Formal Aspects of Computing (1998) 3: 1–000

A Program Refinement Tool

D. Carrington[1], I. Hayes[1], R. Nickson[2], G. Watson[3] and J. Welsh[3]

[1]Department of Computer Science and Electrical Engineering, and the Software Verification Research Centre, The University of Queensland, Australia 4072;
[2]School of Mathematical and Computing Sciences, Victoria University of Wellington, Wellington, New Zealand;
[3]Software Verification Research Centre, The University of Queensland, Australia 4072.

Keywords: refinement calculus, refinement tools, window inference

Abstract.
The refinement calculus for the development of programs from specifications is well suited to mechanised support. We review the requirements for tool support of refinement as gleaned from our experience with existing refinement tools, and report on the design and implementation of a new tool to support refinement based on these requirements.
The main features of the new tool are close integration of refinement and proof in a single tool (the same mechanism is used for both), good management of the refinement context, an extensible theory base that allows the tool to be adapted to new application domains, and a flexible user interface.

1. Introduction

Formal methods are increasingly seen as an important technique for improving the quality of software development [BH95]. The prime area for applying formal methods to software development is high-integrity systems, most importantly, safety-critical applications where the cost of failures is measured in terms of injuries and lives, but also systems where the cost of failure in economic terms is high.

Formal methods, by their rigorous nature, generate a lot of inter-related detail that needs to be managed if the benefits of formality are to be achieved. To date,

Correspondence and offprint requests to: David Carrington, Department of Computer Science and Electrical Engineering, The University of Queensland, Australia 4072. email: davec@csee.uq.edu.au

tools that support formal methods have not been widely adopted even though tools are often seen as necessary to achieve industrial usage of formal methods.

This paper describes the result of a research project to improve the available tool support for the refinement calculus. The project investigated requirements for such tools and reviewed existing tools before developing a new tool known as PRT. PRT is a research prototype[1] but it demonstrates some innovative ways of supporting the refinement calculus and managing the web of detail that results from using the calculus. The distinctive features of the tool are:

- the refinement tool and theorem prover are integrated, with the same development paradigm being used for both refinement and proof;
- support for contextual information representing preconditions, types and invariants;
- an extensible theory base, supporting the Z mathematical toolkit and allowing adaptation to new application domains;
- a flexible user interface that supports construction and reuse of program derivations.

The paper is structured as follows: Section 2 provides an overview of the refinement calculus and an example; Section 3 analyses roles for tools supporting users of the refinement calculus and hence determines tool requirements; Section 4 reviews previous tools for the refinement calculus; Section 5 describes the new tool PRT; and Section 6 assesses PRT relative to the requirements derived in Section 3.

2. Refinement

This section introduces the reader to the concept of refinement as a formal method of developing programs from specifications. It focuses on the method known as the refinement calculus which was developed by Back, Morgan, and Morris [Bac88, Mor94, Mor87] and discusses how the calculus is used.

2.1. Goal of refinement

Refinement methods seek to establish a sequence of transformations that, when applied to a formal abstract specification, generates an executable program that satisfies the original specification. The sequence of transformations is an *idealised* record of the development process [PC86] that provides explicit traceability between specification and code in both directions. Refinement cannot be completely automated: not all specifications can be implemented, and each transformation represents a design decision leading towards an executable program that meets any nonfunctional requirements such as performance bounds.

Software design involves two kinds of decisions: how to represent information and how to process it. Consequently, transformational approaches such as refinement need to be able to describe both types of decisions. Data refinement [Hoa72, Mor89, GM93] is the class of transformations that replaces one data

[1] PRT is available for anonymous FTP from `ftp.cs.uq.edu.au/pub/SVRC/software/` — it runs under Solaris 2.4; a Linux version is expected to be available in the near future.

representation by another, while procedural refinement is the class of transformations that incorporates processing decisions.

2.2. The refinement calculus

The refinement calculus provides a uniform method for deriving procedural programs from abstract specifications. It extends a programming language with specification constructs to give a *wide-spectrum* language [BBG+78] for programs containing both specification and executable-code fragments. The semantics of the wide-spectrum language is defined using Dijkstra's weakest precondition [Dij76].

The primary specification construct is the specification statement [Mor88] $\vec{w}\mathbf{:}[\,Pre\,,\ Post\,]$ representing a program fragment that, executed in a state satisfying the predicate Pre, terminates in a state satisfying $Post$ while modifying only variables in the list $\vec{w}$ (known as the frame). The frame $\vec{w}$ is a subset of the program variables $\vec{v}$. Pre is a predicate in $\vec{v}$ while $Post$ is a predicate in $\vec{v}$ and $\vec{v_0}$ where, by convention, the decorated variables $\vec{v_0}$ represent the corresponding values in the initial state.

The weakest precondition is also used to define the refinement ordering ($\sqsubseteq$) on the constructs of the wide-spectrum language. The notation $S_i \sqsubseteq S_j$, where S_i and S_j are both programs in the wide-spectrum language, asserts that S_j is a satisfactory replacement for S_i in all contexts. Typically, S_j is derived from S_i by applying an instance of a refinement law to S_i that guarantees the appropriate relationship between S_i and S_j, subject to the validity of some *applicability condition*.

2.3. Example: Exponentiation

To illustrate how executable code can be derived from an abstract specification, the refinement calculus is applied to a small, well-known example [Dij76, pp. 65–66] that demonstrates only procedural refinement.

The abstract specification consists of a context defining the variables and their type:

$$\mathbf{var}\ a, b, z : \mathbb{N}$$

and a specification statement requiring that z be made equal to a raised to the power b:

$$z\mathbf{:}[\ a \neq 0\ ,\ \ z = a^b\ ]$$

Our first design decision introduces an additional variable and, implicitly, an enclosing scope.

$$\begin{array}{ll} \sqsubseteq & \mathbf{var}\ n : \mathbb{N} \bullet \\ & n, z\mathbf{:}[\ a \neq 0\ ,\ \ z = a^b\ ] \end{array}$$

The refinement law for new variables has an applicability condition that the new variable is not one of the variables in the frame of the specification to which the law is applied nor appears free in the precondition or the postcondition. This proviso is satisfied by the variable n.

The second decision is to split the specification into a sequential composition of two components with a view to introducing a loop.

$$\begin{array}{lll} \sqsubseteq & I \mathrel{\widehat=} n \leq b \wedge z = a^n \bullet & \\ & n, z\!: [\, a \neq 0\, ,\ I \,]; & \text{(i)} \\ & n, z\!: [\, I\, ,\ z = a^b \,] & \text{(ii)} \end{array}$$

The predicate I, which becomes the loop invariant, represents the intermediate state in the composition. The choice of I determines the partitioning of effort between the two sub-specifications. An important feature of refinement is that the two sub-specifications can be refined independently.

The specification statement labelled (i) is refined to an assignment.

$$\text{(i)} \quad \sqsubseteq \quad n, z := 0, 1$$

The law for introducing an assignment has a proviso that, in this instance, requires $a \neq 0 \Rrightarrow 0 \leq b \wedge 1 = a^0$ which is satisfied.

The specification statement labelled (ii) is refined to a loop with one guard.

$$\begin{array}{lll} \text{(ii)} \quad \sqsubseteq & \mathbf{do}\ n \neq b \longrightarrow n, z\!: [\, n \neq b \wedge I\, ,\ I \wedge 0 \leq b - n < b_0 - n_0 \,] & \text{(iii)} \\ & \mathbf{od} & \end{array}$$

This loop has invariant I, variant $b - n$ and a proviso that $I \wedge n = b \Rrightarrow z = a^b$.

The remaining specification statement can be refined to an assignment statement:

$$\text{(iii)} \quad \sqsubseteq \quad n, z := n + 1, a * z$$

with the proviso:

$$n \neq b \wedge I \Rrightarrow n + 1 \leq b \wedge a * z = a^{n+1} \wedge 0 \leq b - n - 1 < b - n$$

The proof requires the property of exponentiation that $a^{b+1} = a * a^b$.

The refinement is now complete and the executable program can be extracted:

$$\begin{array}{l} [\![\ \mathbf{var}\ n : \mathbb{N} \bullet \\ \qquad n, z := 0, 1; \\ \qquad \mathbf{do}\ n \neq b \longrightarrow n, z := n + 1, a * z\ \mathbf{od} \\ ]\!] \end{array}$$

While this refinement does not produce an efficient implementation, it demonstrates how the refinement calculus is applied.

