View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Queensland eSpace

SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 95-49

A Tool for Developing Correct Programs by Refinement

D. Carrington, I. Hayes, R. Nickson
G. Watson and J. Welsh

October 1996

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533

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

A tool for developing correct programs by refinement

D. Carrington, I. Hayes, R. Nickson, G. Watson and J. Welsh
Software Verification Research Centre
Department of Computer Science
The University of Queensland

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 a number of 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

The refinement calculus of Back [Bac88], Morgan [MV94, Mor94] and Morris [Mor87] neatly formalises the stepwise refinement ideas of Wirth [Wir71] using the weakest precondition formalism of Dijkstra [Dij76]. Using a wide-spectrum language, that incorporates both specification and executable code constructs, and a set of refinement rules, the calculus enables an abstract specification to be transformed into an executable program whose correctness depends on the correctness of each refinement step.

Developing software using the refinement calculus requires a large number of small steps; it is tedious and error-prone to do this by hand for other than small examples. To achieve larger-scale development with the refinement calculus, it seems natural to consider computer-based tool support, and a number of tools have been developed for this purpose. One of the earliest was built by Carrington and Robinson [CR91, CR88] with the Synthesizer Generator which uses attribute grammars to create language-based editors. This tool 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. 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. See [CHN+94c] for a more detailed analysis of the features of some of these tools.

In this paper, we examine the requirements for a refinement tool and describe a tool known as the Program Refinement Tool (PRT) that we have developed to meet these requirements. The distinctive features of our tool are:

- the use of a theorem prover, both for applying refinement rules and for proving obligations;
- a new logic and proof paradigm (*program window inference*) that handles context in both refinement and proof;
- an extensible theory base, supporting the Z toolkit and allowing adaptation to new application domains;
- a flexible user interface that supports construction and reuse of program derivations.

Section 2 investigates the requirements of a tool to support the refinement calculus as a program development method, and discusses design issues. Section 3 describes how the PRT tool meets these requirements, Section 4 compares PRT with other existing tools, and Section 5 summarises the results.

2 Requirements

There are several potential roles for tools that support software development with 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; however many of them involve relatively shallow proofs. The support of a sophisticated proof tool is necessary for this task.

Recording the development steps Because of the large number of steps associated with non-trivial refinements, a computer-based tool can assist by recording and managing the relationships between the stages in the development of the program being refined, and rule applications and provisos at each stage. This management should allow for extending and modifying parts of the development while protecting its integrity. By capturing the development history, which includes all the design decisions, we provide traceability from the specification to the implementation. 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 is achieved by having a single tool that integrates the multiple roles. Our objective is to create such a tool for the refinement calculus.

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'.

2.1 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.

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 predicate logic (for the pre- and post-condition of the specification statement) and Boolean expressions (for the guards of the **if** and **do** commands). We need a way to define that logic, and a way to extend it to include the data types used in specifications and programs.

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 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.

Applicability conditions

Applicability conditions are components of refinement rules. Many of them 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.

Proofs

If the tool is to manage the proof obligations 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.

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 required for discharging proof obligations.

2.2 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.

2.3 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.

From a set of potentially applicable rules, selection of a rule and (perhaps) instantiation of its parameters is required. When this is done, the rule can be applied and the resulting program fragment calculated.

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.

2.4 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 2.2). 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, we need not develop additional mechanisms for handling them.

Proof style

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]. 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 [UW94].

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.

2.5 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. Ideally, the user interface should be easy to use, so that its mechanisms do not hinder, or interfere with, the user's tasks.

The user 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 these customisations.

Two user interface styles can be compared for a refinement tool. The first style is based on a 'symbolic calculator' model which concentrates on rule application. If the tool also manages the relationships between successive refinement steps, the user is likely to be aware of the underlying representation, typically a refinement tree.

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 with hypertext links possibly providing information hiding, for example, linking each proof to the corresponding proof obligation.

2.6 The Design Goals

We now discuss the design of a refinement tool, PRT, based on the above requirements. Within this framework, a set of more specific goals was adopted as the basis of the design. These are a balance between 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 2.2 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 [MMSS95], or to encompass Back's Action Systems [BS91].

