Collaborative Interactive Theorem Proving with Clide

Martin Ring[1] and Christoph Lüth[1,2] ⋆

[1] Deutsches Forschungszentrum für Künstliche Intelligenz, Bremen, Germany
[2] Universität Bremen, FB 3 — Mathematics and Computer Science, Germany

Abstract. This paper introduces Clide, a collaborative web interface for the Isabelle theorem prover. The interface allows a document-oriented interaction very much like Isabelle's desktop interface. Moreover, it allows users to jointly edit Isabelle proof scripts over the web; editing operations are synchronised in real-time to all users.
The paper describes motivation, user experience, implementation and system architecture of Clide. The implementation is based on the theory of operational transformations; its key concepts have been formalised in Isabelle, its correctness proven and critical parts of the implementation on the server are generated from the formalisation, thus increasing confidence in the system.

1 Introduction

Just like mathematics, interactive theorem proving is at its heart a social activity. Mathematical proof is rarely a solitary activity, it is most often done in collaboration with others. It is thus unfortunate that present theorem prover interfaces have very much been single-user; a *real-time collaborative* user interface, where many users can jointly edit the same proof in the vein of the late Google docs[3] should add much to the user experience, enhance productivity and enable new patterns of interaction between theorem provers and humans. Until now, there have hardly been real-time collaborative user interface for theorem provers, so this hypothesis had to remain untested. This paper presents a first prototype of a real-time collaborative, web-based user interface for a state-of-the-art interactive theorem prover, Isabelle, allowing us to experiment with the collaborative user experience.

As the experience with Google docs shows, collaborative user interfaces thrive when they are available on the web. The web has collaboration built-in, with many users connecting to a single server, and web interfaces offer *eo ipso* a lot of advantages: they are inherently cross-platform, portable and mobile, they require little installation effort (a recent web browser is enough), and only need few resources on the user side. Recent advantages in web technology (collectively and somewhat inaccurately known as 'HTML5') allow the development of

⋆ Research supported by BMBF grants 01IW10002 (SHIP), 01IW13001 (SPECifIC).
[3] Now available as Google Drive.

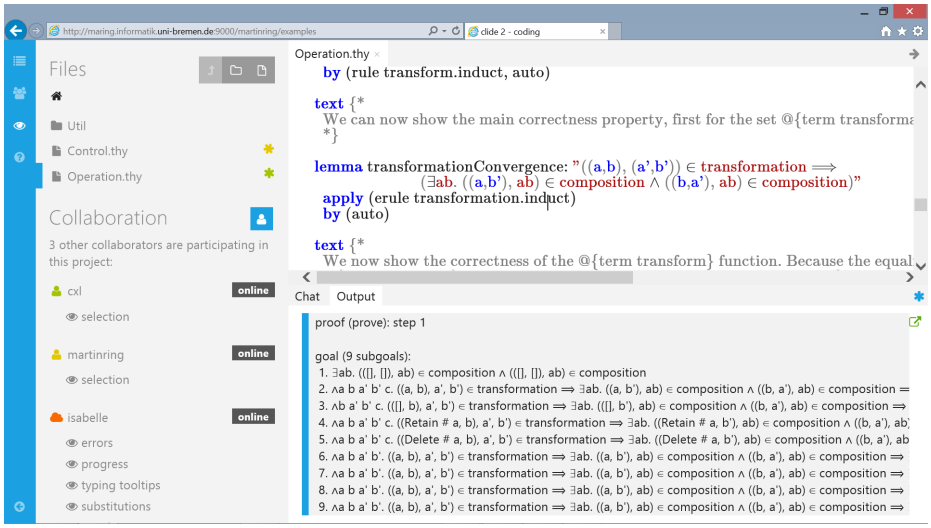


Fig. 1. The Clide user interface: On the left there is a toolbar, on the right a theory has been opened (above). The Isabelle output, here the current proofstate, is shown in a separate part of the window; it can also be inlined.

web interfaces of near-desktop quality. The first version of the Clide system [1] demonstrated a web-based interface for Isabelle; the present work extends this to a truly collaborative setting. This is not a completely trivial exercise; the basic problem is keeping the documents *synchronised* across the different clients (user interfaces), the server and the theorem prover. Fortunately, well explored solutions for this problem exist which we could draw on when implementing our system, namely the theory of operational transformation. We have formalised the basic algorithms of this theory in Isabelle, and generate parts of our implementation from this formalisation.

This paper is structured as follows: we first introduce Clide from the users' perspective, then give the theory and pragmatics of the implementation. We explain the system architecture and the underlying design decisions, and finish with conclusions, where we review related and future work.

2 The User Experience

The interface was designed with an emphasis on typography over superfluous graphics, with a clear arrangement reducing it to the basics such that it does not distract from the main center of attention, the proof script. Pervasive use of HTML5 and JavaScript make the interface very responsive; because the entire user interface is implemented as a single-page application which dynamically changes views through JavaScript the interaction more resembles a desktop application than a web interface of old.

