

**SOFTWARE VERIFICATION RESEARCH CENTRE**  
**DEPARTMENT OF COMPUTER SCIENCE**  
**THE UNIVERSITY OF QUEENSLAND**

Queensland 4072  
Australia

**TECHNICAL REPORT**

No. 94-43

**Requirements for a  
Program Refinement Engine**

**David Carrington, Ian Hayes,  
Ray Nickson, Geoffrey Watson and  
Jim Welsh**

**July 1995**

**Phone: +61 7 365 1003**

**Fax: +61 7 365 1533**

**Note:** Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

# Requirements for a Program Refinement Engine

David Carrington      Ian Hayes      Ray Nickson  
Geoffrey Watson      Jim Welsh

## Abstract

Refinement is a mathematically-based technique for developing a program from an abstract specification so that the program satisfies the specification. The aim of the Program Refinement Tool project is to develop a generic refinement tool suitable for supporting a methodology for the interactive development of programs based on the refinement calculus. This report summarizes our investigation into an appropriate engine to use for the refinement calculator and theorem prover in this tool.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Refinement Engine</b>	<b>3</b>
2.1	Structure of Refinement Rules . . . . .	3
2.2	Applicability Conditions . . . . .	5
2.3	Parameters . . . . .	5
2.4	Monotonicity . . . . .	7
2.5	Proving Refinement Rules . . . . .	8
2.6	Schematic Developments . . . . .	8
2.7	Customizing the Language and Logic . . . . .	9
2.8	Heuristic Content of Refinement Rules . . . . .	10
2.9	Context . . . . .	10
2.10	Other Issues . . . . .	12

<b>3</b>	<b>Proof Engine</b>	<b>13</b>
3.1	Genericity . . . . .	14
3.2	Proof Style . . . . .	15
3.3	Context . . . . .	16
<b>4</b>	<b>Combining the Transformation and Proof Engines</b>	<b>16</b>
<b>5</b>	<b>Conclusions</b>	<b>17</b>

## 1 Introduction

Refinement is a mathematically-based technique for developing a program from an abstract specification so that the program satisfies the specification. The refinement approach starts with a formal, abstract specification of the required product and transforms this via a sequence of small steps into a form suitable for execution.

The refinement calculus, developed by Back, Morgan and Morris, [Bac78, Bac88, MR87, Mor94, Mor87] provides a uniform method for deriving programs from specifications. The calculus extends a programming language with abstract specification constructs, to give a *wide-spectrum language* [B<sup>+</sup>85, B<sup>+</sup>87] that can express specifications, programs, and every step in between. The semantics of the wide-spectrum language is defined using Dijkstra's weakest preconditions [Dij76]. The weakest precondition is also used to define an ordering ( $\sqsubseteq$ ) on constructs in the wide-spectrum language. If *Spec* is a specification and *Prog* is a program, then  $Spec \sqsubseteq Prog$  if and only if *Prog* is an implementation of *Spec*. More generally, the ordering allows one specification or command (*S*) to be replaced by another (*T*): correctness is maintained as long as  $S \sqsubseteq T$ . Typically, *T* is derived from *S* by applying an instance of a *refinement rule*, which guarantees the appropriate relationship between *S* and *T*, subject to the validity of some *applicability condition*.

The aim of this ARC-funded project, based in the Software Verification Research Centre at the University of Queensland, is to develop a generic refinement tool suitable for supporting a methodology for the interactive

development of programs based on the refinement calculus. A particular focus of the project is the close coupling of a generic language-based editor, instantiated to support the refinement calculus, with an engine for applying refinement rules and calculating proof obligations and a theorem prover with which to discharge those obligations. This report summarizes our investigation into an appropriate engine to use for the ‘back-end’ refinement calculator and theorem prover. It does not cover the details of our requirements from the tool’s user interface, except where those requirements directly affect the functions we expect from the back-end engine.

## 2 Refinement Engine

The first requirement for a tool supporting program refinement is a way to represent specifications, programs and refinement rules, and an engine for applying rules. We shall assume a calculus similar to Morgan’s [Mor94], where the specification and programming languages are unified, though we shall defer discussion of the details of that wide-spectrum language, henceforth referred to as *WSL*. We also assume that valid transformations are defined in terms of some *refinement relation* over commands in *WSL*. To allow stepwise development, the refinement relation must be reflexive and transitive (a *preorder*). Refinement then proceeds by expressing the specification as a command in *WSL*, and transforming it (by a series of valid applications of refinement rules) into another (complex) command in *WSL* that is considered executable *code*. The exact definition of what subset of *WSL* constitutes acceptable code can vary from one derivation to the next. Each application of a refinement rule transforms some command (the *subject*) into another command (the *result*), with the obligation to show that some *applicability conditions* hold.