3 The Resulting Tool

3.1 Architecture

From the requirements discussed in Section 2, we developed an abstract architecture comprising:

- **A Refinement engine.** This applies the refinement transformation rules. It also generates the obligations incurred when rules are applied.
- **Proof support.** A partially automated assistant to discharge the proof obligations.

- **A Relationship Manager.** This 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.** This 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+94b, CHN+94a, CHN+96]; the technical requirements for the refinement engine are described in detail in [CHN+94b].

Combining the refinement engine and prover immediately 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.

3.2 The theorem prover and refinement engine

Window inference

In Section 2.4, we discussed the advantages of using a theorem prover based on term rewriting in a refinement tool. 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. At any stage, the proof

can contain a hierarchy of unsolved subproblems, each of which has its own context. Each node in the hierarchy is called a *window*, which is a data structure comprising:

- a *focus*, which is the term being transformed;
- some *hypotheses*, which are assumed true for the transformation of the focus;
- a *relation*, indicating what relationship is to be preserved under the transformation; and
- a *goal*, which indicates the desired outcome of transforming the focus.

Using this paradigm, a problem is solved by successively *focusing* on component subproblems. Each focus opens a new window with hypotheses appropriate for that subproblem. Within this window, the subproblem can be tackled by applying transformations and perhaps decomposing it further by opening new windows. When a subproblem is solved, for instance by reducing the focus to *true* for a proof or, in the case of refinement, transforming a specification statement to code, the window is *closed*. The final proof is thus represented by a hierarchy of windows and transformations, that together constitute a valid transformation from the original problem to the required result.

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 [UW94], 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.

Developing a logic of program window inference

Reasoning about programs requires a consideration of *states*. For example, the specification statement
$x{:}\left[\, x = a \;,\; x = b \,\right]$ 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 modelling 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 modelling of states.

Program window inference [NH95] is based on a modal logic, 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.

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 this 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.

Program variables

Program variables are modelled 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 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.

Applying refinement rules

In Ergo, an axiom or theorem can be interpreted as a directional inference 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].

Ergo axioms and theorems can be interpreted as *conditional* inference rules. In this case, 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 a table that can be extended.
- 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.

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.

3.3 The user interface

Interaction style

The detailed design of the user interface concentrated on presenting various aspects of the refinement, and in 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 2.5, with navigation via the structure of the proof.

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].

The current version of PRT uses the Ergo Emacs interface.

The display

The Emacs Ergo interface supports two styles of interaction. Initially the user is presented with the original 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. 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), which has panes that display the current state of the refinement, and the proof script frame in which 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 selection and display 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.