Clide organises the user's work in *projects*, which are collections of files and folders. Projects are the basic unit of granularity for sharing. The Clide user interface has two basic views: the *backstage view* is the starting point, where users can review their projects, create new ones, and select a current one, and the *project view*, where users can create, edit and delete files and folders in the current project.

Fig. 1 shows a screenshot of the project view. On the left, there is a (hideable) tool sidebar, where users can select files, invite collaborators, and access common editing operations such as cut, copy & paste. The tool sidebar further shows the other collaborators and their details. In the center of the view, there is the main file editor, where files are opened in tabs. The editor is based on the CodeMirror editor, and offers a seamless editing of mathematical text in a web-browser, with features such as integration of mathematical symbols, Greek letters, and other Unicode symbols, flexible-width font, super/subscripting and tooltips for text spans. The interaction with Isabelle is very similar to the Isabelle/jEdit interface [2]: users edit the theory while Isabelle processes it asynchronously on the server, sending back the results as they become available. These results can be the prover's state, error or warning messages after executing this particular prover command. The prover messages can be displayed inlined or in a separate window, which is useful for larger proof states. In addition to the messages, the inner syntax of the theory as well as special symbol substitutions are annotated and type information for hovered terms is provided. All these annotations can be deactivated individually if desired. There is also a chat window which allows short text messages to be sent to collaborators. The collaboration is unintrusive, and should be familiar to users of Google docs: each user can see the cursor of other users and their editing operations, taking effect immediately without blocking. Users also see the selection area of other users, which is useful for communication purposes ("Where here is the error?"), and to warn other users that this area of the file is about to be deleted; this feature can be deactivated if it gets too intrusive.

A public evaluation version of the system is online at http://clide.informatik.uni-bremen.de/. The evaluation version features public projects, which are open to all users of the system (normally not a desirable state of affairs), which is great to get quickly up and collaborating.

2.1 Use Cases

Collaboration should not be end unto itself. We envisage at least the following use cases for collaborative theorem proving:

- *Scientific collaboration:* two (or more) users are working jointly on a proof, all contributing actively and staying in close contact; collaboration ensures that all participants know the proof, and can continue working on it. In a normal situation, collaborators would be sitting around the same machine; with a collaborative interface this situation can be extended to collaborating across countries and continents (timezone issues notwithstanding).

- *Proof review:* one user is going through the proof, explicating it to others who do not contribute actively, but try to understand what is being formalised. This situation is useful in the classroom, both for lecturers to explain a proof (while the students can interactively explore it), but possibly also for teachers to see how students progress and be able to assist them if needed.
- *Machine-assisted collaboration:* here, other collaborators are software processes. A simple example of this is Clide used as a single-user web-interface: the user still collaborates with Isabelle.

We would be surprised to see massive open online collaboration, where thousands of people work on one single theory. In this situation, an underlying version management and revision control system is needed; the ProofPeer project recently started in Edinburgh is an interesting step in this direction [3].

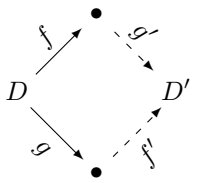
3 Implementation

Research in the area of *computer-supported cooperative work* (CSCW) goes back to the early eighties of the last century. One of the more challenging concerns has always been real-time activity awareness and coordination [4]. A collaborative system with these properties requires a mechanism to synchronise the distributed document states as quickly as possible across all users without loss of information or diverging documents. This is far from trivial because communication always involves delays (ranging from usual network delay to temporary failure), and thus edits will occur concurrently.

If we only consider the insertion of content the problem has a reasonably easy solution by introducing a partial ordering of concurrently inserted document positions (*e.g.* via vector clocks). Problems come with concurrent insertions and deletions especially if we do not only want state consistency but also basic *intention preservation* [5], which is essential for a usable system. The most popular approach to this problem has been *operational transformation* (OT) [6].

3.1 The Basics of Operational Transformation

The problem of synchronisation is that we may have situations where two operations f and g are applied concurrently to the same document D, and we need to complete the resulting span again with operations f', g' into a common document D' as in (1). Writing $applyOp\ f\ D$ for the application of an operation f



(1)

to a document D, the completion of (1) is written as

$$\forall D.\ applyOp\ g'\ (applyOp\ f\ D) = applyOp\ f'\ (applyOp\ g\ D). \tag{2}$$

The basic idea behind OT is to solely consider the operations, and not the documents, and to restrict ourselves to a tractable set of basic operations. Hence, operations are sequences of basic actions, where an actions is: advance one character; insert one character; or delete one character. To apply an operation, we traverse the document and operation simultaneously, and apply the basic actions. Note that this application is partial; we can only apply an operation to a document of the appropriate length. We can then transform these operations against each other, written as $\mathit{transform}\ f\ g = \langle f', g' \rangle$, to obtain the completion (1); applying first f, then g' should be the same as first applying g, then f'.

