

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

Queensland 4072
Australia

TECHNICAL REPORT

No. 93-17

Methodology Modelling: Combining Software Processes with Software Products

Jun Han and Jim Welsh

August 1993

Phone: +61 7 365 1003

Fax: +61 7 365 1533

Methodology Modelling: Combining Software Processes with Software Products

Jun Han and Jim Welsh
Software Verification Research Centre
Department of Computer Science
University of Queensland, Qld 4072, Australia
phone: +61 7 3651631, fax: +61 7 3651999, e-mails: {han,jim}@cs.uq.oz.au

Abstract

Software processes and software products are the two inseparable issues that a software development methodology addresses. Based on the notion of software objects which combine software processes with software products, we introduce a methodology-oriented approach to process modelling, to facilitate effective environment support for software development. Using this approach, we develop a process model for rigorous software development. We demonstrate that this model can be instantiated to individual methodologies to capture their software processes.

1 Introduction

The significance of software processes in improving the quality of software products has been widely recognised for some time. Identification, improvement, representation and environment support for software processes are major areas of concern. This has been demonstrated by the recent work on process modelling. The approaches to process modelling are quite diverse: from activity-based to product-based. This diversity is due to the treatment of the relationships between software processes and software products. The strategies adopted greatly affect the effectiveness of the process models, in terms of granularity, prescriptiveness, enactability, manageability and applicability.

This paper introduces a methodology-oriented approach to process modelling, based on the notion of software objects which combine software processes with software products. Our objectives are as follows.

1. Coherent support for software processes and software products. We believe that unified

support for software processes and software products is a major factor in improving software quality.

2. Fine-grained, non-prescriptive and definitive support for software processes. This is to enable unconstrained and automated support for the software engineer's personal technical process.
3. Methodology-based support for the software engineer's conceptual model of software development. Systematic software development by following well-established methodologies is essential to achieve manageable software processes.

The next section gives an overview of our approach to process modelling. Using this approach, section 3 develops a process model for rigorous software development. Section 4 gives examples of instantiating this model to individual methodologies to capture their software processes. Section 5 makes a comparison with related work. Finally, section 6 concludes this paper with a few further remarks on environment support and future directions.

2 An approach to process modelling

A software development methodology concerns the software products and the software processes. Generic environment support for such methodologies requires methodology models. A methodology model equally addresses software processes and software products, and therefore is a process model in a more general sense.

In modelling software development methodologies, we adopt an object-oriented approach. In this

approach, the software process or sequence of development operations, that a given methodology permits on a software product, is inherent in the product itself. We call such a software product a *software object*. What a software development environment supports is the manipulation of all software objects allowed by the underlying methodology. A methodology model defines the software objects and their manipulation required by a range of methodologies.

Software objects. The above presentation of our object-oriented approach gives a dynamic view of software objects, i.e., a software object is developed through a sequence of operations and the application and effect of these operations are captured by the software object itself. An alternative, static view is that a class of software objects can be defined by their structure and operations. A structure instance for a specific software object reflects that object's development process by recording the initial, intermediate and final results as well as the development relations between them.

The operations applicable to a software object can be classified into *construction*, *editing* and *query* operations. Construction operations introduce new features into the object by creating them. Editing operations make changes to the object based on existing features of itself and/or other objects. Query operations access existing features of the object, but do not change them. The construction and editing operations of an object do not create or destroy the object as a whole, but its components. The object itself is created or destroyed as a component of its enclosing object and by the operations of this enclosing object.

At any moment, the user perceives the structure of a software object as a record of the (partially ordered) natural sequence of construction operations by which the user would construct the object, as now understood, from scratch. Editing operations enable the user to adjust this natural sequence, as better understanding of the software object is attained. Query operations provide mechanisms to extract components of the software object for purposes like review and reuse. As such, the editing and query operations that have been applied are of no subsequent interest to the user or any other reviewer of the software object, and we consider the software object's structure to be determined by the construction operations.

In a methodology model, the object structures should be general enough and the object operations should be primitive enough, to represent all

possible requirements of the targeted methodologies.

Model instantiation. By instantiating a model to a given methodology, we capture its requirements for environment support. In the instantiation process, methodology-specific meanings are assigned to the model features, and the constraints that the methodology has on these features are recognised. This includes formulation of object structures and definition of object operations in the context of the given methodology. The result of this instantiation is an environment support architecture for the given methodology.

