ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**He Jifeng, Oxford University Computing Laboratory, UK, John Cooke,
Loughborough University, UK, and Peter Wallis, University of Bath, UK (Eds)**

BCS-FACS 7th Refinement Workshop

Proceedings of the BCS-FACS 7th Refinement Workshop, Bath, 3-5
July 1996

Models for Configuration Management of Refinement Calculus Developments

Kelvin J. Ross

Published in Collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

ISBN 3-540-76104-7

Models for Configuration Management of Refinement Calculus Developments

Kelvin J. Ross

Software Verification Research Centre

Department of Computer Science

The University of Queensland

Queensland 4072, Australia

email: kjross@cs.uq.oz.au

Abstract

This paper considers aspects of Software Configuration Management (SCM) in its role of supporting configurations that arise in development of software using the refinement calculus. From an SCM perspective, the primary difference between formal development methods, such as the refinement calculus, and traditional development methods is the nature of relationships within the development configuration. The relationships are very precise; occur at much finer levels of granularity; and evolve through an iterative/intertwined development process. In analysing support for the refinement calculus, this paper proposes a configuration model which defines the relationships that occur during development and how these relationships evolve as development activities are applied and modifications are made. The configuration model enables properties regarding relationships to be formally inferred. The results of these inferences may be used in developing automated support for refinement calculus developments, such as change impact analysis, traceability and remanufacturing services.

1 Introduction

During software development a system will evolve to ultimately satisfy particular requirements. In addition to user requirements that are delivered by the system developer, there may be methodology requirements which require the system to be developed according to a particular development methodology. Many requirements of the methodology involve how the components of the development documentation (ie. the development *configuration*) must ‘fit’ together: how components of the configuration must relate to other components.

The *refinement calculus* [7] defines a development methodology for the correct transformation of program specifications into implementations. In the refinement calculus there is a precise notion of a *refinement relationship* which indicates when an implementation satisfies a specification. In providing SCM support for the refinement calculus there are many requirements of the methodology based on the nature of the refinement relationship.

This paper introduces the use of configuration models to capture properties of software development configurations, in particular those which arise using the refinement calculus. A *configuration model* defines the structure of the configurations which occur within the development documentation, as well as the relationships which exist within and between these configurations. The model not only identifies the relationships required to meet the methodology (e.g. the refinement relationship), but also identifies how these relationships are established as process activities are applied and how they are affected as changes are made.

1.1 Overview

The structure of the paper is as follows. Section 2 provides a brief overview of Software Configuration Management (SCM) and the support it can offer refinement calculus development. An overview of configuration models is provided in Section 3, and then the configuration model for refinement calculus developments is provided in a layered approach in the following sections: Section 4 introduces the refinement relationship; Section 5 adds the notion of refinement rules; and Section 6 adds a complex development document representation for recording refinement developments.

In Section 7 we consider properties that are consequences of the configuration model and how they convey useful information relating to development, such as conformance to the methodology and change impact analysis. Section 8 examines how the configuration model can provide information for constructing SCM support for the refinement calculus. Finally in section 9 we present conclusions, indicate where simplifications have been incorporated into the model and review related work.

Within the paper a particular presentation style has been adopted. The definitions for the configuration models are displayed in figures enclosed in oval boxes. Other supporting figures are displayed in shadowed boxes. The notation includes atomic type and primitive relationship names being underlined; activity names are capitalised; and assumption and inference rule names are shown in typed font.

2 Software Configuration Management

Software Configuration Management (SCM) is the means through which the integrity and traceability of the software system are recorded, communicated and controlled during both development and maintenance [15].

SCM is concerned with the management of configurations of components which collectively represent the software development, including management of the relationships between components and management of the persons and processes involved in the development of components.

2.1 SCM Services

SCM offers a wide range of services for managing software development configurations as they evolve. These services include:

Repository: Efficient storage and retrieval of versions of configuration components as well as relationships between components. For example, a repository for refinement could offer storage of the versions of development components (e.g. program fragments and proofs) and relationships between component (e.g. modification between versions, refinement between program fragments and uses dependencies between proofs).

Change Control: Techniques employed to initiate and regulate changes to the development. For example, procedures and practices may be employed to prevent developers from changing refinement developments until approval has been granted by a review committee.

Manufacturing: Support for automatic reconstruction of configuration components from other components; and efficient support for reconstruction as changes are realised. For example, a refinement tool may be reapplied automatically as changes are made to determine if the refinement is still up-to-date. The application of the tool may be restricted so that it is only reapplied to those components that are affected by the change.

Concurrency Control: Management of the cooperation between multiple developers to enable development activities to occur concurrently. For example, multiple developers may be permitted to carry out refinements of separate parts of a program fragment independently, but not refine the same part concurrently.

Configuration Consistency: Facilities to determine whether components of a development configuration are compatible, consistent or conformant. For example, consistency checking may determine if the specification and the implementation are those which have refinement documented within the development configuration.

Dependency Analysis: The representation and consulting of dependencies may indicate how changes to a component will impact on other components of the configuration. For example, between an implementation and a specification there are dependencies based on the refinement dependencies. These dependencies may enable the tracing of a change to a specification through the refinement to indicate the parts of the implementation which are affected by the change.

Process Support: Reporting the status of the development; monitoring the status as changes are applied; and offering guidance and control of the developer's activities. For example, process support may monitor what refinement activities have been performed and what still remains to be completed. The support may offer advice to the developer as to the appropriate activities to continue development (e.g. suggest discharging of relevant proof obligations before continuing with further refinements).

2.2 SCM and the Refinement Calculus

Existing SCM systems, established to support traditional developments, are equally applicable to formal developments, such as refinement calculus developments. Both traditional and formal developments contain many products at different life-cycle phases, contain components which are structured into larger components (configurations), and evolve iteratively as changes are made. As such many SCM services offered for traditional development may be suitably adapted for support of formal development, e.g. repositories (RCS [16]) and remanufacturing support (MAKE [4]).

From an SCM perspective, there are a number of differences between formal development and traditional development which have an impact on the SCM systems [14]. In particular, the nature of relationships within formal development lead to a number of significant differences:

- The relationships are very precise and are frequently defined in terms of other development relationships using complex rules. For example, the refinement calculus provides precise definitions as to what constitutes a refinement between different program fragments. In many cases the refinement relationship between program fragments is based on refinement relationships between subcomponents of the fragments.
- Formal development relationships, due to their precise nature, are often the basis for managing the integrity of the software development. In cases where machine-checking of the relationships is built into SCM support the integrity of the support systems become a concern. For example, a SCM system may manage a refinement development by controlling the invocation of various integrated refinement tools. If the SCM system is responsible for reporting when refinement is completed, based on the tools that have been applied, then it must use the tools appropriately to ensure that the reporting of a completed refinement is correct. If the system incorrectly reports that refinement is achieved then the developer may have a false notion of integrity of the software development.
- Relationships within formal development exist at various levels of granularity within the configurations, even down to extremely fine levels. For example, the refinement calculus contains relationships at high levels, such as between entire programs, as well as relationships between fine levels, such as between individual statements.
- There may exist a wide variety of relationships between components of the development: the developments are not restricted to just one type of relationship. The refinement calculus contains many relationships other than the refinement relationship between components. It also contains links to generated proof obligations, to proofs of these obligations, etc.
- The evolution of a relationship is often deeply intertwined in the development process. A particular relationship between components may not exist until a number of other process activities have been combined appropriately. The way in which the SCM controls the order in which activities are applied should not be over-restrictive. For example, the development of refinement between two fragments may involve a number of intertwined activities: an appropriate refinement rule is selected; the rule is applied to a program fragment, along with user input, to generate a refined fragment; resulting proof obligations are generated from the refinement; and the proof obligations are discharged by formal proof. The SCM system should allow refinement activities to be intertwined: the developer should be capable of commencing other refinements and returning later to discharge the proof obligations of an earlier refinement.

