Supporting Proof in a Reactive Development Environment *

Farhad Mehta
ETH Zurich, Department of Computer Science
Claussiusstr. 49, 8092 Zurich, Switzerland
fmehta@inf.ethz.ch

Abstract

Reactive integrated development environments for software engineering have lead to an increase in productivity and quality of programs produced. They have done so by replacing the traditional sequential compile, test, debug development cycle with a more integrated and reactive development environment where these tools are run automatically in the background, giving the engineer instant feedback on his most recent change.

The RODIN platform provides a similar reactive development environment for formal modeling and proof. Using this reactive approach places new challenges on the proof tool used. Since proof obligations are in a constant state of change, proofs in the system must be represented and managed to be resilient to these changes. This paper presents a solution to this problem that represents proof attempts in a way that makes them resilient to change and amenable to reuse.

1 Introduction

One of the major issues of software engineering is the inevitability of change. Change can occur as the result of a change in requirements, the need for new functionality, the discovery of bugs, etc. But these are not the only sources of change. Smaller incremental changes are unavoidable during the development phase of large, real world systems. The construction of such systems is an incremental activity since, in practice, getting things right the first time around is not possible [4]. Modern development environments, such as the Eclipse IDE for Java [9], support incremental construction by managing change and giving the engineer quick feedback on the consequences of the latest modification of a program.

Formal model development is also an incremental activity[16]. One of the major criticisms of using formal methods in practice is the lack of adequate tool support, especially for proof. The RODIN platform [14] addresses this by providing a similar reactive development environment for the *correct construction* of complex systems using refinement and proof. It is based on the Event-B [1] formal method. The development process consists of *modeling* the desired system and *proving* proof obligations[1] arising from it.

In the *correct by construction* setting advocated by the Event-B method, the proving process is seen as not merely justifying that certain properties of the system hold, but also as a *modeling aid* to help steer the course of the modeling process. In contrast to *post-construction verification*, the proving process assists modeling, in the same way testing and debugging assists programing.

But the way the proving process is supported by formal development tools used today does not take advantage of this interaction between modeling and proving. The main reason for this is that modeling and proving are treated as isolated activities, making it hard for the user to move freely between them in order to use proving insights to aid modeling.

1.1 Reactive Formal Development

From the experience of how a previous generation of development tools [7, 3] were used, the RODIN platform proposes a reactive development environment, similar to a modern IDE, where the user working on a model is constantly notified on the status of his proofs. To achieve this, the tools present in the RODIN platform are run in a reactive manner. This means that when a model is modified, it is automatically:

- Checked for syntax and type errors,
- its proof obligations are generated, and

*This research was carried out as part of the EU research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems) http://rodin.cs.ncl.ac.uk.

[1] A proof obligation is a logical statement derived from a model that must be proved in order to guarantee that some property of the model holds.

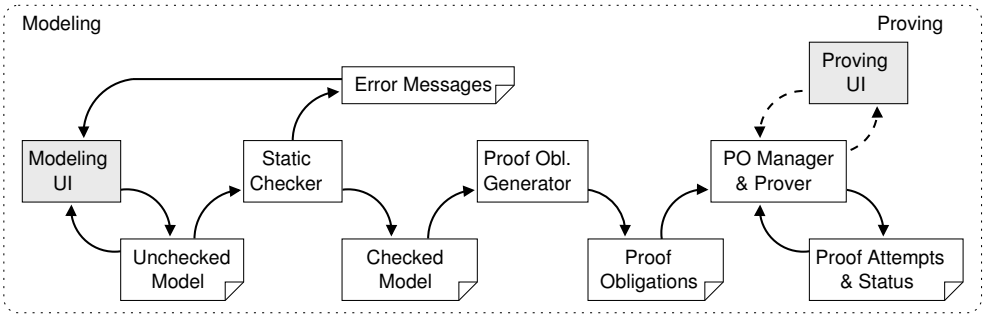


Figure 1. The RODIN Tool Chain

- the status of its proofs are updated, possibly by calling automated provers, or reusing old proof attempts.

The RODIN tool chain is illustrated in figure 1. The user gets immediate notification on how his last modeling change affects his proofs. Inspecting failed proofs often gives the user a clue on how to further modify his model. The user may then use this feedback to make further modifications to his model, or decide to work interactively on a proof that the prover was unable to discharge automatically. A detailed description of the RODIN tools and user interface, and further benefits for using the reactive approach for formal model development can be found elsewhere [2].

Supporting such a reactive development environment poses new challenges for the tools involved, particularly the prover.

1.2 A Reactive Prover - Challenges

In a reactive development environment, a change in a model is immediately reflected as changes in proof obligations. The *main challenge* for a proof tool in such an environment is to make the best use of previous proof attempts in the midst of changing proof obligations. Reusing previous proof attempts is important since considerable computational and manual effort is needed to produce a proof.

The main difference between current approaches to proof reuse and what is required for proof reuse in the reactive setting is the frequency with which changes occur, and the efficiency with which these changes need to be processed in the reactive setting. Current approaches to represent and reuse proof attempts are not well suited for the frequency with which some common types of change occur in the reactive setting.