In order to state (2) point-free, *i.e.* without referring to a document D, we need to define the composition $\circ$ of two operations. Note that this is not simply the concatenation of the two sequences of actions; rather, we merge two sequences into one new sequence which combines the effects of the two operations. We can then drop the document D from the correctness property (2) and state:

$$\mathit{transform}\ f\ g = \langle f', g' \rangle \Longrightarrow g' \circ f = f' \circ g \tag{3}$$

The correctness of the composition operation $\circ$ is stated as

$$\mathit{applyOp}\ (g \circ f)\ D = \mathit{applyOp}\ g\ (\mathit{applyOp}\ f\ D) \tag{4}$$

and together these easily imply (2).

3.2 Formalisation in Isabelle/HOL

We introduce the formalisation of the theory of operational transformation on which our implementation is based. For reasons of space, we do not show the full formalisation; we give enough details to show the actual algorithms, but we elide most lemmas and all Isabelle proofs (most of which are very short anyway).[4]

We start with the basic concepts. For documents, we keep the actual character set as a type parameter, actions are as mentioned above, and operations are then lists of actions:

```
type_synonym 'char document = 'char list
datatype 'char action = Retain | Insert 'char | Delete
type_synonym 'char operation = 'char action list
```

We can now recursively define the application function:

```
fun applyOp :: 'char operation ⇒ 'char document ⇒ 'char document option
where
  applyOp [] []                    = Some []
| applyOp (Retain# as) (b# bs) = Option.map (λds. b# ds) (applyOp as bs)
| applyOp (Insert c# as) bs      = Option.map (λds. c# ds) (applyOp as bs)
| applyOp (Delete# as) (_# bs)  = applyOp as bs
```

[4] The full theory can be found at http://www.informatik.uni-bremen.de/~cxl/papers/itp2014-appendix.pdf for reference.

```
| applyOp _ _                    = None
```

However, reasoning about this function directly is not straightforward because of its partiality: *applyOp f d* is only defined for a document d of a certain *inputLength*, given by the number of *Retain* and *Delete* actions in that operation. A straightforward induction on the definition of *applyOp* would leave us with an induction assumption where it is not immediate that *applyOp* is applicable to its arguments. In order to get around this difficulty, we define the graph of the function as an inductive set:

```
inductive_set application :: (('char operation × 'char document) × 'char document)
set where
  empty[intro!]: (([],[]),[]) ∈ application
| retain[intro!]: ((a,d),d') ∈ application ⟹ ((Retain#a,c#d),c#d') ∈ application
| delete[intro!]: ((a,d),d') ∈ application ⟹ ((Delete#a,c#d),d')    ∈ application
| insert[intro!]: ((a,d),d') ∈ application ⟹ (((Insert c)#a,d),c#d') ∈ application
```

We can show that *application* is exactly the graph of *applyOp*:

```
lemma applyOpSet: ((a,d),d') ∈ application ⟷ applyOp a d = Some d'
```

The composition of two operations traverses through the two operations and combines the actions pointwise. In the following definition, the second argument of the composition is executed after the first one (the other way around as $\circ$); so *e.g.* a delete action first executed is always kept, because nothing can undo a delete, and an insert action executed second is kept for the same reason. An insert action followed by a retain is just that insert, and insert followed by delete cancel each other out:

```
fun compose :: 'char operation ⇒ 'char operation ⇒ 'char operation option
where
  compose [] []                          = Some []
| compose (Delete# as) bs                = Option.map addDeleteOp (compose as bs)
| compose as (Insert c# bs)              =
        Option.map (Cons (Insert c)) (compose as bs)
| compose (Retain# as) (Retain# bs) = Option.map (Cons Retain) (compose as bs)
| compose (Retain# as) (Delete# bs)  = Option.map addDeleteOp (compose as bs)
| compose (Insert c# as) (Retain# bs) =
        Option.map (Cons (Insert c)) (compose as bs)
| compose (Insert _# as) (Delete# bs) = compose as bs
| compose _ _                          = None
```

The above function uses *addDeleteOp* to insert a delete action. This is an optimisation, where we permute deletes over inserts as much as possible, so we get contiguous sequences of delete and insert actions, which we can later compress for transmission. *addDeleteOp* is defined as follows:

```
fun addDeleteOp :: 'char operation ⇒ 'char operation
where
```

```
  addDeleteOp (Insert c#next) = Insert c# addDeleteOp next
| addDeleteOp as = Delete#as
```

The effect of *addDeleteOp* is to remove the first element of a document:

lemma *addDeleteOpValid*: *applyOp* (*addDeleteOp a*) (*c#d*) = *applyOp a d*

Again, in order to be able to show anything about *compose* we explicitly define its graph as an inductive set.

inductive_set *composition* :: ((*'char operation* × *'char operation*) × *'char operation*) *set* **where** ...