Existing SCM systems fail to meet the extended requirements resulting from the above differences and enhanced functionality is required. The main source of complexity that Software Configuration Management (SCM) systems must manage are relationships [17], yet they form a basis for many of the SCM services. Defining the relationships of formal development provides a much needed systematic study which is essential to any advances in SCM support in this area [6].

3 Configuration Models

A configuration model classifies the components of software developments and defines the relationships that occur within and between the components. The model also provides a mechanism for analysing how relationships within the development are established and evolve as activities are applied. A configuration model can be divided into four main components: types; relationships; activities; and assumptions.

A configuration model is provided using a notation which is a slight variation on VDM-SL [3, 5]. The main differences from VDM-SL are the inclusion of assumptions, which are additional properties which must be satisfied by the relationships, and the use of primitive definitions for certain relationships and types.

3.1 Types

The types of the configuration model define a classification of *Software Configuration Items* (SCIs), which are the components of the software development which are subject to SCM support. The type identifies the groups into which SCIs are categorised. This classification distinguishes relationships and process activities relevant to particular SCIs.

SCIs are either *composite* or *atomic* SCIs. Composite SCIs have a defined structure and are designated by a configuration of subcomponent SCIs. Composite SCIs may include structures defined by abstract collections types, such as records, sets, bags, maps, etc. Atomic SCIs, on the other hand, have no explicit internal structure: they are specified as primitive types.

3.2 Relationships

Relationships convey structural, process and semantic information regarding the SCIs of the development. They are particularly useful for expressing correctness requirements of the development methodology.

A relationship is designated by a function definition that specifies the name of the relationship as well as the types of the operand SCIs. The function definition can be thought of a test function for the operands: it determines whether the operand SCI passed as parameters meet the criteria of the relationship.

A relationship is designated as either *derived* or *primitive*. A derived relationship has a derivation rule: its value is determined from the values of other relationships and/or contents of the operand SCIs. Primitive relationships are the basic relationships of a development methodology which are established from applying process activities.

3.3 Activities

Activities are the tools and methods used in the development process to analyse or manipulate components of the configuration. They are synonymous with steps in the development process, e.g. editing, compilation, proof checking.

The side effect of activities is that they may establish relationships between components of the configuration. The relationships established by an activity are made available so that this information may be extended with new relationships about the components. For example, a single refinement step may indicate a refinement relationship exists between the input and output programs of the refinement step. Additional activities may continue to make new refinements until a sequence of refinement steps exist to establish refinement between the requirements specification and the implementation.

3.4 Assumptions

Assumptions define properties of relationships that result from development activities or which are consequences of the development methodology. An assumption provides a rule for establishing relationships for specific instances of SCIs. For example, an assumption may indicate that if two program fragments have identical contents then refinement is guaranteed.

4 The Refinement Relationship

Refinement is a relationship between program fragments based on the notion of weakest preconditions (see Fig. 1). The functional definition of the relationship indicates that whether two program fragments satisfy the refinement can be determined purely from the contents of the program fragment operands. The refinement relationship is defined as a primitive relationship, which has no explicit derivation.

A program fragment is a design representation which embodies both code and specification (see Fig. 2). It provides a notation which enables the stepwise transition from specification to code: as such a specification may be embedded deep within the implementation structure.

A program fragment is one of a number of alternative program constructs:

$$\sqsubseteq : PROGRAM \times PROGRAM \rightarrow \mathbf{B}$$

Figure 1: Refinement is a primitive relationship defined between program fragments.

$PROGRAM = SEQ \mid IF \mid DO \mid BLOCK \mid \underline{REF_SPEC} \mid \underline{ASSIGNMENT} \mid \underline{SKIP} \mid \underline{ABORT}$

$SEQ :: \begin{array}{l} first: PROGRAM \\ second: PROGRAM \end{array}$

$IF :: \begin{array}{l} conds: \underline{CONDITION}^* \\ branches: PROGRAM^* \end{array}$
 $inv \text{ mk-}IF(conds, branches) \triangle$
 $len \ conds = len \ branches$

$DO :: \begin{array}{l} conds: \underline{CONDITION}^* \\ branches: PROGRAM^* \end{array}$
 $inv \text{ mk-}DO(conds, branches) \triangle$
 $len \ conds = len \ branches$

$BLOCK :: \begin{array}{l} decl: \underline{VAR_DEFNS} \\ code: PROGRAM \end{array}$

$\underline{CONDITION}$ is not yet defined
 $\underline{VAR_DEFNS}$ is not yet defined
 $\underline{REF_SPEC}$ is not yet defined
 $\underline{ASSIGNMENT}$ is not yet defined
 $\underline{SKIP}$ is not yet defined
 $\underline{ABORT}$ is not yet defined

Figure 2: SCI types utilised in the refinement relation.

- *SEQ* represents sequential composition of two sub-program fragments: the second program is executed after the first;
- *IF* represents invocation of sub-programs based on the satisfaction of the associated guard condition;
- *DO* represents iterative invocation of sub-programs based on the satisfaction of the associated guard condition – when no condition is satisfied the iteration terminates;
- *BLOCK* represents a sub-program into which localised variables have been declared;
- *REF_SPEC* represents the abstract (formal) specification of a program;
- *ASSIGNMENT* represents a program statement that updates the contents of a variable with a new value;
- *SKIP* represents a program that does not cause any change to the program state; and
- *ABORT* represents a pathological program: it is never guaranteed to terminate, and when it does it may update the program state in any way.

Each program fragment is represented as either atomic or composite SCIs. The *SEQ*, *IF*, *DO* and *BLOCK* program fragments are composite SCI types which are represented by records. These record types contain programs as subcomponents and enables the representation of complex programs to be defined using the constructs recursively. The fields of the *IF* and *DO* record types utilise sequence abstract types: the SCI contains two ordered collection of subcomponent SCIs. Subtyping is also used to restrict *IF* and *DO* fragments to only those which have record field sequences of equal length. Other remaining (non-record) SCI types are atomic types: there is no explicit knowledge of the internal structure.

Back [1] identified some basic properties of the refinement relationship: reflexivity; transitivity; and monotonicity over the *SEQ*, *IF*, *DO* and *BLOCK* program fragments. These properties are expressed as configuration model assumptions regarding the $\sqsubseteq$ relationship (see Fig. 3). The reflexivity and transitivity assumptions are similar to traditional definitions. However monotonicity is defined by separate assumptions for each type of composite program fragment.

5 Refinement Calculus Transformations

The previous section provides the first layer of a configuration model for refinement. The model so far defined the refinement relationship, the SCI types of program fragments utilised in the relationship, and some basic assumptions of the relationship. As yet no method has been provided to initially establish refinement between two arbitrary program fragments – the basic assumptions only provide useful information once some refinement between program fragments has been realised.

The refinement calculus [7, 9] provides a method for establishing refinement between two arbitrary program fragments. This method utilises a number of refinement rules which are used to transform a program into a refined program. The use of a refinement rule may involve the user providing input to direct the generation towards a particular implementation. Generating the refined program using a given refinement rule and user input may require a number of proof obligations to be discharged as theorems in order for the transformation to be considered correct.

Consider the refinement of a simple specification into an assignment statement (see Fig. 4). In using the refinement calculus the developer provides a refinement rule for the transformation (**Law 1.3** assignment) as well as providing input that defines the assignment statement that is to be refined to $(x := 0)$. Together, a supplied program fragment, a refinement rule and user input produce a refined program fragment and consequent proof obligations. The proof obligations must be discharged as theorems in order for the refinement to be valid.