### 2.1 Structure of Refinement Rules

A refinement rule is a schema whose instances are theorems in the refinement calculus. Each rule defines:

1. the set of subject commands that the rule can be applied to;

2. the set of achievable result commands;
3. the applicability conditions, which may involve semantic and syntactic constraints on the subject and result; and
4. the relationship between particular subjects, results and applicability conditions.

It seems natural to limit the allowable subjects for some rule by writing *patterns* in WSL; that is, each refinement rule has as its *subject pattern* a skeletal command in which some components are *metavariables* (possibly with *meta-types*) standing for the missing constructs. Then, the result of many refinement rules can be expressed by writing a *result pattern* using those same metavariables; to apply the rule, one instantiates the metavariables in the subject pattern to match the subject, and then replaces the subject by the result pattern. For example, one might express the refinement rule that transforms a specification statement  $x: [true / x = a]$  (in which  $x$  is a program variable and  $a$  is a constant) into an assignment  $x := a$  by writing a rule whose subject pattern is  $X: [true / X = A]$  (in which  $X$  is a metavariable standing for an arbitrary program variable and  $A$  is a metavariable standing for an arbitrary constant) and whose result pattern is  $X := A$  (there is no applicability condition, apart from the implicit requirement that the bindings for  $X$  and  $A$  are of the correct meta-types).

In Morgan's WSL, the precondition and postcondition of a specification statement and the guards in an **if** or **do** command are formulas in a first-order theory that includes predicate and function symbols corresponding to the predicates and functions in our application domain; therefore, we also require a way to define that logic. Terms in that logic are used to denote expressions in assignment commands and other constructs. As with the constructs of WSL itself, we shall defer discussion of how the logic is specified, and for now assume there is some notation for terms and formulas within it. We shall make one further simplifying assumption now: that the logic used in 'code' constructs is the same as (or a subset of) that used in 'specification' constructs (so we have to deal with only a single logic). For example, we usually allow arbitrary formulas to appear as guards in WSL, but only those guards that correspond to the Boolean expressions of real programming languages are allowed in code.

## 2.2 Applicability Conditions

We also require a logic for expressing the applicability conditions of refinement rules. Many such conditions can be expressed using the same logic as in WSL, though other constraints on refinement rules are syntactic in nature (such as ‘the identifiers on the left-hand side of a multiple assignment command are distinct’). For now, we shall make the simplifying assumption that there is a single logic rich enough to express the formulas and terms needed in WSL and (at least) those applicability conditions that are semantic in nature. Now, we can write a conditional refinement rule whose subject and result patterns contain metavariables; its applicability condition too may contain those metavariables, so the obligation when applying that rule to some subject is to show that the appropriately instantiated applicability condition is a valid formula of the underlying logic. For example, we can generalize the simple refinement rule for assignment above to one in which the subject pattern is  $X: [Pre / Post \wedge X = A]$ , the result pattern is  $X := A$ , and the applicability condition pattern is  $Pre \Rightarrow Post[X \setminus A]$ <sup>1</sup>. We shall write refinement rules using this layout:

$$\frac{\textit{Applicability condition pattern}}{\textit{Subject pattern} \quad \sqsubseteq \quad \textit{Result pattern}}$$

Where there are syntactic constraints on the rule that cannot be expressed by metavariables or in the applicability condition logic<sup>2</sup>, we shall write the constraints (in English) to the right of the rule and call them *side conditions*.

## 2.3 Parameters

In the (rather contrived) refinement rules given so far, the result has depended in a fixed, quite simple way on the subject; given a rule and a subject, there is only one possible result. In general, we want to be able to express refinement rules whose results can be affected by other factors.

---

<sup>1</sup>We use  $Pre \Rightarrow Post$  as in [Mor94] to stand for a relation between formulas  $Pre$  and  $Post$  that holds when  $Pre \Rightarrow Post$  is valid for every possible binding of program variables. The notation  $Post[X \setminus A]$  stands for the formula resulting from replacing all free occurrences in  $Post$  of the program variable  $X$  by the expression (in this case, constant)  $A$ .

<sup>2</sup>In practice, we hope to use a logic that can express all such constraints.

We shall do this by allowing the result pattern (and applicability condition) to contain metavariables that do not appear in the subject pattern. For example, we can generalize the assignment rule to permit introduction of an assignment command  $X := E$  where  $X = E$  is *not* an explicit conjunct of the postcondition:

$$\frac{Pre \Rightarrow Post[X \setminus E]}{X: [Pre / Post] \quad \sqsubseteq \quad X := E}$$

In this rule, the expression  $E$  (we have also generalized the former constant  $A$ ) appears in the applicability condition and result patterns, but not in the subject pattern. We shall view most such extra metavariables as *parameters*, since their instantiation is not determined by the rule and subject alone.