We leave out the lengthy definition; it follows the recursive definition of *compose* just as *application* follows the definition of *applyOp*. However, we show that *composition* is the graph of the *compose* operation:

lemma *composeSet*: ((*a*,*b*),*ab*) ∈ *composition* ⟷ *compose a b* = *Some ab*

The first proper result is the correctness of composition (4). It is first shown for the relation *composition* (omitted), and then for the function *compose*:

theorem *composeCorrect*:
⟦ *compose a b* = *Some ab*; *applyOp a d* = *Some d'*; *applyOp b d'* = *Some d''* ⟧
⟹ *applyOp ab d* = *Some d''*

Finally, we define the *transform* function, the core algorithm of operational transformation. Recall that the transformation of a and b are two operations a', b' such that a composed with b' is the same as b composed with a'. The transformation is defined recursively, with a lengthy case distinction on the first action of each: e.g., insert actions remain, but cause a retain action to appear in the transformed operation (in order to keep the inserted character); transforming two retain actions results in two retain actions; or a retain and a delete transform to a delete and nothing (reflecting the fact that we either first keep an element, then delete it, or delete it first, without need for a subsequent action):

fun *transform* :: *'char operation* ⇒ *'char operation* ⇒ (*'char operation* × *'char operation*) *option*
where

```
  transform [] []                 = Some ([], [])
| transform (Insert c#as) bs      =
        Option.map (λ(at, bt). (Insert c# at, Retain# bt)) (transform as bs)
| transform as (Insert c# bs)     =
        Option.map (λ(at, bt). (Retain# at, Insert c# bt)) (transform as bs)
| transform (Retain# as) (Retain# bs) =
        Option.map (λ(at, bt). (Retain# at, Retain# bt)) (transform as bs)
| transform (Delete# as) (Delete# bs) = transform as bs
| transform (Retain# as) (Delete# bs) =
        Option.map (λ(at, bt). (at, Delete# bt)) (transform as bs)
| transform (Delete# as) (Retain# bs) =
```

```
                Option.map (λ(at, bt). (Delete# at, bt)) (transform as bs)
| transform _ _                                          = None
```

To our minds, this definition is intricate enough to warrant a formal treatment; at least, we feel more confident about its correctness having done so. We can show that the domain of this function is pairs of operations a and b which have the same input length. To show the main correctness property (3), we define the graph of *transform* as an inductive set (we leave out the lengthy definition):

inductive_set *transformation* :: (('*c operation* × '*c operation*) × ('*c operation* × '*c operation*)) *set* **where** ...

and show that this is a superset of function graph. With this, we can show the second main result, the correctness of transformation. This is even slightly stronger as (3) as it also states that the composition of a and b' (and implicitly b and a') is defined:

theorem *transformCorrect*: *transform a b* = *Some* (*a'*,*b'*)
⟹ *compose a b'* ≠ *None* ∧ *compose a b'* = *compose b a'*

Further, we define identity operations *ident*, which consist only of retain actions, and show that they are the left and right unit to the composition operator. Unfortunately because of the optimisation underlying the *addDeleteOp* function, these properties only hold up to normalisation, *i.e.* sorting operations such that a delete action is never followed immediately by an insert action. Moreover, transformation against an identity does not change an operation:

lemma *transformIdL*:
transform (*ident* (*inputLength b*)) *b* = *Some* (*ident* (*outputLength b*), *b*)

3.3 Implementing Operational Transformation

The previous section showed the formalisation of the core algorithms of operational transformation. The implementation uses these algorithms to synchronise document states on one server and many clients. Our implementation follows the approach by Google [7] which is a simplification of the original algorithms.

The server keeps a single history $h = \langle a_1, a_2, \ldots, a_n \rangle$ which is a sequence of operations a_i. A revision r_i refers to the state after operation a_i (starting with initial revision r_0). Clients report operations together with a revision number, $\langle b, i \rangle$. On receiving $\langle b, i \rangle$, the operation b is transformed with respect to all operations a_j for $i < j \leq n$, resulting in an operation b', which is appended to the history, and distributed to all other clients as a remote edit. Additionally, the client which sent the operation receives an acknowledgment (see Fig. 2). The correctness property (2) means that all squares in Fig. 2 commute, so the server only needs to append the transformed operation to its history. The server does not need to keep track of the actual document states, it is enough to keep track of the operations.

On the client side, clients need to cater for both local edits (affected by the user) and remote edits (sent from the server). To this end, the client keeps track