The *main contribution* of this paper is a new way of representing proof attempts (proof skeletons) that can be reused incrementally, but where explicit reuse may even be avoided for some (recoverable) changes where proof skeletons are guaranteed to be resilient.

Structure of the paper In the next section (§2), we present some background information on what a proof is (§2.1), and how it is constructed in our system (§2.2). The paper is then divided into two parts. The first part (§3) is about proof obligation changes. We describe the types of changes that proof obligations go through (§3.1), how we want to react to them (§3.2), and derive requirements on how proofs attempts should be recorded and managed (§3.3). The second part (§4) is the main contribution of the paper. It discusses how these requirements are met. We start by evaluating three standard approaches used to represent proof attempts (§4.1, §4.2, §4.3) and show why they do not fulfil our requirements. We then propose a mixed approach (§4.4) and refine it to derive a solution (§4.5) that fulfils our requirements. We then mention some extensions (§5) of this solution that we have implemented, and compare our solution to related work (§6). We conclude (§7) by stating what we have achieved and its impact on proving in the RODIN platform.

Relevance The results presented here are independent of the Event-B method and the logic it uses. They are equally applicable in many settings where computer aided proof is performed. The logic used in Event-B is set theory built on first order predicate calculus, but in this paper we only assume that our logic is monotonic[2].

2 Proofs and their Construction

We use the same notion of proof as the one in the standard sequent calculus. We first describe this formal notion of what a proof is.

2.1 Sequents, Proof Rules & Proof Trees

A *sequent* is a generic name for a statement we want to prove. A proof obligation is represented as a sequent. A

[2]By monotonic, we mean that the assumptions of any derived fact may be freely extended with additional assumptions without affecting its validity

sequent has the general form $\mathsf{H} \vdash G$, where H is a finite set of predicates (the *hypotheses*) which can be used to prove the predicate G (the *goal*). In our setting, proof obligations can easily contain many hundreds of hypotheses. A sequent is proved by applying proof rules to it. A proof rule has the general form:

$$\frac{\mathsf{A}}{C}$$

where A is a finite set of sequents (the *antecedents*), and C is a single sequent (the *consequent*). A proof rule of this form yields a proof of sequent C as soon as we have proofs of each sequent in A. The set of antecedents A may be empty. In this case the rule is said to discharge the sequent C. Here is an example of a proof rule:

$$\frac{A, B, C \vdash D \qquad A, B, C \vdash E}{A, B, C \vdash D \wedge E}$$

A *proof tree* is a data-structure that combines individual proof rules to form a larger chain of inference. In this paper we use the terms 'proof' and 'proof tree' interchangeably. Each node in this tree corresponds to a sequent, and may contain a single rule application. In case a node contains a rule application, its sequent is identical to the consequent of the rule, and this node has child nodes whose number and corresponding sequents are identical to the antecedents of the rule. The root of the tree is the sequent we want to prove. The leaf nodes of the tree that do not contain rule applications are the remaining sub-goals that still need to be proved. A proof is either:

Complete when there are no remaining sub-goals.

Incomplete when it has at least one remaining sub-goal.

Here is an example of an incomplete proof tree with two rule applications and one pending sub-goal ($A, B, C \vdash E$):

$$\frac{\dfrac{}{A, B, C \vdash D} \qquad A, B, C \vdash E}{A, B, C \vdash D \wedge E}$$

An example of a complete proof tree can be seen in the next section, which describes how proofs are constructed in our system.

2.2 Reasoners and Tactics

In the previous section we defined the structure of proof rules, but didn't mention where they come from. In our system, proof rules are generated by so-called *reasoners*. A reasoner is a computer program that is capable of generating proof rules. A reasoner is provided with a sequent to prove[3]. It is assumed that reasoners either fail, or generate logically valid proof rules. How this is done is not of interest here. What we should take note of here is that proof rules are *generated* artifacts and the result of some (possibly time consuming or user directed) computation done by the reasoner.

As a first example, we may have a reasoner called $conjI$ that splits a conjunctive goal into two sub-goals. Taking the sequent $A, B, C \vdash A \wedge D$ as input, it produces the following proof rule (named using the name of the reasoner that generated it):

$$\frac{A, B, C \vdash D \qquad A, B, C \vdash E}{A, B, C \vdash D \wedge E}\ conjI$$

We may also have reasoners that internally use specialised decision procedures (say, resolution or linear arithmetic) to completely discharge their input sequents by generating rules without any antecedents:

$$\frac{}{A, B, C \vdash D}\ res \qquad \frac{}{A, B, C \vdash E}\ arith$$

It is assumed that these two reasoners discharge their respective input sequents as shown above. The internal details of how these reasoners work is not of interest here.

Generating individual proof rules and putting them together is a tedious process. In order to make the task of constructing proofs easier, our system provides the convenience of *tactics*. Tactics provide a way to structure strategic or heuristic knowledge about proof search. They provide control structures to call reasoners or other tactics to modify the proof tree.

