Object Organisation in Software Environments for Formal Methods

Jun Han and Jim Welsh

Software Verification Research Centre
Department of Computer Science
University of Queensland, Qld 4072, Australia

Abstract. Software development by formal methods involves an overwhelming amount of technical and managerial detail. Systematic organisation of this information in a method's support environment is an important engineering concern. In this paper, we introduce a model for object organisation in software environments for formal methods, with particular emphasis on easy construction, modification, review and reuse of software objects. We demonstrate how the model can be instantiated to individual methodologies to obtain the object organisation architectures for their support environments.

1 Introduction

The value of formal methods in the development of reliable software systems is increasingly recognised. Because of its complexity, software development by formal methods requires computer-aided support. There have been many efforts to develop computer-based support systems for software development by formal methods. However, most of them concentrate on demonstrating the feasibility of automated semantic support while providing limited clerical and syntactic assistance. Few of them provide systematic support for software development from the software engineering viewpoint.

One of the major engineering concerns in providing environment support for software development by formal methods is the organisation of software objects. In principle, these objects and their organisation should record the development results, reflect the development process, and conform to the user's conceptual model of software development in the given formal method. In this paper, we introduce a model for object organisation in software environments for formal methods, with particular emphasis on easy construction, modification, review and reuse of software objects. This model provides an architectural framework for developing environments supporting software development by formal methods.

The paper is organised as follows. Section 2 overviews our approach to methodology modelling. Based on this approach, section 3 introduces our model for object organisation. Sections 4 to 6 present examples of instantiating the model to individual methodologies. Section 7 reviews related work. Finally, section 8 concludes this paper with a few further remarks on environment support.

2 An Approach to Methodology Modelling

Systematic software development requires that the software engineer follow well-established methodologies. To facilitate environment support for such methodologies, methodology models are required to capture their support requirements.

A software development methodology concerns software products and the software processes that produce these products. In modelling software development methodologies, we adopt an object-oriented approach. A software product is regarded as an object upon which various operations are performed in the course of its development. In this sense, the process or sequence of development operations permitted in the production of a software artifact is held to be inherent in the product itself. A software object is usually developed in a context composed of a large number of other software objects. These software objects are related to each other as the given methodology permits.

A software development environment supports the manipulation of the software objects and their relations allowed by the supported methodology. A methodology model defines the software objects, their relations and their manipulation as allowed by a range of methodologies.

Instantiating a model to a given methodology captures its requirements for environment support. In the instantiation process, methodology-specific meanings are assigned to the model features, and the constraints that the methodology imposes on these features are recognised. The result is an environment support architecture for the given methodology.

In our approach, environment support for methodologies' structural aspect can be achieved in a generic fashion, i.e. the support provided by a generic system according to the model can be specialised to meet the requirements of individual methodologies. Environment support for their semantic aspect is provided by additional methodology-specific tools.

3 An Object Organisation Model

Based on the above approach, we introduce a model for object organisation in software environments for formal methods. Software development by formal methods has program refinement and theorem-proving as two major sub-tasks. We first analyse each of them in its own right, and then discuss their relationships in the context of software development by formal methods.

3.1 Theorem-Proving

Proof Theory System. An interactive theorem-proving system maintains the information useful in constructing and understanding proofs. The user updates the information from time to time. Since a system is usually developed to support theorem-proving in different application domains, the information is divided into domain-specific groups. The information in one group forms *a proof theory*, and all the proof theories constitute the *proof theory system*.

Between the proof theories, there exist relations capturing the fact that one application domain is a sub-domain of, or similar to, another application domain. If the application domain of theory A contains the application domain of theory B, the information in B may all be included in A. To retain the fact that they are two separate theories and to achieve maximum reuse and consistency, an inter-theory relation may be introduced to indicate that theory A inherits all the information of theory B. We call this relation *an inheritance reference* from theory A to theory B. Concrete examples can be found in Demo2 [12], HOL [4] and Mural [7].

If theory C's domain has features similar to those of theory D's domain, certain information in D may be used in C. To realise this, an inter-theory relation of another kind may be introduced to indicate that theory C can access certain information of theory D upon translation. We call this relation *a morphism reference* from theory C to theory D. Concrete examples can be found in Mural [7].

The inheritance and morphism reference relations are called *theory dependence relations* from *child theories* to *parent theories*. The proof theory system records the individual theories and the dependence relations between them.