3. Tool Requirements

There are several potential roles for tools that support software development via the refinement calculus.

Selecting rules Each refinement step involves selecting a component of the current development and an instance of a refinement rule to apply to that component. Choosing rules is not deterministic but the tool might identify those rules that match components of the current development and let the user make the final selection.

Applying rules Once the rule is selected, the application is usually straightforward symbolic manipulation but with opportunities for formula simplification. This calculation step is also a candidate for tool support.

Discharging proof obligations Many refinement rules have an applicability condition or proviso associated with them to ensure that they are applied in an appropriate context. For confidence in the refinement results, it is important that whenever a rule is applied, its proviso is shown to hold. These proof obligations are lemmas in the first-order predicate calculus and normally cannot be discharged completely automatically. A desirable tool would

- automatically discharge proof obligations where possible, and
- support interactive construction of proofs for any remaining proof obligations.

Recording the development steps Because of the large number of steps associated with non-trivial refinements, a computer-based tool can assist by recording the development process. The development history should contain sufficient information to provide traceability from the specification to the implementation by capturing all the design decisions embodied in the rule applications. It should allow for parts of the development to be extended and modified without compromising its overall integrity. The structural information stored in this way can be used for the extraction of code and for navigation through the development. It also supports the reuse and adaptation of derivations to suit changed specifications.

Tools to support each of these activities would be useful, but considerable synergy could be achieved by having a single tool that integrates the multiple roles. We partition our discussion of requirements for a refinement tool into five sections: representation of information, customisation, refinement transformations, proof support and user interface. These requirements are not independent but the partitioning allows some 'separation of concerns'.

3.1. Representation of information

A tool supporting refinement must represent many different forms of information:

- specification and code fragments,
- refinement rules,
- applicability conditions,
- proofs, and
- the relationships between the instances of these elements that are created as a refinement is performed.

3.1.1. Specification and code fragments

The wide-spectrum language approach of the refinement calculus means that specification and code fragments are expressed in a unified notation. At a minimum, this notation incorporates normal programming language constructs (including boolean expressions) while the specification statement requires predicate logic for the pre- and post-conditions. We need a way to define that logic, and a way to extend it to include the data types used in specifications and programs.

3.1.2. Refinement rules

Each refinement rule is a schema (or template) representing a set of instances. Each instance is a theorem in the refinement calculus, defined by:

- a subject program fragment that the rule instance can be applied to,
- the corresponding result fragment, and
- the applicability condition, which may involve syntactic and semantic constraints on the subject and result fragments.

The sets of instances are described by patterns. The subject pattern for a rule may contain metavariables that can be instantiated when the rule is applied. For example, a rule for weakening a precondition is:

$$\frac{Pre \Rrightarrow Pre'}{w\!:[\,Pre\,,\ Post\,] \sqsubseteq w\!:[\,Pre'\,,\ Post\,]}$$

The result pattern contains a metavariable ($Post$) that occurs in the subject pattern, and also an independent metavariable (Pre'). Independent metavariables can be considered as parameters to the rule, since their instantiation is not determined by matching the subject pattern with the actual subject fragment, but the instantiation affects the result.

To apply a rule to a particular program fragment, the metavariables in the subject pattern are instantiated to match the fragment and the fragment is transformed to the result pattern instantiated with the values for the metavariables. Independent metavariables can either be instantiated when the rule is applied or their instantiation can be deferred. There are benefits from allowing uninstantiated metavariables in the program during the development [NG94]:

- it allows the instantiation of metavariables to be deferred until later in the development when the choices for the metavariables are clearer; and
- it allows derived rules to be developed and proved.

A derived rule is a composition of other rules, possibly with a simplified applicability condition. Derived rules provide a mechanism for extending the set of rules in the refinement calculus.

Refinement tactics are an alternative to the concept of derived rules. They are more flexible combinations of existing refinement rules where the combination is computed dynamically when the tactic is applied, based on the current context. Tactics require a language for expressing how rules may be combined and how the context can influence the outcome.

3.1.3. Applicability conditions

Many applicability conditions associated with refinement rules can be expressed using the same logic as used in the wide-spectrum language, but other constraints are syntactic in nature, such as 'the identifiers on the left-hand side of a multiple assignment command are distinct'. Applicability conditions in rule schemas can contain metavariables that appear in subject and result patterns. Applying a refinement rule generates an instance of the corresponding applicability condition. This formula needs to be shown to hold in the context of the rule application.

3.1.4. Proofs

If the tool is to manage proofs associated with applicability conditions arising from applying refinement rules, some representation of proofs is required. The particular representation will depend on the mechanism used to discharge these obligations but it is desirable that the representation facilitates the checking and incremental modification of proofs if the refinement is changed.

3.1.5. Relationships

Applying the refinement approach to program development generates a collection of programs related by a refinement relation. If we are to achieve the goal of capturing an *idealised* record of the development process, we need to represent the refinement relationships between program fragments.

The refinement process can be captured as a refinement tree where the initial specification is the root node. Each node is a program fragment and each edge represents a refinement rule application. Associated with each rule application are the values of any instantiated parameters, the consequent proof obligation and its proof. When the refinement is complete, the implementation code can be collected by traversing the tree with interior nodes contributing the program structure and the leaf nodes contributing the individual commands. This representation does not record the actual order of rule applications but does allow an idealised sequence to be generated.

Contextual information must also be managed by a refinement tool. Contextual information is environmental data that is not necessarily represented explicitly in the current program fragment but which is required for refinement. Examples of contextual information are types of variables, procedure declarations, invariants, and local definitions. Contextual information is typically used for discharging proof obligations.

3.2. Customisation

It is important that users can define new notation within the refinement tool to suit particular applications. This new notation may include additional mathematical formalisms in specifications, and notation to customise the wide-spectrum language to correspond to a target programming language. To define new notation requires the ability to extend the syntax of the wide spectrum language and the syntax and semantics of the logic of applicability conditions.

It is also desirable that new refinement rules can be defined. This raises the issue of the correctness of refinement rules within the tool. Rules could be justified externally to the tool, but a more desirable option is to allow rules to be proved within the tool. The latter option requires the ability to define the semantics of the wide spectrum language and the refinement relation. It also increases the requirements for proof support.

3.3. Refinement transformations

For a refinement tool to be effective at applying rules, several requirements must be satisfied. Mechanisms are required for structuring collections of rules to make them easy to use. It is desirable to be able to identify those rules whose subject pattern matches a given fragment. For schematic developments, pattern matching generalises to unification because of metavariables in both the fragment and the subject pattern.

Once a set of potentially applicable rules has been identified, we expect user interaction to select a particular rule and to supply any parameter instantiations that are required. The rule can then be applied and the resulting program fragment calculated. Any applicability condition associated with the rule should also be instantiated with parameter values and context information.

3.4. Proof support

To discharge proof obligations arising from refinement, one needs a theorem prover. The prover should be integrated into the refinement tool so that obligations can be discharged as they arise. The prover and the refinement tool should present a common interface to the user for both activities, so that the user only need develop a single conceptual model of the interaction.

The prover should be able to handle any applicability condition generated by applying a refinement rule, and it is essential that the underlying logic can be customised to suit particular application domains. As a minimum, we need to be able to extend the logic to define the function and predicate symbols of our application theory.

It is also desirable that the proof support is capable of justifying new refinement rules (Section 3.2). This requires the ability to reason schematically about commands and the refinement relation. If our theorem proving logic can model commands, it can also express many applicability conditions that are normally considered syntactic. For example, to model the semantics of assignment commands, program identifiers need to be objects in the domain of discourse, distinct from the values those variables take on in different computational states. With such a logic, it is easy to express applicability conditions like 'all identifiers on the left-hand side are distinct' for a rule that introduces a multiple assignment command. If we can express all such syntactic applicability conditions in our theorem proving logic, we need not develop additional mechanisms for handling them.