Internally, all tactic applications are translated, via reasoner calls, into proof rule applications. Tactics provide the scaffolding that makes proof construction more convenient, but once constructed, a proof is capable of standing on its own. We should note here that tactics introduce a second layer of execution (and therefore delay) when constructing a proof.

The notion of tactics is present in some form in many proof tools used today [12, 11]. Typically, all user level interactions for proof construction are expressed via tactics since they provide a concise and high level language to express proof construction steps.

We may, for instance, define a tactic $auto$ that splits all conjunctions in a goal (by calling the reasoner $conjI$ in a loop till it is no longer applicable) and then tries a series of automated reasoners (res and $arith$) on all pending subgoals on the resulting proof tree. Applying this tactic on a proof tree with only the initial root sequent $A, B, C \vdash D \wedge E$ gives us the following complete proof tree:

$$\frac{\dfrac{}{A, B, C \vdash D}\ res \qquad \dfrac{}{A, B, C \vdash E}\ arith}{A, B, C \vdash D \wedge E}\ conjI$$

This proof tree, and its construction (using $auto$) will be used as a running example throughout this paper.

[3] A reasoner may also be provided with some extra input, but this has been omitted in the presentation of this paper

We use the term *proof attempt* to denote some record of the construction of a proof. We say that a proof attempt is *applicable* for a given proof obligation if this proof attempt can be used to *reconstruct* a proof for the given proof obligation. For applicability we also require that if the proof attempt is the record of a complete proof, the reconstructed proof should also be complete.

Since proof construction is a layered process (tactics call reasoners, generating rules), we have a choice of what we want to record as a proof attempt. This choice is governed by the requirements we have on the applicability of proof attempts when their proof obligations change. The next section discusses proof obligation changes and concludes with these requirements.

3 Proof Obligation Changes

We start by describing the different types of proof obligation changes that we treat in this paper and the required effects of these changes on the applicability of proof attempts. The changes are presented in the order of the frequency (from our experience) with which they occur:

1. **Adding hypotheses:** Common modeling changes, such as importing a new theory, or adding a new property to the system being modeled, result in new hypotheses being added to almost all proof obligations. Since the logic we use is monotonic, we should not allow this sort of change to make a previous proof attempt inapplicable.
2. **Removing unused hypotheses:** It is often the case that the user chooses to remove or replace a property he thinks is wrong or redundant. The properties of a system are reflected as hypotheses in almost all proof obligations. The removal of a property therefore results in the removal of a hypothesis in a large number of proof obligations. Removing a hypothesis from a proof obligation *per se* renders its previous proof attempt inapplicable. But proof attempts typically only use a small subset of the hypotheses present in a proof obligation. Removal of unused hypotheses from a proof obligation should not make its previous proof attempt inapplicable.
3. **More severe changes** Next come changes to proof obligations that do not fit in the above two categories, for instance the removal of a hypothesis that is used in a proof, or the change of a goal that the proof manipulates. In these cases it is clear that the previous proof attempt is no longer applicable and needs to be modified, either automatically or interactively.

3.1 Characterising Changes

In general, changes to a proof obligation can be classified into two categories with respect to how they affect the applicability of a previous proof attempt:

Recoverable Those changes for which the system can automatically and efficiently guarantee that a previous proof attempt is still applicable. Previous proof attempts in this case need not be modified. We want changes 1 and 2 to fall in this category.

Irrecoverable Those changes for which the system cannot guarantee that a previous proof attempt is still applicable. In this case a proof attempt may need to be modified (automatically or interactively) to account for this change. Change 3 falls under this category.

The next section describes how the system should react to changes in each of these categories.

3.2 Reacting to Changes

Automatically reacting to proof obligation changes as they happen gives the user *immediate feedback* on how his last modeling change affects his proof development. But automatically reacting to a change should not result in a *loss of some previous proof effort*. In this section we describe how the system should react to change, keeping these two points in mind.

We start by making a distinction between *reacting* to a change, and *recovering* from it. The system *reacts* every time it notices that a proof obligation has changed. Recovering from a change means that the system, at the end of the recovery process, ensures that the current proof attempt is applicable for the changed proof obligation. Reacting to a change may mean recovering from it, but later in this section we will see that this is not always desirable.

We would always like to recover from a recoverable proof obligation change (as defined in §3.1) since:

- Recovery is guaranteed to succeed.
- Recovery is not computationally intensive.
- The previous proof attempt is not modified.

In this case, recovery only involves *checking* that the proof attempt is still applicable and marking it as such.

On the other hand, if the change is irrecoverable, the previous proof attempt is now inapplicable. Recovering from such a change requires the creation of a new proof, possibly with the help of a previous proof attempt. Reacting to such changes by trying to immediately recover from them is undesirable because:

- Recovery may fail.
- Recovery may be computationally intensive
- Recovery may need user interaction.
- The previous proof attempt may be lost.