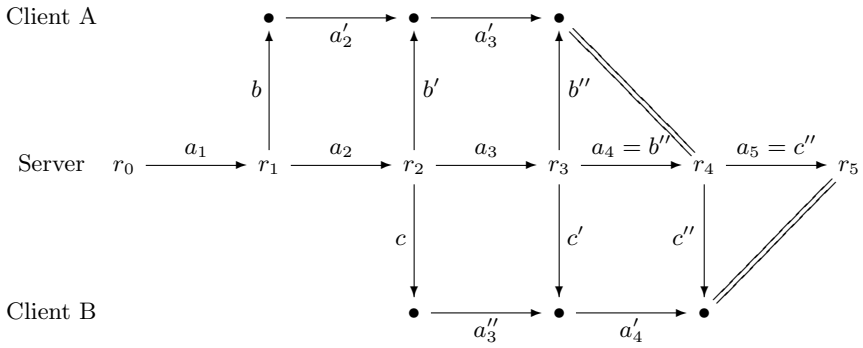


Fig. 2. History management on the server. Here, the current revision is r_3 when client A sends in operation $\langle b, r_1 \rangle$ and client B sends in $\langle c, r_2 \rangle$. The operation b is transformed with respect to $a_3 \circ a_2$ to b'', and appended as a_4 to the history; similarly, the operation c is transformed to c'' with respect to $a_3 \circ a_4$. Client A is sent $a'_3 \circ a'_2$ as the first remote edit, an acknowledgment, and a_5 as a second remote edit, while Client B is sent one remote edit $a'_4 \circ a''_3$, and an acknowledgment. In the end, r_5 becomes the new current revision on the server and both clients A, B. Note that we can resolve the convergence the other way, with first resolving $\langle c, r_2 \rangle$ with respect to a_3 (then a_4 would be c'), and then $\langle b, r_1 \rangle$ with respect to $c' \circ a_3 \circ a_2$ (the result of which would become a_5).

of which operation $\langle a, r \rangle$ has last been passed on to the server, and waits for an acknowledgment of this operation (we say the operation is *pending*). If further local edits occur while waiting, these are *buffered*. (Note that we only need to buffer one operation, as we can compose multiple edits.) Once the operation $\langle a, r \rangle$ has been acknowledged, the revision is increased, and the buffered operation (if there is any) is passed on and becomes pending. If a remote edit is received before the operation $\langle a, r \rangle$ has been acknowledged, we know it refers to revision r, so the operation is transformed with respect to the pending and buffered operations, applied to the local document, and the revision is increased. In turn, the remote edit operation transforms the pending and buffered operations (see Fig. 3). (Note that the client does not receive its own local edits back as remote edits from the server.)

A huge advantage of our Isabelle formalisation is the ability to generate Scala code [8] for the composition and transformation of operations which we can use in the server application. However, we need an implementation of the same algorithms on the client. Unfortunately, due to the restrictions of the web environment there is no practical alternative to JavaScript as a programming language. This leads to potentially divergent implementations. Experiments showed that in principle we can even go one step further and generate the JavaScript code from Scala with the new *Scala.js* compiler [9], but until this tool moves out of the alpha stage we have to rely on a manual port.

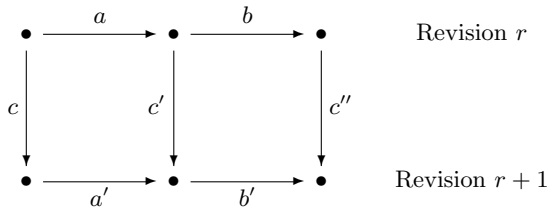


Fig. 3. History management on the client. Operation a is pending, operation b is buffered. If a remote edit c is received, it is transformed with respect to a and b to c'', and applied. a' and b' become the new pending and buffered operations. The revision number r is increased every time a remote edit or acknowledgment is received.

3.4 Further Extensions

In addition to plain text operations we need a mechanism to annotate the document state to integrate information like remote cursors and selections, syntax highlighting, or prover output. The original Google approach was somewhat cumbersome to allow for rich text editing; here, because we do not require annotations to be editable we chose a simpler approach by keeping annotations separate from the operations. In our implementation, annotations consist of two types of actions: $\mathit{plain}(n)$ is equivalent to n retain actions, and $\mathit{annotate}(n, a)$ is equivalent to n retain actions and additionally annotates the document with the annotation a starting at the current position, and ranging n characters.

This means annotations are operations consisting only of retain actions, with the side effect of augmenting the document with additional information. Recall from above that such operations are identities, hence annotations cannot interfere with other operations (see Lemma *transformIdL* in Sect. 3.2). However, other operations have an effect on annotations, so we have to define how delete and insert actions interact with $\mathit{annotate}(a, n)$. We chose to simply extend or shrink the annotation accordingly; this feels like natural behaviour that you would expect if you type inside an annotation of a collaborator.

This leads to a very straightforward integration of annotations. On the client side, annotations are transformed with respect to pending and buffered operations, and only sent on once the pending and buffered operations have been sent on. On the server side, an incoming annotation $\langle a, i \rangle$ is transformed and sent on to the other clients just as with editing operations, except that the sending client receives no acknowledgment and the revision number is not increased. Annotations are identified by their origin and an origin-unique name, and remain until overridden by a subsequent annotation of the same origin and name.

4 System Architecture

Building a modern web application has become an increasingly complex task. The demands placed on such an application have grown rapidly over the last

couple of years. To offer an adequate experience to users, the interface must always stay responsive. To be able to handle fluctuating numbers of visitors, the system needs to be scalable. In addition to these universal demands, our application involves very expensive computations on the theorem proving side on one hand, and a highly distributed state due to the real time collaboration between users on the other. The first version of Clide [1] was scalable, responsive and resilient, but to properly integrate collaboration, it soon became obvious that most of the architecture had to be carefully rethought.

