Electronic Notes in Theoretical Computer Science 70 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 20 pages

Refactoring in Maintenance and Development of Z Specifications and Proofs

Susan Stepney [1] ,[2]

Logica UK Ltd,
Betjeman House, 104 Hills Road, Cambridge, CB2 1LQ, UK.

Fiona Polack [3] Ian Toyn [4]

Department of Computer Science,
University of York,
Heslington, York, YO10 5DD, UK.

Abstract

Once you have proved your refinement correct, that is not the end. Real products, and their accompanying specifications, develop over time, with new improved versions having added functionality. There are new maintenance issues that arise when altering and upgrading pre-existing large specifications and their respective proofs.

We show how concepts from refactoring can be used to structure this process, and provide a means for well-defined, disciplined modifications. Additionally, we discuss how the analogy between proof and refactoring, as meaning preserving transforms, can be used to suggest the development of a refactoring toolset, and thence a refinement toolset.

Keywords: Z, refinement, refactoring, patterns

1 Introduction

Logica's Formal Methods Team (LFM) has been involved in several industrial-scale Z specifications and proofs: of a compiler [9]; of an electronic Purse [12]; of a Smart Card Operating System [11]. [10] reports on some issues to do with performing these industrial-scale Z proofs, and sketches some requirements for proof tool support to help in this task.

[1] Email: susan.stepney@cs.york.ac.uk
[2] current address: Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK.
[3] Email: fiona@cs.york.ac.uk
[4] Email: ian@cs.york.ac.uk

As a system undergoes maintenance and further development, its accompanying specification and proof have to keep track. Here we discuss issues that have arisen from the subsequent maintenance of these large-scale specifications and proofs. We then discuss how the concepts of refactoring can be used to illuminate the requirements for tool support, and discuss how a proof tool can be viewed as the first step towards a refactoring tool.

The examples presented here are given in Z, since they have been derived from Z specifications. However, the concepts are largely language independent.

2 Maintenance

Even when starting a specification and proof task from scratch, a commercial development rarely starts from a clean sheet of paper. Often implementation details constrain what can be done, and how the specification can be structured. When enhancing an existing specification, say upgrading functionality, these constraints are even more important.

2.1 Impact analysis

Often the customer decides on the upgrade required, and requires this to be added to the specification. In simple cases this may merely require the adding of an operation (assuming a state and operations style specification) to capture the new functionality. In more complex cases it may also require the adding of more state to capture how the new function works. Even more complex cases may require radical alterations to the specification, because the change may subvert a modelling assumption of the original.

Part of the maintenance process is *impact analysis*: determining the cost and consequences of a proposed change before making that change. The effect on the formal specification and proof should be included in this impact analysis, so that the customer can realise the actual effect of what looks to them to be a simple implementation change.

For example, in one of LFM's projects, we experienced a problem when a customer added an operation with a different execution style. This new operation used a *push* model (the system waiting for a command, then responding to whatever command it got), whilst all the other operations used a *pull* model (the system actively looping until it finished processing a sequence of commands). The change of model had little impact on the implementation, requiring only a simple flag local to that one operation. The impact on the specification was somewhat larger. We had to modify the specification to include a flag as a global state component, that got explicitly switched *off* in every operation except the new one, thereby affecting every operation. We also had to provide some subtle argumentation as to how this global specification variable corresponded to the implementation's local variable. Earlier consultation with the formal methods team might have resulted in a different design.

2.2 Maintaining conventions

It is essential during maintenance also to keep the non-formal parts of the specification up to date. The indexing, commenting, and layout conventions all need to be maintained. This can be difficult if the conventions have not been documented. In LFM's projects, there was a degree of continuity of the staff involved, but the conventions in use did have to be explained to new staff. Consequently, we have started to describe some simple Patterns for these conventions [13], to aid both the original specification process, and to help with maintenance.

3 Refactoring

When upgrades involve addition of operations and state components, the specification gets steadily more complex. It is essential to make structural improvements to a specification that is undergoing such continual development, else even the most elegant one soon degenerates into incomprehensible symbol soup.

Refactoring [4] is a technique for improving the structure of code in a disciplined, controlled, and manageable way. It is a technique to improve design without changing behaviour. In particular, refactorings can be used to ensure that evolving code makes use of the required Patterns [6].

Each individual refactoring change is very small, such as renaming a component, or moving a feature, or splitting a method. Thus each change can be understood and tested in isolation. Large improvements to the structure of the code are made by applying a sequence of such small controlled changes towards some goal.