Proof Theory. A proof theory provides a logical context for proof problems in the concerned application domain. This logical context contains, as *elements*, symbols, axioms, definitions, theorems, inference rules, tactics, and so on. Between these elements, there exist relations capturing the fact that one element refers to another element for definitional or inferential purposes. For instance, a function may be used in stating an inference rule, and an axiom may be referred to when justifying assertions in a theorem's proof. We call such relations *reference relations* between proof theory elements. A proof theory records the theory elements and the reference relations between them.

Proof Theory Elements. A proof theory element usually has a number of *components*. Most of these components have a very simple structure. The most complicated element components are proofs of theorems.

Proof. Constructing a proof involves an inference process aimed at establishing the validity of an assertion which expresses a proof problem. This inference process is usually composed of a number of inference steps. Each step relates an assertion to other assertions according to an inference rule, in the hope that establishment of these latter assertions guarantees establishment of the former assertion. The relation from the former assertion to each of the latter assertions is called *a use relation* between assertions. This inference process continues until all the assertions involved either have immediate proof or have been related to other assertions.

A proof records each of the assertions involved and the information relevant to each proof step. The use relations between assertions are embedded in

their corresponding proof step information, and reflect the proof's construction process.

Proof Object Hierarchy. We call all the objects involved in the proof process *proof objects*, including the proof theory system, proof theories, theory elements, element components (e.g. proofs), assertions and proof step information. The inclusion relations among these objects tailor them into a proof object hierarchy. The dependence relations between proof theories, the reference relations between proof theory elements and the use relations between assertions are also structural features of interactive theorem-proving that we are interested in. These relations are embedded in the relevant proof objects.

3.2 Program Refinement

As in theorem-proving, an interactive program refinement system assists the user in manipulating all the objects involved in the refinement process, which we call *refinement objects*. By analogy, a hierarchy of refinement objects may also be formulated as an object organisation model for interactive program refinement.

Refinement Theory System. Corresponding to the proof theory system, we have the *refinement theory system* which contains all the information useful in developing and understanding refinements. This information is divided into groups specific to application domains of program refinement. Each group forms *a refinement theory*. There exist inheritance and morphism reference relations between refinement theories to capture the fact that one application domain is a sub-domain of, or similar to, another application domain. The refinement theory system records the individual refinement theories and the dependence relations between them.

Refinement Theory. A refinement theory is composed of different kinds of *elements* useful for program refinement in the concerned application domain, such as function definitions, refinement rules, applications, and so on. An application encapsulates the development of a software system in a way similar to that in which a theorem encapsulates the proof of an assertion. It has a name, a statement which specifies the targeted software system, and a refinement which records the refinement results and process of the system.

There exist reference relations between elements of a refinement theory for definitional or refinement purposes. For instance, an application's statement may be defined using functions, and its refinement steps use refinement rules.

A refinement theory records the theory elements and the reference relations between them.

Refinement Theory Elements. Elements of a refinement theory have a number of *components*. Most of these components have a very simple structure. The most complicated components are refinements of applications.

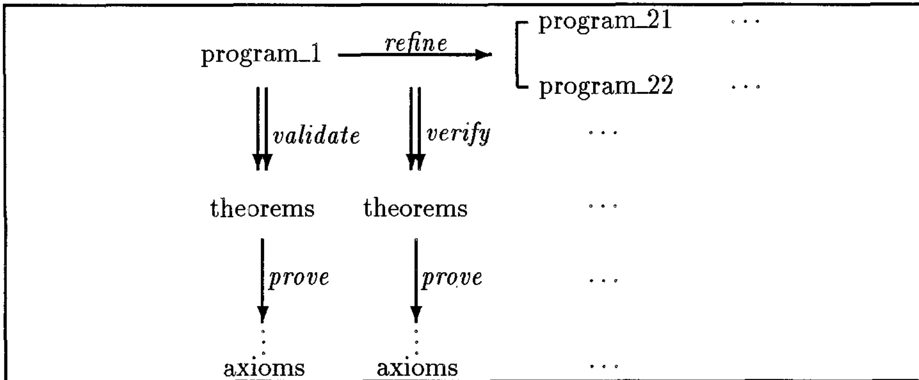

Fig. 1. Relationship between refinements and proofs

Refinement. Developing a software system in a formal method involves formal specifications and program code. In our discussion, specifications and programs are not distinguished, and they are all referred to as programs. Therefore, a program is a specification segment, a code segment or a mixture of both. A program may have additional validation obligations stating its properties.