The inclusion of the refinement calculus approach into the development method requires a number of new SCI types to be added to the configuration model (see Fig. 5). These SCI types classify refinement rules that encapsulate the basic laws of the refinement calculus, user input that accompany the application of the refinement rule, and logical statements to capture the resulting proof obligations of the refinement generation.

The method of generating a refinement between program fragments can be formally captured as a relationship (see Fig. 6). The *generated* $\sqsubseteq$ relationship is satisfied when the refinement calculus method has been applied successfully: a valid refinement rule and user input generate the refined program fragment and the resulting proof obligations have been formally discharged to become theorems. The *generated* $\sqsubseteq$ relationship is a derived relationship, and is explicitly represented in terms of other primitive relationships:

Figure 3: Assumptions that capture basic properties of the refinement relationship.

Figure 4: The refinement calculus provides a method of transforming a program fragment into a refined program fragment using refinement rules.

REFINEMENT_RULE is not yet defined
INPUT is not yet defined
INF_STMT is not yet defined

Figure 5: Additional SCI types found in the refinement calculus methodology.

- valid_rule determines whether a refinement rule is defined within the refinement calculus;
- ref_generates_program derives the refined program fragment through transformation of an initial program fragment, a refinement rule and user input;
- ref_generates_POs derives the proof obligations that result from the above transformation;
- is_theorem determines whether a logical statement is a theorem.

$$\begin{aligned}
 & generated_ \sqsubseteq : PROGRAM \times PROGRAM \rightarrow \mathbf{B} \\
 & generated_ \sqsubseteq(p1, p2) \triangleq \\
 & \quad \exists rule: \underline{REFINEMENT_RULE}, in: \underline{INPUT} . \\
 & \quad \quad \underline{valid_rule}(rule) \wedge \\
 & \quad \quad \underline{ref_generates_program}(p1, rule, in) = p2 \wedge \\
 & \quad \quad \forall obl \in \underline{ref_generates_POs}(p1, rule, in) . \\
 & \quad \quad \quad \underline{is_theorem}(obl) \\
 \\
 & \underline{valid_rule} : \underline{REFINEMENT_RULE} \rightarrow \mathbf{B} \\
 \\
 & \underline{ref_generates_program} : PROGRAM \times \underline{REFINEMENT_RULE} \times \underline{INPUT} \rightarrow PROGRAM \\
 \\
 & \underline{ref_generates_POs} : PROGRAM \times \underline{REFINEMENT_RULE} \times \underline{INPUT} \rightarrow \underline{INF_STMT_set} \\
 \\
 & \underline{is_theorem} : \underline{INF_STMT} \rightarrow \mathbf{B}
 \end{aligned}$$

Figure 6: The refinement calculus provides relationships for generating refinements.

To provide a facility for establishing the ref_generates_program and ref_generates_POs relationships an activity for generating the refinement is included (see Fig. 6). The GENERATE_REFINEMENT activity is analogous to an automated refinement tool: it is supplied with a program fragment, a refinement rule and the user input to carry out the refinement transformation; it returns a refined program, a set of proof obligations and a result flag. If the result flag is true the returned program fragment and proof obligations are a correct transformation of the supplied parameters.

Notice that the definition of the activity has been under-specified: it does not specify the behaviour when the result flag is false. In this situation the activity may return any program and/or proof obligations it chooses. The intention of the result flag is an indication of whether any errors or exceptions may have occurred in the generation.

Finally an assumption is included to associate the refinement generation to the refinement relation defined in the previous section. This assumption indicates that any two programs that satisfy the generated_ $\sqsubseteq$ relationship must also satisfy the $\sqsubseteq$ relationship (see Fig. 8).

This section has added to the configuration model to define a method to initially establish a refinement relationship. Ultimately it provides a means to associate the GENERATE_REFINEMENT activity with the $\sqsubseteq$ relationship, providing that some means is made available to establish the valid_rule and is_theorem relationships. Once the refinement relationship is initially generated it can be extended to indicate other refinements between program fragments which follow from the assumptions regarding the refinement relationship provided in the previous section (see Fig. 3).

$GENERATE_REFINEMENT(p1: PROGRAM, ref_rule: \underline{REFINEMENT_RULE}, user: \underline{INPUT})$
 $p2: PROGRAM, proof_obls: \underline{INF_STMT_set}, result: \mathbf{B}$
 post $result \Rightarrow$
 $ref_generates_program(p1, ref_rule, user) = p2 \wedge$
 $ref_generates_POs(p1, ref_rule, user) = proof_obls$

Figure 7: The *GENERATE_REFINEMENT* is an activity used to establish relationships which lead to refinement.

$\boxed{\sqsubseteq_by_generation} \text{ ————— }$
 $\forall p1, p2: PROGRAM .$
 $generated_ \sqsubseteq (p1, p2) \Rightarrow \sqsubseteq(p1, p2)$
 —————

Figure 8: An assumption that indicates that a generated refinement satisfies the refinement relationship.

6 Refinement Development Document

Typically a refinement development is recorded within a (linearised) development document. The purpose of the development document is to convey the refinement of a specification into an implementation: it records design decisions and provides evidence that the implementation is a correct refinement of the specification.

Consider the development document which describes the refinement of a program to compute the Greatest Common Divisor (GCD) of two natural numbers (see Fig. 9). This document explicitly records refinement steps between program fragments. The refinement steps are sufficient to indicate refinement between the specification and the implementation based on the properties of the refinement relationship provided in previous sections.

In this document the development started with an initial specification (shown in the top box of the document) and produced an implementation (shown in the bottom box). Markers are used in the document to indicate where refinement occurs on the separate parts of program fragments, including a special marker ($\triangleleft$) to indicate when refinement of a subcomponent of a fragment immediately follow.

Refinement steps within the development document may contain cross-reference labels to the proof obligations resulting from the refinement step. These labels identify conjectures that formally state and discharge the proof obligations (see Fig. 10).

6.1 Configuration Representation

In the same way that there are many ways of documenting a refinement (Fig. 9 is one approach), there are also many ways of representing the structure of the chosen development document. Alternative representations exist for storing a refinement that increase the ability to develop and maintain the development document. A suitable representation of the development document achieves four main objectives [17]:

Manage complexity: smaller components are easier to understand, design and manage than larger components.

Divide labour: developments can be decomposed into units according to the work breakdown structure.

Produce a maintainable system: minimise the impact of changes within manageable parts of the structure, and enable efficient and flexible regeneration.

Reuse items: minimise the amount of development required by utilising previously carried out work.

On its own the development document simply records the outcome of the refinement process: the representation of the document falls short of achieving the above objectives. The document has omitted information which could be

Figure 9: Sample Greatest Common Divisor (GCD) development document.

$$\begin{array}{c}
 \boxed{\text{PO-1}} \frac{x, y: \mathbf{N}, \quad x > 0 \wedge y > 0}{x > 0 \wedge y > 0 \wedge \text{gcd}(x, y) = \text{gcd}(x, y)} \\
 \\
 \boxed{\text{PO-2}} \frac{a, b, x, y: \mathbf{N}, \quad a > 0 \wedge b > 0 \wedge \text{gcd}(a, b) = \text{gcd}(x, y) \wedge a = b}{a = \text{gcd}(x, y)} \\
 \\
 \boxed{\text{PO-3}} \frac{a, b, x, y: \mathbf{N} \quad a > 0 \wedge b > 0 \wedge \text{gcd}(a, b) = \text{gcd}(x, y)}{\Leftrightarrow a > 0 \wedge b > 0 \wedge \text{gcd}(a, b) = \text{gcd}(x, y)} \\
 \\
 \boxed{\text{PO-4}} \frac{a, b, x, y: \mathbf{N} \quad a > 0 \wedge b > 0 \wedge \text{gcd}(a, b) = \text{gcd}(x, y) \wedge a = b}{\Leftrightarrow a > 0 \wedge b > 0 \wedge \text{gcd}(a, b) = \text{gcd}(x, y) \wedge \neg(a > b \vee a < b)} \\
 \\
 \boxed{\text{PO-5}} \frac{a, b, x, y: \mathbf{N}, \quad a < b \wedge a > 0 \wedge b > 0 \wedge \text{gcd}(a, b) = \text{gcd}(x, y)}{a - b > 0 \wedge b > 0 \wedge \text{gcd}(a - b, b) = \text{gcd}(x, y) \wedge 0 \leq a - b + b < a + b} \\
 \\
 \boxed{\text{PO-6}} \frac{a, b, x, y: \mathbf{N}, \quad a < b \wedge a > 0 \wedge b > 0 \wedge \text{gcd}(a, b) = \text{gcd}(x, y)}{a > 0 \wedge b - a > 0 \wedge \text{gcd}(a, b - a) = \text{gcd}(x, y) \wedge 0 \leq a + b - a < a + b}
 \end{array}$$