Refactoring is particularly emphasised as part of the XP discipline [2], which relies heavily on an incremental approach to design and coding. The Cleanroom development process [3], which allows a component to be changed provided the input to output function is unchanged, can be viewed as an early example of meaning-preserving refactoring.

3.1 Refactoring a specification and proof

These concepts of refactoring are readily applicable to formal specification and proof developments, as we show below. Mathematicians in particular are fond of rewriting proofs to make them clearer, or more general, or shorter. In this paper we are discussing a more specific discipline: a particular process (refactoring in small controlled steps) to achieve a goal (better structure to help with maintenance and upgrades).

A formal refactoring should not itself change the meaning of the specification. (We refrain from defining what the meaning of a specification is. It may be the set of models as given by something such as the International Standard for Z [7]. It may be a weaker concept such as “essential meaning” [5], being the specification restricted to those names of interests. It may be some weaker meaning still, relevant to the particular specification context.) Refactoring may, however, be a desirable step before or during upgrading the specification and proof (which upgrading will then change the meaning).

A refactoring may be applied to the specification to improve its structure, in which case the changes need to be propagated through any affected proofs. A refactoring may be applied to the specification in order to improve the structure of a subsequent proof. A refactoring may be applied to a proof alone, to improve its structure, leaving the specification unaffected.

3.2 Small steps

The first rule for formal refactoring, as for code, is to make the changes in small, controlled steps. Each refactoring step should be the smallest logically complete change that can be made, and pushed all the way through the specification and proof. (A small change to the specification can have a large knock-on effect on the proof. This is another reason refactoring steps should be as small as possible.) Do not succumb to the temptation to make several large sweeping changes in one go: it is very easy to get lost in the morass of changes, and forget or miss a needed change to a proof. Errors are easier to locate if the change is small. Small steps are also easier to document, and to record in the change lists.

For example, one of LFM’s specifications has a system state comprising about 30 components, with about 15 predicates constraining them. Early on in the original specification development, we partitioned the state into four fairly independent sub-states (independent because few of the predicates involve components in different substates, and most of the operations change components in a single sub-state). During one subsequent upgrade cycle, it became clear to us that one of the state components, and its associated predicates, would fit better in a different sub-state, because there it would result in significantly fewer of the predicates and operations referencing multiple substates. We moved the declaration and the predicates in a single refactoring step, pushing the required changes through the operation definitions and proofs. This was the smallest logically complete step we could make: taking two steps (declarations, then predicates, say) would have left the intermediate specification type-incorrect and unprovable.

It is possible to introduce errors during an attempted refactoring step (especially at the present, where tool support is not yet fully developed). Also, some refactoring steps can result in larger than anticipated changes to the specification, and particularly the proof. If the benefit from the refactoring is small, it might be better to revert rather than continue with the big restructuring. So make sure it is possible to roll back each change (using the version control system) so that it is possible to recover from mistaken attempts to refactor.

3.3 Identifying refactorings

Refactoring opportunities become more apparent as the specifier gains experience. Additionally, in formal refactoring, the process of performing the proof gives additional insight into the structure of the system. Lessons learned in this way can suggest refactorings to improve structure.

Lessons learned while doing the proofs can be used to suggest refactorings to the specification. For example, in the Purse project (see earlier), we noticed that certain combinations of state components appeared in the refinement proofs, and that these combinations mapped to meaningful concepts in the application domain. So we introduced these into the specification as derived state components. This refactoring improved the clarity and structure of both the specification and the proofs.

Lessons learned while doing the proof can be used to refactor the proof itself. For example, in the Purse project, we became aware that we were proving a similar property several times. We parameterised this property, extracted it as a proved lemma, then used it several times in the main proof. This shortened and simplified the main proofs. The lemma made sense as a property in the application domain, so this refactoring also made the proof structure easier to understand.

3.4 Change control lists

Refactoring can inflate change control lists, and make the changes look bigger than they really are, which might dismay any third party expecting to evaluate only a small upgrade. In LFM's projects, we took care to separate out the list of changes that were refactorings from those that were real functionality changes, to help structure the evaluation task.

4 Proving Refactorings

The code refactoring process that [4] describes has to be modified for refactoring non-executable specifications.

With code refactoring, there is heavy emphasis on the testing. One must execute the regression test suite after every small refactoring step to ensure the code has not been broken, and that the meaning has not been changed. Since each step is small, any bug introduced by the change should be rapidly detectable and fixable.