3.4.1. Proof styles

There are many styles of proof: resolution, as used in Otter [McC90]; natural deduction as used in Mural [JJLM91] and in LCF [GMW79] and its descendants; proof theories tailored to various constructive logics as used in NuPRL [CAB+86] and induction-based strategies such as those used in the Boyer-Moore prover [BM79] and Gypsy [Goo84]. The choice of proof style will influence interactions between the user and the tool to discharge proof obligations and to justify new refinement rules. A particularly effective proof style is term rewriting [HKLR92]. With this approach, a sequence of formal objects is constructed, starting with the formula to be proved, in which each object is related to the preceding one by some validity-preserving relation (typically, but not essentially, equivalence). If the final element of the sequence is a formula known to be valid, the original formula is valid. Usually, successive elements of the sequence are calculated by applying conditional rewriting rules to the previous element. Generally, the rewriting relation and term structure of the object logic allow replacement of subterms (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+93], and is a major proof style in Isabelle [Pau86]. Term rewriting is also a foundation of the window inference proof technique used in the Ergo theorem prover [BBNU96].

3.5. User interface

The user interface of any tool provides mechanisms for accessing the tool's functionality. Different user interfaces are possible for a fixed set of tool functions, so it is valuable to consider the required capabilities of the user interface in addition to the tool's functionality.

Users needs to be able to create and modify program fragments based on the wide spectrum language, preferably in a form with which they are familiar. (The special symbols commonly used with the refinement calculus can cause some difficulties.)

To apply a refinement rule, the user must identify a fragment to be refined within the current development, select a rule to be applied and instantiate any parameters of that rule. It is also convenient to have browsing facilities for the collection of refinement rules.

During the development process, the user needs to be able to view the current state of the refinement and to navigate over the complete development record. Because there is a lot of information generated during refinement, the user interface must be capable of hiding detail not currently relevant.

As well as being able to navigate over the development record, it is desirable that the user can modify parts of the refinement record and observe the consequences (a form of 'what if' analysis). Examples are changing the initial frame of the specification or the parameter values of a rule application.

Discharging proof obligations also has user interface implications. If we assume that this cannot be done fully automatically, mechanisms for controlling and viewing the proof activity are required.

Customisation is a major requirement of a refinement tool. The user interface must provide the mechanisms for performing the customisations discussed in Section 3.2. Equally, the user interface itself should be capable of user customisation.

3.5.1. Interface styles

For maximum ease of use, the user interface of any tool should reflect and reinforce the user's mental model of the process and products involved. In practice, the tool designer must choose a particular model and users must map or adjust their mental models to that chosen. Two interface styles, reflecting distinct mental models, can be identified for a refinement tool.

The first style is based on a 'symbolic calculator' model which concentrates on rule application. At each step, the calculator state represents the result of the overall refinement to date and the specific focus for the next refinement operation. User input is normally the refinement rule to be applied at this focus, but to support flexible refinement, the user must be able to reset the focus to an alternative refinement point if one exists. With this model, support for review, modification or reuse of a prior refinement involves replay of an explicit or implicit history of the sequence of refinement steps involved. Such a sequence is an ideal history in the sense that it does not reflect any replays involved in its creation, but is more specific than necessary in that it reflects the specific order of development chosen. Developments other than pure top-down refinement can only be conceptualised as creation and collation of fragments of the sequence involved.

The second style is an 'active document' model [CHN+96] where the emphasis is on manipulating a refinement record that approximates the textbook style for presenting refinements. This style is a WYSIWYG approach in which each refinement step expands, i.e., edits, the refinement record at the relevant point. Modification or reuse of an existing refinement is achieved by the same editing paradigm, with the 'active document' deducing the consequences for other parts of the overall refinement as appropriate. Review of a refinement simply involves browsing the document that records it. Within such a document, hypertext links can significantly enhance its readability by, for example, linking each proof to the corresponding proof obligation. The development history captured in such a document is truly ideal since all details of the actual history are reflected only by the (unrecorded) sequence of edit operations that produced it. Development processes other than pure top-down are supported by this model since it allows refinement documents to pass through incomplete and inconsistent states.

The example in Section 2.3 is presented top-down in the style used by Morgan in [Mor94], yet this may not be the way in which it was developed. As Parnas [PC86] explained, there is normally a discrepancy between the actual development process with its trials and backtracking and the form in which we want to record the final product. Support tools for software development need to handle the actual development process and the need to subsequently present the idealised, rational development. Naftalin [Naf92, Naf94] and Han [Han92] have described how the actual development process using the refinement calculus need not be purely top-down and they have assessed some of the consequences for support tools.

4. Existing Tools

Because software development using the refinement calculus requires a large number of small steps, it seems natural to consider computer-based tool support and several such tools have been developed. To illustrate the range of tools available prior to the development of PRT, we describe six tools according to five criteria: depth of formalisation, support for proving obligations, user interface, genericity, and support for managing context.

One of the earliest tools was built by Carrington and Robinson [CR91, CR88] with the Synthesizer Generator [RT89] which uses attribute grammars to create language-based editors. We refer to this tool as CRSG. It was limited to integer expressions and simple and array variables. Refinement rules were built-in as editing transformations. The Red tool [Vic90] was developed at Oxford. Proof obligations could not be discharged within the tool but they could be piped to an independent theorem prover. Red had support for developing derived refinement rules from the base set.

Back has been associated with several refinement tools. The Centipede environment [BHS92] used a graphical display for refinement diagrams [Bac91] that represent the structure of a refinement. Another tool developed with von Wright [BvW90] was implemented using HOL [Gor88]. It was subsequently extended [vW94] to use the window inference paradigm. Grundy [Gru92] also used HOL and window inference for developing a refinement tool, which treated programs as predicates with extensions to model undefinedness and nontermination. Nickson [Nic93, GNU92] developed a refinement editor with a graphical user interface and its own proof system based on rewrite rules. We refer to this tool as RRE. It contained support for refinement tactics as a mechanism for creating more powerful rules. The Proxac editor [vdS94] developed by van de Snepscheut is built on top of a general term rewriting engine.

Depth of formalisation Von Wright's HOL tool models specifications and programs by defining everything in higher order logic. The tool relies only on the correctness of the definitions since the refinement rules are proved from first principles using HOL. This leads to highly trustworthy proofs that refinement developments are correct, and facilitates the use of results from conventional mathematics. On the other hand, using HOL notation directly leads to notation and formalism that is sometimes cumbersome to use for refinement. More recently, Butler and Långbacka [BL96] have developed a graphical user interface for the HOL refinement calculus theory that supports a conventional programming syntax and hides the underlying representation of program state.

The other tools treat specifications and programs as uninterpreted terms, manipulated syntactically. Refinement rules cannot be proved, and developments do not have the formal status of mathematical theorems. No distinction between program and logical variables is typically made, so some caution must be exercised when using standard results of classical predicate logic and mathematics.

Support for proving obligations All of the tools provide (or intend to provide) some support for proving refinement obligations. The tools differ in the kind of support they provide for this activity. CRSG and RRE attempt fully automatic proof (though each supports manual application of rewriting rules). Like RRE, Proxac uses the same style (based on term rewriting) for proof as it does for refinement. The HOL tool includes suitable tactics for assisting with the kinds of proofs that arise in formal development, but does not attempt to fully automate proofs. Red and Centipede do not incorporate proof support directly, but can be linked to external proof tools.

User interface Proxac, RRE and Centipede emphasise usability and include sophisticated graphical user interfaces. This makes it easier to experiment with the tools, but difficult to record derivation steps for off-line browsing, adaptation and reuse. We would classify them as examples of the 'symbolic calculator' style described in Section 3.5.1.

CRSG's interface is the Synthesizer Generator, so specifications, refinements and proofs are constructed by expanding templates. It uses the 'active document' model style. The other tools have simple, conceptually powerful command-driven interfaces. These are less pleasant to use, but support the construction of human- and machine-readable derivation scripts that can be edited textually and fed back into the tool for reuse. The creation of a graphical user interface for the HOL tool [BL96] improves its usability.