Figure 10: Proof obligations generated in the sample GCD development.

used to better address the maintainable system and reuse objectives. For example, recording the tools and their inputs used in generating the separate refinement steps enables the step to be regenerated when changes are applied. This information can be stored within an extended development document representation from which the original document can be extracted.

This section considers a particular representation which meets many of the above objectives. This representation was based on the configurations that occur in the *Centipede* environment [2], which decomposed the development according to *refinement diagrams* [1].

The development configuration represents the refinement development as a complex graph. For example, the refinement diagram for the GCD case study above illustrates the graph representation (see Fig. 11). Properties of refinement, such as transitivity and monotonicity are implicit in the diagram.

Figure 11: Refinement diagram of sample GCD refinement development.

Nodes of the configuration graph representation are program fragments of the refinement development (see Table 1). Each node may contain a number of other nodes, but circularities in containment are not permitted. The nodes of the refinement graph utilise references to other nodes rather than storing the node in its entirety.

Edges of the graph represent individual refinements that are generated using refinement rules (see Table 2). Each edge of the graph is attributed with information regarding which refinement rule was applied, the input provided by the user and the resulting proof obligations of the refinement.

The representation outlined above provides support for flexible and efficient development of refinements. In addition to the structures illustrated above, the representation enables many alternative structures to be utilised (see Fig. 12). These provide support for reusing refinements, exploring alternative possibilities, etc.

Figure 12: Some other refinement diagram structures: (i) recomposition, (ii) convergence, (iii) divergence and (iv) sharing.

The representation would also include a proof theory to enable the discharging of proof obligations of the refinement development. The proof theory consists of inference rules, made up of conjectures and their proof. A conjecture is a theorem when its proof is complete with respect to other theorems in the theory. The references to proof obligations in the development would be references to a named inference rule within the proof theory.

This representation also enables the earlier refinement document to be extracted from the configuration using traversals of the refinement graph representation. It does not over-impose any order on development of documentation. For example, it may allow refinements to be developed and proofs to be discharged as independent activities.

Label	Type	Value
a0	BLOCK_FRAG	$= \text{mk-BLOCK_FRAG}("x, y, g: \mathbf{N}", b0)$
b0	REF_SPEC	$\sim "g: [x > 0 \wedge y > 0, g = \text{gcd}(x, y)]"$
b1	BLOCK_FRAG	$= \text{mk-BLOCK_FRAG}("a, b: \mathbf{N}", c0)$
c0	REF_SPEC	$\sim "g, a, b: [x > 0 \wedge y > 0, g = \text{gcd}(x, y)]"$
c1	REF_SPEC	$\sim "g, a, b: [x > 0 \wedge y > 0, g = \text{gcd}(x, y)]"$ with context augmented with I defn.
c2	SEQ_FRAG	$= \text{mk-SEQ_FRAG}(d0, e0)$
d0	REF_SPEC	$\sim "g, a, b: [x > 0 \wedge y > 0, I]"$
d1	ASSIGNMENT	$\sim "a, b: = x, y"$
e0	REF_SPEC	$\sim "g, a, b: [I, g = \text{gcd}(x, y)]"$
e1	SEQ_FRAG	$= \text{mk-SEQ_FRAG}(f0, g0)$
f0	REF_SPEC	$\sim "g, a, b: [I, I \wedge a = b]"$
f1	REF_SPEC	$\sim "a, b: [I, I \wedge a = b]"$
f2	DO_FRAG	$= \text{mk-DO_FRAG}(["a > b", "a < b"] [h0, i0])$
g0	REF_SPEC	$\sim "g, a, b: [I \wedge a = b, g = \text{gcd}(x, y)]"$
g1	ASSIGNMENT	$\sim "g: = a"$
h0	REF_SPEC	$\sim "a, b: [a > b, I, 0 \leq a + b < a_0 + b_0]"$
h1	ASSIGNMENT	$\sim "a: = a - b"$
i0	REF_SPEC	$\sim "a, b: [a < b, I, 0 \leq a + b < a_0 + b_0]"$
i1	ASSIGNMENT	$\sim "b: = b - a"$
a0'	BLOCK_FRAG	$= \text{mk-BLOCK_FRAG}("x, y, g: \mathbf{N}", b1')$
b1'	BLOCK_FRAG	$= \text{mk-BLOCK_FRAG}("a, b: \mathbf{N}", c2')$
c2'	SEQ_FRAG	$= \text{mk-SEQ_FRAG}(d1, e1')$
e1'	SEQ_FRAG	$= \text{mk-SEQ_FRAG}(f2', g1)$
f2'	DO_FRAG	$= \text{mk-DO_FRAG}(["a > b", "a < b"] [h1, i1])$

Table 1: Program fragments in the GCD development.

Trans	From	To	Rule	Input	Obligations
1	b0	b1	intro vars	" $a, b: \mathbf{N}$ "	{}
2	c0	c1	intro spec constant	" $I \triangleq a > 0 \wedge b > 0 \wedge \text{gcd}(a, b) = \text{gcd}(x, y)$ "	{}
3	c1	c2	seq comp	" $MID = I$ "	{}
4	d0	d1	intro ass	" $a, b: = x, y$ "	{PO1}
5	e0	e1	seq comp	" $MID = I \wedge a = b$ "	{}
6	g0	g1	intro ass	" $g: = a$ "	{PO2}
7	f0	f1	contract frame	" g "	{}
8	f1	f2	intro iteration	" $INV = I, VAR = a + b, GUARDS = [a < b, a > b]$ "	{PO3, PO4}
9	h0	h1	intro ass	" $a: = a - b$ "	{PO5}
10	i0	i1	intro ass	" $b: = b - a$ "	{PO6}

Table 2: Development transitions corresponding to refinement steps in the GCD development.

6.2 Types

In defining the configuration of the refinement development a composite SCI captures the information for a whole development document. This SCI is classified as a *DEVELOPMENT* SCI type (see Fig. 13).

Figure 13: SCI types utilised in a refinement development.

The development configuration includes the definition of the following:

start : the program fragment to be refined (reference to program in top box of document in Fig. 9);

end : the program fragment resulting from the refinement (reference to program in bottom box of document in Fig. 9);

frags : the representation for a collection of uniquely labeled program fragments (see Table 1);

steps : the ordered sequence of refinement steps (see Table 2) – the order of the refinement steps is only used to indicate the order in which the refinement document is printed; and

theory : the mathematical theory in which proof obligations are discharged.

The definition of the SCI types for the development representation included the definition of a number of subtypes which are restricted by an invariant. These subtypes are used primarily to restrict the SCI to those which are sensible. For example, a subnode label must reference a fragment in *frags* and there are no circularities in the program fragment references (this would produce a program fragment that is infinitely recursive). However these place restrictions on activities that are provided to modify the SCIs: they must preserve the invariants.