The development of an application usually involves a number of design or refinement steps. Each step relates a program to other programs according to a refinement rule. Under the specific semantic relation determined by the refinement rule, these latter programs constitute a refinement of the former program. The relation from the former program to each of the latter programs is called *a use relation* between programs. Among the information relevant to a refinement step are the verification obligations stating the conditions that the step has to satisfy. This refinement process continues until all the programs involved are either efficient code segments or have been related to other programs.

A refinement records each of the programs involved and the information relevant to each refinement step. The use relations between programs are embedded in their corresponding refinement step information, and reflect the refinement's construction process.

3.3 Software Development by Formal Methods

As discussed above, program refinement gives rise to validation and verification obligations, which are theorems to be proved. This relationship between refinements and proofs is shown in Fig. 1.

Conceptually, program refinement (excluding the establishment of proof obligations) and theorem-proving are separate tasks. Therefore, we retain the refinement and proof object hierarchies in the support environment. They constitute the overall software object hierarchy for software development by formal methods, rooted at the *development theory system*, as shown in Fig. 2.

To realise the above relationship between refinements and proofs on this overall hierarchy, we introduce reference relations from proof obligations of programs

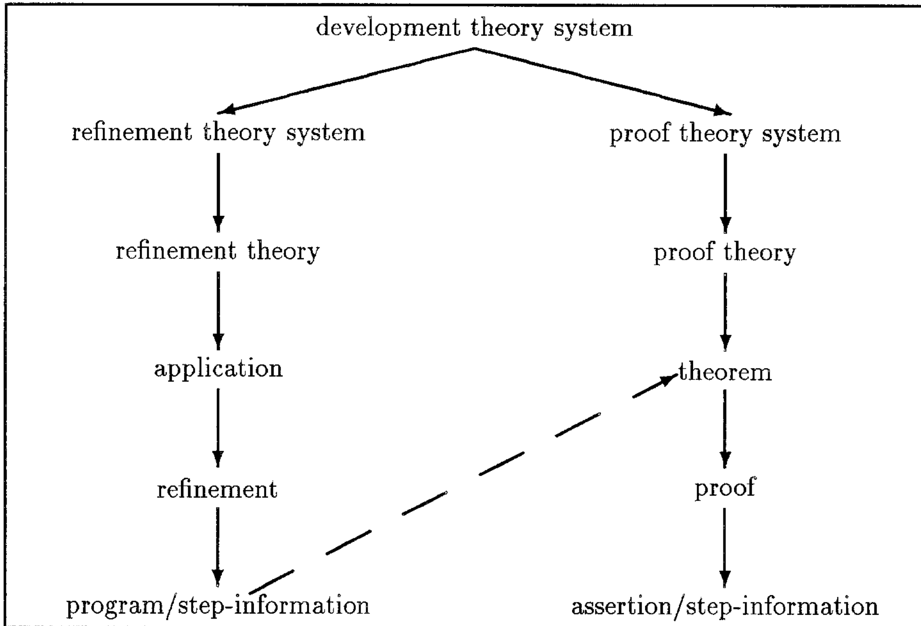

Fig. 2. Hierarchical relationship among software objects

and refinement steps on the refinement sub-hierarchy to theorems on the proof sub-hierarchy (see Fig. 2). These reference relations are called *proof allocation relations*. Under a proof allocation relation, the proof obligation concerned is regarded as being discharged by the proof of the referred theorem.

The above structural organisation of software development by formal methods supports the user's conceptual model in that the distinction between the program refinement task and the theorem-proving task is maintained by separation of refinement theories from proof theories and that the close relationship between refinements and their proofs is supported by proof allocation relations. In general, this approach gives an object organisation model for the entire software development task, which is conceptually natural and structurally simple.

This object organisation model has been instantiated to a number of theorem-proving and program refinement methodologies, to obtain their environment support architectures [5]. They include Demo2 [12], Mural [7], HOL [4], Nuprl [3], B [14] and the refinement calculus [8]. The model is also applicable to other methodologies such as VDM [6] and RAISE [9]. In the following three sections, we concentrate on the instantiations to the refinement calculus and the Demo2 theorem-proving system. We also discuss the refinement-proof relationships in using Demo2 to meet the theorem-proving requirements of the refinement calculus.

theory name	*inheritance references (to)*	*elements*
`root`		$< elements \ldots >$
`fol`	`root`	$< elements \ldots >$
`func_logic`	`fol`	$< elements \ldots >$
`computation`	`func_logic`	$< elements \ldots >$
`arithmetic`	`computation`	$< elements \ldots >$
⋮	⋮	⋮

Fig. 3. A Demo2 proof theory system

4 Demo2

Demo2 is an interactive proof editor developed based on a *window inference* approach [10, 12]. Proofs, proof theory elements, proof theories and the proof theory system in our model largely correspond to proofs, theory elements, theories and the theory hierarchy in Demo2, respectively.