The user typically does not want to spend time and resources on proving proof obligations after every modeling change, but only at points where he thinks his model is stable enough to be worthy of proof effort. At other points in the development, the model may be unstable or incomplete, resulting in incomplete or unstable proof obligations. Immediate recovery from such changes would mean that time and effort is spent on proving proof obligations that will change in the very near future. A more serious consequence of this is that any previous proof attempt would be modified, or even lost. The user should be able to make experimental changes without worrying about the system modifying his proofs. Since the system does not have a way of knowing the intent of a modeling change, the best thing to do in this case would be to mark any previous proof attempt as now being inapplicable. The user can later make this proof attempt applicable again, either by undoing his modeling change, or explicitly asking the system to recover by reusing a previous proof attempt.

Proof attempts generated automatically (i.e. without any user interaction) are an exception to this rule. Losing such proof attempts is not that serious since they can be regenerated automatically. In this case, a system may try to immediately recover from an irrecoverable change (by possibly rerunning automated provers) and replace the previous proof attempt with a new one in case this was successful.

Table 1 summarises how the system should react to recoverable and irrecoverable changes. From this table we see that it is important that proof attempts are represented such that they are recoverable whenever possible.

Change	Previous Proof Attempt	Reaction
Recoverable	-	Recover by marking proof attempt applicable
Irrecoverable	Interactive	Mark proof attempt inapplicable
	Automatic	Recover with new automatic proof attempt

Table 1. Reacting to Changes

3.3 Requirements

We conclude this section by placing three concrete requirements on the way proof attempts are represented:

1. Addition or removal of unused hypotheses should be guaranteed to be recoverable proof obligation changes.
2. The check to see if a proof attempt is applicable for a proof obligation should be efficient.
3. For irrecoverable changes, proof attempts should be reused incrementally to construct new proof attempts.

The next section talks about how to represent proof attempts so that they fulfil these requirements.

4 Representing Proof Attempts

This section compares various alternatives we can consider to represent proof attempts. As mentioned before, a proof attempt is a record of the construction of a proof. A proof tool must allow proofs to be saved and reloaded from saved proof attempts. We deliberately talk of recording proof *attempts*, and not of explicitly recording proofs as they are defined in §2. This is because we are not bound to recording these proofs explicitly, but only require that the proof tool be able to *reconstruct* these proofs from their proof attempts.

There are two types of information that a proof attempt can record about the construction of a proof:

- *what* has been proved.
- *how* a proof is constructed.

Proofs are the result of a proof construction activity. They record *what* has been proved. The way a proof is constructed though may provide more information on *how* the proof was done which cannot be expressed within the proof itself. A proof attempt may therefore record more information from the proof construction phase than what is present in the constructed proof. In our setting where proof obligations are subject to change, using information about *how* a proof was constructed is of great importance while reusing a proof attempt when its proof obligation changes[4]. At the same time, it is also important to record *what* was proved since we would like to do this reuse incrementally, and do not want to reconstruct a proof just to check if it is still applicable for a proof obligation.

We first consider three approaches of how to record a proof attempt that follow naturally from the way proofs are constructed. The type of information they record are in parenthesis:

1. Recording proofs explicitly (what)
2. Recording reasoner calls (how)
3. Recording tactic applications (how)

In the subsections that follow (§4.1, §4.2, §4.3), we first evaluate these three ways of recording a proof attempt with the following criteria in mind:

Efficiency: How efficient is this approach with respect to the space required to store proof attempts, the computation required to show that a proof attempt is applicable for a proof obligation, and the computation required to reconstruct an actual proof from it?

Reusability: What possibilities do we have to reuse proof attempts as they are represented using this approach?

[4] A central assumption here is that proof obligations normally do not change so drastically that they require a radically different approach to prove them.

As we will see, each approach has its advantages and shortcomings. Our aim in this section is to make a reasonable trade-off, keeping our three requirements from §3.3 in mind. In §4.4 we propose a mixed approach, and refine this approach in §4.5 to derive a solution (proof skeletons) that fulfils our requirements.

4.1 Recording Proofs Explicitly

As a first attempt, we represent a proof attempt explicitly as a proof (i.e. as a proof tree defined in §2). This approach corresponds to the *proof object* view of looking at a proof attempt and records *what* has been proved in full detail.

Efficiency Proof trees are large and cumbersome data structures. Because of this, most proof tools in use today by default avoid the explicit construction of proof trees all together. In our setting, a proof obligation may easily contain hundreds of hypotheses which are mostly repeated in all nodes of a proof tree. Representing proof attempts in this way therefore raises serious storage concerns. Reconstructing a proof tree though is a trivial task; it only has to be reloaded. Checking if a proof attempt is recoverable for a proof obligation is also trivial; it has to be identical to the root sequent of the proof tree. No reasoner calls are required for both these operations.

Reusability Although this approach eliminates the need for proof reconstruction, it gives us absolutely no way of reusing proof attempts when proof obligations change. A proof can only be associated with a single proof obligation: the sequent occurring in its root node. When a proof obligation changes, its proof is no more applicable.