We shall consider two mechanisms for instantiating parameters:

1. The parameters must be explicitly instantiated before applying a rule.
2. Parameters need not be explicitly instantiated before applying a rule.

If the first option is taken, it is useful to list the parameters along with the rule, perhaps with annotations specifying how instantiations are to be obtained. Then, parameters can be viewed as formal arguments of the refinement rule, just like the formal parameters of a procedure in a traditional procedural programming language. We thus expect an application of a rule to supply actual parameters that bind the metavariables, so the result and applicability condition patterns are fully instantiated when the rule is applied.

The second option is perhaps more interesting; the applicability condition and result of a rule application will in general contain metavariables, and the result may itself be the subject of some later refinement, so even the subject pattern of that later refinement need not be fully instantiated. Our refinement engine (and our proof assistant) must thus be able to deal with *schematic* developments, and some applicability conditions will need to be deferred until the metavariables within them are instantiated. We shall see in Section 2.6 how such a mechanism may be useful in other ways. Eventually, we shall normally want all metavariables to be instantiated (so that the resulting program is code). Therefore, even with this option there must be

a mechanism for supplying instantiations, though we have the flexibility to do so when appropriate. For example, it is possible that some collection of partly instantiated applicability conditions can be viewed as a set of *constraints* on the metavariables within them for which there are only a few solutions [NG94], and some constraint sets may be automatically solvable.

To choose between the options for instantiating parameters, we should consider how we expect a refinement tool to be used. It is likely that, most often, the user knows precisely what instance of a refinement rule is desirable in some circumstance; this suggests that the first option (with explicit arguments) is appropriate. On the other hand, the ability to defer parameter bindings is occasionally useful, so we should not rule out the possibility. We shall aim to support both styles of operation, providing a means for the user to supply an instantiation with a rule application, or defer it until later. To achieve this, we shall give refinement rules explicit argument lists, so the user will normally furnish bindings as actual arguments when applying a rule, and provide some mechanism for choosing *not* to furnish actual arguments for some formals.

## 2.4 Monotonicity

We have assumed that the refinement relation is reflexive and transitive (a *preorder*), so that a sequence of (zero or more) valid refinement rule applications is a valid refinement. An essential feature of the *stepwise refinement* method [Wir71], of which the refinement calculus is a formalization, is the ability to replace subcommands by their refinements, yet retain a valid refinement of the enclosing commands. To allow subcommand replacement, it is also necessary that the constructs of WSL are *monotonic* with respect to the refinement relation. This means that if  $S \sqsubseteq T$  and  $C[S]$  is a command containing  $S$  as a subcommand, then  $C[S] \sqsubseteq C[T]$ . This is the case for all of the constructors in Morgan's refinement calculus language, though there are useful constructs [Nel89] for which it does not hold. We shall (initially at least) assume that all constructs are monotonic. We shall also consider the replacement of components of command constructs that are not themselves commands (like the guards of an **if** command), and see how that affects the meaning of the enclosing command.

## 2.5 Proving Refinement Rules

It is possible to support the refinement calculus with a tool that includes a fixed set of refinement rules. One of the goals of this project is to develop a *generic* tool, in which refinement rules can be tailored to a particular application. One way to do this is to allow the tool's user to supply refinement rules that have been justified by hand (or, at least, outside the tool); the refinement rules then act as the definition of the semantics of WSL and of the refinement relation itself (of course, for a particular refinement system to be useful, the rules should correspond to some sensible intuition about the underlying semantics). Then, each particular refinement theory contains a fixed set of rules (perhaps including rules that can be derived from them: see Section 2.6).

A more interesting possibility is to provide support for proving refinement rules *within* the tool. With that approach, we can be much more confident that the rules in the system are valid; this is useful when postulating new rules, but also helps to avoid mistakes when encoding existing, trusted rules in the system. To support the proving of refinement rules from first principles, it will be necessary to include in the tool a specification of the semantics of WSL and a definition of the meaning of refinement. This approach demands much greater flexibility in the theorem prover, as we shall see in Section 3.1. Another approach is to allow new rules to be proved by appealing to existing rules (or, better still, to *derive* the new rules from the old); this is the subject of the next section.

## 2.6 Schematic Developments

It is useful to be able to handle schematic developments: those in which some parts of the initial specification, and/or some of the parameters of rules used, have not been fully instantiated. Allowing such developments brings two benefits:

1. The ability to defer design decisions in a refinement by leaving some rule parameters uninstantiated, as discussed above.
2. The ability to prove *derived* refinement rules.