4.1 Proof Theory System

The proof theory system in our model corresponds to the theory hierarchy in Demo2. Structurally, the proof theory system is composed of the model theories which correspond to the Demo2 theories on the hierarchy. All the dependence relations between theories in Demo2 are inheritance relations in our model. Except for the theory `root` which does not have parent theories, all other theories have exactly one parent theory.

Figure 3 shows an example proof theory system in Demo2. The theory `root` contains the basic facts about theorem-proving. The theory `fol` is a theory of first order logic. The theory `func_logic` is a theory of functional logic [11]. The theory `computation` is a theory of computation in functional logic. The theory `arithmetic` is a theory of arithmetic. The inheritance relations between these theories are also shown in Fig. 3.

4.2 Proof Theory

A theory in Demo2 comprises the information available for proof construction in a particular application domain. A proof theory in our model corresponds to a Demo2 theory, and may contain additional information such as tactics. The following are some typical theory elements: primitive and defined quantifiers, primitive and defined functions, window opening rules, hypothesis splitting rules, simplification rules, axioms, theorems, and tactics.

Reference relations exist among elements of a proof theory, including the inferential reference relations among axioms, theorems and inference rules. Demo2 enforces strict "proof-before-use", i.e. only axioms and theorems with complete proofs can be used to prove other theorems and to introduce inference rules.

theory name: `arithmetic` *inheritance references (to)*: `computation`

theory elements:

name	*type*	*contents*
`is_int`	prim_fun	(1,default,`A`)
$+$	prim_fun	(2,500,`yfx`)
$*$	prim_fun	(2,490,`yfx`)
$<$	prim_fun	(2,520,`xfx`)
$>$	def_fun	(2,520,`xfx`); $[x_1, x_2], x_2 < x_1$
$\geqslant$	def_fun	(2,520,`xfx`); $[x_1, x_2], (x_1 > x_2) \vee (x_1 = x_2)$
`ltsucc`	axiom	`is_int`$(A) \Rightarrow A < A + 1$
`posinc`	axiom	$A \geqslant 0 \Rightarrow A + 1 \geqslant 1$
`geonem`	axiom	$A < B \wedge C \geqslant 1 \Rightarrow A < B * C$
`th`	theorem	`is_int`$(S) \wedge S \geqslant 0 \Rightarrow$ $(S \geqslant 0) \wedge (S < (S+1) * (S+1))$ < *proof*... >
⋮	⋮	⋮

Fig. 4. Demo2 theory `arithmetic`

Figure 4 shows some elements of the Demo2 theory `arithmetic`, where the theorem proofs are omitted. Note that a primitive function has, as its contents, an arity, a precedence and an associativity. A defined function has, in its contents, an additional definition term preceded with a list of object variables. An axiom has a statement as its contents. A theorem has a statement and a proof as its contents. The use of the defined function $\geqslant$ in the statement of the theorem `th` is an example of definitional references between proof theory elements. In the next subsection, we will see examples of inferential references.

4.3 Proof

The construction of a proof in Demo2 is carried out primarily by a sequence of goal-directed equivalence transformations in a logical window. Each transformation is either simple or complex. If the user transforms a window from one version to another version by a simple operation such as appealing to an axiom or a hypothesis, this transformation is classified as simple. On the other hand, a complex transformation achieved by transforming a term relevant to the current window version requires justification of how the term transformation is performed. The justification itself is a proof carried out in a nested subwindow and is composed of a sequence of simple and complex transformations.