To illustrate this, here is how the proof attempt of the proof presented in §2.2 (our running example) would look like in this setting:

$$\frac{\overline{A, B, C \vdash D} \quad \overline{A, B, C \vdash E}}{A, B, C \vdash D \wedge E} \qquad \textit{(proof attempt 1)}$$

The rule names have been omitted since they have no significance here. Trying to reuse this proof attempt for the identical proof obligation $A, B, C \vdash D \wedge E$ is fine. Any slight changes though (for instance adding a hypothesis) make this proof attempt inapplicable.

The next two sections describe proof attempts that record *how* a proof was constructed.

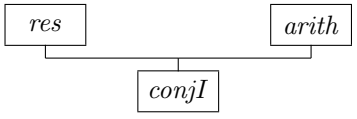
4.2 Recording Reasoner Calls

In this approach, a proof attempt is represented as a tree that records all reasoner calls required to construct a proof. The structure of this tree is identical to that of a proof tree. The nodes of this tree though do not store sequents or proof rules, but the reasoner calls that were used to generate these proof rules. In contrast to the previous approach, a proof attempt now stores *how* a proof was constructed.

Efficiency This representation of a proof attempt requires much less storage since we do not need to store sequents. Reconstructing a proof from such a proof attempt now requires extra computation since each proof rule needs to be regenerated by calling the reasoner that originally generated it. Checking that a proof attempt is applicable for a proof obligation (even if it has not changed) is expensive since it requires the proof to be reconstructed.

Reusability Compared to the previous approach, proof attempts stored in this way give us more possibilities for reuse when proof obligations change. This is because we now have a layer of computation between a proof attempt and the proof reconstructed from it. This makes it possible for a proof attempt to recover from proof obligation changes. How much change a proof attempt is able to recover from depends on how well each reasoner deals with change in its input sequent. In practice, reasoners are specialised for a particular task and are ignorant of most details present in their input sequents. For instance, the reasoner $conjI$ only needs the goal to be a conjunction and is ignorant of the hypotheses.

Here is how the proof attempt of our running example would look like in this setting:



(*proof attempt 2*)

This proof attempt is no more a proof tree, but has the same structure. As in the previous case, trying to reuse this proof attempt for the identical proof obligation $A, B, C \vdash D \wedge E$ succeeds. If we added an extra hypothesis F, or removed a hypothesis that was not needed by any reasoner in the proof (say A), this proof attempt would still be reusable. In both these cases though, this cannot be guaranteed without re-executing each reasoner call. If the goal changes from $D \wedge E$ to just D, reuse would fail since the first reasoner call to $conjI$, which requires a conjunctive goal, would fail.

4.3 Recording Tactic Applications

In §2.2 we introduced the notion of tactics and mentioned that all user level interactions for proof construction are expressed using tactic applications. If one could keep track of the sequence (i.e. a list) of tactic applications the user used to construct a proof, this information could also be recorded as a proof attempt. This is how proof attempts are represented in most proof tools [12, 11].

Efficiency Proofs are reconstructed by replaying all tactic applications in the order they were applied when constructing the proof. This approach simulates the original construction of the proof at the user interaction level. Since tactics introduce a second level of execution to proof construction, and may call reasoners that fail, reconstructing a proof from such a proof attempt requires the greatest amount of computation of all three approaches. The same is true for checking if a proof attempt is applicable for a proof obligation. The amount of storage required for such a proof attempt is the least of all three approaches.

Reusability Since this way of representing a proof attempt records each user interaction with the proof tool, it could in principle give us the greatest possibility for reusing proof attempts. But since the way a tactic works is unconstrained (it may globally modify the entire proof tree) and highly unpredictable (we have no control over what it does), in practice we have very weak guarantees for when such a proof attempt is reusable for changes in a proof obligation.

Going back to our running example, its proof attempt in this setting is a list with only one tactic application:

$apply(auto)$ (*proof attempt 3*)

It may be reused successfully for $A, B, C \vdash D \wedge E$, and for variations of it with the addition or removal of unused hypotheses, and even for $B, C \vdash D$. But we can make this claim only because we know what the tactic $auto$ does. In general, it can be the case that the addition of a hypothesis makes a serious change in the way a tactic works (for instance the addition of a disjunctive hypothesis may trigger a tactic to start a case distinction), leaving the remainder of a proof attempt irrelevant and unusable. This is a recurrent problem in proof tools where proof attempts are stored as tactic applications [8].

We see that this approach gives us even weaker guarantees (compared to §4.2) about the applicability of a tactic to a changed proof obligation. Tactic applications are therefore not well suited for reuse in our setting.

Till now we have seen three approaches to represent proof attempts that followed naturally from the way proofs are constructed. The first three rows of table 2 (that appears towards the end of the paper) compares these representations with respect to our requirements in §3.3. From this table we see that no single approach fulfils our requirements. The aim of the next section is to combine the advantages of two of these approaches to devise a proof attempt representation that can be *incrementally* reused.