With specification refactoring the analogue of running the regression test suite is doing type checking, theorem proving, and validation.

4.1 Type checking

Type checking catches the small, silly mistakes. This is easily performed by a formal notation support tool.

4.2 Theorem proving

Next the refactoring is propagated through the proofs: this involves first making the changes and then ensuring that all the reasoning steps are still valid after these changes. This can sometimes require quite significant changes to the proofs. Refactoring steps should be kept as small as possible partly to ensure these proof changes are not too dramatic.

This step serves to expose any changes to the meaning of the specification that affect those theorems being proved.

4.3 Validation

There is still the possibility that a change may affect the meaning of the specification, but not the validity of any of its theorems. The question arises, is this a problem? If the specification and proof are capturing a refinement relation, a change to meaning that does not invalidate the refinement is probably not a problem. More care needs to be taken if properties other than refinement are of interest, for example, a security or safety property. The change should be revalidated against the informal requirements.

For example, one of LFM's specifications originally had several predicates constraining the abstract state that were capturing the actual functioning of the concrete system. In "benefactoring" steps (see later), we removed these from the abstract specification, thereby simplifying it, leaving them only in the concrete specification. Since nothing required the constraints in the abstract model, removing these constraints did not affect the proofs. We justified this change by (informal) validation: these constraints were not a necessary property of the system being specified, merely a property of the more concrete design.

4.4 More theorems

Refactoring should be considered as an opportunity to capture further desirable but implicit properties of the specification. These desirable properties can be expressed as theorems, requiring reproof after refactorings. These theorems act as a 'regression proof suite' to ensure further refactorings are correct.

4.5 Incremental refactoring process

Although a refactoring is the smallest possible incremental change, some single refactorings can additionally be broken down into smaller, partially checkable steps. Even though the entire specification and proof is inconsistent part way through the steps, portions of it can be checked. So, for example, in the context of a refinement proof

- modify and check the abstract specification
- modify and check proofs about the abstract specification
- modify and check the concrete specification
- modify and check proofs about the concrete specification
- modify and check the retrieve relation specification
- modify and check the refinement proof

At any step an error might be discovered (for example, that a global property does not hold). This might require earlier steps to be modified.

5 Useful Refactorings

In this section we describe some of the refactorings that the Logica Formal Methods team has found useful in its various Z projects. Some of these are done simply to improve and clarify structure. Others are done also to conform to various Z conventions, as expressed in [1], and currently being captured as Z Patterns in [13].

The proofs in these projects were all performed by hand, with minimal tool support (type checking only). These specifications and proofs were independently evaluated; some of the structural improvements are designed to make evaluation of such proofs easier. Different refactorings might be more appropriate for tool-supported proofs. (See also the discussion on tool support for the refactoring process later.)

Many refactorings can be applied in either direction (since they are meaning preserving). The choice of which direction to take in a particular case depends on the specific context. The purpose is to improve the structure, readability, and modifiability of the specification and proof.

The statement of each refactoring has the following structure (some parts may be omitted for brevity)

- name
- short statement of the problem
- short statement of the refactoring change
- discussion
- process steps to be followed to achieve the refactoring
- example

5.1 Rename a Component

You have a specification component with a name that does not indicate its purpose, or otherwise breaks the naming convention. *Change the name.*

This refactoring is most useful in the early stages of specification. Initial names choices can become inappropriate as the specification develops and the purpose of a component becomes clarified. The initial name might be overly suggestive (containing more "semantics" than does its definition), or otherwise misleading. The naming convention is often evolving at this point, too. Keeping the names meaningful and uniform makes the specification more readable.

- choose the new name
- update the naming convention documentation if necessary
- rename the definition
- propagate the name change throughout the specification and proofs. The typechecker can be used to help find all the places the name needs to be changed.

5.2 Extract Commonality

You have a term used in several places in the specification or proof. *Introduce a new definition to provide a name for the term, and use that name in place of the term.*

Make sure the newly named term is a meaningful concept in its own right, not just derived from a textual coincidence. Let the name capture this meaning (and follow any naming convention Pattern in use). The use of the name makes the specification more readable, and more concise.

The "term" might be an expression, a predicate, a part of a schema, or even a chunk of proof. The new name can be introduced as a global definition (possibly a new toolkit definition), a schema, a derived state component, a local definition (existentially quantified), or a lemma (in which case the "name" occurs in the informal commentary rather than the formal text).