A proof in our model corresponds to a Demo2 proof, with assertions corresponding to window versions and proof steps to transformation steps. Therefore, a model proof for Demo2 is composed of a linear sequence of proof steps. Except for the initial window version, all other assertions are generated from proof steps. An assertion has a focus F, a goal G, a number of hypotheses H, and an

1_1^1: $\vdash ((\mathtt{is_int}(S) \land S \geqslant 0 \Rightarrow (S \geqslant 0) \land (S < (S+1) * (S+1))) \Leftrightarrow \mathtt{true})$

openwin on $(S \geqslant 0) \land (S < (S+1) * (S+1)$

2_1^1: $\mathtt{is_int}(S), S \geqslant 0 \vdash ((S \geqslant 0) \land (S < (S+1) * (S+1)) \Leftrightarrow \mathtt{true})$

openwin on $S \geqslant 0$ of focus

3_1^1: $\mathtt{is_int}(S), S \geqslant 0, S < (S+1) * (S+1) \vdash ((S \geqslant 0) \Leftrightarrow \mathtt{true})$

trans focus by hyp 2

3_1^2: $\mathtt{is_int}(S), S \geqslant 0, S < (S+1) * (S+1) \vdash (\mathtt{true} \Leftrightarrow \mathtt{true})$

closewin

2_1^2: $\mathtt{is_int}(S), S \geqslant 0 \vdash ((S < (S+1) * (S+1)) \Leftrightarrow \mathtt{true})$

lemma proof $S < S+1$

3_2^1: $\mathtt{is_int}(S), S \geqslant 0 \vdash ((S < S+1) \Leftrightarrow \mathtt{true})$

trans_true focus by ltsucc and hyp 1

3_2^2: $\mathtt{is_int}(S), S \geqslant 0 \vdash (\mathtt{true} \Leftrightarrow \mathtt{true})$

closelemma

2_1^3: $\mathtt{is_int}(S), S \geqslant 0,\ 1: S < S+1 \vdash ((S < (S+1) * (S+1)) \Leftrightarrow \mathtt{true})$

lemma proof $S+1 \geqslant 1$

3_3^1: $\mathtt{is_int}(S), S \geqslant 0, S < S+1 \vdash ((S+1 \geqslant 1) \Leftrightarrow \mathtt{true})$

trans_true focus by posinc and hyp 4

3_3^2: $\mathtt{is_int}(S), S \geqslant 0, S < S+1 \vdash (\mathtt{true} \Leftrightarrow \mathtt{true})$

closelemma

2_1^4: $\mathtt{is_int}(S), S \geqslant 0,\ 1: S < S+1,\ 1: S+1 \geqslant 1$
$\vdash ((S < (S+1) * (S+1)) \Leftrightarrow \mathtt{true})$

trans_true focus by geonem, hyp 3 and hyp 8

2_1^5: $\mathtt{is_int}(S), S \geqslant 0,\ 1: S < S+1,\ 1: S+1 \geqslant 1 \vdash (\mathtt{true} \Leftrightarrow \mathtt{true})$

closewin

1_1^2: $1: \mathtt{is_int}(S) \land S \geqslant 0 \Rightarrow (S < S+1),\ 1: \mathtt{is_int}(S) \land S \geqslant 0 \Rightarrow (S+1 \geqslant 1)$
$\vdash (\mathtt{true} \Leftrightarrow \mathtt{true})$

Fig. 5. th's Demo2 proof

equivalence relation $\equiv$. It defines an intention to prove that given the hypotheses, the focus and the goal are equivalent according to the equivalence relation: $H \vdash (F \equiv G)$. The focus, goal and hypotheses are all logical terms. The equivalence relation can be logical equivalence, arithmetic equality, and so on. The information attached to a proof step records the proof command applied, including references to theory elements such as axioms and theorems. A complex proof step also has a nested justification proof.

Figure 5 shows a Demo2 proof of the theorem `th` in the theory `arithmetic` in a condensed form. At the beginning of the proof, the initial assertion labelled 1_1^1 is introduced. The proof is carried out top-down, and is recorded with assertions to the left and (explanations of) proof operations to the right in an interleaved manner. Because the proof operations are self-explanatory, we shall not go through the proof process step by step. Note that the use of the axiom `ltsucc` in the above proof is an example of inferential references between theory elements.

The modelling exercise in this section shows that the organisation require-

theory name	*inheritance references (to)*	*elements*
⋮	⋮	⋮
`list`	...	$< elements \ldots >$
`arith`	`list`	$< elements \ldots >$
`sorting`	`list`	$< elements \ldots >$
`UI`	...	$< elements \ldots >$
`language`	`arith, sorting`	$< elements \ldots >$
`environment`	`language, UI`	$< elements \ldots >$

Fig. 6. A refinement theory system for the refinement calculus

ments of Demo2 can be naturally met by our model. Like the current Demo2 system, our model supports object construction. In addition, the modelling suggests improvements as to easy editing, checking, review, replay and reuse of proof objects by recording the object structures and inter-object relations.