Genericity Proxac attains great genericity by its simplicity. It is easy to add a new (unverified) refinement or proof rule to the system, and such rules can manipulate novel program and logical constructs without prior definition. RRE is partly generic, since new, unproved refinement and proof rules can be added, but it is not easy to add new program constructs. The HOL tool is, in theory at least, equally generic. To add a new refinement rule, one proves a theorem in the underlying logic. To add a new program construct, one provides a definition of the construct, either in terms of existing constructs or using the semantic model. To make such additions practical, one would need to also define suitable high-level tactics for using the new rules and manipulating the new constructs. Red supports the construction of new, derived refinement rules by combining existing rules, but not the introduction of new program constructs, nor the addition of primitive rules. Centipede's user interface is driven by a "BNF-like" definition while its transformation rules are defined in a Prolog library. These mechanisms allow extensions at the syntactic level. It is not possible to extend CRSG easily.

Support for managing context RRE maintains a structure that encapsulates the context of a program fragment (including, but not limited to, the types of all program variables), and makes this context available in a specialised way when discharging proof obligations. In HOL, the context is a formal part of the definition of constructs. The other tools do not represent context at all.

5. Program Refinement Tool

5.1. The Design Goals

We now discuss the design of a refinement tool, PRT, based on the requirements discussed in Section 3. Within this framework, a set of more specific goals was adopted as the basis of the design. These goals balance the practical limitations of producing a working tool and the desire to extend the functionality beyond that of existing tools. The goals are also a compromise between an emphasis on the logical basis of the tool and its theoretical foundations, and more practical issues such as usability. The design goals adopted for PRT are to develop an integrated tool that:

- provides support for the process of selecting refinement rules;
- automates the process of applying rules as far as possible;
- enables the user to prove obligations, by providing a seamless integration of the theorem prover into the tool; and
- provides a customisable user interface specifically engineered for the display of refinement and proof structures.

Genericity is desirable at all levels of the design. The requirement for customisation is discussed in Section 3.2 and includes the ability to extend the syntax, refinement rules and underlying logic. Possible applications of genericity are in the extension of the refinement theory to include data refinement, a probabilistic extension to the refinement calculus [MMS96], or to encompass Back's Action Systems [BS91].

5.2. Tool Architecture

From our analysis of the requirements, we developed an abstract architecture comprising:

- **A Refinement engine** that applies the refinement transformation rules and generates the obligations incurred when rules are applied.
- **Proof support** whereby a partially automated assistant has responsibility for managing the proof obligations.
- **A Relationship Manager** that handles the structural relationships between components of the refinement: between specifications and their refinements, between obligations and their proofs, and perhaps between alternative refinements.
- **A User Interface** that presents the user with a unified interface to the other components.

This architecture is realised by implementing PRT as an application of a theorem prover, in which the roles of the refinement engine, proof support tool and relationship manager are combined.

The integrity of a program refinement depends on the validity of the applications of the refinement rules. This in turn depends on the correctness of the rules, their correct application, and the validity of the associated proof obligations. The application of rules is the function of the refinement engine. The justification of rules and discharging of obligations are functions of the theorem prover. Substantial benefits accrue if these two components are combined and operate in a similar fashion. The advantages of this arrangement are discussed further in [CHN+94a, CHN+96]; the technical requirements for the refinement engine are described in detail in [CHN+95].

Combining the refinement engine and prover achieves the requirement of presenting the user with a unified user interface and a single conceptual model for both activities. Also the refinement and its associated proofs are held in a common data structure or set of related structures. This simplifies the presentation and navigation of the refinement for the user. It also means that relevant information is shared between obligations and the refinement steps that give rise to them. This is significant for a refinement tool since each obligation is presented in the context of the current state of the refinement, and having this context readily available in the proof of the obligations is an advantage.

Formal refinement is typically done when the correctness of the refinement of resulting program from its specification is important. Combining the refinement engine and prover means that the tool is dependent on a single formal system, so the effort of establishing the soundness of the tool is reduced. This unification also simplifies the task of managing the structural relationships since the complete refinement is a single theorem and its components are subproofs of this theorem. Thus, the prover acts as the relationship manager as well.

5.2.1. Window inference

In PRT, we use the *window inference* proof paradigm [RS93] which is based on term rewriting. This supports goal-directed reasoning within a hierarchical proof structure.

Probably the greatest benefit of using a prover based on term rewriting is the similarity of the proof process to the refinement process. Both involve selecting a component to transform, choosing and instantiating a transformation rule, discharging its applicability condition and replacing the fragment by the result of the 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 composed); and monotonicity of constructors (so subterms can be replaced). Using a proof style that is similar to refinement reduces the cognitive load on the user when switching between refinement and proof steps.

Each node in the proof hierarchy is a subproof (or *window*), which has the following components:

- a unique identifying subproof number, used to navigate the proof tree;
- a *relation* R, indicating what relationship is to be preserved in the subproof;
- a collection $H_1, \ldots, H_m$ of *hypotheses*, which are assumed true throughout the subproof;
- a numbered sequence $F_1, \ldots, F_n$ of *focus terms*;
- for each pair F_i, F_{i+1}, a *justification* showing that $F_i \mathrm{R} F_{i+1}$ (under the assumption that the hypotheses hold); and
- a *goal* which indicates the desired outcome of transforming the focus.

The justification of a transformation $F_i \mathrm{R} F_{i+1}$ can be:

- an appeal to an inference rule, giving the name of the rule and the numbers of the hypotheses in the current subproof that discharge the rule's applicability conditions; or
- a vector selecting some subterm S of the input focus F_i, and the number of another subproof that transforms S to S', such that F_{i+1} results by replacing the selected subterm S of F_i by S'.

Hypotheses can also have justifications, showing how they are derived from the context. For example, in the subproof shown in Figure 1:

- the subproof number is 17;
- the relation is *refsto* (i.e. $\sqsubseteq$);
- the hypotheses are the terms numbered $20, \ldots, 28$;
- the focus terms are numbered $1, 2, 3$ (the '$' at the end of focus 2 in Figure 1 indicates truncation of the term to fit the window);
- the transition from 1–2 is justified by a forwards application (`f`) of the `rules:semI` inference rule, with hypothesis `28` used to discharge the applicability condition;
- the transition from 2–3 is justified by another subproof, numbered `34`, which transforms the first (`[1]`) subterm of focus 2 into the first subterm of focus 3, leaving the other subterms unchanged; and
- the goal `unknown`, because the goal of a refinement is typically discovered during the refinement process, not determined beforehand.

A subproof can be *open*, indicating that the final focus may be further transformed. An open subproof may not be used in any justification. In Figure 1, the final focus is in bold face, to indicate that the subproof is open and further refinements may be applied to a focus. When a subproof is finished, it is *closed*;

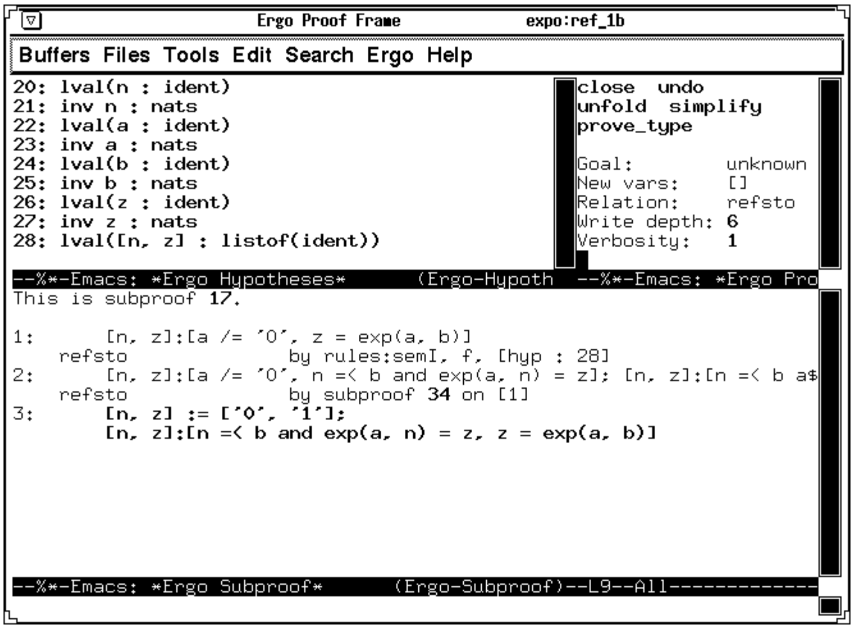

Fig. 1. PRT Proof frame — the '$' at the end of the line labelled 2 indicates truncation.

it then becomes the justification of a step in the parent subproof, and its final focus replaces its initial focus in the selected subterm of the parent subproof.