This second capability is discussed now. To derive a new refinement from existing rules, we can start a refinement with a specification that contains metavariables. Then, we apply refinement rules to it to construct the result we want from the derived rule, possibly leaving some of the parameters for those constituent rules uninstantiated. If parts of the applicability conditions of the constituent rules cannot be discharged (generally because they are contingent on the bindings of metavariables within them), they are collected together. When we have applied all the desired constituent rules, we have a result that contains metavariables, and can form the new derived rule by treating the original subject as a subject pattern, the result as a result pattern, and the conjunction of the collected applicability conditions as an applicability condition pattern; any metavariables occurring in the applicability condition or result patterns become the parameters of the new rule. We may wish to tidy up the applicability conditions by providing some stronger condition. This is essentially the approach taken in the *Red* refinement editor [JMG, Vic90], and is quite effective.

## 2.7 Customizing the Language and Logic

Our aim is to build a tool that is *generic*: that is, it should be possible to customize the tool to suit different particular applications. At the least, we should like to be able to define new mathematical formalisms for use within terms and formulas in our specifications, and new programming data types for our code. This means that we shall need the ability to extend those parts of the syntax of WSL and the semantics of the logic in which we express and discharge proof obligations. We shall probably also want to define new refinement rules specialized for the new data objects; if we take the approach of proving such new rules, the semantics of WSL will thus need to be extensible.

A deeper genericity is possible if we allow completely novel WSL constructs to be defined by the tool's user. Once again, we need to be able to extend the syntax of WSL, to add refinement rules involving the new constructs, and (if we are to prove those rules) to define the semantics of the extended WSL. A secondary goal of the project is to investigate the representation of *context* within the refinement calculus, and how it can be managed by a tool. To do that effectively, we shall want the ability to define new WSL constructs that introduce context, and probably variant refinement relations (such as the indexed refinement relations of [MV90, VM94b, Nic94]).

## 2.8 Heuristic Content of Refinement Rules

A refinement rule can be viewed formally as a theorem in some theory of refinement. A rule with applicability condition  $C$ , subject  $S$  and result  $T$ , involving the metavariables  $V$ , can be seen as the theorem  $\forall V \bullet C \Rightarrow (S \text{ R } T)$ , where  $\text{R}$  may be refinement ( $\sqsubseteq$ ) or refinement equivalence ( $\sqsubseteq\!\!\sqsubseteq$ , where  $S \sqsubseteq\!\!\sqsubseteq T$  iff  $(S \sqsubseteq T)$  and  $(T \sqsubseteq S)$ ). There are other *heuristic* aspects to a refinement rule, though: how its parameters are intended to be instantiated; the kind of situation in which it is most likely to be useful; and (for rules with conclusion  $S \sqsubseteq\!\!\sqsubseteq T$ ) the direction in which the rule is most likely to be applied. We shall probably want the ability to express some of this heuristic information along with our rules. Direction is easy: we can always write the conclusions with the intended subject before the intended result. For parameter bindings, we can annotate each rule with information about the intended bindings of parameters; these annotations might be expressed using *meta-types* (such as ‘fresh-identifier’ or ‘Boolean-expression’), or perhaps by procedural code instructing the transformation engine how to obtain suitable bindings (such as ‘supply the postcondition and allow the user to edit it to construct an invariant’). To provide information about likely applicability, we can add annotations expressing data like precedence, context, etc.; these annotations might be used to rank applicable rules or provide sophisticated browsing facilities.

## 2.9 Context

An important goal of our project is to investigate the use and management of *context* during refinements. Context is needed to determine the applicability of some refinements; as such, it must be available to the user and communicated to the engine used in discharging proof obligations. On the other hand, it is desirable to avoid overwhelming the user with too much context information, since most applicability conditions will refer to only a small amount of the possible context. Context must be carefully managed so that its presentation to the user (and to the proof engine) is as unobtrusive as possible.

One of the simplest aspects of context is *type* information about the variables in a WSL program. This type information may be needed when proving

refinement obligations involving program variables: for example, a specification statement  $x: [x > 0 / x \geq 1]$  is refined by **skip** if  $x : \mathbb{N}$ , but not if  $x : \mathbb{R}$  (the obligation is  $x > 0 \Rightarrow x \geq 1$ ). A generalization of types has been defined by Morgan and Vickers [MV90]: this generalization allows the specification of arbitrary relations (*invariants*) over program variables that are maintained by type-correct code. A simple modification of the refinement calculus is used to formalize the treatment of invariants: the applicability condition of every refinement rule has an antecedent that includes the conjunction of all invariants in scope.