Structural and semantic aspects. A model captures the structural aspect of the targeted methodologies. Its features are given semantic meanings in the instantiation process. In an environment, such semantic support is provided by methodology-specific tools.

Methodology-specific constraints. Object structures in a given methodology may be constrained and become simpler than those suggested by a model. Object operations in the given methodology may also be constrained as to their applicability.

Model and methodology-specific operations. Several operations in a given methodology may correspond to one primitive model operation, if the difference between them has no structural implication in the model. Conversely, a methodology-specific operation may be composed of a number of primitive model operations. We assume that the mechanisms for operation composition are the constructs of a general-purpose programming language.

3 A process model for rigorous software development

Software development by formal methods embraces formal specification and verified design. A formal specification describes precisely the (functional) requirements for a software system in a mathematical notation. It may be validated by deriving expected properties of the system.

Development of a software system from its top-level formal specification, i.e., the most abstract program¹, usually involves many design or refinement steps. Each step implements an abstract specification with a more concrete specification by

¹We do not distinguish specifications from programs.

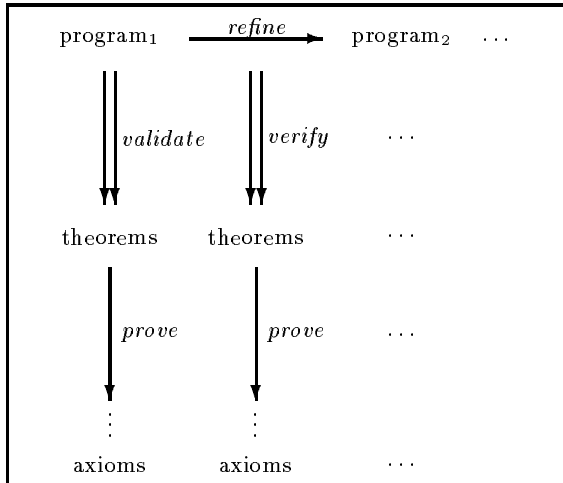


Figure 1: Relationship between refinements and proofs

making a design decision. To verify that this concrete specification satisfies the abstract specification, certain proof obligations must be discharged.

Discharging a validation or verification obligation involves proving a theorem and often requires a formal reasoning process of many steps. Refinements and proofs are all software objects involved in rigorous software development. The relationship between refinements and proofs is shown in Figure 1.

In the remaining part of this section, we introduce a simplified process model for rigorous software development by concentrating on simple proofs and refinements. (Discussions of more complex proofs and refinements with nested and encapsulated subordinate proofs and refinements can be found in [3].)

3.1 Proofs

Constructing a proof involves an inference process aimed at establishing the validity of an assertion which expresses a proof problem. This process is usually composed of a number of inference steps. Each of these steps 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. This inference process continues until all the assertions involved either have immediate proof or have been related to other assertions.

The assertion which describes the initial proof problem is the *main assertion*. If assertions a_1, a_2, \dots and a_n are used in an inference step to es-

tablish another assertion a , we call a_1, a_2, \dots and a_n the *sub-assertions* and a the *super-assertion*. We also say that $a_i (i = 1, 2, \dots, n)$ is *used* in a 's proof or by a . The relation from a to $a_i (i = 1, 2, \dots, n)$ is called a *use relation* between assertions.

3.1.1 Construction

The following are the primitive construction operations for proofs.

Assertion introduction introduces an assertion into a proof. For instance, the main assertion of a proof is usually introduced at the beginning of the proof process.

*Decomposition inference*² carries out an inference step by applying an inference rule to a number of existing assertions, one of which is the super-assertion while the remaining (possibly zero) are sub-assertions. A number of new sub-assertions are generated.

Composition inference carries out an inference step by applying an inference rule to a number of existing assertions, which are sub-assertions. A super-assertion and some (possibly zero) new sub-assertions are generated.

Connection inference carries out an inference step by applying an inference rule to a number of existing assertions, one of which is the super-assertion while the remaining (one or more) are the sub-assertions. No further sub-assertions are generated.

The systems Nuprl [1] and Demo2 [15] support decomposition inference, while Mural [7] supports all the three inference styles.

3.1.2 Structure

The structure of a proof should in principle record all the assertions and inference steps involved in the proof process. The inference steps reflect the proof process by relating the assertions in a necessary order. The above analysis of the proof construction process suggests that a directed graph structure is appropriate to represent a proof with assertions as vertices and use relations between assertions as edges.