- create the new definition
- typecheck, to ensure there are no name clashes
- for each use of the term in the scope of the definition
  - · replace the use of the term with the name
  - · typecheck, to ensure the name has been used properly in this case
  - · propagate the replacement through the proofs (this may require the addition of an expansion step, replacing the name with its definition, at each point in the proof when the definition is used)

5.3 Inline a Name

You have a name with a relatively simple definition, used only once or a few times. *Remove the name, and replace its use(s) with its definition.*

This is essentially the Extract Commonality refactoring in reverse. Getting the right size of chunking is an art. Too few names and the reader has to puzzle out large swathes of mathematics. Too many names, and the reader has to remember their meaning when puzzling out their uses.

- for each use of the name
  - · replace the name with its definition
  - · typecheck, to ensure the name has been used properly
- delete the definition of the name
- typecheck, to ensure no uses have been missed

5.4 Change a Cartesian Product to a Schema Product

You have a cartesian product type being used as a record, with lots of component references. *Introduce a schema product type, and use it instead.*

The components of a cartesian product are labelled by their positions, a not-very meaningful number. The components of a schema product are labelled by their names, and these can be chosen to be much more meaningful.

- create the new definition, with a new name (if the name of the new and old definitions are to be the same, first do a refactoring to rename the old definition)
- typecheck, to ensure there are no name clashes
- change each occurrence of the old name to the new one, and each occurrence of component reference from number to name (this will require more than just mechanical changes, see the example).
- typecheck, to ensure the name has been used properly
- propagate the replacement through the proofs
- delete the old definition
- typecheck, to ensure no uses have been missed

Example before: a syntactic structure is defined as a cartesian product; its semantics by meaning functions applied to the numbered components.

$$binOp == EXPR \times OP \times EXPR$$

$$\forall\, b : binOp \bullet M_e\ b = M_o\ b.2\ (M_e\ b.1, M_e\ b.3)$$

Example after: the syntactic structure is defined as a schema product; its semantics by meaning functions applied to the named components.

$$BinOp == [\ lhs, rhs : EXPR;\ op : OP\ ]$$

$$\forall\, BinOp \bullet M_e\ \theta BinOp = M_o\ op\ (M_e\ lhs, M_e\ rhs)$$

5.5 Split a State Component

You have a state component that is a product type, where each component is being referenced independently of the others. *Replace the single component with a separate component for each part of the product.*

The parts of the product type are acting independently, and so do not need to be bundled together. Structure before:

$$S == [\ c : A \times B;\ \ldots \mid \mathcal{P}(c.1);\ \mathcal{Q}(c.2);\ \ldots\ ]$$

Structure after:

$$S == [\ ca : A;\ cb : B;\ \ldots \mid \mathcal{P}(ca);\ \mathcal{Q}(cb);\ \ldots\ ]$$

5.6 Merge State Components

You have two or more state components that are constantly being referenced together. *Replace the separate components with a single product type component (cartesian or schema product).*

The separate components are acting as parts of a greater whole, and so can be combined into that whole. Structure before:

$$S == [\ ca : A;\ cb : B;\ \ldots \mid \mathcal{P}(ca, cb);\ \mathcal{Q}(ca, cb, \ldots);\ \ldots\ ]$$

Structure after (cartesian product):

$$S == [\ c : A \times B;\ \ldots \mid \mathcal{P}(c);\ \mathcal{Q}(c, \ldots);\ \ldots\ ]$$

Structure after (schema product):

$$\begin{aligned}&C == [\ ca : A;\ cb : B\ ]\\ &S == [\ c : C;\ \ldots \mid \mathcal{P}(c);\ \mathcal{Q}(c, \ldots);\ \ldots\ ]\end{aligned}$$

If some of the predicate part can also be bundled, alternative structure after (cartesian product):

$$\begin{aligned}&C == \{\ ca : A;\ cb : B \mid \mathcal{P}(ca, cb)\ \}\\ &S == [\ c : C;\ \ldots \mid \mathcal{Q}(c, \ldots);\ \ldots\ ]\end{aligned}$$

Alternative structure after (schema product):

$$\begin{aligned}&C == [\ ca : A;\ cb : B;\ \ldots \mid \mathcal{P}(ca, cb)\ ]\\ &S == [\ c : C;\ \ldots \mid \mathcal{Q}(c, \ldots);\ \ldots\ ]\end{aligned}$$

5.7 Split a State into Substates

You have a large number of state components. *Structure the state into substates.*

Smaller chunks of state may be easier to understand, and may simplify operation definitions. Components that constrain each other, and components that change together, are candidates for being in the same substate. Most state predicates are on substates, with few on the global state. Also most operations affect only a single substate, and the unchanging nature of the other substates can be captured with a few Ξ schemas, rather than a long list of unchanging state components.