A second form of context that is familiar to programmers is the set of *procedures* available to be called at some point in the program development. Morgan [Mor94] uses a scoped procedure declaration whose semantics are derived using the copy rule; the issues of parameters and of recursion are treated independently. The semantics of such procedure declarations and their interactions with invariants are explored in [VM94b].

In formal specifications and developments, we need to consider other kinds of context too. We shall almost certainly want to make *local definitions* to introduce abbreviations and other names for formulas and terms. We also need to consider carefully the theory in which the semantics of our specification and programming data types are expressed (and in which proof obligations should be discharged). Both of these concepts can be formally modelled using contexts [Nic94].

A final form of context that we can usefully model is motivated by the technique of writing *assertions* within programs. Such assertions are formulas that we intend to be true at some point in the execution of a program (unlike invariants, which are true throughout their scope). Assertions form the basis of Floyd's [Flo67] and Hoare's [Hoa69] approaches to program verification. We shall assume implicit assertions before and after every command in a WSL program: for example, after a specification statement  $S \triangleq w: [Pre / Post]$  we can assume (at least) the assertion *Post*, since *S* (or any refinement thereof) must establish that, miraculously if necessary. This assertion may be useful in a refinement of a command *T* that immediately follows *S* in a sequential composition, and by treating it as part of *T*'s context (its *implicit precondition*) we can better structure the proof obligations for the refinements of *T*.

## 2.10 Other Issues

Numerous other issues will arise in the design of the fine details of the refinement rule representation and means by which the refinement engine applies them. We shall not be able to fully address all those issues in this project; in many cases, we shall make simplifying assumptions to avoid complications in this prototype tool. Some of the issues we can foresee are:

**Organizing refinement rules:** A practical tool will contain numerous refinement rules: some of those rules will be expressed and proved from first principles; others may be derived beforehand from other rules; still others may be expressed as tactics for combining other rules. Many of those rules will be specialized to particular kinds of problems or algorithm design strategies. Organizing such a large (and open-ended) library of rules in a useful way, and providing efficient facilities for browsing and selecting rules, will be an important issue in the design of a usable tool.

**Matching of rules:** The instantiation of a refinement rule (actually a rule schema) to some particular problem involves matching. In the simplest case, that matching involves a simple, deterministic choice of values to which to bind the subject variables, parameters and other metavariables in the rule to yield a ground theorem that can be applied. If we allow schematic developments (as in Section 2.6), the problem may be generalized to a unification problem (since we might choose to instantiate existing metavariables in the program being refined). As we consider richer wide-spectrum languages, the need for more sophisticated matching and unification algorithms will increase: for example, if subject patterns involve associative and commutative operators, we might want matching and unification algorithms that avoid the need for explicit rearrangement of arguments to match a fixed rule.

**Calculating results:** A simple pattern-matching language for expressing rules limits us to rather simple relationships between subjects, parameters and results. We may wish to generalize those relationships. For example, we might choose to express the guarded command set of an **if** command as a list; then, we would like a rule that takes a parameter list of arbitrary length (the guards) and yields an **if** command with the appropriate number of branches. This will require list-building

facilities to express how the subject and parameters combine to yield the instantiated applicability condition and result.

**Other refinement relations:** We shall initially assume the simple refinement relation of [MR87], defined using  $wp$ . Later, it may be useful to use different refinement relations: to handle different forms of context, for example, we might want to index that relation in various ways. Another example of a useful variant refinement relation is Nelson's relation [Nel89] defined in terms of both  $wp$  and  $wlp$ ; this relation permits arbitrary refinement inside a guarded command set.

**Tactics:** It is useful to package commonly-occurring sequences of refinement rules as *tactics*: parameterized fragments of a derivation that can be applied as a unit [GNU92, VM94a]. Tactics are a common mechanism for adding usability to theorem provers (see e.g. [Nip89b, GW91, Fel93, MGW]) that can often express more powerful concepts than can derived rules. For example, we might write a tactic that can select an **if**-introduction rule in which the number of guarded commands is determined as the tactic is applied. This is an alternative to the approach of permitting more powerful matching and calculation within a single rule.

### 3 Proof Engine

Our second requirement is for an engine to discharge proof obligations. This engine needs to be able to handle any applicability condition generated by the transformation engine; given the genericity of the latter, it is clearly essential that the logic underlying the proof engine can be easily customized to suit each particular application domain. It will also be useful if the proof engine uses a proof method that is similar to refinement, reducing the cognitive load on the user when moving from program transformation to proof. Finally, an important feature of the proofs that arise in refinement is that they often depend on context (including type information about local variables): it will be useful if that context can be efficiently managed by the prover.