In a meaningful proof, an assertion should never directly or indirectly use itself. This implies that the proof representation graph is acyclic.

²We choose not to use the traditional terminology *backward inference* or *goal-directed inference* because "decomposition" is closer to the terminology the software engineer uses. Similar arguments also apply to other kinds of inference that follow.

Definition 1 (Proof structure) *The representation structure of a proof is an acyclic directed graph $\mathcal{P} = \langle \mathcal{V}^p, \mathcal{E}^p \rangle$.*

\mathcal{V}^p is a set of vertices denoting the assertions involved in the proof. An assertion $v \in \mathcal{V}^p$ if and only if it is introduced into the proof by assertion introduction, decomposition inference or composition inference.

\mathcal{E}^p is a set of edges denoting the use relations between the assertions. A use relation $\langle v_1, v_2 \rangle \in \mathcal{E}^p$ if and only if assertion v_2 is used by assertion v_1 under decomposition, composition or connection inference. All the edges originating from a single assertion vertex represent a proof step relation which records the inference operation applied.

The different types of vertices and edges are distinguishable in terms of the operations which introduce them.

This proof structure records the proof results and the necessary proof process. Note that recording the order of commutable proof steps is regarded as unnecessary.

3.1.3 Editing and query

The following are the editing and relevant query operations applicable to proofs.

Assertion deletion removes an assertion from a proof. *Assertion copying* copies an assertion into a buffer. *Assertion pasting* pastes a buffered assertion into a proof.

Proof step deletion removes a proof step relation from a proof. *Proof step copying* makes a copy of a proof step relation in a buffer. *Proof step pasting* pastes a buffered proof step relation into a proof.

Assertion attachment connects an assertion to an incomplete proof step relation as a super- or sub-assertion.

The deletion, pasting and attachment operations are editing operations, and the copying operations are query operations. In a given methodology, editing and even construction operations may introduce inconsistency into a proof with respect to its structure definition. Such inconsistency, if allowed, may be rectified by further operations.

Most existing proof systems concentrate on mechanisation of proof procedures and provide little support for proof editing. In general, providing more editing operations such as those listed above is necessary to achieve more effective support.

3.2 Refinements

The refinement process starts with a formal specification of a software system (i.e., an abstract program), experiences a number of verified refinement steps, and produces the executable code which satisfies the initial specification. Each refinement step relates a program to a number of 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.

We call the initial system specification the *main program*. If programs p_1, p_2, \dots and p_n are organised together as a refinement of another program p in a refinement step, we call p_1, p_2, \dots and p_n the *sub-programs* and p the *super-program*. We also say that $p_i (i = 1, 2, \dots, n)$ is *used* in p 's refinement or by p . The relation from p to $p_i (i = 1, 2, \dots, n)$ is called a *use relation* between programs.

3.2.1 Construction

Program refinement and proof construction are both problem-solving tasks. Analysis of their processes suggests that they have strong structural similarity, with programs corresponding to assertions and refinement steps to inference steps. Corresponding to the proof construction operations, we have the following refinement construction operations: *program introduction*, *decomposition refinement*, *composition refinement* and *connection refinement*. However, these refinement operations have the following special features.

1. A program is a segment of formal specification, a segment of code, or a mixture of both. Unlike assertions, a program may have a number of additional validation obligations stated by the user. Each of them has a statement and a proof. The statement is called the validation condition and expresses a desired property of the program. The proof establishes the validation condition. The construction of the proof is as discussed in the previous subsection.
2. A refinement step may require justification by a number of verification obligations, to show that the sub-programs under the refinement rule satisfy the super-program. A verification obligation has a statement and a proof. The statement is called the verification condition and states a property to be satisfied by the refinement step. The proof establishes the verification condition. For example, some algorithm-

mic refinement rules in the refinement calculus require verification justifications [10].

An additional refinement construction operation is *result extraction*. It extracts the refinement result of a program – the source program, introduces it into the refinement as another program – the result program, and relates them. The relation from the source program to the result program is called the result extraction relation.

Result extraction can be used to retrieve the final executable code from a refinement, or to retrieve the intermediate refinement result of a program before further refining it. If the refinement result of a program p is $\langle \dots x := 1; x := 0; \dots \rangle$, for example, we may like to transform it into $\langle \dots x := 0; \dots \rangle$ by a further refinement. To do so, we need to retrieve the refinement result of p by a result extraction operation.