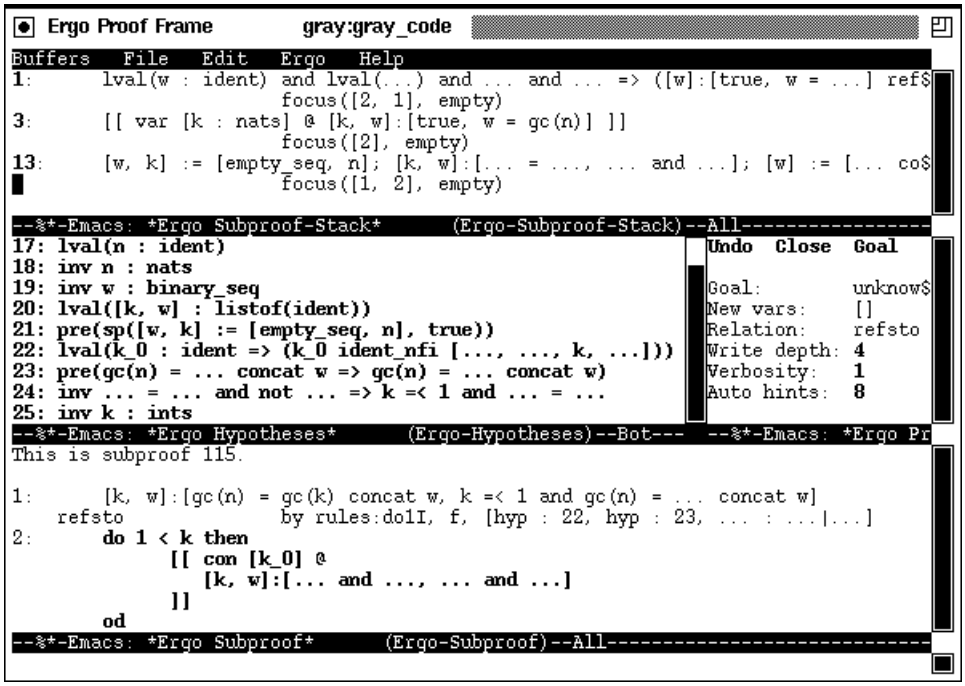

Figure 1: Emacs Ergo — Proof Frame

The components of the proof frame can be seen in Figure 1. This, and the other figures, represent stages in the refinement of a program to generate Gray codes. This is an iterative version of the example in Morgan [Mor94, pages 132ff.]. The details of the refinement are not given here — it is one of our case studies and is used as an example in the PRT User Manual [CHN+95].

The proof frame has four panes. The top pane is the subproof stack, which displays all the ancestors of the current subproof. This pane shows the layers

of the refinement. At the point in the derivation shown in Figure 1 there are three:

1 The initial specification with context;

3 A focus to the specification, which has been refined to a **var** block;

13 A focus to the specification inside the block, which has been refined to a composition of an assignment, a specification statement, and another assignment.

Note that specification statements are presented in a format similar to that of Morgan, for example:
`[k,w]:[true , w = gc(n)]`. In the display, ellipses are used for detail suppression within large terms, and can be expanded by the user if required. The bottom pane is the subproof pane which shows the steps taken in the current subproof. In Figure 1, it shows a specification statement being refined to a **do** loop, using the rule `do1I` with several hypotheses discharging the provisos. The next step in the development is to refine the specification statement inside the **do** loop to an assignment. Above the subproof pane is a pane showing the hypotheses of the current window, and also a control pane, in which global options can be set.

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 labelled `Close` and `Undo` which respectively close a window, and undo the last proof step. Opening a new window on a sub-term 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.

Finding rules

The PRT interface has a menu option to guide the user in choosing a refinement rule. 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 (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 and, at the right (as the parameter to `oterm`), 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 explicitly defer their instantiation).