In a standard theorem prover, the relation being preserved is typically equivalence or implication, however other relations may be appropriate. When using window inference for refinement transformations, the relationship is that of 'is refined by' [Gru92]. It is also possible to replace refinement rules such as 'weaken precondition' by operations that allow one to open a window directly on a precondition and weaken it.

PRT is built as an extension of the Ergo theorem prover [BBNU96], which uses the window inference proof paradigm. Ergo has other characteristics that suit its use in a refinement tool: it is designed to be extensible, and it supports automated proof through a comprehensive tactic language [Whi92]. Tactics can be invoked automatically at each proof step, which provides a mechanism to discharge simple obligations without user intervention.

5.3. Program Window Inference

Reasoning about programs requires a consideration of *states*. For example, the specification statement $x{:}[\,x = a\,,\; x = b\,]$ refers to two distinct states (initial and final), and the three occurrences of x refer to different values:

- In the frame, x denotes the variable (*l-value*) itself: the frame constrains the variables that can change from initial to final state.
- In the precondition, x denotes the value taken by the variable in the initial state. The precondition constrains the set of initial states from which the command is applicable.

- In the postcondition, x denotes the value taken by the variable in the final state. The postcondition constrains the set of acceptable final states.

Standard treatments [Bac88, MV94, Mor87] of refinement are based on higher-order logic. Program variables are an explicit domain, distinct from the domains of values they can take. States are functions that map program variables to values; expressions and predicates map states to values and truth values. Programs are then identified with their meanings as predicate transformers; there is no explicit weakest precondition operator.

An advantage of the refinement calculus is that most developments can be done using refinement rules whose conditions are apparently first-order, and do not refer to state at all. Some 'extralogical' conditions are syntactic in nature (for example, 'x is a fresh variable'), while in other 'logical' conditions the program variables behave just like logical variables (for example, '$x = 0 \Rightarrow x \leq 1$'). This reduction to first-order concepts is possible because the logical conditions of rules are generally implicitly universally quantified 'for all states'.

With the higher-order approaches to modeling refinement, this universal quantification must be made explicit, the different denotations of identifiers must be distinguished, and the appearance of states as arguments of expressions and predicates cannot be easily hidden. On the other hand, with these approaches it is possible to formally prove refinement rules, reasoning explicitly about states and their relationships. This is not possible with classical first-order logic without an explicit modeling of states.

Program window inference [NH96] is based on a modal logic [NH95], in which the possible worlds are states. This modal logic can readily be related to the more traditional higher-order semantics. Using a modal logic allows us to avoid higher-order constructs, so that program variables behave like (modal) logical variables in the applicability conditions of refinement rules. It is still possible to distinguish occurrences of identifiers, using modal operators that constrain the sets of states under consideration; it is possible to prove refinement rules because states are implicit parameters [SRH94] of predicates.

5.3.1. The program window inference theory

Ergo has an extensive theory hierarchy covering classical logic, set theory, arithmetic and structures such as sequences. To support refinements, a new theory has been added that implements the modal logic of program window inference. The program window inference theory includes special support for dealing with program variables, applying refinement rules and managing the kinds of context that arise during refinement. It includes:

- A definition of the refinement relation in terms of a primitive *weakest precondition* function (wp).
- The refinement rules, which are theorem schemas in this theory.
- Window opening and closing rules that apply to the refinement relation.
- Mechanisms for handling program variables and their substitution.
- Tactic support for applying and instantiating refinement rules, which hides much of this detail from the user and allows substitution to be done in a simple manner.
- A proof interface that has been adapted for refinement.

Each refinement step is an application of an instance of a theorem schema in the underlying theory, with the metavariables instantiated to correspond to the current context. Within the refinement theory, program variables behave in special ways, and machinery to handle them is built into the theory. For further details on how program variables are handled in Ergo, see [CHN+94a]. Such detail is usually hidden from the user of PRT, but is necessary for a full understanding of how the theory works and the soundness of its results. The result of a development is an Ergo theorem that asserts that the final program is a refinement of the original specification, perhaps with some conditions; the theorem is available for use in other developments as a derived refinement rule.

5.3.2. Program variables

Program variables are modeled by an infinite collection of constants. It is not possible to use logical variables directly, as Ergo has built-in notions of substitution and quantification that do not concur with that of the modal logic used. Substitution of expressions for program variables (needed, for example, in the applicability condition of the refinement rule for introducing an assignment command) is represented as a modal function on predicates, and is supported by a tactic that calculates such substitutions where possible. Similarly, it is possible to bind program variables with universal and existential quantifiers (as in the definition of weakest precondition for local variable and constant constructs), and tactics exist that support reasoning with those quantified terms.

5.3.3. Applying refinement rules

In Ergo, an axiom or theorem can be interpreted as a directional transformation rule. The rule is applied by instantiating it to match the situation in which it is applied. By default, other schematic variables are left uninstantiated. These metavariables can be instantiated at any stage, later in the derivation. For refinement, it is more often convenient to completely instantiate rules when they are applied. To facilitate this, each refinement rule is annotated with a list of *parameters*. These are schematic variables in the rule that normally require explicit instantiation. The parameters have meta-types, which control how instantiations are obtained and checked, for example by prompting the user. It is possible to leave parameters uninstantiated when a rule is applied, thus retaining the flexibility of refinements using metavariables [NG94].

Some refinement rules are conditional. For these rules, it is necessary to discharge the conditions when the rule is used. When a refinement rule is applied, the conditions of the rule are discharged as follows:

- Some conditions (in particular, most of the syntactic conditions) are discharged automatically using tactics. Automatic discharging of conditions is driven by an extensible table associating tactics with condition patterns.
- The remaining obligations are handled in one of three ways.
  - The default is that the remaining obligations are recorded as conjectures that must be discharged before the current window is closed.
  - Alternatively these obligations may be deferred. They are recorded as postulates in the theory, which the user can discharge later as separate proofs.

– The final option is that these obligations must be discharged as they are generated.

Which option is currently in use can be selected by the user.

5.3.4. Context

Several kinds of context are used for determining the applicability of refinements and for discharging obligations.

l-value context captures information about the names of variables, procedures, etc. that are in scope. It is used to discharge obligations such as 'x is a fresh variable' or 'x and y are distinct program variables'.

invariant context captures information about the values of variables that is preserved throughout a scope. Invariants [MV90] are used to incorporate type information, while retaining an untyped refinement logic.

precondition context captures assumptions about initial values of variables that can be deduced from prior assignments and tests. For example, immediately after the guard $x = y$ in an **if** command, the precondition context includes $x = y$.

These three kinds of context are represented in the Ergo hypothesis list by terms annotated `lval`, `inv` and `pre`. Window opening and closing rules automatically manage the hypothesis list, deducing new hypotheses from the context and manipulating existing hypotheses according to their annotations.

5.4. The User Interface

The detailed design of the user interface concentrated on presenting various aspects of the refinement, and on representing common operations by simple actions in the interface. These issues are also central to the design of the interface of the base prover, and an Emacs interface to the Ergo theorem prover addressing these issues was developed jointly with the Ergo developers [NU95]. The Emacs Ergo interface uses the 'active document' style of Section 3.5, with navigation via the structure of the proof.

5.4.1. Emacs Display

The current version of PRT uses the Ergo Emacs interface. The interface supports two styles of interaction. Initially the user is presented with a command-line interface to Ergo in an Emacs buffer. With this interface, the user can select a theory to work in and start a refinement. For the exponentiation example from Section 2, we create a new theory and define the exponentiation (`exp`) function:

```
declare_nonmodal exp(A,B).
axiom exp_type === A:nats and B:nats => exp(A,B):nats.
axiom exp_base === A /= 0 => exp(A,0) = 1.
axiom exp_def === exp(A,B+1) = A*exp(A,B).
```

The initial specification is entered as:

```
refinement ref_1 === [z]:[a/=0, z=exp(a,b)]
               in_context [var(a:nats), var(b:nats), var(z:nats)].
```

The in_context clause supplies type information about the program variables. Other tasks such as theory management are also done at this level.