3.2.2 Structure

The structure of a refinement should in principle record all the programs, refinement steps and result extraction steps involved in the refinement process. The refinement and result extraction steps reflect the refinement process by relating the programs in a necessary order. The above analysis of the refinement process suggests that a directed graph structure can be used to represent a refinement: programs are denoted by vertices; use relations and result extraction relations between programs by edges.

As in the case of proofs, a program in a refinement should never directly or indirectly relate to itself under program use relations and/or result extraction relations. This implies that the refinement representation graph is acyclic.

Definition 2 (Refinement structure)

The representation structure of a refinement is an acyclic directed graph $\mathcal{R} = \langle \mathcal{V}^r, \mathcal{E}_u^r \cup \mathcal{E}_e^r \rangle$.

\mathcal{V}^r is a set of vertices denoting the programs involved in the refinement. A program $v \in \mathcal{V}^r$ if and only if it is introduced into the refinement by program introduction, decomposition refinement, composition refinement or result extraction. A program may contain a number of validation obligations.

\mathcal{E}_u^r is a set of edges denoting the use relations between the programs. A use relation $\langle v_1, v_2 \rangle \in \mathcal{E}_u^r$ if and only if program v_2 is used by program v_1 under decomposition, composition or connection refinement. All the use relation edges originating from a single program vertex represent a refinement

step relation which records the refinement operation applied and may contain a number of verification obligations.

\mathcal{E}_e^r is a set of edges denoting the result extraction relations between the programs. A result extraction relation $\langle v_1, v_2 \rangle \in \mathcal{E}_e^r$ if and only if program v_2 is extracted from program v_1 's refinement under result extraction. If $\langle v_1, v_2 \rangle, \langle v_1, v_3 \rangle \in \mathcal{E}_e^r$, then $v_2 = v_3$, i.e., a program may have at most one result program. A result extraction relation records the result extraction operation applied.

\mathcal{E}_u^r and \mathcal{E}_e^r partition the entire set of edges. The different types of vertices and edges are distinguishable in terms of the operations which introduce them.

3.2.3 Editing and query

The editing and query operations of refinements are similar to those of proofs with programs corresponding to assertions, and refinement steps and result extraction steps to proof steps.

4 Examples

In this section, we give two examples to demonstrate how the above process model for rigorous software development can be instantiated to individual methodologies. Before doing so, we first introduce a visualisation system for the proof and refinement structures to help understand them. Figure 2 shows the visual symbols for assertions, programs, use relations and result extraction relations, where

- \mathcal{V}_i and \mathcal{V}_g contain introduced and generated assertions/programs respectively,
- \mathcal{V}_m contains main assertions/programs,
- $\mathcal{E}_d, \mathcal{E}_c$ and \mathcal{E}_l contain decomposition, composition and connection use relations respectively, and
- \mathcal{E}_r contains result extraction relations.

4.1 A Mural proof

Figure 3 shows a Mural proof of **and-ass-left** [7]:

$$\{(e_1 \wedge e_2) \wedge e_3\} \vdash e_1 \wedge (e_2 \wedge e_3)$$

It is composed of an ordinary hypothesis line h_1 , a number of ordinary lines 1 to 5, and a conclusion line c . Every line contains an expression – a logical statement. Non-hypothesis lines have an

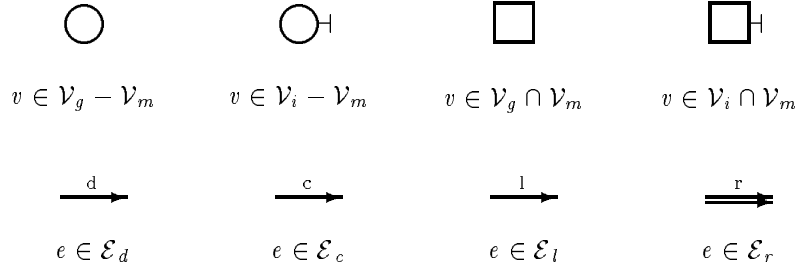


Figure 2: Visual symbols for proof and refinement structures

additional evidence part – the information of their establishment. The proof uses the following inference rules:

$$\begin{aligned}
 \text{and-E-right: } & \{e1 \wedge e2\} \vdash e1 \\
 \text{and-E-left: } & \{e1 \wedge e2\} \vdash e2 \\
 \text{and-I: } & \{e1, e2\} \vdash e1 \wedge e2
 \end{aligned}$$

Proof modelling. An assertion in our model corresponds to a line in the Mural proof, and contains the expression of that line. The assertion corresponding to the conclusion line is the main assertion of the proof.