### 3.1 Genericity

Many early theorem provers were designed to work with only a single, fixed logic. For the kind of proofs we want from our generic refinement tool, it is important that at least some details of the logic can be varied to suit the application domain of the programs being developed. Developers of automated theorem provers over the last decade have tended to favour parameterizing the logic used by their provers [Pau86, FGT92].

We need at least to be able to extend the logic to define the function and predicate symbols of our application theory. Our refinement engine will be generic enough to allow the use of such symbols (together with control over concrete syntax, such as the ability to declare infix operators); this means that proof obligations containing symbols for application-specific data types will arise. The prover must be able to treat these obligations in a sensible way; most usefully, it should allow its users to (easily) build theories containing definitions, axioms and inference rules for manipulating terms and formulas involving novel data types.

We should also like to be able to use the prover to justify proposed new refinement rules from first principles (Section 2.5). In this case, the prover will need some ability to reason about commands themselves, since our usual definition of refinement is in terms of weakest preconditions, and the refinement rules we want to express generally involve metavariables (so weakest preconditions cannot in general be calculated explicitly). Such reasoning cannot be done easily in a prover that is based on first-order logic and extensions thereof<sup>3</sup>, since  $wp$  is essentially a modal operator [CHN<sup>+</sup>94]. The need to reason about commands also arises if we have refinement rules whose conditions are themselves refinements: such rules are sometimes useful (see e.g. [NG94]).

If our theorem proving logic can model commands, we can express many rule applicability conditions that are normally considered syntactic within that logic. For example, to model the semantics of assignment commands within a logic, program identifiers will need to be objects in the domain of discourse, distinct from the values those variables take on in different

---

<sup>3</sup>Actually, higher order logic is sufficient [BvW90], and that can be defined in first-order logic and set theory (by modelling functions as their graphs); but reasoning in such a theory using a tool whose basis is in first-order logic would almost certainly be cumbersome.

computational states. With such a logic, it is an easy matter to express applicability conditions like ‘all identifiers on the Left-Hand Side are distinct’ on a rule that introduces an assignment command. If we can express all applicability conditions within our theorem proving logic, our transformation engine need not deal with side-conditions at all.

### 3.2 Proof Style

There are many styles of proof: *resolution*, as used in Otter [McC90]; *natural deduction* as used in Mural [JLM91] and in LCF [GMW79] and its descendants; proof theories tailored to various *constructive logics*, such as the constructive type theory used in NuPRL [CAB<sup>+</sup>86], and induction-based strategies such as those used in the Boyer-Moore prover [BM79] and Gypsy [Goo84]. A proof style that is particularly effective uses *term rewriting* [HKLR92]. In term rewriting approaches, one constructs a sequence of formal objects that starts with the formula to be proved, in which every object is related to the one before it by some validity-preserving relation (typically, but not essentially, an equivalence relation); if the sequence terminates in a formula known to be valid, the formula itself was valid. Usually, successive elements of the sequence are calculated by applying conditional *rewriting rules* to previous elements. Generally, the rewriting relation and term structure of the object logic are such that subterm replacement is valid (i.e. the constructs of the object language are monotonic with respect to the relation). This style of proof forms the basis of the provers Affirm [EM80] and Eves [KPS<sup>+</sup>93], and is a major proof style used in Isabelle [Nip89a]. Finally, term rewriting is at the foundations of the *window inference* proof technique [RS93], upon which the theorem prover Ergo [UW94] is based.

Perhaps the greatest benefit in using a prover based on term rewriting is the similarity of that process to refinement. Both activities can be seen in a similar way: they involve selection of a subject component to transform, the choice and instantiation of a transformation rule, the discharging of its applicability condition (perhaps by a recursive application of the same procedure), and the replacement of the subject by the result of transformation. Both activities depend on the same properties of the relations and objects concerned: reflexivity (so that ‘no change’ is always an option); transitivity (so transformations can be sequentially composed); and monotonicity of constructors (allowing subterm replacement). Thus, it is feasible to support

both activities with a similar interface, reducing the disruption of the user's train of thought when moving between refinement and proof activities.

### 3.3 Context

We have already discussed (Section 2.9) how refinement rules will need to refer to various kinds of information about the context within which they are being applied. Generally, the proof of an applicability condition of a refinement rule will depend on some of that context: an example involving type context appeared in Section 2.9. It is possible to design refinement rules in a way that includes all the relevant context information in every applicability condition; this is the approach of [MV90], where almost every applicability condition includes the conjunction of the types and invariants in scope as an explicit antecedent. This approach is undesirable in a typical top-down development, since many refinements occur in the same context, so repeating explicitly it in every obligation to be discharged by the prover becomes wasteful. A better approach involves close integration between the transformation engine and prover, so that each maintains its own notion of the current context as the user navigates the derivation, and there is no need to repeatedly communicate it, or indeed to explicitly refer to it in refinement rules.