For the new iteration of Clide, we chose the *Typesafe Reactive Platform* [10] as a basis because it is event-driven and resilient by design, leading to a responsive and scalable application. The platform includes the *Akka* actor library for concurrency control [11], the functional relational mapper *Slick* as an efficient database integration, and the *Play!* web framework. The uses of *Slick* and *Play!* are obvious, but *Akka* became in fact the most important component for Clide, as it is very well suited for a collaborative architecture.

4.1 Universal Collaboration

To reflect the collaborative nature of the application, instead of offering a specialised API for plug-ins, we utilise the same API for human and non-human users. We call this approach *universal collaboration*. The unification has several advantages: On one hand it simplifies the core system itself, on the other hand it also makes it easier to write plug-ins that involve heavy computations (and thus delays). All the management of distributed asynchronous document states is achieved in the operational transformation framework. This way plug-ins can focus on the important aspects — annotating or otherwise contributing content to documents — in a simple, synchronous manner. Moreover, it is easy for plug-ins to work together without knowing anything about their respective implementations. It is not even required for plug-ins to run on the same machine as the server. Neither is it necessary for the server to know anything about the plug-ins a priori; the plug-ins can actively register with the server via a TCP connection. Users can choose to invite a plug-in into a project just like they would with human collaborators.

4.2 The Clide Infrastructure

The Clide infrastructure consists of modules which are loosely coupled, standalone applications whose actor systems communicate with each other via Akka remoting. The modules themselves can easily be further divided and distributed across different machines which leads to great scalability. Fig. 4 shows an example setup with several modules connected to the `clide-core` module. The modules can be configured to connect to a specific address and are implemented in a way that they retry until they are connected to an instance of `clide-core` and reconnect on network failure or in case the peer is restarted. This way it is

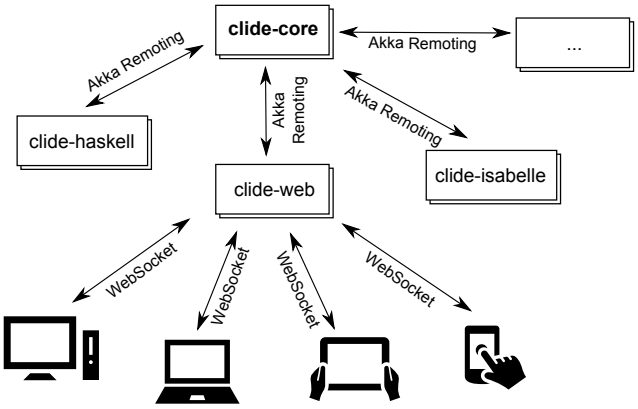


Fig. 4. Overview of the system architecture.

very easy to configure a Clide infrastructure because only a couple of configuration files have to be adjusted. The modules do not have to be started in any particular order and individual failures do not propagate.

The purpose of the `clide-web` module is somewhat special in that it mainly serves as a translator between JSON messages transmitted via WebSockets and the internal message representation. *WebSockets* are a very good fit for the communication between web clients and `clide-web` because their properties are similar to those of message channels connecting actors. WebSockets are full-duplex and thus allow us to send messages from the server to the client without any overhead. `clide-web` directly connects the client with an actor in `clide-core`; that way web clients have the same access to all levels of the API as any other client. The second task of `clide-web` is to provide the resources of the user interface (HTML, CSS and JavaScript files) to the clients. For the user interface we utilised the *angular.js* library which allows for declarative data bindings defined in HTML code. The client side logic and thus also the client side implementation of the operational transformation framework are implemented in CoffeeScript, a language that compiles to JavaScript but compensates for many deficiencies of that language. Because we only used technology from the HTML5 standard, it is possible to use the web interface on any modern, HTML5 compliant web client, i.e. not only on classic computers but also tablets, and given an adapted user interface even smartphones, or television sets.

4.3 The `clide-core` API

Fig. 5 shows a simplified view of the internal actor system with instances of all available types of actors as well as the ownership hierarchy and message flow indicated. The starting point from the outside world is the `UserServer` in the *Global API* which authenticates users, and acts as a message router. It is also possible to sign up a new user here.

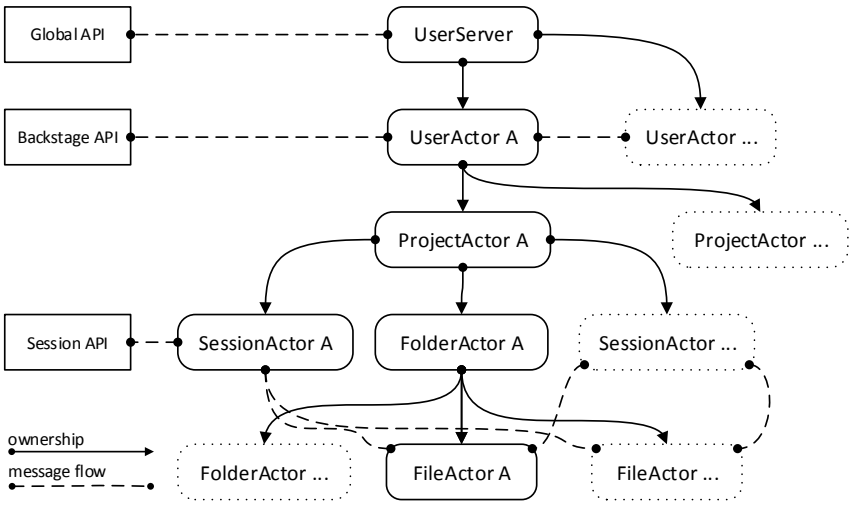


Fig. 5. Actor infrastructure in `clide-core`.