- define substate schemas; typecheck
- modify the state schema to use these; typecheck
- modify each operation schema to use these, including the use of Δ and Ξ substate schemas; typecheck
- define substate initialisation operations; typecheck
- use schema calculus to define the state initialisation operation using the substate initialisations; typecheck
- modify the initialisation operation to use these; typecheck
- propagate through the proofs.

This refactoring may be followed by Expand schemas slowly, to take advantage of the opportunity to expand only the relevant substates during schema expansion steps (this may result in some extra steps being added, as the substates are expanded one by one).

Example before

$$\begin{aligned}
&S == [\ x, y : \mathbb{Z};\ a, b : \mathbb{P}\,\mathbb{Z} \mid x \in a;\ y \notin b;\ a \neq b\ ] \\
&\mathit{Op} == [\ \Delta S;\ x? : \mathbb{Z} \mid x' = x?;\ a' = a \cup x?;\ y' = y;\ b' = b\ ] \\
&\mathit{InitS} == [\ S' \mid x' = 0;\ y' = 1;\ a' = \{x'\};\ b' = \varnothing\ ]
\end{aligned}$$

Example after

$$\begin{aligned}
&\mathit{Sx} == [\ x : \mathbb{Z};\ a : \mathbb{P}\,\mathbb{Z} \mid x \in a\ ] \\
&\mathit{Sy} == [\ y : \mathbb{Z};\ b : \mathbb{P}\,\mathbb{Z} \mid y \notin b\ ] \\
&S == [\ \mathit{Sx};\ \mathit{Sy} \mid a \neq b\ ] \\
&\mathit{Op} == [\ \Delta \mathit{Sx};\ \Xi \mathit{Sy};\ x? : \mathbb{Z} \mid x' = x?;\ a' = a \cup x?\ ] \\
&\mathit{InitSx} == [\ \mathit{Sx}' \mid x' = 0;\ a' = \{x'\}\ ] \\
&\mathit{InitSy} == [\ \mathit{Sy}' \mid y' = 1;\ b' = \varnothing\ ] \\
&\mathit{InitS} == \mathit{InitSx} \land \mathit{InitSy}
\end{aligned}$$

5.8 Move a State Component between Substates

You have a substate component frequently being used along with components in another substate. *Move the component into the other substate.*

- Move the component and any predicates, as necessary.

5.9 Split an Operation into Disjuncts

You have an operation with a top level disjunct amongst its predicates. *Split the operation into two parts, one for each disjunct.*
Example before

$$Op == [\ \Delta S \mid \mathcal{P};\ \mathcal{Q} \lor \mathcal{R};\ \mathcal{S}\ ]$$

Example after

$$\begin{aligned}
&OpQ == [\ \Delta S \mid \mathcal{P};\ \mathcal{Q};\ \mathcal{S}\ ]\\
&OpR == [\ \Delta S \mid \mathcal{P};\ \mathcal{R};\ \mathcal{S}\ ]\\
&Op == OpQ \lor OpR
\end{aligned}$$

5.10 Split an Operation into a Composition

You have an operation with some existentially quantified state components that are acting as "intermediate" variables. *Split the operation into two on these components, and compose the parts.*

This is effectively expanding out the definition of schema composition [5].

Example before

$$Op == [\ \Delta S \mid \exists S_0 \bullet \mathcal{P}(\theta\ S, \theta\ S_0) \land \mathcal{Q}(\theta\ S_0, \theta\ S\ ')\ ]$$

Example after

$$\begin{aligned}
&Op1 == [\ \Delta S \mid \mathcal{P}(\theta\ S, \theta\ S\ ')\ ]\\
&Op2 == [\ \Delta S \mid \mathcal{Q}(\theta\ S, \theta\ S\ ')\ ]\\
&Op == Op1 \mathbin{⨾} Op2
\end{aligned}$$

5.11 Genericise Common Definitions

You have several similar definitions acting on different types. *Define a generic construct that captures all the behaviours.*

This is a special case of Extract Commonality.