Window inference is a good method of managing some simple kinds of context. The technique is to maintain a list of *hypotheses* that may be taken to hold while the proof is *focused* on some particular component; any transformation of that component may assume all the hypotheses, and they are available to discharge applicability conditions of transformation rules. Normally, window inference hypotheses are formulas; we are considering extensions to window inference that allow context information such as typed variable declarations to be included directly as hypotheses.

## 4 Combining the Transformation and Proof Engines

We have already justified the use of a term rewriting style of prover, so that refinement and proof become similar activities. This argument can be

taken further: if the *same tool* is used for both activities, it is much easier to present a similar interface. Notwithstanding this, the way we want to manage refinements is slightly different from the way we want to manage proofs; an important difference is that the *goal* during proof activity is clear (namely, ‘transform the obligation to *true*’), whereas in refinement it is usually considerably less precise (‘transform the specification to code’). Thus, goal-directed strategies are likely to be used much more frequently during proof than during refinement. Still, there are enough similarities between the two activities that there is much to be gained by providing a consistent interface: if window inference is used both to manage program context and as the proof technique, for example, we get a clear benefit by combining the two components of the tool.

Use of a theorem prover as the transformation engine for refinement increases our confidence in the overall correctness of developments, since the same formalism is used for stating refinement rules, their applicability conditions, and the rules used in discharging them. Furthermore, it is possible for the refinement rules themselves to be theorems that are proved with that theorem prover; then, we have a full justification within a single formal system for every development (based on an embedding of the semantics of the wide-spectrum language in that formal system). There is no reliance on the correctness of (possibly complex) interactions between transformation engine, proof obligation generator and prover, since they are the same entity.

## 5 Conclusions

We have set out some of our requirements for representing and applying refinement rules and for discharging proof obligations. These requirements led us to choose an interactive theorem prover based on term rewriting as our engine for discharging proofs, and to embed refinement in that same prover so it can also serve as our engine for applying refinements.

Ergo [RW93, UW94] is the Software Verification Research Centre’s theorem prover. It is implemented in Qu-Prolog [RC93]. Proofs are done by rewriting, using window inference [RS93] to manage contexts. Rewrite rules are deduced from *axioms*, *theorems* and *definitions*, which are organized into theories. Users can prove their own theorems from existing axioms and

theorems, or can add theories that define new notation and give axioms or definitions to suitably constrain its interpretations.

Ergo was designed with an intention to support a modal logic known as *functional logic* [SRH94]. Ergo's base theory is a subset of intuitionistic logic upon which classical, constructive and modal logics can be built. In particular, Ergo does not depend on classical equality: *substitution of equals for equals* is not essential, though Ergo does assume  $=$  is an equivalence relation (so that it can use it to express definitions).

Ergo is ideally suited for use in our project, because of its advanced facilities for theory development, its rewrite-based proof style, its handling of schematic proofs, its management of context through window inference, and its handling of non-classical logics. We have been able to construct a usable refinement theory in Ergo based on weakest preconditions and a simple modal logic (in which  $wp$  is a modal operator: compare [Gol82]), without the need to build a deep embedding of refinement and predicate transformer semantics involving higher-order logic. This refinement theory will form the foundation upon which our refinement tool is built. The details of the embedding of the refinement calculus in Ergo can be found in [CHN<sup>+</sup>94].

## References

- [B<sup>+</sup>85] F. L. Bauer et al. *The Munich Project CIP, Volume I*, volume 183 of Lecture Notes in Computer Science. Springer-Verlag, 1985.
- [B<sup>+</sup>87] F. L. Bauer et al. *The Munich Project CIP, Volume II*, volume 292 of Lecture Notes in Computer Science. Springer-Verlag, 1987.
- [Bac78] R. J. R. Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, 1978.
- [Bac88] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.
- [BM79] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.
- [BvW90] R. J. R. Back and J. von Wright. Refinement concepts formalised in Higher Order Logic. *Formal Aspects of Computing*, 2:247–272, 1990.