Each `UserActor` is responsible for one authenticated user via the *Backstage API*. It allows to create and manage projects and access rights. Peers also get informed about new projects and invitations.

Each project is coordinated by a `ProjectActor`. It accepts requests from a `UserActor` to access a project (after checking the rights), and creates a `SessionActor` for each user accessing the project. The `SessionActor` provides the *Session API* which is the core API in the sense that it provides the editing and annotation operations for one client. The `ProjectActor` coordinates the operations, distributing them to all clients as described in Sect. 3.3. Access to files and folders is given by other dedicated child actors of the `ProjectActor`. Individual `FileActors` manage a single file each and are responsible for the server side operational transformation framework.

4.4 Assistants and Integration with the PIDE Framework

One intriguing aspect of Clide is that collaborators need not be human. We call such non-human collaborators *assistants*, and provide a simplified interface to implement them. It allows easy access to state changes, edit operations and annotations. An assistant only needs to reference the state on which its actions are based, the underlying framework takes care of everything else. This way assistants do not drastically differ from usual plug-ins for IDEs.

There are two ways of building an assistant. The first is to directly use Akka remoting to communicate with the `UserServer` via the messages available in the `clide.actors` package. However, this requires detailed knowledge of the

messaging protocol which might still be subject to future modifications and in addition can only be comfortably used from Scala. For this reason there is also a simpler way: Implementing the `AssistantBehavior` interface is all it takes to build an Assistant the easy way. The interface contains a number of abstract methods which will be called as events occur. The implemented interface can then be registered with the generic `AssistantServer` class which will take care of everything else. All method calls within the interface are synchronized. If a calculation takes a long time, subsequent modifications to the document as well as annotation activities (such as cursor movements) will be combined so that the assistant does not lag behind if many changes occur in a short period of time. Communicating back is done via the `AssistantControls` interface which is supplied to the behavior. A short introduction on implementing assistants as well as an API reference can also be found on the GitHub project page.

With this simplified framework, the integration of the PIDE framework [2] (and hence Isabelle) is very straightforward, and a lot of code from Isabelle/jEdit could be reused in the process of implementing the `clide-isabelle` module. When the PIDE framework reports that new information about one of the viewed theories is available, we translate the prover state, type information, inner syntax as well as output messages into Clide annotations and report them back to the assistant framework.

As a case study, we have also implemented an assistant to handle Haskell files (try inviting Haskell as a collaborator on the web site). It calls the Haskell compiler for each source file, parse its error and warning messages and passes them back as annotations. In about two hundred lines of code, we implemented some usable Haskell assistance.

The integration of native processes into a publicly available web interface bear a serious security risk which must be considered when implementing an assistant for Clide. For this reason we start Isabelle in safe mode, disabling all ML integration because that would grant easy access to the server file system.

5 Conclusions

This paper has introduced Clide, a real-time collaborative web interface for the Isabelle theorem prover. It combines modern web technologies with the Isabelle PIDE back-end to offer a user interface which in terms of responsiveness and display of mathematical notation and symbols equals conventional desktop interfaces for Isabelle, such as Isabelle/jEdit or ProofGeneral. The single-user experience resembles Isabelle/jEdit, with the user editing documents which are processed asynchronously on the server in the background, and results appearing as they become available. However, Clide is easier to set up, is mobile, and most of all offers real-time multi-user collaboration. It implements the theory of operational transformations; we have formalised the key concepts in Isabelle, proved its correctness, and derive critical parts of the implementations on the server side from this theory, thus increasing confidence in the system.

5.1 Related Work

There is a lot of related work in the community of computer-supported cooperative work (CSCW) [4,12], including of course the theory of operational transformations [13], but to our knowledge, this is the first fully operational collaborative interface for an interactive theorem prover.

Other web interfaces for theorem provers include Proof Web [14], which is based on older web technologies and hence not as interactive as Clide. There are also a number of mathematical wikis (*e.g.* [15,16,17]) based on interactive provers such as Mizar or Coq, but they resolve collaboration in the typical wiki-style, namely by versioning the edited files or pages; users do not edit the same page at the same time, but instead create different versions.