In defining the development configuration a variation of program fragments is utilised which contain references to subcomponent program fragments (see Fig. 14).

The theory SCI type is a composite SCI made up of named inference rules (see Fig. 15). Each inference rule has a statement (the conjecture) and a proof. The proof may designate the conjecture as an axiom of the theory or it may contain a proof which may discharge it in terms of other inference rules.

$PROGRAM_FRAG = SEQ_FRAG \mid IF_FRAG \mid DO_FRAG \mid BLOCK_FRAG \mid \underline{REF_SPEC}$
 $\mid \underline{ASSIGNMENT} \mid \underline{SKIP} \mid \underline{ABORT}$
 $SEQ_FRAG :: \text{first: } \underline{LABEL}$
 $\text{second: } \underline{LABEL}$
 $IF_FRAG :: \text{conds: } \underline{CONDITION}^*$
 $\text{branches: } \underline{LABEL}^*$
 $\text{inv mk-IF_FRAG(conds, branches)} \triangle$
 $\text{len conds} = \text{len branches}$
 $DO_FRAG :: \text{conds: } \underline{CONDITION}^*$
 $\text{branches: } \underline{LABEL}^*$
 $\text{inv mk-DO_FRAG(conds, branches)} \triangle$
 $\text{len conds} = \text{len branches}$
 $BLOCK_FRAG :: \text{decl: } \underline{VAR_DEFNS}$
 $\text{code: } \underline{LABEL}$

Figure 14: SCI types of development program fragments, which contain references to other fragments (labels).

$THEORY = \underline{INF_NAME} \xrightarrow{m} \underline{INF_RULE}$
 $\underline{INF_RULE} :: \text{stmt: } \underline{INF_STMT}$
 $\text{proof: } \underline{PROOF} \mid \text{AXIOM}$
 $\underline{INF_NAME}$ is not yet defined
 $\underline{PROOF}$ is not yet defined

Figure 15: A theory contains inference rules used to formally discharge the refinement's proof obligations.

6.3 Relationships

Many relationships can be formulated within a refinement development. This section identifies relationships relevant to the completion of a refinement between the initial and final program fragments.

Firstly, a number of structural relationships are evident within the development representation (see Fig. 16). These include extraction functions for determining labels that are used within refinement steps, names of obligations generated in steps, containment relationship between program fragment labels and to dereference the development fragment structures to complete program fragments.

The definition of relationships presents many complex uses of the VDM-SL notation. In particular, the structural relationships featured the following:

- *Set comprehension* is used in *from_labels* to return the set of program fragment labels obtained from the *from* attribute of the steps in the development.
- *Distributed union* is used in *obligation_names* to return all the proof obligation references by combining all proof obligation sets from each refinement step.
- A *case statement* is used in *subnode_labels* to return the references to other program fragments based on the type of program fragment utilised.
- A *transitive closure* of the *part_of* relationship is provided by the *part_of_trans* relationships. This is defined without using recursion by constructing a sequence of *part_of* relationships.
- A *precondition* is used to restrict the domain of the *get_program* relationship – only labels which are defined within fragments can extract a program fragment.

Many other VDM-SL constructs are utilised within definitions, however the discussion of their use is not considered further in this paper. The reader should refer to VDM-SL references for in-depth discussions [3, 5].

The goal of refinement development is to have a complete development, which requires a refinement from the initial program fragment to the final program fragment using a given set of refinement rules (see Fig. 17).

The configuration model given here could be extended to enforce certain other pragmatic constraints, such as disallowing diverging refinement steps (so that there is only one version of development presented in the document) or ensuring the final program is that generated from the leaves of the refinement path.

Refinement between two components is any transitive collection of single refinement steps (see Fig. 18). A single step is either equality of the two fragments, a refinement between all subcomponents of the program fragments (monotonicity) or a generated refinement. Checking for refinement by monotonicity recursively checks the *refinement_between* relationship.

Ultimately the refinement between two fragments must be provided as a path of complete generated refinement steps (see Fig. 19). In order for a generated step to be complete, it must satisfy the following:

- The resulting extracted program fragment must be the refinement of the initial program fragment using the designated refinement rule and input.
- The referenced inference rule statement in the verification theory must correspond to the proof obligations that are to be proven for the generated refinement; and
- The referenced inference rule in the verification theory must be discharged.

Whether an inference rule is discharged as a theorem is provided as a primitive relationship. In other work the definition of this relationship and its consequent properties have been examined in detail [12]. However for brevity, this work has been simplified to a single primitive relationship.

6.4 Activities

There are many activities that can be formulated for developing the configuration defined in this section. These would include: creation and deletion of program fragments, editing and modifications to program fragments, creation and deletion of refinement steps; merging of program fragments; analysis of refinement paths; and so on. These activities could be defined in terms of tools which modify the structure and contents of the configuration.

```

fromLabels : DEV_STEPS → LABEL-set
fromLabels(steps)  $\triangleq$  {step.from | step ∈ elems steps}

toLabels : DEV_STEPS → LABEL-set
toLabels(steps)  $\triangleq$  {step.to | step ∈ elems steps}

obligation_names : DEV_STEPS → INF_NAME-set
obligation_names(steps)  $\triangleq$   $\bigcup$ {step.obls | step ∈ elems steps}

rules_used : DEV_STEPS → REFINEMENT_RULE-set
rules_used(steps)  $\triangleq$  {s.rule | s ∈ elems steps}

subnodeLabels : PROGRAM_FRAG → LABEL-set
subnodeLabels(p)  $\triangleq$ 
  cases true of
    is-SEQ_FRAG(p) → {p.first, p.second}
    is-IF_FRAG(p) → elems p.branches
    is-DO_FRAG(p) → elems p.branches
    is-BLOCK_FRAG(p) → {p.code}
    others {}
  end

part_of : LABEL × LABEL × DEV_FRAGMENTS → B
part_of(sub, super, frags)  $\triangleq$ 
  sub ∈ dom frags ∧ super ∈ dom frags ∧
  sub ∈ subnodeLabels(frags(super))

part_of_trans : LABEL × LABEL × DEV_FRAGMENTS → B
part_of_trans(sub, super, frags)  $\triangleq$ 
  ∃path: LABEL+ .
  len path ≥ 2 ∧
  path(1) = sub ∧ path(len path) = super ∧
  ∀i ∈ inds path .
  i < len path ⇒ part_of(path(i), path(i + 1), theory)

get_program : LABEL × DEV_FRAGMENTS → PROGRAM
get_program(label, frags)  $\triangleq$ 
  let node = frags(label) in
  cases true of
    is-SEQ_FRAG(node) → mk-SEQ(get_program(node.first, frags), get_program(node.second, frags))
    is-IF_FRAG(node) → mk-IF(node.conds,
      [get_program(node.branches(i), frags) | i ∈ inds node.branches])
    is-DO_FRAG(node) → mk-DO(node.conds,
      [get_program(node.branches(i), frags) | i ∈ inds node.branches])
    is-BLOCK_FRAG(node) → mk-BLOCK(node.decl, get_program(node.code, frags))
    others node
  end

pre label ∈ dom frags
    
```

Figure 16: Structural relationships provide information that is extracted directly from the configuration structure.

```

complete_development : DEVELOPMENT × REFINEMENT_RULE-set → B

complete_development(dev, rules)  $\triangle$ 
    refinement_between(dev.start, dev.end, dev) ∧
    rules_used(dev.steps) ⊆ rules
    
```