4.4 A Mixed Approach

Each of the three proof attempts presented till now either recorded exactly *what* was proved, or *how* it was proved. The aim of this section is to combine these two types of information in order to reuse proof attempts *incrementally*, and to avoid reconstructing a proof just for checking that a proof attempt is applicable for a proof obligation.

A natural choice for this mixed approach is to combine the approach of recording entire proofs (§4.1) with the one recording reasoner calls (§4.2). We use the same reasoner call tree, but additionally store the proof rules generated by each reasoner call in its node. This is how the proof attempt of our running example would look in this setting:

$$\dfrac{\boxed{\dfrac{}{A, B, C \vdash D}\ res} \quad \boxed{\dfrac{}{A, B, C \vdash E}\ arith}}{\boxed{\dfrac{A, B, C \vdash D \quad A, B, C \vdash E}{A, B, C \vdash D \wedge E}\ conjI}}$$

(*proof attempt 4*)

From this mixed proof attempt we can easily extract its explicit proof tree (*proof attempt 1*), or its reasoner call tree (*proof attempt 2*). As in §4.1, we are guaranteed that this proof attempt is applicable for the proof obligation $A, B, C \vdash D \wedge E$ without the need to reconstruct its proof. For any modifications of this proof obligation, its proof needs to be reconstructed as described in §4.2.

But this reconstruction can now be done *incrementally*. Each node in this tree contains a reasoner call, as well as the proof rule it generated. The proof rule can be seen as a *cache* of the result of the reasoner call. When reconstructing a proof at such a node, this cached proof rule can be used directly if its conclusion is identical to the sequent to be proved. If not, the reasoner can be called again with the new sequent to be proved. In case this succeeds, we have a new proof rule that we can use to carry on with the proof reconstruction. In this way it is possible to save expensive calls to an automated reasoner at the ends of a proof tree.

For instance, reusing (*proof attempt 4*) when the conjunct E in the goal changes to E', gives us the following proof:

$$\dfrac{\boxed{\dfrac{}{A, B, C \vdash D}\ res} \quad \dfrac{}{A, B, C \vdash E'}\ arith}{A, B, C \vdash D \wedge E'}\ conjI$$

where we did not need to call res again to generate the boxed inference rule.

Evaluation This approach allows us to reuse a proof attempt incrementally and gives us a guarantee on its applicability (when its proof obligation has not changed) without having to reconstruct its proof. But this guarantee is much weaker than the one we desire. We would like proof attempts to still be guaranteed applicable (without reasoner replay) for addition and removal of unused hypotheses. This

approach is also not usable in practice since it requires storing entire proof trees (via their proof rules). The next section describes the alternative that we use in place of storing proof rules to refine this mixed approach to solve the above problems.

4.5 Proof Skeletons

As seen earlier (§4.1), the problem with recording proof rules is that they are too verbose. This not only makes them too large to store, but also too *hard-wired* to reuse. A proof rule basically tells us how a pending sub-goal changes in a proof. A change in a pending sub-goal should give us two pieces of information:

Applicability When is this change applicable for a pending sub-goal.

New sub-goals If applicable, what are the new pending sub-goals that replace this pending sub-goal.

A proof rule gives us exactly this information, but not in a modular and incremental form. The idea behind *proof skeletons* is to store the *changes* a reasoner makes to a pending sub-goal instead of an entire proof rule as it appears in a proof.

In §4.2 we observed that a reasoner is typically specialised for a particular task and is ignorant of most of the details present in its input sequent. But since its output is a proof rule, the fine details of what it actually depended on, and what it actually modified are lost.

In order to keep this information, the output of a reasoner needs to be made more precise. The next subsection shows how this can be done to track how hypotheses change and how they are used in a proof.

4.5.1 Tracking Hypotheses

The aim of this section is to come up with a way of recording proof attempts that is guaranteed to be recoverable after adding or removing unused hypotheses from proof obligations. We will use ideas from how hand-written proofs are done to achieve this. When doing a proof by hand, one does not start by stating all assumptions one *may* depend upon, but by keeping these assumptions in mind, and letting the proof steps dictate which assumptions are used as the proof unfolds.

To use this idea of computing used hypotheses as the proof unfolds, instead of returning a proof rule, a reasoner needs to return a reduced version of the proof rule, called a *proof rule skeleton*. A proof rule skeleton is identical to a proof rule except:

1. The hypotheses of the conclusion are restricted to those that were actually used by the reasoner.
2. Only changes to these used hypotheses (i.e. their addition or removal) are recorded in place of all hypotheses of each antecedent.

This is what the proof rule skeletons look like for our running example:

$$\frac{\vdash D \quad \vdash E}{\vdash D \wedge E}\ conjI \qquad \frac{}{C \vdash D}\ res \qquad \frac{}{B \vdash E}\ arith$$

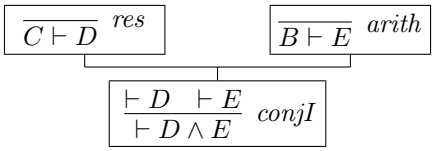
In contrast to the complete proof rules that appear in §4.4 (*proof attempt 4*), from the rule skeletons above we can clearly see what hypotheses were used by each reasoner (none for $conjI$, C for res and B for $arith$). This representation also corresponds more closely to how proof rules are expressed in hand-written proofs.