5.2 Future Work

There are two ways in which this work can be extended. Firstly, there is nothing specific about Isabelle in our framework, except for the fact that the PIDE framework with its Scala API provides a good foundation of our work: it really improves interaction if the theorem prover is multi-threaded and asynchronous, and it helps if everything runs on the same platform. But the system architecture is generic, and could be used to implement a collaborative IDE as well, because all this needs in the first instance is to replace Isabelle with a compiler which analyses the code. There is already a simple assistant for Haskell files; the same approach could be used to integrate batch-based ITPs like HOL4 or HOL-light. Another possible extension would be a 'ProofGeneral module', a generic implementation of script management.

The second main avenue to pursue would be to add more, and richer, assistants. Systems for Proof General such as ML4PG [18], which uses machine learning techniques to help the user find similar proofs, could be adapted to our system, but one could also envisage for example tutoring systems, where the collaborating machine analyses the users errors, and offers helpful advice when certain erroneous patterns occur, or assisted document authoring [19], where for example the collaborating machine suggests an induction scheme based on the current proof state. Apart from actively contributing assistants, there is also a surprising benefit of passively listening collaborators: Clide can be used as a simple API for external applications which can delegate proofs to humans and wait until the theorem prover agrees and then return. The possibilities are endless, and we would like think the generic architecture of our system makes it easy to explore these exciting new avenues of research.

References

1. Lüth, C., Ring, M.: A web interface for Isabelle: The next generation. In Carette, J., ed.: Conferences on Intelligent Computer Mathematics CICM 2013. LNAI 7961 Springer (2013) 326– 329

2. Wenzel, M.: Isabelle/jEdit - a prover IDE within the PIDE framework. In Jeuring, J., *et al*, eds.: Intelligent Computer Mathematics — 11th International Conference MKM 2012. LNCS 7362, Springer (2012) 468–471
3. ProofPeer web page. http://www.proofpeer.net (Accessed: 29.01.2014).
4. Dourish, P., Bellotti, V.: Awareness and coordination in shared workspaces. In: Proceedings of the 1992 ACM Conference on Computer-supported Cooperative Work. CSCW '92, ACM (1992) 107–114
5. Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D.: Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. ACM Trans. Comput.-Hum. Interact. **5**(1) (March 1998) 63–108
6. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. SIGMOD Rec. **18**(2) (June 1989) 399–407
7. Wang, D., Mah, A., Lassen, S.: Google Wave operational transformation. http://www.waveprotocol.org/whitepapers/operational-transform/operational-transform.html (Accessed: 30.01.2014).
8. Haftmann, F., Nipkow, T.: A code generator framework for Isabelle/HOL. In: Theorem Proving in Higher Order Logics (TPHOLs 2007), Emerging Trends Proceedings, Dept. of Computer Science, University of Kaiserslautern (2007) 128– 143
9. Doeraene, S.: Scala.js website. http://www.scala-js.org (Accessed: 29.01.2014).
10. Typesafe Inc.: Typesafe reactive platform overview. http://typesafe.com/platform (Accessed: 29.01.2014).
11. Wyatt, D.: Akka Concurrency. Artima Press (2013)
12. Greenberg, S., Marwood, D.: Real time groupware as a distributed system: Concurrency control and its effect on the interface. In: Proc. 1994 ACM Conference on Computer Supported Cooperative Work. CSCW '94, ACM (1994) 207–217
13. Sun, C., Ellis, C.: Operational transformation in real-time group editors: Issues, algorithms, and achievements. In: Proc. 1998 ACM Conference on Computer Supported Cooperative Work. CSCW '98, ACM (1998) 59–68
14. Kaliszyk, C.: Web interfaces for proof assistants. In Autexier, S., Benzmüller, C., eds.: Proc. of the Workshop on User Interfaces for Theorem Provers (UITP'06). Number 174[2] in ENTCS (2007) 49– 61
15. Urban, J., Alama, J., Rudnicki, P., Geuvers, H.: A wiki for Mizar: Motivation, considerations, and initial prototype. In Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P., eds.: Intelligent Computer Mathematics. LNCS 6167 Springer (January 2010) 455–469
16. Alama, J., Brink, K., Mamane, L., Urban, J.: Large formal wikis: Issues and solutions. In Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F., eds.: Intelligent Computer Mathematics. LNCS 6824, Springer (January 2011) 133–148
17. Tankink, C.: Proof in context — web editing with rich, modeless contextual feedback. In Kaliszyk, C., Lüth, C., eds.: Proc. 10th International Workshop On User Interfaces for Theorem Provers (UITP'12). EPTCS **118** (2013) 42– 56
18. Komendantskaya, E., Heras, J., Grov, G.: Machine learning in Proof General: Interfacing interfaces. In Kaliszyk, C., Lüth, C., eds.: Proceedings 10th International Workshop On User Interfaces for Theorem Provers, UITP 2012. EPTCS **118** (2013) 15–41
19. Aspinall, D., Lüth, C., Wolff, B.: Assisted proof document authoring. In Kohlhase, M., ed.: Mathematical Knowledge Management MKM 2005. LNAI 3863, Springer (2006) 65– 80