5.12 Curry a Function

You have a function argument that is a product type, but want to apply the function to only part of that product. *Replace the product type with a curried form.*

Example before:

[5] This suggests there should be a similar refactoring for schema piping. However, since we have yet to come across a realistic case of piping in a large specification, we leave out the description.

$$\begin{array}{|l}
add : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \\
increment : \mathbb{N} \rightarrow \mathbb{N} \\
\hline
\forall\, m, n : \mathbb{N} \bullet add(m, n) = m + n \\
\forall\, n : \mathbb{N} \bullet increment\ n = add(1, n)
\end{array}$$

Example after:

$$\begin{array}{|l}
add : \mathbb{N} \nrightarrow \mathbb{N} \rightarrow \mathbb{N} \\
increment : \mathbb{N} \rightarrow \mathbb{N} \\
\hline
\forall\, m, n : \mathbb{N} \bullet add\ m\ n = m + n \\
increment = add\ 1
\end{array}$$

5.13 Reorder Product Arguments

You have two functions, one with a product range, one with a product domain, that you want to compose, but the products have their items in different orders. *Reorder one of the products to match the other.*

Example before:

$$\begin{array}{|l}
f : X \nrightarrow Y \times Z \\
g : Z \times Y \nrightarrow W \\
h : X \nrightarrow W \\
\hline
\forall\, x : X \bullet \exists\, y : Y;\ z : Z \mid (y, z) = f\ x \bullet h\ x = g(z, y)
\end{array}$$

Example after:

$$\begin{array}{|l}
f : X \nrightarrow Y \times Z \\
g : Y \times Z \nrightarrow W \\
h : X \nrightarrow W \\
\hline
h = f \mathbin{⨾} g
\end{array}$$

5.14 Reorder Curried Arguments

You have a function that you want to partially apply, but the argument order does not support that. *Reorder the arguments so that it can be partially applied.*

Example before:

$$\begin{array}{|l}
f : X \nrightarrow Y \nrightarrow Z \\
g : X \nrightarrow Z \\
\hline
\exists\, y : Y \bullet \forall\, x : X \bullet f\ x\ y = g\ x
\end{array}$$

Example after:

$$\begin{array}{|l}
f : Y \mathrel{⇸} X \mathrel{⇸} Z \\
g : X \mathrel{⇸} Z \\
\hline
\exists\, y : Y \bullet f\ y = g
\end{array}$$

5.15 Thin Early, Thin Often

The hypothesis of the goal being proved is large and clumsy. *Thin the hypothesis as early and as often as is possible.*

Thinning declarations and predicates from the hypothesis has two advantages: it keeps the hypothesis small and manageable; and it indicates more clearly what remaining properties are still required to discharge the remaining goal.

5.16 Move a Common Proof Step Before a Branch Point

You have a proof that has branched into several “cases” (for example, when splitting up a disjunct in the hypothesis), and a similar step is used in each branch (for example, cutting in a particular value). *Move the step before the case split, and do it only once.*

If the step is only “similar” in each branch, it is first necessary to apply some other refactorings to make the step the same in each branch (for example, by parameterising the proof on the case branch parameter) before this refactoring can be performed.

5.17 Turn a Common Proof Step into a Lemma

This is a special case of Extract Commonality.

Make sure the lemma is meaningful in isolation, and is not just a “textual macro”.

5.18 Expand schemas slowly

You have a large or deeply nested schema that is being expanded, or flattened, in a single step, resulting in the sudden appearance of a lot of (as yet) unnecessary terms. *Expand the schema incrementally, exposing only those terms needed at any given stage.*

This may require the prior use of the Split a state into substates or the Split an operation into disjuncts refactoring.

If the schema is used as a predicate in the goal’s hypothesis, consider duplicating the required term outside the schema, rather than expanding the schema. Use the term as required, then thin it.

6 Meaning changing “Benefactorings”

Refactoring changes structure without changing meaning. Other steps that look superficially like refactoring steps, but that *do* change the meaning of the specification, can still be used, to fix bugs, to tidy up infelicities, and to upgrade specifications in a manageable way. The same discipline as used in refactoring, of taking only small, provable, controlled steps, is used when applying these “benefactorings”.

For example, see the case of removing unnecessary abstract constraints (earlier).

Separating out refactorings (changing structure without changing meaning) from benefactorings (changing meaning without changing structure) can help provide a disciplined framework to the maintenance and upgrade process.

Some benefactorings that we have applied to LFM’s projects are noted below.