- [CAB<sup>+</sup>86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.
- [CHN<sup>+</sup>94] David Carrington, Ian Hayes, Ray Nickson, Geoffrey Watson and Jim Welsh. Refinement in Ergo. Technical Report 94-44, Software Verification Research Centre, The University of Queensland, 1994.
- [Dij76] E. W. Dijkstra. *A Discipline of Programming*. Academic Press, 1976.
- [EM80] R. Erickson and D. Musser. The Affirm theorem prover: Proof forests and management of large proofs. In Wolfgang Bibel and Robert Kowalski, editors, *5th conference on automated deduction*, volume 87 of Lecture Notes in Computer Science. Springer-Verlag, 1980.
- [Fel93] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11:43–81, 1993.
- [FGT92] W. M. Farmer, J. D. Guttman and F. J. Thayer. Little theories. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of Lecture Notes in Computer Science, pages 567–581. Springer-Verlag, 1992.
- [Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics XIX*, pages 19–32. American Mathematical Society, 1967.
- [GMW79] M. J. Gordon, R. Milner and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of Lecture Notes in Computer Science. Springer-Verlag, 1979.
- [GNU92] Lindsay Groves, Raymond Nickson and Mark Utting. A tactic driven refinement tool. In Cliff B. Jones, Roger C. Shaw and Tim Denvir, editors, *Fifth Refinement Workshop*, Workshops in Computing, pages 272–297. BCS FACS, Springer-Verlag, 1992.

- [Gol82] Robert Goldblatt. *Axiomatising the Logic of Computer Programming*, volume 130 of Lecture Notes in Computer Science. Springer-Verlag, 1982.
- [Goo84] D. Good. Mechanical proofs about computer programs. Technical Report 41, Institute for Computing Science, University of Texas at Austin, 1984.
- [GW91] P. H. B. Gardiner and J. C. P. Woodcock. A tactic language for B. 1991.
- [HKLR92] Jieh Hsiang, Hélène Kirchner, Pierre Lescanne and Michaël Rusinowitch. The term rewriting approach to automated theorem proving. *Journal of Logic Programming*, 14:71–99, 1992.
- [Hoa69] C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12:576–583, 1969.
- [JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.
- [JMG] M. H. Jarvis, C. C. Morgan and P. H. B. Gardiner. *The Red Manual*. User documentation as distributed with the Refinement Editor from the Programming Research Group, Oxford University.
- [KPS<sup>+</sup>93] Sentot Kromodimoeljo, Bill Pase, Mark Saaltink, Dan Craigen and Irwin Meisels. A tutorial on EVES. Technical report, ORA Canada, 1993.
- [McC90] W. McCune. OTTER-2.0 user’s guide. Technical Report ANL-90/9, Argonne National Laboratories, 1990.
- [MGW] A. P. Martin, P. H. B. Gardiner and J. C. P. Woodcock. Tactic semantics and reasoning. Draft 1.1 December 20, 1993.
- [Mor87] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
- [Mor94] Carroll Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

- [MR87] Carroll C. Morgan and Ken Robinson. Specification statements and refinement. *IBM Journal of Research and Development*, 31(5):546–555, September 1987.
- [MV90] Carroll Morgan and Trevor Vickers. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14:281–304, 1990.
- [Nel89] Greg Nelson. A generalization of Dijkstra’s calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.
- [NG94] Raymond G. Nickson and Lindsay J. Groves. Metavariables and conditional refinements in the refinement calculus. In David Till, editor, *Sixth Refinement Workshop*, Workshops in Computing, pages 167–187. BCS FACS, Springer-Verlag, 1994.
- [Nic94] Raymond George Nickson. *Tool Support for the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 1994.
- [Nip89a] Tobias Nipkow. Equational reasoning in Isabelle. *Science of Computer Programming*, 12:123–149, 1989.
- [Nip89b] Tobias Nipkow. Term rewriting and beyond – theorem proving in Isabelle. *Formal Aspects of Computing*, 1:320–338, 1989.
- [Pau86] L. C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.
- [RC93] Peter J. Robinson and Anthony Cheng. Qu-Prolog 3.2 reference manual. Technical Report 93-18, Software Verification Research Centre, University of Queensland, 1993.
- [RS93] Peter Robinson and John Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47–61, 1993.
- [RW93] Peter J. Robinson and K. Whitwell. The demonstration interactive theorem prover Demo3.3. Technical Report 93-4, Software Verification Research Centre, University of Queensland, 1993.
- [SRH94] John Staples, Peter J. Robinson and Daniel Hazel. A functional logic for higher level reasoning about computation. *Formal Aspects of Computing*, 6:1–38, 1994.

- [UW94] Mark Utting and Keith Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Centre, Department of Computer Science, The University of Queensland, St. Lucia, QLD 4072, Australia, 1994.
- [Vic90] T. Vickers. An overview of a refinement editor. In *Proceedings of the Fifth Australian Software Engineering Conference*, pages 39–44, 1990.
- [VM94a] Trevor Vickers and Carroll Morgan. A language of refinements. Technical Report TR-CS-94-05, Australian National University, May 1994.
- [VM94b] Trevor Vickers and Carroll Morgan. Procedures and invariants in the refinement calculus. Technical Report TR-CS-94-04, Australian National University, May 1994.
- [Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.