During a refinement development or a proof, a different style of interaction is used. In this style the tool displays several Emacs frames and panes in which information about the current state of the refinement is presented. The most important frames are the proof frame (Figure 1) that has panes that display the current state of the refinement, and the proof script frame where the commands required to perform the refinement are recorded. A number of other frames are used for different purposes: the rules frame is used to control the display and selection of matching rules; Help is displayed in its own frame; if the proof browser is invoked, this also appears in its own frame; and the original Ergo command-line interface frame is retained in the background.

The components of the proof frame can be seen in Figure 1. This, and the other figures, represent stages in the refinement of the exponentiation program from Section 2.

The proof frame has three panes. The bottom pane is the subproof pane which shows the steps taken in the current subproof. Figure 1 shows a specification statement being refined to a sequential composition, using the rule semI. The second step in the development is the refinement of the first component of the sequential composition to an assignment. Details of this step are elided by hiding the discharging of the proof obligation as a subproof. These details are accessible by clicking on the subproof number (65) to display that subproof. This approach is used to elide complete refinement subtrees so that the user can choose an appropriate level of abstraction. Note that specification statements are presented in a format similar to that of Morgan, for example:

```
[n, z]:[n =< b and exp(a,n) = z, z = exp(a,b)]
```

Above the subproof pane is a pane showing the hypotheses of the current window, and a control pane containing buttons for common commands and editable values for global options.

The interface is arranged so that point and click and menu selection can be used for common operations. For instance, the control pane has buttons labeled Close and Undo which respectively close a window, and undo the last proof step. Opening a new window on a subterm of the current focus can also be performed with the mouse. The user can highlight any subterm of the focus, and use the mouse to open a window on that subterm.

5.4.2. Finding Rules

The PRT interface has a menu option to guide the user in choosing refinement rules. When the user selects this option, Ergo searches the refinement rules, matching them against the current context, and displays a list of those that match in a rules frame (see Figure 2 for an example). The matching rules are displayed in the central pane. Each line shows the name of a rule at the left (followed by f,c which are Ergo annotations indicating a forward application of the refinement rule and that there is a condition associated with the rule) and, at the right, the output term of the rule as instantiated in the current context.

Where matching the subject does not fully instantiate the rule, the result pattern contains metavariables, and, if this rule is selected, the user is prompted to supply values for these. For instance, the assignment rule is displayed in the form B := C, and if applied, the user is prompted to supply values for the variable and expression lists (or to defer their instantiation explicitly). When the

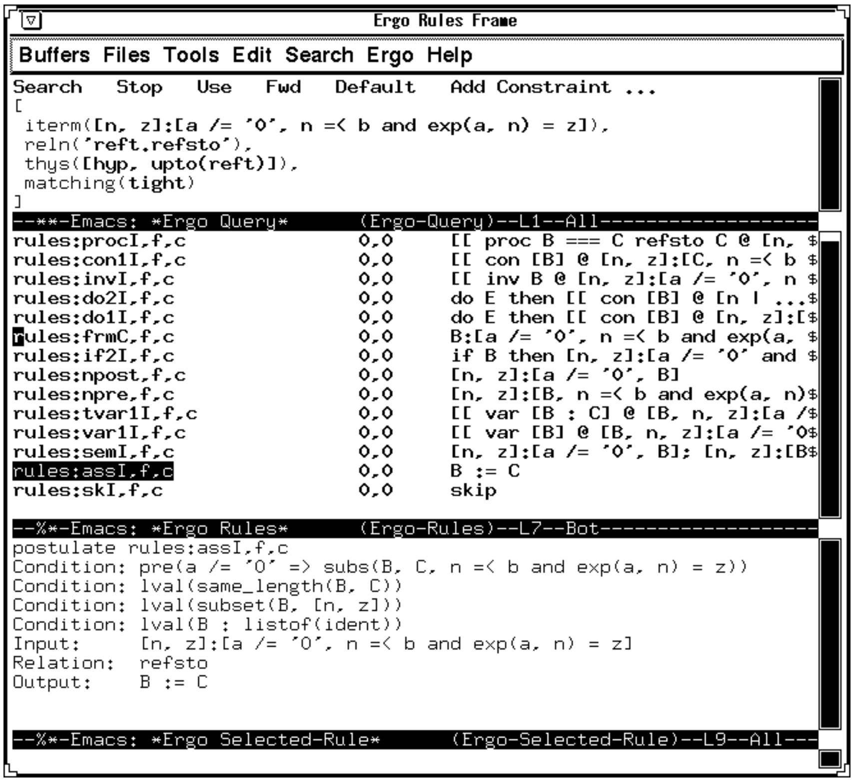

Fig. 2. PRT Rules Frame

user selects a rule from this list, the lower pane displays a full description of the rule selected in the form:

$$\mathit{Condition} \Rightarrow \mathit{Input} \sqsubseteq \mathit{Output}$$

with *Condition*, *Input* and *Output* instantiated in the current context. In Figure 2, the assignment rule is selected. When the currently selected rule is applied, the corresponding command is also inserted into the proof script. The search for applicable rules is a heuristic process and the user can opt for a looser or tighter matching as the situation demands. The parameters used by the searching mechanism can be adjusted by editing the fields in the template displayed in the top pane. For instance to control the theories that are searched, the user can edit the *thys* field. Rules are chosen by pattern matching: one pattern matches the rule and another matches the output of the rule, when unified with the current context. These patterns can also be modified by the user to control the search.

5.4.3. Scripts

The proof script is a simple text file which is displayed in its own frame. It includes the commands for all the rule applications and the proofs of their obligations as subproofs. This frame is editable, allowing the user to add comments and correct errors, and the file is saved when the user exits a proof.

```
/* Introduce the initial assignment*/
        apply(rules:assI).
        /* Identifiers: */ [n,z].
        /* Expressions: */ [0,1].
        discharge(_).
            [2].
              [1].
                hyp:(33).
                  use([iarith:def_nats, f, u]).
                  close.
              use([expo:exp_base, t, c]).
```

Fig. 3. The Fragment of the Proof Script for: $n, z := 0, 1$

Some of the commands in the proof script will have been typed by the user into this frame, while others record operations performed in other ways. For instance, the user may open a window by selecting a subterm of the focus with the mouse, in which case the explicit command to open the window is recorded in the proof script at the appropriate place. Where parameters are required for a rule application, the user types these into the proof script frame, so that they are recorded as part of the script. The fragment of a proof script in Figure 3 shows the commands to apply the first 'introduce assignment' rule in the example and discharge the resulting obligation. The numbers in braces are commands to open windows on subterms; these are normally generated by selecting the required subterm with the mouse. The fragment also contains use of the type declaration for the variable b (`hyp:(33)`), an application of an arithmetic theory rule, and the base axiom for exponentiation. The script is automatically saved and reloaded as part of the proof. It can also be replayed line by line to recreate the proof. Scripts allow reuse at a basic level, since, being simple text files, they can be readily copied, combined and edited. After editing, replaying a script is necessary to generate a valid refinement.

5.4.4. Browsing and printing refinements

The subproof pane of the proof frame shows all the steps performed in the current window. In this pane, subproof numbers are highlighted, and clicking on any of these brings up a browser for that subproof in a separate frame. In PRT, Ergo subproofs may represent either the application of refinement rules or the discharge of obligations. For instance, in Figure 1, subproof **65** in the bottom pane is the transformation of the first part of specification to an assignment. Clicking on the subproof number brings up a browser for this transformation. The browser is a read-only display that has hypotheses and subproof panes as in the proof frame, and an additional ancestors pane that defines the current subproof's position in the proof tree. By clicking on further subproof numbers in the ancestor pane, the user can navigate through the entire refinement.

The result of a refinement in PRT is a theorem of the form:

$$\mathit{Context} \Rightarrow (\mathit{Initial\ Specification} \sqsubseteq \mathit{Final\ Program})$$

On completion, the theorem and its proof are stored and can be viewed subsequently using the browser. The theorem can be used in subsequent developments;

if it contains metavariables, it is in effect a derived schematic rule. The full proof or any subproof from it can be saved and printed in ASCII format.

5.4.5. Help

The Ergo reference manual is available as a hypertext document. This document is generated automatically from special comments in the Ergo code and user-developed tactics.