Using such a rule skeleton to construct a proof rule is straightforward. Since our logic is monotonic[5] we can systematically add hypotheses to the conclusion and all antecedents to get a valid proof rule. For instance, we can use the rule skeleton $conjI$ above to construct the following proof rule that we have used in our running example by systematically adding the hypotheses A, B, and C (here boxed) to its conclusion and antecedents:

$$\frac{\boxed{A, B, C} \vdash D \quad \boxed{A, B, C} \vdash E}{\boxed{A, B, C} \vdash D \wedge E}$$

A rule skeleton has greater applicability (and therefore reusability) compared to a proof rule. For hypotheses, it only requires that the used hypotheses from the conclusion be present in a pending sub-goal.

We get a proof skeleton by replacing proof rules in the mixed proof attempt (§4.4) with proof rule skeletons. The proof skeleton for our running example is:



(*proof attempt 5*)

Using rule skeletons instead of proof rules makes (*proof attempt 5*) much more concise compared to (*proof attempt 4*). Proof skeletons do not present serious storage problems in practice since they typically do not contain a large number of rule skeletons (which themselves only contain relevant hypotheses).

From (*proof attempt 5*) and the initial sequent $A, B, C \vdash D \wedge E$ we can reconstruct its explicit proof (*proof attempt 1*) using just the rule skeletons. But what we are more interested in is the applicability of this proof skeleton for other initial sequents. For this we can recursively compute what hypotheses need to be present in the initial sequent so that each rule skeleton is applicable. For (*proof attempt 5*) we

[5] Monotonicity asserts that the assumptions of any derived fact may be freely extended with additional assumptions. Over here we interpret 'derived fact' to be a proof rule.

Proof Attempt Representation	Is it recoverable for:				How is it reused for irrecoverable changes?
	Original PO	How?	Original PO ± (unused) hyps	How?	
Explicit Proofs	Yes	guaranteed	No	-	Not possible
Reasoner calls	Yes	reasoner replay	Yes	reasoner replay	reasoner replay
Tactics	Yes	tactic replay	maybe	tactic replay	tactic replay
Mixed approach	Yes	guaranteed	Yes	reasoner replay	incremental
Proof skeletons	Yes	guaranteed	Yes	guaranteed	incremental

Table 2. Comparing Proof Attempt Representations

can compute that if the initial sequent contains the hypotheses C and B (and trivially, the goal $D \wedge E$), each of its rule skeletons will be applicable. This information can be stored along with the proof skeleton as a cache of its *dependencies*:

Used Hyps : C, B Goal : $D \wedge E$	(*proof attempt 5 dependencies*)

These cached dependencies can be used later to efficiently check if a proof skeleton is still applicable for a proof obligation when it changes.

For (*proof attempt 5*) we are guaranteed that we can reconstruct a proof for *any* initial sequent containing the hypotheses C and B, and the goal $D \wedge E$, without doing this reconstruction explicitly. We may now safely add all the hypotheses we want, or remove any unused hypotheses without making our proof attempt inapplicable.

In this section we have seen five approaches to represent proof attempts. Table 2 compares these representations with respect to the requirements that we placed on proof attempts in §3.3. From this table we see how we improved our representation of a proof attempt until we arrived at one (i.e. proof skeletons) that fulfilled our requirements.

The next section describes some extensions of the proof skeleton approach that we use to represent proof attempts in the RODIN platform.

5 Our Solution

For the RODIN platform we use proof skeletons to record proof attempts as described in the previous section, with added support for:

Hypothesis dependencies as explained here, also taking forward inferences[6] into account. A forward inference contributes to the used hypotheses only if it produces a hypothesis that is used further up in the proof.

Goal independence for proofs that succeed irrespective of what goal the initial sequent has (for instance a proof due to contradictory hypotheses).

Type environment dependencies to track the identifiers used in a proof, along with their types.

Free identifier dependencies to track which identifiers were assumed to be undeclared in an initial sequent.

Refactoring of proof attempts when identifiers are renamed.

The proof skeleton representation also opens up new possibilities for revalidating and refactoring proof attempts. Due to space limitations we are unable to elaborate on these points and present a formal treatment of this work which will appear later.

6 Related Work

The topic of proof reuse has been heavily studied in the context of program verification. The KIV and KeY systems use dynamic logic to verify imperative and object oriented programs. Both systems support a form of proof reuse [13, 5] based on reasoner replay as discussed in §4.2. They show impressive results using heuristics that are fine tuned for their particular setting. Similar heuristics can be incorporated into our setting to better handle irrecoverable changes. Our solution independently contributes to their approach (and other replay based approaches) since proof attempts are guaranteed resilient to simple changes where explicit replay may be avoided.