Figure 17: Completion of the refinement development is designated as a relationship.

```

refinement_between : LABEL × LABEL × DEVELOPMENT → B

refinement_between(start, end, dev)  $\triangle$ 
    trans_steps_between(start, end, dev)

pre start ∈ dom dev.fragms ∧ end ∈ dom dev.fragms

trans_steps_between : LABEL × LABEL × DEVELOPMENT → B

trans_steps_between(start, end, dev)  $\triangle$ 
    ∃ path: LABEL+ .
        len path ≥ 2 ∧
        path(1) = start ∧ path(len path) = end ∧
        ∀ i ∈ inds path .
            i < len path ⇒ single_step_between(path(i), path(i + 1), dev)

single_step_between : LABEL × LABEL × DEVELOPMENT → B

single_step_between(start, end, dev)  $\triangle$ 
    start ∈ dom dev.fragms ∧ end ∈ dom dev.fragms ∧
    (start = end ∨
    mono_step_between(start, end, dev) ∨
    generated_step_between(start, end, dev))

mono_steps_between : LABEL × LABEL × DEVELOPMENT → B

mono_steps_between(start, end, dev)  $\triangle$ 
    start ∈ dom dev.fragms ∧ end ∈ dom dev.fragms ∧
    let snode = dev.fragms(start) ∧ enode = dev.fragms(end) in
    cases true of
        is-SEQ_FRAG(snode) ∧ is-SEQ_FRAG(enode) → refinement_between(snode.first,
                                                                    enode.first, dev) ∧
                                                                    refinement_between(snode.second,
                                                                    enode.second, dev)
        is-IF_FRAG(snode) ∧ is-IF_FRAG(enode) → snode.conds = enode.conds ∧
        ∀ i ∈ inds snode.branches .
            refinement_between(snode.branches(i),
                            enode.branches(i), dev)
        is-DO_FRAG(snode) ∧ is-DO_FRAG(enode) → snode.conds = enode.conds ∧
        ∀ i ∈ inds snode.branches .
            refinement_between(snode.branches(i),
                            enode.branches(i), dev)
        is-BLOCK_FRAG(snode) ∧ is-BLOCK_FRAG(enode) → snode.decl = enode.decl ∧
            refinement_between(snode.code,
                            enode.code, dev)
    others false
    end
    
```

Figure 18: Refinement between two fragments is determined by finding a suitable path of refinement steps.

```

generated_step_between : LABEL × LABEL × DEVELOPMENT → B
generated_step_between(start, end, dev)  $\triangleq$ 
   $\exists \text{step} \in \text{elems dev.steps} \cdot$ 
    step.from = start  $\wedge$  step.to = end  $\wedge$ 
    complete_generated_step(step, dev)

complete_generated_step : DEV_STEP × DEVELOPMENT → B
complete_generated_step(step, dev)  $\triangleq$ 
  let from_prg = get_program(step.from, dev.frgs)  $\wedge$ 
    to_prg = get_program(step.to, dev.frgs) in
  ref_generates_program(from_prg, step.rule, step.input) = to_prg  $\wedge$ 
  ref_generates_POs(from_prg, step.rule, step.input) = {dev.theory(name) | name  $\in$  step.obls}  $\wedge$ 
   $\forall \text{name} \in \text{step.obls} \cdot$ 
    rule_discharged(name, dev.theory)

pre step  $\in$  elems dev.steps
rule_discharged : INF_NAME × THEORY → B
    
```

Figure 19: A complete generated refinement step indicates refinement between its two program fragments.

A sufficient basis for development requires the addition of an activity to establish the *rule_discharged* primitive relationship. As noted earlier extensive support for this relationship has been considered in other work, but for an adequate activity set adding a *CHECK_THEOREM* activity would enable refinements to be developed to completion (see Fig. 20).

```

CHECK_THEOREM (name: INF_NAME, theory: THEORY) result: B
post result  $\Leftrightarrow$  rule_discharged(name, theory)
    
```

Figure 20: The *CHECK_THEOREM* activity checks whether a given inference rule is a theorem of a given theory.

6.5 Assumptions

Only one further addition to the refinement assumptions (see § 4) is provided for the purposes of analysis presented in this paper. This assumption indicates that when an inference rule is discharged within a theory then the inference rule's statement has been proven to be a theorem. (see Fig. 21).

```

proven_theorem
 $\forall \text{name: } \underline{\text{INF\_NAME}}, \text{theory: } \text{THEORY} \cdot$ 
  rule_discharged(name, theory)  $\Rightarrow$  is_theorem(theory(name).stmt)
    
```

Figure 21: An assumption indicates the statement of a discharged inference rule is a theorem.

7 Consequences of Refinement Developments

The configuration model provides many definitions regarding the structure and relationships of SCIs involved in refinement developments. Other theorems may be formally proven as consequences of configuration model definitions, and are thus further properties of the development representation. These theorems indicate properties such the logical implications of some relationships on other relationships and the effect of modifications to the SCIs on the development relationships.

This section defines some sample consequences that may be determined from the configuration model definitions outlined in the previous sections. Many of the properties defined in this section are properties that are assumed when building SCM support. An adequate method is required for proving the consequences as theorems from the configuration model: otherwise SCM environments may be constructed to support the development configurations using incorrect assumptions.

7.1 Conformance to Methodology

It is a difficult problem to provide assurance that the chosen representation for the development documentation is appropriate for representing refinements. The configuration model introduced in Section 6 above is structured to enable efficient and flexible development of refinement documents. The representation of the development document has introduced concerns beyond that originally considered for refinement: a separate notion of completion of a refinement development has been constructed based on notions such as separation of the discharging of proof obligations, storage of evidence for refinements, and reuse of program fragments. Formally associating completion of the development back to the definition of refinement establishes confidence in the document representation being appropriate.

The following example shows that the original intention of refinement is achieved: i.e. when a development is deemed complete (*complete_development*) the development provides evidence to ensure the program fragments of the development are a refinement ($\sqsubseteq$). This can be formally stated as a consequence of the configuration model using the following theorem:

$$\boxed{\text{dev_adequate}} \frac{\begin{array}{l} \text{dev: DEVELOPMENT,} \\ \text{rules: REFINEMENT_RULE-set,} \\ \forall r \in \text{rules} \cdot \text{valid_rule}(r) \end{array}}{\text{complete_development}(\text{dev}, \text{rules})} \Rightarrow \sqsubseteq(\text{get_program}(\text{dev.start}, \text{dev.fragments}), \text{get_program}(\text{dev.end}, \text{dev.fragments}))$$

Based on the definition of *complete_development*(*dev*, *rules*) a proof of this consequence uses the following theorem that indicates refinement follows from the *refinement_between* relationship:

$$\boxed{\text{steps_adequate}} \frac{\begin{array}{l} \text{start, end: LABEL,} \\ \text{dev: DEVELOPMENT,} \\ \forall r \in \text{rules_used}(\text{dev.steps}) \cdot \text{valid_rule}(r) \end{array}}{\text{refinement_between}(\text{start}, \text{end}, \text{dev})} \Rightarrow \sqsubseteq(\text{get_program}(\text{start}, \text{dev.fragments}), \text{get_program}(\text{end}, \text{dev.fragments}))$$

Based on the reflexivity, transitivity and monotonicity properties of the $\sqsubseteq$ relationship (see § 4) this ultimately reduces to showing that a generated refinement step is a refinement:

$$\boxed{\text{generation_adequate}} \frac{\begin{array}{l} \text{step: DEV_STEP,} \\ \text{dev: DEVELOPMENT,} \\ \text{valid_rule}(\text{step.rule}) \end{array}}{\text{complete_generated_step}(\text{step}, \text{dev})} \Rightarrow \text{generated_}\sqsubseteq(\text{get_program}(\text{step.from}, \text{dev.fragments}), \text{get_program}(\text{step.to}, \text{dev.fragments}))$$

which follows from the definition of *complete_generated_step* and the $\sqsubseteq$ -by-generation assumption (see § 4).

7.2 Change Impact Analysis

Change impact analysis is a technique employed to determine the effects of changes; typically it is used before the changes are made to assess the impact they will have. A configuration model provides a formal basis for change impact analysis. We can deduce consequences of a configuration model to indicate those relationships that are affected by a change or those that are independent of the change.