The assertion corresponding to the ordinary hypothesis line is regarded as having immediate proof. An assertion corresponding to a non-hypothesis line may or may not be established, depending on whether or not this line has an evidence part. If it does, the assertion is established according to the recorded rule, and has the sub-assertions formulated based on those lines referred to by the evidence part. Therefore, the evidence part reflects the assertion use relations and the proof step relation.

The following are the Mural proof operations that we are interested in.

- **add line.** It adds a line containing an expression into a proof. In our model, it is an assertion introduction operation.
- **inference by rule.** It establishes a line

h_1	$(e1 \wedge e2) \wedge e3$	
1	$e1 \wedge e2$	by and-E-right on $[h_1]$
2	$e1$	by and-E-right on $[1]$
3	$e2$	by and-E-left on $[1]$
4	$e3$	by and-E-left on $[h_1]$
5	$e2 \wedge e3$	by and-I on $[3,4]$
c	$e1 \wedge (e2 \wedge e3)$	by and-I on $[2,5]$

Figure 3: A Mural proof

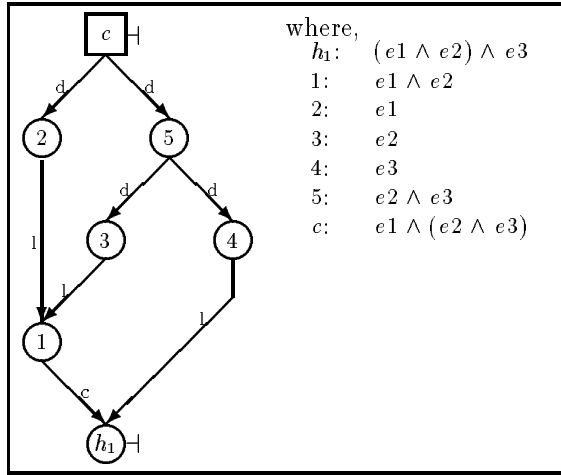


Figure 4: Representation of a Mural proof

based on other lines by applying an inference rule. If the line being established has a previous evidence part, it is replaced. In our model, it is a compound operation. Applying the inference rule is a decomposition, composition or connection inference operation, depending on which lines are given and which lines are generated. Removing the evidence part is a proof step deletion operation.

- **remove line.** It removes a line, including the expression and the evidence part (if any), from a proof. In our model, it is a compound operation composed of assertion deletion and proof step deletion.

Representation. Figures 4 shows the representation of the given Mural proof. A Mural proof in itself does not record the detailed proof construction process, i.e., how each inference step is carried out. In contrast, the representation shown assumes that the proof is constructed as follows. First, the main assertion c and the hypothesis assertion h_1 are introduced by **add line**. (In reality, they are

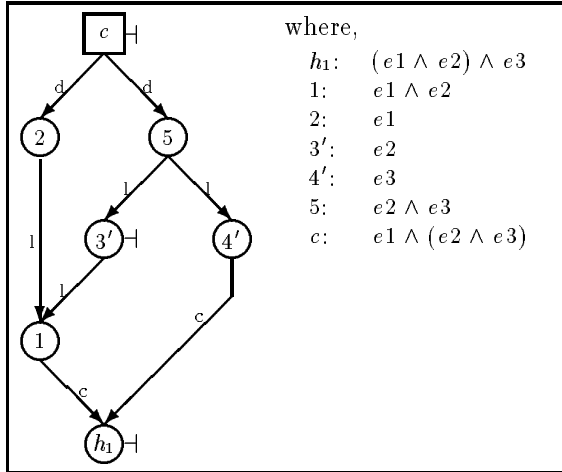


Figure 5: Representation of an edited Mural proof

introduced when creating the enclosing theorem.) Then, the main assertion c is decomposed into assertions 2 and 5 by **inference by rule** under the **and-I** rule. Assertion 5 is decomposed into assertions 3 and 4 also under the **and-I** rule. Assertion 1 is composed from assertion h_1 under the **and-E-right** rule. Assertions 2 and 3 are connected to assertion 1 under the **and-E-right** rule and the **and-E-left** rule respectively. Finally, assertion 4 is connected to assertion h_1 under the **and-E-left** rule. Note that in the representation, the operation applied at each proof step is attached to the corresponding proof step relation.