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

$$\mathit{Obligation} \Rightarrow \mathit{Subject} \sqsubseteq \mathit{Result}$$

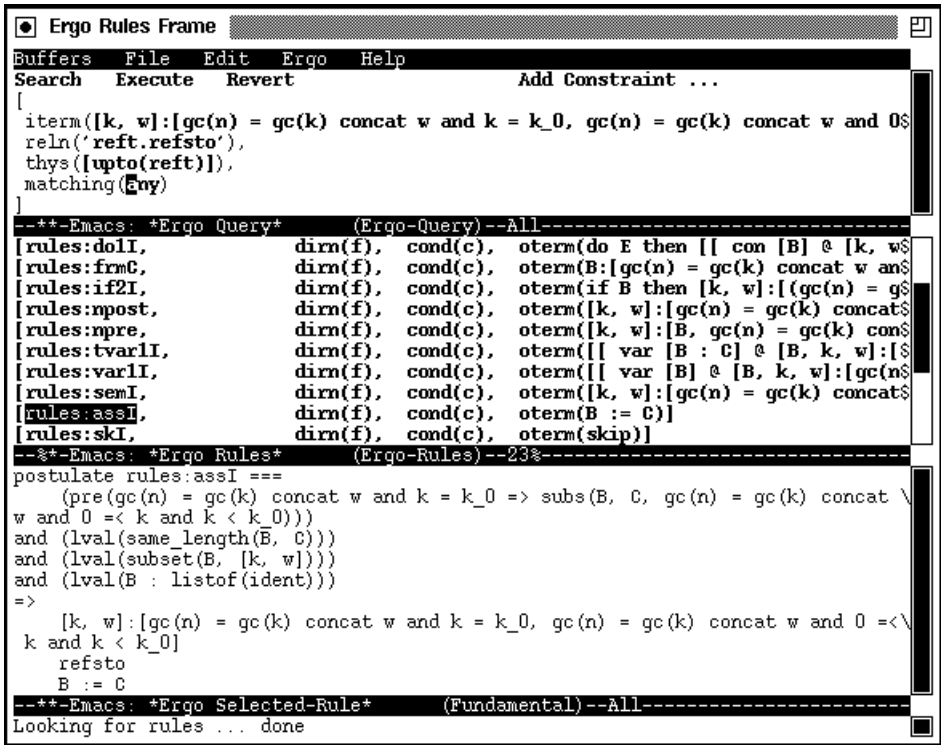

Figure 2: Emacs Ergo — Rule Frame

with *Obligation*, *Subject* and *Result* instantiated in the current context. In Figure 2, the assignment rule is selected. The mouse is then used to apply the currently selected rule, inserting a command in the proof script, and executing it.

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, the user can edit the *thys* field so that only certain theories are searched. 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.

Scripts

The proof script is a simple text file which is displayed in its own frame. It includes 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.

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 sub-term 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 'introduce assignment' rule and discharge the resulting obligation.

```
/* Having modified the pre-condition, do the assignment*/
          apply(assI).
        /* Identifiers: */ [w].
        /* Expressions: */ [(seq([k])) concat w].
        discharge(hyp:( _ )).
            [2].
              subs.
              use(jprop:sym_eq).
        close.
```

Figure 3: The Fragment of the Proof Script: $w := \langle k \rangle \frown w$

The script is automatically saved and reloaded as part of the proof. It can also be replayed line by line to recreate the proof. Such scripts allow reuse at a

basic level, since, being simple text files, they can be readily copied, combined and edited.

Browsing and printing refinements

The proof frame has two panes showing the current state of the proof tree. One shows all the steps performed in the current window, and the other shows all the ancestors of the current window. In these panes, 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 **13** in the top pane is the transformation of the specification inside the focus above (labelled **3**) to a sequence of three statements. Clicking on the **13** will bring up a browser for this transformation. The browser is a read-only display that has the same set of panes as the prover. By clicking on further subproof numbers in the browser, the user can navigate through the entire refinement.

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

$$Context \Rightarrow (Initial\ Specification \sqsubseteq Final\ Program)$$

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.

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.

Customisation

Customisation of the user interface is available at several levels. The Ergo interface is built using X-windows Emacs, 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.

3.4 Extensibility

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.

For the Gray code refinement we need a theory of Gray codes. This theory inherits theories of integer arithmetic (including `div` and `mod`) and sequences from the standard Ergo library. The theory also contains the type, definition and properties of the Gray code function and associated definitions such as the predicate "these two binary sequences differ in one just place". The treatment of Gray codes in [Mor94] relates it to the parity function, so this must also be defined. Theorems about Gray codes can then be proved in this theory, and used as lemmas in the refinement.

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 specialized tactic support for doing this at present.
- Derive the rule by completing a schematic refinement. It is possible to do 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, which 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 refinement rule.

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.

New program constructs

It is possible to add new constructs to the wide-spectrum language. 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.

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. A more practical approach may be to add signatures to predicates, and define data refinement using *encoding* and *decoding* commands [BvW92].

4 Comparisons

Existing refinement tools can be classified in a number of ways. We consider some of these classifications, using the following existing tools to illustrate the differences and allow comparisons with PRT.

CRSG	The tool built by Carrington and Robinson [CR88, CR91] using the Synthesizer Generator.
Red	The refinement calculus tool [Vic90] from Oxford University.
HOL	Tools [BvW90, Gru92, vW94] based on the HOL theorem prover.
Centipede	A tool for manipulating refinement diagrams [Bac91, BHS92].
RRE	The refinement calculus tool [GNU92, Nic93] from Victoria University of Wellington.
Proxac	A generic transformation tool with an instantiation [vdS94] for the refinement calculus.
Cogito	A methodology and suite of tools [BKKT94] that includes refinement from Z.

See [CHN+94c] for a more detailed review of some of these tools.