Ergo has a set of commands (the *show* commands) that list various types of information about the current Ergo environment, for instance, one can display the set of operators defined in the current theory or the available tactics. The Emacs interface has a hierarchy of menus that give access to the different *show* commands in a convenient way.

5.4.6. Customisation

Customisation of the user interface is available at several levels. The Ergo interface is built using GNU Emacs for the X Window System, and the display can be tailored by using standard X-windows and Emacs facilities. For instance, the fonts can be adjusted by overriding the X-windows defaults, and the mouse can be used to restructure the pane display. The Ergo Emacs interface has a number of customisation features; for example, there are buttons to adjust runtime parameters such as the level of detail suppression.

6. Assessment

We can assess PRT using the criteria introduced in Section 4 where previous support tools for the refinement calculus were reviewed.

Depth of formalisation PRT uses a purpose-built logic with commands, predicates and program and logical variables as separate syntactic classes. The syntax of specifications, programs and logical formulae is close to that traditionally used. Considerable benefit is gained by a formal treatment of states as possible worlds in a modal logic, because a deep modeling is possible, yet the first-order flavour of refinement provisos is retained and standard mathematical results are available. A possible disadvantage is that the logic is novel, perhaps reducing confidence in its soundness.

Support for proving obligations PRT supports the discharge of proof obligations using the program window inference logic. Because the applicability conditions of most rules need be proved for all states, the modalities can usually be ignored and the obligations discharged by appealing to results from classical logic. The close integration of refinement and proof logics exploits the similarities between these activities, reducing the number of different process models and interface styles that must be understood by the users. Using the same underlying engine for refinement and proof also increases one's confidence in the validity of refinements, since we can be sure that the semantics underlying the activities are identical.

User interface The prototype user interface supports navigation and browsing of proofs and generates commands for the underlying engine. The refinement and proof engine underlying PRT is command-driven, and can be fed a script.

Genericity PRT can be extended by defining weakest precondition semantics for new program constructs, and defining how program window inference context is affected by these constructs. New refinement rules can be postulated, and they can be proved with the definition of refinement and the weakest precondition semantics of the program constructs used. Application theories can be defined, building upon an existing library that includes first-order predicate logic, arithmetic and ZFC set theory. We discuss this aspect of PRT in the following section.

Support for managing context PRT uses program window inference, which has a powerful notation for representing different kinds of context, including preconditions, types and invariants, and distinctness and aliasing properties of variables. Window rules update contextual information automatically as the focus of attention moves in a derivation.

6.1. Extensibility

PRT was designed to be extensible in terms of the wide-spectrum language, the refinement rules, and the supporting proof theories and tactics.

6.1.1. New program constructs

It is possible to add new constructs to the wide-spectrum language (or change to a different executable programming language subset). To do this, one must define abstract and concrete syntax for the new constructs and extend the definition of wp to include the new constructs. Also, one will normally want to provide window opening and closing rules for the new constructs, specifying how they interact with program window inference context. Finally, one will normally prove refinement rules to introduce (and perhaps remove) the new constructs.

6.1.2. New rules

Every refinement rule in the system is a theorem of the program window inference theory, proved using the definitions of refinement and weakest precondition. There are two ways to add a new refinement rule:

- Prove the rule from first principles, as for the built-in rules. This can be a difficult process, since it generally requires abstract reasoning about concepts such as substitution. PRT has little specialised tactic support for doing this at present.
- Derive the rule by completing a schematic refinement. It is possible to perform a refinement from an initial specification that contains metavariables, and finishing with a fragment that also contains metavariables. Such a refinement will generally introduce assumptions about the allowable bindings for those metavariables; such assumptions should be included in the initial context for the refinement. When the refinement is complete, the result is a theorem,

as described above; this theorem is in the correct form for use by PRT as a (derived) refinement rule.

6.1.3. Application theories

The Ergo design is based on a hierarchy of theories and it is relatively simple to add theories specific to particular application domains. Typically, theories are built and extended incrementally.

6.1.4. New tactics

The 'apply' command is implemented by a tactic in the program window inference theory. It is possible to write new tactics, perhaps encapsulating a recurring sequence of refinement steps [GNU92]. Since the full power of Prolog is available for writing Ergo tactics, they can perform arbitrarily sophisticated input and output, branching, looping, etc., as well as invoking refinement and proof rules.

6.2. Performance

We have completed several small case studies, including GCD [CHN+94b], and the Gray code [Mor94], and the exponentiation example in this paper. These case studies have demonstrated the usability of the tool and suggested areas for improvement. Early versions of Ergo limited the size and complexity of refinements but these problems have been overcome.

To give an idea of the amount of information in typical small refinements, we present some statistics.

	Gray	Expo	GCD
No. of refinement steps	8	6	8
No. of refinement provisos	20	15	20
No. of 'lval' provisos	15	11	15
Total no. of inference steps	363	419	559
Inference steps associated with 'lval' provisos	108	193	247
No. of refinement and proof commands in script	48	80	115
Time taken to replay the script (sec.)	253	336	669

The 'lval' provisos are the ones normally considered syntactic, such as 'x is a fresh identifier'. PRT discharges all 'lval' provisos fully automatically. A window inference step is an application of a transformation rule or a window opening rule. Each primitive step has roughly the same complexity, so should take roughly the same amount of time.

We pay a significant run-time penalty for using a logic that models syntactic obligations explicitly (though the automation means that these conditions do not burden the user). In a tool that did not do this, the automatically-discharged 'lval' provisos and the associated inference steps would disappear (or be replaced by simple side-conditions). Because we can model these conditions in the logic, we can be more confident that the conditions of proved refinement rules are correct and sufficient. Inadequate syntactic conditions are a common source of error in postulated refinement rules.

The simple automation currently used is beneficial, but more is needed. At present there is little automation for obligations involving propositional logic, arithmetic, etc. This will improve as Ergo matures and the collection of theories and tactics expands. Apart from 'lval' manipulation, most of the automatic inference steps are associated with type conditions, for which Ergo does have reasonable automation.

Abstraction and reuse are vital to managing complexity. PRT provides several facilities for structuring refinements, and significant benefits accrue from:

- Exploiting structure in application theories, so that components can be shared among developers.
- Proving theorems in application theories, to reduce the size of proofs of provisos within refinements and avoid repetition of inference patterns.
- Writing tactics that automate recurring proof patterns.
- Proving derived refinement rules.
- Writing refinement tactics.

Ultimately, good large-scale performance will be achieved only by partitioning problems: by using procedures, proving refinement lemmas, and incorporating a module system. PRT is a reasonable foundation for investigating these possibilities.

7. Conclusions

In this paper, we have investigated the roles of computer-based tools for supporting users of the refinement calculus. The analysis of roles led to a list of requirements for such support tools. We reviewed previously developed tools against these requirements before describing the new tool, PRT. PRT seeks to achieve a balance between formal rigour and usability. It is part of an ongoing research program to scale up formal methods and supporting tools to achieve industrial relevance.

There are many possible ways to extend and improve PRT. We are planning extensions in three major areas:

User interface An alternative interface using the UQ* editor [WH94] is also under development. UQ* is a generic syntax-directed editor that supports multiple interacting tools. The UQ* design is based on the 'active document' model, with a central store that can contain multiple documents and arbitrary relations between documents and parts of documents. In the UQ* version of PRT, the refinement development is a document that is created jointly by the user (via the editor) and by Ergo which is an attached tool.

Features of UQ* that make it suitable for PRT include:

- UQ* relations allow structures such as refinement trees to be imposed on documents.
- The display and navigation of such relational structures is an integral feature of the UQ* editor interface.
- UQ* supports multiple relational structures. This feature could be used to implement such features as alternative refinements.
- UQ* supports WYSIWYG output for the display of the symbols of the refinement language. This is generic, so that special symbols for application theories can be added readily.

- UQ* supports multiple tools operating on the central store. So tools such as LaTeX converters, or code collectors can be added.
- A graphical tool for attachment to UQ* is under development, which will allow the display and navigation of the refinement document at a high level, e.g., by refinement diagrams [Bac91].

Data refinement PRT has no support for data refinement. This could be added by defining (in terms of weakest preconditions — cf. [Mor94, Section 23.3.10]) a family of data refinement relations, indexed by new and old variables and coupling invariant, and prove augmentation and diminution laws. An alternative approach may be to add signatures to predicates, and define data refinement using *encoding* and *decoding* commands [BvW92].