7.2.1 Functional Dependency

The functional definition of the relationships indicate that the result of the function is dependent only on the parameters of the function. As such any change to SCIs only affects the value of a relationship if it affects the values of its operands.

For example, a change to a program fragment affects the relationship *completed_development* only if it changes the development or the allowed refinement rules. This property is captured by the following theorem:

$$\boxed{\text{no_change}} \frac{\begin{array}{l} \text{old_dev}, \text{new_dev}: \text{DEVELOPMENT}, \\ \text{old_rules}, \text{new_rules}: \text{REFINEMENT_RULE-set}, \\ \text{old_dev} = \text{new_dev}, \\ \text{old_rules} = \text{new_rules} \end{array}}{\begin{array}{l} \text{completed_development}(\text{old_dev}, \text{old_rules}) \\ \Leftrightarrow \text{completed_development}(\text{new_dev}, \text{new_rules}) \end{array}}$$

This inference is typical of change impact analysis consequences developed from the configuration model. The consequence determines the effect of a change on a relationship by considering versions of the relationship operands, the relationship between the versions, which indicates the type of change being considered, and the consequence of the change in terms of a logical association between the old result of the relationship and the new result. Above we consider two versions of the development and rules, but they said to be equal in value (ie. they have the same contents). In these cases the result of *completed_development* applied to the alternative parameter versions is the same.

7.2.2 Independence of Changes to Parameters

The above section provided a consequence which indicate the effects on relationships when no change to the parameters is made. In many cases changes which affect parameters of a relationship may still not affect the result of the relationship. In these cases the result of a relationship is independent of the components that are changed. The independence of relationships on certain changes to their operands provides the basis for useful change impact analysis.

For example, changes to a development which only add to and/or reorder the refinement steps will maintain that the new development is complete if the old development is complete:

$$\boxed{\text{add_refinements}} \frac{\begin{array}{l} \text{old_dev}, \text{new_dev}: \text{DEVELOPMENT}, \\ \text{rules}: \text{REFINEMENT_RULE-set}, \\ \text{only_add_or_reorder_steps}(\text{old_dev}, \text{new_dev}) \end{array}}{\begin{array}{l} \text{completed_development}(\text{old_dev}, \text{rules}) \\ \Rightarrow \text{completed_development}(\text{new_dev}, \text{rules}) \end{array}}$$

This theorem indicates that once a development is complete whether it is still complete is not affected by modifications which add new refinement steps or reorder the steps. The inference rule is restricted to considering only a particular class of change to the versions of operands: only those which have added to or reordered the refinement steps of the development – no other changes are permitted. The class of change is expressed by a relationship which defines the permitted differences in the contents of the versions of the development (see Fig. 22). This is a normal configuration model relationship and can be augmented to the existing definitions of the configuration model.

$$\begin{array}{l} \text{only_add_or_reorder_steps} : \text{DEVELOPMENT} \times \text{DEVELOPMENT} \rightarrow \mathbf{B} \\ \text{only_add_or_reorder_steps}(\text{old_dev}, \text{new_dev}) \triangleq \\ \quad \text{old_dev.start} = \text{new_dev.start} \wedge \\ \quad \text{old_dev.end} = \text{new_dev.end} \wedge \\ \quad \text{old_dev.frag} = \text{new_dev.frag} \wedge \\ \quad \text{elems old_dev.steps} \subseteq \text{elems new_dev.steps} \wedge \\ \quad \text{old_dev.theory} = \text{new_dev.theory} \end{array}$$

Figure 22: A relationship captures a particular difference between versions of developments.

The above change impact analysis, even though it may be intuitive to those well grounded in the refinement calculus, provides a formal justification that adding new steps have no effect on whether a development remains completed. Furthermore it indicates that the layout order of the development document can be reordered without affecting whether the development is complete.

The impact analysis above is restricted to changes that added new information to the configurations. Similarly we can reason how relationships evolve for any change to the SCI operands, such as modifying or deleting components. For example, changing the contents of program fragments has a limited effect on the *complete_generated_step* relationship. The only generated steps affected by changing program fragments are those which transitively contain the changed program fragment. This is formally stated by the following theorem:

$$\begin{array}{c}
 \text{step: DEV_STEP,} \\
 \text{old_dev, new_dev: DEVELOPMENT,} \\
 \text{step} \in \text{elems old_dev.steps,} \\
 \text{frags: LABEL-set,} \\
 \text{only_modifies_frags(frags, old_dev, new_dev),} \\
 \forall f \in \text{frags.} \\
 \quad \neg \text{part_of_trans}(f, \text{step.from, old_dev.frags}) \wedge \\
 \quad \neg \text{part_of_trans}(f, \text{step.to, old_dev.frags}) \\
 \hline
 \boxed{\text{modify_frags}} \quad \text{complete_generated_step}(\text{step, old_dev}) \\
 \Leftrightarrow \text{complete_generated_step}(\text{step, new_dev})
 \end{array}$$

This theorem indicates that any given *step* within the old development still retains whether it is completed in a new development if certain conditions regarding the changes between the old and new developments are satisfied: the modification to the developments are restricted to changing the contents of set of identified program fragments (*frags*); and the program fragments involved in the given generated step (*step.from* and *step.to*) do not (transitively) contain a fragment *f* which is within those changed fragments *frags*. The *only_modifies_frags* relationship is used to restrict the developments to those that have only *frags* changed (see Fig. 23).

$$\begin{array}{l}
 \text{only_modifies_frags : LABEL-set} \times \text{DEVELOPMENT} \times \text{DEVELOPMENT} \rightarrow \mathbf{B} \\
 \text{only_modifies_frags(frags, old_dev, new_dev)} \triangleq \\
 \quad \text{old_dev.start} = \text{new_dev.start} \wedge \\
 \quad \text{old_dev.end} = \text{new_dev.end} \wedge \\
 \quad \text{old_dev.steps} = \text{new_dev.steps} \wedge \\
 \quad \text{old_dev.theory} = \text{new_dev.theory} \wedge \\
 \quad \text{let unaffected} = \text{dom old_dev.frags in} \\
 \quad (\text{unaffected} \triangleleft \text{old_dev.frags}) = (\text{unaffected} \triangleleft \text{new_dev.frags})
 \end{array}$$

Figure 23: Difference relationship which indicates that only *frags* have been effected by modifying a development.

Many more theorems which capture impact analyses may be formally established as consequences of the configuration model. For example, we could reason about effects of modifying the theory on refinement steps, or the effect of changing refinement steps on other steps. The approach of establishing these effects as consequences follows that shown in the examples above.

8 Configuration Models as a Basis for SCM Support

The configuration model and its inferred consequences provide valuable information which can be utilised in providing SCM support for managing development configurations. This information provides properties which can be used in the design of SCM systems as well as a mechanism for analysing the systems after they have been constructed.

This paper has shown how the development document representation can be related back to the theoretical refinement model. The configuration model enabled an alternative structure to be constructed and the considerations for completion of development using the structure could be related back to more abstract concerns.

As a further illustration of how the configuration model provides assistance in building SCM support briefly consider process modelling as a means of support. Process support environments provide management of the evolution of developments by controlling the employment of development activities, monitoring and recording of the status of the development and providing valuable guidance to further the development.