5 Refinement Calculus

The refinement calculus is a formal method for software development, developed primarily by Back, Morgan and Morris. In the following discussion, we follow Morgan's presentation of the method in [8].

Experimental tools have been developed to support the refinement calculus [2, 13, 1]. Because of their experimental nature, these tools concentrate on the support for the core refinement activities, and give little consideration to the object organisation issue. Instead of modelling an existing system, therefore, we propose an object organisation system for the refinement calculus.

5.1 Refinement Theory System

The refinement theory system for the refinement calculus is composed of individual refinement theories specific to application domains. For simplicity, only inheritance relations between these theories can be introduced to allow one theory to inherit all the information of another theory.

Figure 6 shows a refinement theory system for the refinement calculus and contains a number of refinement theories: `list` for list-processing, `arith` for arithmetics, `sorting` for sorting algorithms, `UI` for user interface applications, `language` for processing programming languages, and `environment` for developing programming environments. The inheritance relations between these theories are also shown in Fig. 6.

5.2 Refinement Theory

A refinement theory for the refinement calculus contains the information available for program refinement in a particular application domain, and is composed

theory name: `arith`		*inheritance references (to)*: `list`
theory elements:		
name	*type*	*contents*
$+$	prim_fun	$(2,500)$; $\mathbb{R} + \mathbb{R} : \mathbb{R}$
$*$	prim_fun	$(2,490)$; $\mathbb{R} + \mathbb{R} : \mathbb{R}$
$\sqrt{\ }$	def_fun	$(1,480)$; $\sqrt{\mathbb{R}} : \mathbb{R}$; $\sqrt{a} = b$ **iff** $b \geqslant 0 \wedge b * b = a$
$\lfloor\ \rfloor$	def_fun	$(1,480)$; $\lfloor \mathbb{R} \rfloor : \mathbb{Z}$; $\lfloor a \rfloor = b$ **iff** $b \leqslant a < b + 1$
$\mathtt{assn}_1$	rule	**if** $pre \Rrightarrow post[w \backslash E]$, **then** $w, x : [pre, post] \sqsubseteq w := E$
`sqrt`	app	$\lbrack\!\lbrack$ **var** $r, s : \mathbb{N} \bullet r : [\mathbf{true}, r = \lfloor \sqrt{s} \rfloor]$ $\rbrack\!\rbrack$
		< *refinement*... >
⋮	⋮	⋮

Fig. 7. Refinement theory `arith`

of various elements. The following are some typical elements in a refinement theory: primitive functions, defined functions, refinement rules and applications. Definitional and refinement reference relations exist among these elements. For example, the primitive and defined functions may be used to define other functions and to write programs involved in applications. The refinement rules are used to carry out refinement steps of applications. One application may be used in another application's refinement as part of the development.

Figure 7 shows some of the elements in the refinement theory `arith`, where the refinements of applications are omitted. Note that a primitive function has, as its contents, an arity-precedence pair and a concrete syntax in terms of the types of the function's parameters and result. A defined function has an additional definition in its contents. A refinement rule has a rule statement as its contents. An application has a statement (i.e. an abstract program) and a refinement as its contents. The use of $\lfloor\ \rfloor$ in the statement of the application `sqrt` is an example of definitional reference between refinement theory elements. An example of refinement references can be found in the next subsection.

5.3 Refinement

Constructing a refinement in the refinement calculus involves many steps. Each step refines a program by applying a refinement rule, and generates a number of other programs. The refinement records all the programs and refinement steps.

The refinement calculus embeds a specification mechanism, the *specification statement*, in Dijkstra's Guarded Command Language. As such, a program in the refinement calculus may contain both abstract and executable constructs, and is a simple or complex statement[1]. A simple statement is an assignment statement

[1] For simplicity, we only consider those programs that do not contain validation obligations or advanced language features such as procedures and modules.

or a specification statement. The specification statement has the form:

$w : [pre, post]$

where w is the *frame* containing the variables whose values may be changed by the statement, *pre* is the *precondition* describing the initial state of the program, and *post* is the *postcondition* describing the final state of the program. A complex statement is composed of a number of statements organised by one of the following constructs: sequential composition (;), alternation (**if fi**), iteration (**do od**) and local block (|[ ]|). A local block may contain variable (**var**), invariant (**and**) and logical constant (**con**) declarations. A program in our model corresponds to a program in the refinement calculus, except that a model program may have an *environment* to record the declaration information of its enclosing context when it is isolated from this context: [*environment*] *statement*.