Depth of formalisation HOL and Cogito model specifications and programs deeply, using classical logic. Refinement rules are proved from first principles. This leads to highly trustworthy proofs that refinement developments are correct, and facilitates the use of results from conventional mathematics. On the

other hand, this deep modelling leads to notation and formalism that is sometimes cumbersome to use at the level of refinement. 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.

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 formulas 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 modelling 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 our confidence in its soundness.

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). HOL and Cogito include suitable tactics for assisting with the kinds of proofs that arise in formal development, but do not attempt to fully automate proofs. Red and Centipede do not incorporate proof support directly, but can be linked to external proof tools.

PRT supports the discharge of proof obligations using the program window inference logic. Because the applicability conditions of most rules need be proved in only a single state, 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 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. CRSG's interface is the Synthesizer Generator, so specifications, refinements and proofs are constructed by expanding templates. 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 refinement and proof engine underlying PRT is command-driven, and can be fed a script. We have two prototype user interfaces that provide support for navigation and browsing of proofs, which generate commands for the underlying engine.

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. HOL and Cogito are, 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. It is not possible to extend CRSG.

PRT can be extended by defining the weakest precondition semantics of 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 extensive library that includes first-order predicate logic, arithmetic and ZFC set theory.

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 and Cogito, the context is a formal part of the definition of constructs. The other tools do not represent context at all.

PRT uses program window inference, which has a powerful notation for representing different kinds of context, including implicit 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.

5 Evaluation

We have completed several small case studies, including GCD [CHN+94c], a symbol table [Hay93] and the Gray code [Mor94]. These case studies have

demonstrated the usability of the tool, but also its limits. In fact, it is impractical with the current version of PRT to attempt examples larger than those used in our evaluation of existing tools [CHN+94c]. This is largely because of non-linear asymptotic behaviour in the version of Ergo used — the cost of a primitive inference step depends on the number and complexity of the steps that have preceded it. The latest version of Ergo corrects this fault, and this is expected to give at least an order of magnitude improvement in the size of refinements that can be handled with PRT.

To give an idea of the amount of information in a typical small refinement, we present some statistics from the Gray code refinement, which is given in full in [CHN+95]. This does not include statistics relating to the proofs of several lemmas that were used in the development of the Gray code theory.

Number of refinement steps	9
Number of refinement provisos	21
Number of 'lval' provisos[1]	14
Total number of inference steps[2]	443
Number of automatic inference steps	377
Number of inference steps associated with 'lval' provisos	328
Number of refinement and proof commands in script	61

Notes:

1. The 'lval' provisos are the ones normally considered syntactic, such as 'x is a fresh identifier'. PRT discharges all 'lval' provisos fully automatically.

2. 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 14 automatically-discharged 'lval' provisos and the 328 associated inference steps would disappear (or be replaced by very 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 very little automation for obligations involving propositional logic, arithmetic, etc. Apart from the '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.

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.

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

[BKKT94] Anthony Bloesch, Ed Kazmierczak, Peter Kearney, and Owen Traynor. The Cogito methodology and system. In *Proceedings of the 1994 Asia-Pacific Software Engineering Conference*, pages 345–355. IEEE Computer Society Press, 1994.

[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] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. Requirements for a program refinement engine. Technical Report TR94-43, Software Verification Research Centre, November 1994.

[CHN+94c] 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] David Carrington, Ian Hayes, Ray Nickson, Geoffrey Watson, and Jim Welsh. The PRT user manual. version 1.03. Technical Report 95-56, Software Verification Research Centre, The University of Queensland, 1995.

[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)*, volume 18 of *Australian Computer Science Communications*, 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.

[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.

[Hay93] Ian Hayes. *Specification Case Studies*. Prentice-Hall, second edition, 1993.

[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.

[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] 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.

[MMSS95] Carroll Morgan, Annabelle McIver, Karen Seidel, and J W Sanders. Probabilistic predicate transformers. Technical Report PRG-TR-4-95, Oxford University Computing Laboratory, Feb 1995.

[Mor87] J. 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.

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

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

[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.

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

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

[UW94] Mark Utting and Keith Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Centre, The University of Queensland, 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.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.