible to extend the Java editor already present in Eclipse with Coqoon-powered Coq proofs.

Coqoon's predecessor, Kopitiam, provided just such an environment; however, this environment was built using cruder integration techniques and predated the introduction of PIDE. A prototype inspired by Kopitiam, but built using Coqoon, PIDE, and a custom text editor more aware of the interleaving between Coq and Java code, is under development at the IT University.

References

1. David Aspinall. Proof General: A Generic Tool for Proof Development. In *TACAS*, volume 1785 of *LNCS*, pages 38–42. Springer, 2000.
2. David Aspinall, Christoph Lüth, and Daniel Winterstein. A Framework for Interactive Proof. In *Calculemus/MKM*, pages 161–175, 2007.
3. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec$^\sharp$ Programming System: An Overview. In *CASSIS*, pages 49–69, 2005.
4. Bruno Barras, Carst Tankink, and Enrico Tassi. Asynchronous processing of Coq documents: from the kernel up to the user interface. In *Proceedings of ITP*, Nanjing, China, August 2015.
5. Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal. Verifying object-oriented programs with higher-order separation logic in Coq. *Lecture Notes in Computer Science*, 6898:22–38, 2011.
6. Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In *ARITH*, pages 107–115. IEEE Computer Society, 2013.
7. Nicolas Bros and Rafael Cerioli. OcaIDE. Software, available on http://www.algo-prog.info/ocaide/.
8. Julien Charles and Joseph R. Kiniry. A Lightweight Theorem Prover Interface for Eclipse. In *UITP Workshop proceedings*, 2008.
9. Eclipse Foundation. EGit. Software, available on http://www.eclipse.org/egit/.
10. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1994. First edition, 20th printing.
11. Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In *ITP*, pages 163–179. Springer, 2013.
12. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, New York, NY, USA, 1993.
13. Thomas C. Hales. *Dense Sphere Packings - a blueprint for formal proofs*. Cambridge University Press, Sep 2012.
14. John Harrison. HOL Light: An Overview. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 60–66, 2009.
15. Bart Jacobs and Frank Piessens. The VeriFast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven, August 2008.
16. Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2, 2014.
17. K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR-16*, pages 348–370, 2010.

18. Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for proofs and programs*, pages 213–237. Springer, 1994.
19. Hannes Mehnert. Kopitiam: Modular Incremental Interactive Full Functional Static Verification of Java Code. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 518–524, 2011.
20. Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
21. Martin Ring and Christoph Lüth. Collaborative Interactive Theorem Proving with Clide. In *ITP*, pages 467–482. Springer, 2014.
22. The Coq Development Team. The Coq Reference Manual. Available electronically, at http://coq.inria.fr/doc.
23. Andrius Velykis. Isabelle/Eclipse. Software, available on http://andriusvelykis.github.io/isabelle-eclipse.
24. Makarius Wenzel. Asynchronous User Interaction and Tool Integration in Isabelle/PIDE. In *ITP*, volume 8558 of *LNCS*, pages 515–530. Springer, 2014.
25. Makarius Wenzel. System description: Isabelle/jEdit in 2014. In *UITP*, 2014.