Our solution for avoiding explicit reuse is similar to [10, 16], where a proof is generalized using meta-variables that can be reinstantiated for proof reuse. Although more general, this approach requires higher-order unification (which is undecidable) for reinstantiating proofs. The authors in [10] claim that the unification problems they encounter are fairly simple, but we believe that in our setting (where a sequent may have hundreds of hypotheses), unification, even if well behaved, would still take more time than reasoner replay. We opt for a less general, but simpler and more efficient solution to support the types of reuse that we frequently need.

The VSE system supports proof reuse [16] over an evolving specification by explicitly transforming specifications and proofs simultaneously using a set of predefined basic

[6] Forward inferences derive new hypotheses using old ones and are frequently used to simplify hypotheses.

'development transformations'. Although this allows for a more controlled proof reuse, we do not take this approach due to two reasons. First, as discussed in §3.2, we do not want to forcibly recover from every modeling change. Secondly, we believe that a proof tool should use proof obligations, and not models as a source of change. We want the proof tool to work independently of the modeling constructs used, allowing both to evolve independently. Our proof tool does have the possibility of accepting 'hints' from the modeling tools (e.g. relevant hypotheses, constant renaming), but these have a logical meaning independent of the modeling formalism used.

Our approach is similar to earlier work done on the mural system for checking the consistency of proofs when theories change [15]. Proof attempts were represented explicitly as natural deduction style proofs that could be traversed and checked for consistency when their proof obligations changed. Explicit proof dependencies though were not computed. It is unclear from the documentation how such proof attempts could be replayed for irrecoverable changes.

Proof reuse is not such a hot topic in pure logic based proof assistants such as Isabelle [12]. Users do reuse proof attempts, but by rerunning tactic application scripts. This is justified since in such systems one does not prove 'proof obligations' (that are machine generated and change often), but 'theorems' (that are entered by the user and are somewhat stable). Nevertheless, the need for proof management and reuse is felt [8], the more such tools are used for verification. Recent work on theorem reuse [6] in Isabelle is a step in this direction, but this is a post proof consideration and requires explicit user guidance.

7 Conclusion

In this paper we have presented a solution to represent and manage proof attempts such that:

- They are guaranteed to be resilient (still applicable) for some simple (recoverable) changes that proof obligations frequently go through when using a reactive development environment.
- It is efficient to check that a proof attempt is still applicable when its proof obligation changes.
- For irrecoverable changes, proof attempts can be reused incrementally to construct new proof attempts.

We have found the ideas presented in this paper important for supporting proof within the RODIN reactive development environment. This has facilitated better interaction between the modeling and proving processes and has resulted in a marked improvement in the usability of this tool over the previous generation of tools [7, 3] for the B method.

Acknowledgements

The author would like to thank Jean-Raymond Abrial, Cliff Jones, Laurent Voisin, Vijay D'Silva, Stefan Hallerstede, Thai Son Hoang, Adam Darvas, Joseph Ruskiewicz and the anonymous reviewers for their comments.

References

[1] J.-R. Abrial. *System modeling with Event B*. Cambridge, 2007. to appear.

[2] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM 2006*, volume 4260, pages 588–605. Springer, 2006.

[3] J. R. Abrial and D. Cansell. Click'n prove: Interactive proofs within set theory. In *TPHOLs 2003*, volume 2758 of *Lect. Notes in Comp. Sci.*, pages 1–24. Springer-Verlag, 2003.

[4] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[5] B. Beckert and V. Klebanov. Proof reuse for deductive program verification. In J. Cuellar and Z. Liu, editors, *Proceedings, Software Engineering and Formal Methods (SEFM), Beijing, China*. IEEE Press, 2004.

[6] E. Broch Johnsen and C. Lüth. Theorem reuse by proof term transformation. In *International Conference on Theorem Proving in Higher-Order Logics TPHOLs 2004*, pages 152–167, Sept. 2004.

[7] Clearsy. *Atelier B. User Manual*, 2001. Aix-en-Provence.

[8] P. Curzon. The importance of proof maintenance and reengineering, 1995.

[9] Eclipse - an open development platform, official website. http://www.eclipse.org.

[10] A. Felty and D. Howe. Generalization and reuse of tactic proofs. In F. Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, volume 822 of *LNAI*, pages 1–15, Kiev, Ukraine, 1994. Springer-Verlag.

[11] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system, Jan. 15 2001.

[12] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[13] W. Reif and K. Stenzel. Reuse of Proofs in Software Verification. In J. Köhler, editor, *Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs*, IJCAI. Montreal, Quebec, 1995.

[14] Rigorous Open Development Environment for Complex Systems (RODIN) official website. http://rodin.cs.ncl.ac.uk/index.

[15] K. J. Ross and P. A. Lindsay. Maintaining consistency under changes to formal specifications. In *FME*, pages 558–577, 1993.

[16] A. Schairer and D. Hutter. Proof transformations for evolutionary formal software development. In *AMAST '02*, pages 441–456, London, UK, 2002. Springer-Verlag.