Editing. To demonstrate the representation of proof editing, we delete assertions 3 and 4 as well as the proof steps under them by **remove line**. Note that the proof step under assertion 5 becomes incomplete. Then, assertion 4' which is the same as the previous assertion 4 is composed from assertion h_1 by **inference by rule** under the **and-E-left** rule. Assertion 3' which is the same as the previous assertion 3 is introduced by **add line**, and connected to assertion 1 under the **and-E-left** rule. Finally, assertion 5 is connected to assertions 3' and 4' under the **and-I** rule. The proof becomes complete again. The representation of the edited proof is shown in Figure 5.

The above proof representations capture the detailed proof process by recording both the basic inter-assertion dependence relations and the proof styles in which the proof steps are carried out. From the stylistic information, we can derive the possible operation sequences applied, which are

particularly helpful to the understanding of proofs, especially large proofs.

This example also shows that our model is capable of accommodating assertion (and proof) reuse. For instance, assertion 1 is used by both assertions 2 and 3. We also show that even after editing, the proof can be understood as being constructed from scratch.

4.2 A refinement in the refinement calculus

In this subsection, we demonstrate how refinements constructed in the refinement calculus can be represented in our model by giving an example taken from [10]. Suppose that we are given a natural number s and required to set the natural number r to the greatest integer not exceeding \sqrt{s} . The abstract program specifying this problem is

$$\begin{aligned} & \llbracket \text{var } r, s : \mathbb{N} \bullet \\ & \quad r : [\text{true}, r = \lfloor \sqrt{s} \rfloor] \\ & \rrbracket \end{aligned} \quad (1)$$

It is the main program of the refinement and is introduced at the beginning of the refinement process. After a series of decomposition refinement steps, this abstract program is refined to code (see pages 70-73 of [10] for details):

$$\begin{aligned} & \llbracket \text{var } r, s : \mathbb{N} \bullet \\ & \quad \llbracket \text{var } q : \mathbb{N} \bullet \\ & \quad \quad q, r := s + 1, 0; \\ & \quad \quad \text{do } r + 1 \neq q \longrightarrow \\ & \quad \quad \quad \llbracket \text{var } p : \mathbb{N} \bullet \\ & \quad \quad \quad \quad p := (q + r) \text{ div } 2; \\ & \quad \quad \quad \quad \text{if } s < p^2 \longrightarrow q := p \\ & \quad \quad \quad \quad \quad \llbracket s \geq p^2 \longrightarrow r := p \\ & \quad \quad \quad \quad \quad \text{fi} \\ & \quad \quad \quad \quad \rrbracket \\ & \quad \quad \quad \text{od} \\ & \quad \quad \rrbracket \\ & \rrbracket \end{aligned} \quad (20)$$

Refinement modelling. A program in the refinement calculus is a simple or complex statement³. A simple statement is an assignment statement or a specification statement. The specification statement has the following form:

$$w : [pre, post]$$

³Validation obligations can be associated with programs in the refinement calculus. However, we exclude them for simplicity purposes.

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. The following is an example:

$$r : [\mathbf{true}, r = \lfloor \sqrt{s} \rfloor]$$

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 also contain variable (**var**), invariant (**and**) and logical constant (**con**) declarations. For instance, program (1) is a local block.

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 in its enclosing context when it is isolated from its super-program. The declaration information in the environment is qualified by **variables**, **invariants** and **constants**. In general, a model program has an optional environment and a statement:

$$[environment] \text{ statement}$$

The following program is obtained by isolating the specification statement in program (1):

$$[\mathbf{variables} \ r, s : \mathbb{N}] \ r : [\mathbf{true}, r = \lfloor \sqrt{s} \rfloor]$$

For simplicity purposes, we concentrate on a small set of construction operations. (Refinement editing is similar to proof editing and is omitted.) One operation is to introduce the initial specification of the software system to be developed. In our model, it is program introduction. Another operation applies a refinement rule to an existing program (as the super-program), and generates a number of sub-programs. In our model, it is decomposition refinement. The refinement step relation records the applied operation and the required verification obligations. The last construction operation is to retrieve the refinement result of a program. In our model, it is result extraction. The corresponding result extraction relation records the applied operation.