Module structures Modularisation facilities that support data abstraction allow data to be specified and reasoned about abstractly with access to the data provided via a set of operations. This allows the user of the data to work at the level of the abstract specification of the module interface, while allowing the implementer of the module to use a more efficient representation of the data provided its external behaviour satisfies the specification. Extensions of current theory are required for developing programs that use data abstractions and that implement data abstractions.

We are also planning to exploit the generic nature of PRT by using it as the base for two different approaches to refinement. The first of these is a support tool for a refinement calculus for concurrent real-time programs. The refinement rules are similar to the conventional ones but generate precise timing constraints as well as program code. These timing constraints must be discharged either experimentally or through further proof. The second approach is to use PRT to support derivation of logic programs.

Acknowledgments

The work reported in this paper was supported by Australian Research Council grant A4-92-32859. Ray Nickson was a Research Fellow at the Software Verification Research Centre for the project leading to this paper. We wish to thank Luke Wildman for his assistance in improving the output presentation of specifications in PRT.

References

[Bac88] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[Bac91] R. J. R. Back. Refinement diagrams. In Joseph M. Morris and Roger C. Shaw, editors, *Fourth Refinement Workshop*, Workshops in Computing, pages 125–137. BCS FACS, Springer-Verlag, 1991.

[BBG+78] F.L. Bauer, M. Broy, R. Gnatz, W. Hesse, and B. Krieg-Brückner. A wide spectrum language for program development. In *3rd Int. Symp. Programming, Paris*, pages 1–15, 1978.

[BBNU96] H. Becht, A. Bloesch, R. Nickson, and M. Utting. Ergo reference manual. Technical Report 96-31, Software Verification Research Centre, The University of Queensland, 1996.

[BH95] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, April 1995.

[BHS92] R. J. R. Back, J. Hekanaho, and K. Sere. Centipede — a program refinement environment. Ser A 139, Åbo Akademi, 1992.

[BL96] M. Butler and T. Långbacka. Program derivation using the refinement calculator. In *Proceedings of the Higher Order Logic Theorem Proving and its applications workshop*, pages 93–108. Springer-Verlag, 1996.

[BM79] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[BS91] Ralph-Johan Back and Kaisa Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17–30, 1991.

[BvW90] R. J. R. Back and J. von Wright. Refinement concepts formalised in higher order logic. *Formal Aspects of Computing*, 2:247–272, 1990.

[BvW92] R. J. R. Back and J. von Wright. Predicate transformers and higher order logic. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1992.

[CAB+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+94a] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. Refinement in Ergo. Technical Report TR94-44, Software Verification Research Centre, December 1994.

[CHN+94b] David Carrington, Ian Hayes, Ray Nickson, Geoffrey Watson, and Jim Welsh. A review of existing refinement tools. Technical Report 94-8, Software Verification Research Centre, The University of Queensland, 1994.

[CHN+95] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. Requirements for a program refinement engine. In *Fourth Australasian Refinement Workshop*. School of Computer Science and Engineering, The University of New South Wales, 1995. Also available as TR-94-43, Software Verification Research Centre, The University of Queensland.

[CHN+96] David Carrington, Ian Hayes, Ray Nickson, Geoffrey Watson, and Jim Welsh. Structured presentation of refinements and proofs. In Kotagiri Ramamohanarao, editor, *Proceedings of the Nineteenth Australasian Computer Science Conference (ACSC'96)*, pages 87–96, 1996. Also available as TR-95-46, Software Verification Research Centre, The University of Queensland.

[CR88] D. A. Carrington and K. A. Robinson. A prototype program refinement editor. In *Australian Software Engineering Conference*, pages 45–63. ACS, 1988.

[CR91] D. A. Carrington and K. A. Robinson. Tool support for the refinement calculus. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification*, volume 3 of *DIMACS Series in in Discrete Mathematics and Theoretical Computer Science*, pages 381–394. American Mathematical Society, 1991.

[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.

[GM93] P.H.B. Gardiner and C. Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5(4):367–382, 1993.

[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 Jones et al. [JSD92], pages 272–297.

[Goo84] D. Good. Mechanical proofs about computer programs. Technical Report 41, Institute for Computing Science, University of Texas at Austin, 1984.

[Gor88] M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Springer-Verlag, 1988.

[Gru92] Jim Grundy. A window inference tool for refinement. In Jones et al. [JSD92], pages 230–254.

[Han92] J. Han. *A Structural Model for Methodology-Based Interactive Rigorous Software Development*. PhD thesis, University of Queensland, 1992.

[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.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System.* Springer-Verlag, 1991.

[JSD92] Cliff B. Jones, Roger C. Shaw, and Tim Denvir, editors. *Fifth Refinement Workshop*, Workshops in Computing. BCS FACS, Springer-Verlag, 1992.

[KPS+93] S. Kromodimoeljo, B. Pase, M. Saaltink, D. Craigen, and I. Meisels. The EVES system. In P. E. Lauer, editor, *Functional programming, concurrency, simulation and automated reasoning: International lecture series 1991-1992*, pages 349–373. Springer-Verlag, 1993.

[McC90] W. McCune. OTTER-2.0 user's guide. Technical Report ANL-90/9, Argonne National Laboratories, 1990.

[MMS96] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.

[Mor87] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

[Mor88] C.C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[Mor89] J.M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.

[Mor94] Carroll Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.

[MV90] Carroll Morgan and Trevor Vickers. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14:281–304, 1990. Also in *On the Refinement Calculus*.

[MV94] Carroll Morgan and Trevor Vickers, editors. *On the Refinement Calculus.* Springer-Verlag, 1994.

[Naf92] Maurice Naftalin. A model of the refinement process. In Jones et al. [JSD92], pages 211–229.

[Naf94] Maurice Naftalin. Informal strategies in design by refinement. In Maurice Naftalin, Tim Denvir, and Miquel Bertran, editors, *Industrial Benefit of Formal Methods: Second International Symposium of Formal Methods Europe*, volume 873 of *Lecture Notes in Computer Science*, pages 401–420. Springer-Verlag, 1994.

[NG94] Raymond G. Nickson and Lindsay J. Groves. Metavariables and conditional refinements in the refinement calculus. In Till [Til94], pages 167–187.

[NH95] Ray Nickson and Ian Hayes. Program window inference. Technical Report 95-29, Software Verification Research Centre, The University of Queensland, 1995.

[NH96] Ray Nickson and Ian Hayes. Supporting contexts in program refinement. Technical Report 96-29, Software Verification Research Centre, Department of Computer Science, The University of Queensland, 1996. Accepted for publication in Science of Computer Programming.

[Nic93] R. Nickson. *Tool Support for the Refinement Calculus.* PhD thesis, Victoria University of Wellington, 1993.

[NU95] Ray Nickson and Mark Utting. A new face for Ergo: Adding a user interface to a programmable theorem prover. Presented at OZCHI'95, also Technical Report TR-95-42 of the Software Verification Research Centre, The University of Queensland, 1995.

[Pau86] L. C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[PC86] D.L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, 1986.

[RS93] P. Robinson and J. Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47–61, 1993.

[RT89] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-based Editors.* Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[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.

[Til94] David Till, editor. *Sixth Refinement Workshop*, Workshops in Computing. BCS FACS, Springer-Verlag, 1994.

[vdS94] Jan L. A. van de Snepscheut. Mechanized support for stepwise refinement. In Jürg Gutknecht, editor, *Programming Languages and System Architecures*, volume 782 of *Lecture Notes in Computer Science*, pages 35–48. Springer-Verlag, 1994.

[Vic90] T. Vickers. An overview of a refinement editor. In *Proceedings of the Fifth Australian Software Engineering Conference*, pages 39–44, 1990.

[vW94] J. von Wright. Program refinement by theorem prover. In Till [Til94].

[WH94] Jim Welsh and Jun Han. Software documents: Concepts and tools. *Software—Concepts and Tools*, 15(1):12–25, 1994.

[Whi92] Keith Whitwell. A tactical environment for an interactive theorem prover. Technical Report 92-7, Software Verification Research Centre, The University of Queensland, 1992.