A refinement step in the refinement calculus is carried out by application of a refinement rule, and may generate verification obligations. A step which isolates some components of a program without functional refinement is also regarded as a refinement step. For instance, the body of an iteration may be isolated from the overall iteration. The refinement operation applied and the verification obligations generated (if any) constitute the information attached to the refinement step. The refinement operation contains, among other things, a reference to the refinement rule applied.

To demonstrate the modelling of refinements in the refinement calculus, we consider an example taken from [8]. The refinement problem is that we are given a natural number s and required to set the natural number r to the greatest integer not exceeding $\sqrt{s}$. It is formulated as the application `sqrt` in the refinement theory `arith` (see Fig. 7). The abstract program specifying this problem is

$$|[\ \mathbf{var}\ r, s : \mathbb{N} \bullet r : [\mathbf{true}, r = \lfloor\sqrt{s}\rfloor]\ ]| \tag{1}$$

It is the initial program of the refinement and is introduced at the beginning of the refinement process. After a number of refinement steps, this abstract program is refined to code (see pages 70-73 of [8] for details):

$$\begin{array}{l}
|[\ \mathbf{var}\ r, s : \mathbb{N} \bullet \\
\quad |[\ \mathbf{var}\ q : \mathbb{N} \bullet \\
\quad\quad q, r := s + 1, 0; \\
\quad\quad \mathbf{do}\ r + 1 \neq q \longrightarrow \\
\quad\quad\quad |[\ \mathbf{var}\ p : \mathbb{N} \bullet \\
\quad\quad\quad\quad p := (q + r) \operatorname{div} 2; \\
\quad\quad\quad\quad \mathbf{if}\ s < p^2 \longrightarrow q := p \\
\quad\quad\quad\quad [\!]\ s \geq p^2 \longrightarrow r := p \\
\quad\quad\quad\quad \mathbf{fi} \\
\quad\quad\quad]| \\
\quad\quad \mathbf{od} \\
\quad]| \\
]|
\end{array} \tag{2}$$

Due to space limitation, we shall not give all the details of the refinement process (see [5]). The following is one of the refinement steps. Program (3):

$$[\mathbf{var}\, q, r, s : \mathbb{N}]q, r : [\mathbf{true}, r^2 \leqslant s < q^2] \tag{3}$$

is refined to program (4):

$$[\mathbf{var}\, q, r, s : \mathbb{N}]q, r := s + 1, 0 \tag{4}$$

using the refinement rule $\mathbf{assn_1}$ (see Fig. 7). The verification obligation of this refinement step is

$$[\mathbf{var}\, q, r, s : \mathbb{N}]\, \mathbf{true} \Rrightarrow 0 \leqslant s < (s+1)^2 \tag{3.O$_1$}$$

Note that the use of the refinement rule $\mathbf{assn_1}$ in this step is an example of refinement references between theory elements.

As with the modelling of Demo2, the modelling exercise in this section shows that our model can effectively meet the organisation requirements of the refinement calculus.

6 Refinement Calculus and Demo2

The above refinement example has shown that applying certain refinement rules gives rise to verification obligations. In general, validation obligations may also be stated against programs. According to our model, discharging these proof obligations is a theorem-proving task and should be carried out in proof theories.

If we choose Demo2 as the theorem-proving system for the refinement calculus, its development theory system is composed of a refinement theory system for the refinement calculus and a proof theory system for Demo2. For example, we may combine the refinement theory system and the proof theory system given in previous examples, into a development theory system.

In a development theory system for the refinement calculus and Demo2, the proof obligations of refinements may be discharged by proving theorems in Demo2 theories. The relationships between them are captured by proof allocation relations. For instance, we may introduce a proof allocation relation from the verification obligation $(3.\mathrm{O}_1)$ of `sqrt`'s refinement in the refinement theory `arith` to the theorem `th` in the Demo2 theory `arithmetic`. Proving this theorem (as shown earlier) discharges the verification obligation $(3.\mathrm{O}_1)$.

Carrington and Robinson's refinement editor [2] uses Bill Pugh's demonstration proof editor **pv** for theorem-proving, while Back's refinement diagram editor [1] is developed on top of the HOL proof generating system. The distinction between program refinement and theorem-proving tasks is not fully supported in either of these two systems. In contrast, Vickers' refinement editor [13] separates program refinement from theorem-proving completely. The natural relationship between refinements and their proofs is lost.