Representation. The representation structure of the given refinement is shown in Figure 6. The vertices are attached with programs, and edge groups with refinement step relations or result extraction relations. Due to space limitation, we

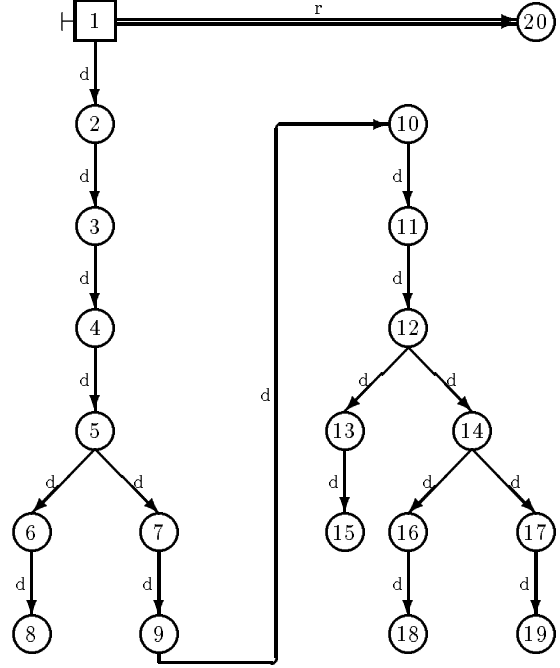


Figure 6: Representation structure of a refinement

shall not give all the details (see [3]). The following is one refinement step. Program (6)

$$\begin{aligned} &[\mathbf{variables} \ q, r, s : \mathbb{N}] \\ &\quad q, r : [\mathbf{true}, r^2 \leq s < q^2] \end{aligned} \quad (6)$$

is refined to program (8)

$$\begin{aligned} &[\mathbf{variables} \ q, r, s : \mathbb{N}] \\ &\quad q, r := s + 1, 0 \end{aligned} \quad (8)$$

by decomposition refinement using the following refinement rule for introducing assignment

$$\begin{aligned} &\text{if } pre \Rightarrow post[w \setminus E], \\ &\text{then } w, x : [pre, post] \sqsubseteq w := E \end{aligned}$$

where \Rightarrow and \sqsubseteq are pronounced as “entails” and “is refined to” respectively, and the formula $post[w \setminus E]$ is obtained by replacing in $post$ all the free occurrences of w by E while avoiding variable capture. The verification condition of this refinement step is

$$\begin{aligned} &[\mathbf{variables} \ q, r, s : \mathbb{N}] \\ &\quad \mathbf{true} \Rightarrow 0 \leq s < (s + 1)^2 \end{aligned} \quad (6.O_1)$$

and can be discharged in a proof system like Mural.

Also note that program (20) is the refinement result of program (1), and is extracted from it by

a result extraction operation.

As with the proof example discussed earlier, this refinement example shows that our model can effectively capture the refinement results and the refinement process.

5 Related work

Many approaches to process modelling have been proposed over the last few years. Activity-based and product-based approaches are two major classes. Activity-based process modelling or process programming takes the view that software processes consist of orderly executed activities, and can be programmed in formal notations [12]. Example approaches include Arcadia [16], PML [14], Marvel [8], Merlin [6], Prism [9] and SPM [18]. These approaches are well suited to modelling the managerial aspect of the software life cycle. The software processes are captured explicitly, and have high enactability and manageability. However, they tend to be coarse-grained and prescriptive, and have difficulties in modelling the technical aspect of software processes. In these approaches, the process aspect of software development is the dominant factor.

Product-based process modelling focuses on both the final and intermediate software products as well as their relations, and takes the view that software products and their relations are created through software activities. Example approaches include Design Recording [13], EPM [5] and the algebraic approach [11]. These approaches have the potential to carry out fine-grained process modelling, and to tackle the technical aspect of software processes. They also give the user freedom to conduct his creative activities. On the other hand, they tend to be less definitive, which leads to low enactability and manageability. The fact that software processes are implicitly embedded in software products presents difficulties in understanding the processes. In general, the product aspect of software development dominates these approaches.

Our approach combines software processes with software products. The structures and operations of software objects are precisely defined, and the object structures explicitly reflect the software processes. It is fine-grained, non-prescriptive and definitive, so it is capable of dealing with the technical aspect of software processes in an enactable and manageable manner. It also emphasises systematic software development by following well-

established methodologies. In our approach, software processes and software products are treated as equally important. However, its applicability to the managerial aspect of software processes needs to be further investigated.