6.1 Change a Type

A functionality change may require a component’s type to be changed. For example, a simple type might need to be extended to a product type. Before adding any new functionality, make the minimal change to the type that can be done (so, add the new component throughout, but don’t use it yet).

- modify the type in the abstract model; typecheck
- modify any global property proofs about the abstract model
- modify the type in the concrete model; typecheck
- modify any global property proofs about the concrete model
- modify the retrieve relation that links the modified types if appropriate; typecheck
- modify the refinement proof as necessary

6.2 Add a State Component

The type change argument above applies in a slightly larger context to adding a state component to a Δ/Ξ spec, in order to add more functionality.

- add the component to the abstract model; typecheck
- add abstract constraints; typecheck
- modify any global property proofs about the abstract model
- add the component to the concrete model; typecheck
- add concrete constraints; typecheck
- modify any global property proofs about the concrete model
- modify the retrieve relation to link the new abstract and concrete components; typecheck

- modify the refinement proof as necessary

6.3 Add an Operation

In the usual Δ/Ξ style model, adding an operation has little effect on the specification. If the model has theorems about global properties, it is necessary to modify their proofs to ensure the properties still hold.

- add the operation to the abstract model; typecheck
- modify any global property proofs about the abstract model
- add the operation to the concrete model; typecheck
- modify any global property proofs about the concrete model
- add a new branch to the refinement proof to cover the new operation

6.4 Add a Function Argument

Similarly, for adding a further argument to a function. First add the argument, then add the constraints separately.

7 Refactoring as a proof technique

The concept of refactoring can be used when performing a (hand) proof. A proof can be considered to be the documentation of a sequence of meaning preserving transformations of a goal that improves its structure in a particular way, to the predicate *true*.

Many proof steps can be applied in either a forward or backward manner, and hence are meaning preserving. Examples include one-pointing or applying Leibniz (replacing equals by equals), schema expansions, and various eliminations. Such steps can be treated as refactorings. The presentation of the proof can be constructed by repeating the following steps:

- copy and paste the most recent form of the goal
- perform the desired refactoring by suitably editing the copy
- typecheck

For brevity of presentation, several refactorings (such as several one-pointings) may be performed on a single copy of the goal. However, to make such a presentation step comprehensible to reviewers, no single part of the goal changed by a refactoring should be further changed by a subsequent refactoring in the same step (that is, restrict multiple refactorings to distinct parts of the goal).

The names of the refactorings performed form part of the documentation of the proof step.

8 Tool Support

A code refactoring can affect an entire program. For example, changing the name of a method involves a change at every place that method is called. However, although widespread, such a change is shallow. Deeper code refactorings tend to be contained to small regions of code. It is by having *small* changes (either small in depth, or small in breadth) that code refactoring is manageable.

Formal specification and proof refactorings tend to have more widespread effects, because of the impact of a specification change on the proofs. For example, adding a component or predicate to the state may involve adding it to the proof of every operation. Again, the changes tend to be shallow: the addition will probably have little or no effect on most operations. However, every proof needs to be checked, and this can be tedious without suitable tool support.

Tools are currently being built to support code refactorings. For example, the latest version of JBuilder includes a refactoring wizard. Similar tool support (within a proof tool) for formal refactoring would be a benefit.

Such a tool could support common refactoring patterns such as

- partition a state into substates
- move a component between substates
- split a complex operation into disjuncts
- split a complex operation into a sequential composition
- extract a common sub-proof as a lemma

Also important is support for pushing a relatively small change through the proof. This requires facilities for parameterising tactics for machine-generated proofs.

9 Unifying refactoring and proof

We have seen that refactoring is a meaning preserving transformation on a specification, moving it closer to a well-structured pattern, while proof is a meaning preserving transformation on a goal, moving it closer to *true*. In this section we use this analogy to explore how concepts from proofs and proof tools could be extended to build refactoring tools and refactoring pattern languages.

9.1 Tactics for refactoring

Proof tools tend to rely on a small core set of elementary inference steps, with the desired soundness and coverage properties, combined together to produce usable inferences using some tactic language.

The analogy suggests that a refactoring tool should rely on a small set of suitable refactorings (somewhat like the ones presented above), and then combine them to produce large scale refactorings (several small refactorings resulting in the desired pattern) using some kind of refactoring combination language, incorporating composition, choice, iteration, and powerful pattern matching.