7 Related Work

Our object organisation model for software development by formal methods is developed based on investigations into existing theorem-proving and program refinement methodologies/systems. Many theorem-proving systems have a mechanism to organise proof information into proof theories. In particular, Mural provides one of the most advanced organisation mechanisms called the theory store, which corresponds to the proof theory system in our model. The concept of proof theories as a basic mechanism for organising proof information has also inspired us to organise refinement information into refinement theories.

In dealing with the relationships between theorem-proving and program refinement, there have been two major approaches. One is to separate them, and the close relationship between refinements and their proofs is not supported. The other is to regard them as a single task based on theorem-proving, and the distinction between the program refinement task and the theorem-proving task is not recognised. As our model suggests, we argue that both the close relationship between refinements and their proofs and the distinction between the program refinement and theorem-proving tasks should be supported to conform to the user's conceptual model. Mural provides such support to a certain degree by relating programs/refinement-steps to proof theories.

Most of the existing systems provide methodology-specific object organisation. The issues involved are not addressed systematically. In particular, there are very limited provisions for easy modification, review and reuse of software objects, and no provisions for refinement organisation by refinement theories.

8 Conclusions

In this paper, we have introduced a model for object organisation in software environments for formal methods. It centres on a hierarchy of software objects, including two related sub-hierarchies for program refinement and theorem-proving. Proof allocation relations are introduced to capture the structural relationships between program refinement and theorem-proving. They tailor the two sub-hierarchies into the overall hierarchy in a simple and consistent manner. The software objects record the development results and reflect the development processes. Their definition and organisation conform to the user's conceptual model of software development in given formal methods.

The proposed model has been systematically instantiated to a number of theorem-proving and program refinement methodologies. These instantiations have captured the relevant methodologies' object organisation requirements, and have consequently reflected their environment support architectures. To this end, our model provides an architectural framework for developing generic and methodology-specific environments supporting software development by formal methods.

A more comprehensive account of the model can be found in [5]. It contains detailed definition of object structures, operations and consistency, and also deals

with object presentation. After some prototype experiment [5] and a feasibility study, we are currently developing a generic, methodology-based environment for software development by formal methods.

Acknowledgements

We would like to thank David Carrington, Ian Hayes, Peter Lindsay, Peter Robinson, John Staples and Nigel Ward for their comments and help.

References

1. R.J.R. Back. Refinement diagrams. In *Proc. 4th BCS-FACS UK Refinement Workshop*, pages 125–137, Cambridge, UK, January 1991.
2. D. Carrington and K. Robinson. A prototype program refinement editor. In *Proc. 3th Australian Software Engineering Conf.*, pages 45–63, Canberra, Australia, May 1988.
3. R.L. Constable, S.F. Allen, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.
4. M.J.C. Gordon. HOL: A proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, MA, 1988.
5. J. Han. *A Structural Model for Methodology-based Interactive Rigorous Software Development.* PhD thesis, University of Queensland, St. Lucia, Australia, 1992.
6. C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall International, London, second edition, 1990.
7. C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. mural*: A Formal Development Support System.* Springer-Verlag, London, 1991.
8. C. Morgan. *Programming from Specifications.* Prentice-Hall International, London, 1990.
9. M. Nielsen, K. Havelund, K.R. Wagner, and C. George. The RAISE language, method and tools. *Formal Aspects of Computing*, 1(1):85–114, 1989.
10. P.J. Robinson and J. Staples. Formalising the hierarchical structure of practical mathematical reasoning. Technical Report 138, Department of Computer Science, University of Queensland, St. Lucia, Australia, December 1989.
11. J. Staples. Functional logic for program verification: Introductory lectures. Technical Report 168, Department of Computer Science, University of Queensland, St. Lucia, Australia, July 1990.
12. T.G. Tang, P.J. Robinson, and J. Staples. The demonstration proof editor Demo2. Technical Report 175, Department of Computer Science, University of Queensland, St. Lucia, Australia, April 1991.
13. T. Vickers. An overview of a refinement editor. In *Proc. 5th Australian Software Engineering Conf.*, pages 39–44, Sydney, Australia, May 1990.
14. T. Vickers. An overview of a theorem proving assistant. In *Proc. 13th Australian Computer Science Conf*, pages 402–411, Melbourne, Australia, February 1990.