6 Conclusions

In this paper, we have presented an methodology-oriented approach to process modelling. It unifies software processes and software products, provides fine-grained, non-prescriptive and definitive support for software processes, and emphasises systematic software development by following well-established methodologies. Using this approach, we have developed a process model for rigorous software development, and demonstrated that it can be instantiated to individual methodologies. The process model accommodates not only the initial construction of software objects but also their subsequent modification and reuse.

A more comprehensive model for rigorous software development has been developed using the proposed approach [3]. It also deals with organisation, consistency and presentation of proofs and refinements. In addition to Mural and the refinement calculus, this model has been systematically instantiated to Demo2 [15], HOL [2], Nuprl [1] and B [17].

After some prototype experiment and a feasibility study, we are currently developing a generic, methodology-based environment for rigorous software development. It has three major components: a methodology description language, a tool integration language and an environment proper. The environment proper accepts a methodology description, a tool integration description and the semantic tools, and customises itself into a methodology-specific environment. The methodology description language is being developed based on the approach and model proposed in this paper.

It becomes clear that different aspects of software processes may need different process models [4]. Integrating our approach with other approaches to achieve effective modelling of all aspects of software processes is a further research topic. Another issue to explore is the potential to incorporate in our approach the capability of direct process programming based on the operations of software objects.

Acknowledgements

Many of our colleagues at Queensland have commented on the work reported in this paper. We are also grateful to Cliff Jones for discussions on proof representation.

References

- [1] R.L. Constable, S.F. Allen, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [2] 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, Massachusetts, 1988.
- [3] J. Han. *A Structural Model for Methodology-based Interactive Rigorous Software Development*. PhD thesis, The University of Queensland, St. Lucia, Australia, March 1992.
- [4] W.S. Humphrey. Modeling implications of the personal software process. In *Experience with Software Process Models, Proceedings of 5th International Software Process Workshop*, pages 74–77, Kennebunkport, Maine, October 1989. IEEE CS Press.
- [5] W.S. Humphrey and M.I. Kellner. Software process modeling: Principles of entity process models. In *Proceedings of 11th International Conference on Software Engineering*, pages 331–342, Pittsburgh, Pennsylvania, May 1989. IEEE CS Press.
- [6] H. Hünnekens, G. Junkermann, B. Peuschel, W. Schäfer, and J. Vagts. A step towards knowledge-based software process modelling. In *Proceedings of 1st International Conference on System Development Environments and Factories*, pages 49–58, Berlin, Germany, May 1989. Pitman Publishing, London, 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] G.E. Kaiser and F.H. Feiler. An architecture for intelligent assistance in software development. In *Proceedings of 9th International Conference on Software Engineering*, pages 180–188, Monterey, California, March-April 1987. IEEE CS Press.
- [9] N.H. Madhavji and W. Schäfer. Prism – methodology and process-oriented environment. *IEEE Transactions on Software Engineering*, SE-17(12):1270–1283, December 1991.
- [10] C. Morgan. *Programming from Specifications*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, London, 1990.
- [11] A.T. Nakagawa and K. Futatsugi. Software process à la algebra: OBJ for OBJ. In *Proceedings of 12th International Conference on Software Engineering*, pages 12–23, Nice, France, March 1990. IEEE CS Press.
- [12] L. Osterweil. Software processes are software too. In *Proceedings of 9th International Conference on Software Engineering*, pages 2–13, Monterey, California, March-April 1987. IEEE CS Press.
- [13] C. Potts. A generic model for representing design methods. In *Proceedings of 11th International Conference on Software Engineering*, pages 217–226, Pittsburgh, Pennsylvania, May 1989. IEEE CS Press.
- [14] C. Roberts. Describing and acting process models with PML. *ACM SIGSOFT Software Engineering Notes*, 14(4):136–141, June 1989.
- [15] T.G. Tang, P.J. Robinson, and J. Staples. The demonstration proof editor Demo2. Technical Report 175, Department of Computer Science, The University of Queensland, St. Lucia, Australia, April 1991.
- [16] R.N. Taylor, F.C. Belz, L.A. Clarke, L. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young. Foundations for the Arcadia environment architecture. *ACM SIGSOFT Software Engineering Notes*, 13(5):1–13, November 1988.
- [17] T. Vickers. An overview of a theorem proving assistant. In *Proceedings of 13th Australian Computer Science Conference*, pages 402–411, Melbourne, Australia, February 1990.
- [18] L.G. Williams. Software process modeling: A behavioral approach. In *Proceedings of 10th International Conference on Software Engineering*, pages 174–186, Singapore, April 1988. IEEE CS Press.