9.2 Refactoring entire documents

A proof tool transforms a single goal; a refactoring transforms an entire specification and proof. However, the specification and proof document can be considered to be a single entity, comprised of three parts: the *definitions* (what is usually called the specification), the *theorem* (we can assume there is only one, without loss of generality, simply by conjoining any separate ones) and the *tactic*, the specification of how to apply elementary inference rules to the theorem in the context of the declarations, in order to generate *true*. (The third part of this document is usually presented to a human reader as the outline proof generated by the tactic, rather than the tactic itself.)

Hence a refactoring tool should be able to transform declarations, theorems, and tactics. Transforming a tactic might be done to improve the structure of a proof. It might also be done to genericise a proof, so that the tactic can still prove the theorem in the context of a refactored specification. This implies that as well as patterns and refactorings for specifications, we want patterns and refactorings for tactics.

9.3 Refactoring for refinements

The small meaning preserving steps, combined into the higher level transformational steps, bridge the gap between the two common styles of proof. *Meaning preserving transforms* (for example, refinement calculi such as [8]) have the advantage of being small easily proved steps, but the disadvantage of being hard to guide towards the desired end point. *Posit and prove*, the conventional technique used for Z refinements, has the advantage of being able to find the desired end point (it is simply posited), but hard to prove, because the step is so large.

Refinement can be considered to be a 'vertical' refactoring (moving between an abstract and a concrete representation of the 'same' specification, possibly changing representation language), since it preserves the meaning of the abstract specification in some sense. Refinement tends to be cast as a 'posit and prove' style: the concrete specification is posited, then a large proof needs to be performed. The concepts of refactoring, with a suitable refinement refactoring pattern language, should be exploited to build a method that combines the advantage of small, provable steps, with the advantage of reaching a desired end point.

9.4 Proof, refactoring, refinement unified

In summary:

- An inference step transforms a goal into another goal with the same meaning. A tactic language combines these small steps to produce large reasoning steps intended to move the goal closer to *true*.
- A refactoring transforms a specification (declaration, theorem and tactic) into another specification with the same meaning. A refactoring language combines these small steps to produce large transformational steps intended to move the specification closer to realising a pattern.
- A refinement transforms an abstract specification into a concrete specification with the same meaning. A refinement refactoring language could combine small refinement steps to produce large refinements intended to move the abstract specification closer to an implementation.

That is, all three areas can be viewed as special cases of *meaning preserving transforms*. Hence techniques currently used in one area may well be applicable in the other two. We intend to investigate how proof tools in particular could provide the basis for refinement tools and refactoring tools.

10 Conclusions

The code changing discipline of refactoring has been applied equally fruitfully to specification and proof maintenance in a number of large commercial formal specification and refinement projects.

The same refactoring discipline can be applied to specification upgrades. Systematically combining refactorings and benefactorings provides a structure for the specification maintenance and upgrade process.

Refactoring can be used as part of the proof process itself. The analogy between proof and refactoring can be extended, to suggest a route to building a refactoring toolset, and a refinement toolset.

References

[1] Barden, R., S. Stepney and D. Cooper, "Z in Practice," BCS Practitioners Series, Prentice Hall, 1994.

[2] Beck, K., "Extreme Programming Explained," Addison-Wesley, 2000.

[3] Dyer, M., "The Cleanroom Approach to Quality Software Development," Wiley, 1992.

[4] Fowler, M., "Refactoring: improving the design of existing code," Addison-Wesley, 1999.

[5] France, R. B. (2002), (private communication).

[6] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns," Addison-Wesley, 1995.

[7] ISO/IEC 13568, "Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics: International Standard," (2002).

[8] Morgan, C. C., "Programming from Specifications," Prentice Hall, 1990.

[9] Stepney, S., *Incremental development of a high integrity compiler: experience from an industrial development*, in: *Third IEEE High-Assurance Systems Engineering Symposium (HASE'98), Washington DC*, 1998.

[10] Stepney, S., *A tale of two proofs*, in: *BCS-FACS Third Northern Formal Methods Workshop, Ilkley* (1998).

[11] Stepney, S. and D. Cooper, *Formal methods for industrial products*, in: J. P. Bowen, S. Dunne, A. Galloway and S. King, editors, *ZB2000: First International Conference of B and Z Users, York, August 2000*, Lecture Notes in Computer Science **1878** (2000).

[12] Stepney, S., D. Cooper and J. Woodcock, *An electronic purse: Specification, refinement, and proof*, Technical Monograph PRG-126, Programming Research Group, Oxford University Computing Laboratory (2000).

[13] Stepney, S. and F. Polack, *A pattern language for Z*, (in preparation).