A particular requirement of process support for refinement is the management of the correctness of developments. The configuration model provides a specification basis for providing support for managing correctness of developments. The concerns of the process support include:

- *A precise view of the requirements for correctness/completion of the development.* Many requirements have been captured using the definition of relationships. The relationships capture not only correctness of the entire development, but also decomposes correctness into individual requirements for subparts of the development, such as individual refinement steps.
- *An effective mechanism for reporting correctness/completion information to the developer.* For instance, statuses may be utilised to indicate the correctness of separate steps. The status has a strong association with the definition of the relationship that formally designate the requirements for correctness/completeness. Statuses may be associated with separate parts of the development to report on the completion of the component. For example, each refinement step may have a status to indicate when it is a complete generated step. Intermediate statuses may provide reporting to indicate partial levels of completion. For example, the refinement step may also have intermediate statuses which indicate when the step has been generated by a tool, as well as whether the resulting proof obligations are discharged.
- *The appropriate tracking of correctness/completion of the development as changes are applied.* For instance, the process model specifies the status updates that occur as changes are applied to ensure necessary redevelopment. The change impact analysis provided through consequences of the configuration model provides a precise analysis of how the completion of the development is affected by certain changes. This analysis can be incorporated into the process support to indicate that a certain change requires a particular update to status depending on the association between status and the level of completion/correctness. For example, the impact analysis in Section 7 indicates that if a program fragment is modified then any refinement step is still complete providing the program fragment is not transitively part of the step. This can be reflected in process status updates: if a program fragment is modified then any step that the fragment is part of has its status updated to ensure that it is rechecked, but other steps where completion is not affected by the changed program fragment may retain the same status.

The configuration model and its formal analysis enables verification of support systems to ensure they correctly manage configurations as changes are applied. For example, in [13] a configuration model was utilised in the verification of the support offered by a process model in managing the correctness of VDM specifications. This approach is especially useful in evaluating refinement environments considering the integrity of the method being supported.

9 Conclusions

In summary this paper provided a configuration model of refinement calculus developments. The configuration model is a systematic study of development configurations and their relationships that evolve as development activities are applied. The model specifies requirements for adherence to the development methodology and provides valuable feedback that can be incorporated into the development of SCM support.

The configuration model presented in this paper provides the following features:

- A formalisation of the requirements for completion of a refinement development in terms of relationships between refinement components.
- A demonstration of the adequacy of the relationships in representing refinement.
- An indication of activities which are used to develop a refinement and establish the necessary relationships.
- An analysis of how the relationships are affected as changes are made.

The configuration model for the refinement calculus development is a major case study arising from PhD research on Software Configuration Management for support of formal development. The PhD thesis [11] provides an in-depth examination of configuration models in their role of expressing SCM requirements for support of formal developments.

9.1 Omissions and Simplifications

For brevity we have not provided the derivation rule for the *rule_discharged* relationship: in this paper it is defined as a primitive relationship. In other work this relationship has been considered in detail [11] and has resulted in many

useful properties regarding the activities used to establish the relationship as well as its behaviour through change impact analysis.

Similarly we have simplified *valid_rule* to be defined as a primitive relationship. The approach adopted simply analysed for validity the refinement rules used in the development. However an additional mechanism is required to enable the extension of new rules to cope with an evolving refinement rule set. It may keep track of accepted rules and allows extensions to usable rules through formally proving them from accepted rules. Many approaches are foreseeable, but have not been considered in this case study.

Alternative representations for context of refinement need to be developed. In the configuration model this information was represented within atomic SCIs of the development fragments or is contained as axioms of the verification theory. However as the refinement scope is expanded to include data refinement this representation needs to be expanded. It will need to store somehow a local context for a development to incorporate locally defined context information to be represented, such as operation specification to enable refinements to procedure calls, etc.

The refinement activities considered in this paper represent a minimal set. Many other activities are useful in assisting in the development of components is a refinement document, such as program fragment editors, sophisticated analysis tools (e.g. graphical tools to illustrate the paths), and theorem proving tools.

9.2 Related Work

Other approaches have defined similar support for refinement developments. Two similar approaches include:

- Centipede [2] provides a development environment based on the structure of refinement graphs, which enables editing and review of the development. Centipede records all refinements collectively in a large pool of refinement steps, and no separation of individual developments is represented.
- Naftalin [10] specified a process implementation, which defines a system in VDM for supporting refinement graphs. In this specification operations are employed to develop the refinement graph, such as adding refined fragments, copying fragments, etc.

These approaches focused on the generation of refinement graphs, which provide the structural basis for the representations in this paper. However they did not provide solutions for monitoring the correctness of the development and hence provided minimal semantic support for refinement developments.

On the other hand, Morgan [8] provided a functional implementation for support of refinement calculus developments. The approach provides a static mechanism for determining correctness. However the implementation approach ignored the effects of changes on correctness: changes require the development to be re-evaluated from scratch. Process support would have enabled a more efficient approach to be taken, such as suitable tracking of the correctness of the development as changes are made.

The approach in this paper provides an abstract specification of the correctness requirements of refinement development. The specification provides the requirements for a configuration correctness without defining the actual process to be followed: it describes a process rather than prescribing it. It has also extended the support scope by considering the interaction of the verification theory in the development theory.

Acknowledgements

I am indebted to my PhD. supervisor, Peter Lindsay, who over the past three years has helped nurture the early ideas for this paper into a sound framework. I also thank other colleagues from the SVRC for the useful discussion and feedback, especially Ray Nickson for reviewing this paper, and David Carrington for providing comments on drafts of this work.

References

- [1] R. J. R. Back. Refinement diagrams. In J. M. Morris and R. C. Shaw, editors, *4th BCS Refinement Workshop*, Workshops in Computing, pages 125–137. BCS FACS, Springer-Verlag, 1991.
- [2] R. J. R. Back, J. Hekanaho, and K. Sere. Centipede – a program refinement environment. Technical Report 139, Dept. of Computer Science, Åbo Akademi University, Lemminkäisenkatu 14, SF-20520 Turku, Finland, 1992.

- [3] J. Dawes. *The VDM-SL Reference Guide*. Pitman Publishing, 1991.
- [4] S. I. Feldman. Make – a program for maintaining computer programs. *Software – Practice and Experience*, 9:255–265, April 1979.
- [5] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, 1986.
- [6] B. Meyer. The software knowledge base. In *Proc. of 8th International Conference on Software Engineering*, pages 158 – 165, London, U.K., August 1985. IEEE CS.
- [7] C. Morgan. *Programming from Specifications*. Prentice Hall, 1st edition, 1990.
- [8] C. Morgan. The refinement calculus, and literate development. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, volume 755 of *LNCS*, pages 161 – 182. Springer-Verlag, 1993.
- [9] C. Morgan and T. Vickers. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14, 1990.
- [10] M. Naftalin. A model of the refinement process. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *Proc. of the 5th Refinement Workshop*, Workshops in Computing, pages 212–229. Springer-Verlag, January 1992.
- [11] K. J. Ross. *Software Configuration Management in the Support of Formal Software Development*. PhD thesis, Dept. of Computer Science, The University of Queensland, Australia, 1995. In Preparation.
- [12] K. J. Ross and P. A. Lindsay. Maintaining consistency under changes to formal specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *Proceedings of Formal Methods Europe - 1993*, LNCS 670, pages 558–577. Springer-Verlag, April 1993.
- [13] K. J. Ross and P. A. Lindsay. A precise examination of the behaviour of process models. In M. Naftalin, T. Denvir, and M. Bertran, editors, *Proceedings of Formal Methods Europe: Industrial Benefit of Formal Methods*, volume 873 of *LNCS*, pages 251 – 270. Springer-Verlag, October 1994.
- [14] K. J. Ross and P. A. Lindsay. Applying software configuration management techniques to formal development. Technical Report No. 95-12, Software Verification Research Centre, Dept. of Comp. Sci., University of Queensland, March 1995.
- [15] IEEE Computer Society. IEEE Standard for Software Configuration Management Plans. ANSI/IEEE Std 828-1990, 1990.
- [16] W. F. Tichy. RCS – a system for version control. *Software – Practice and Experience*, 15(7), July 1985.
- [17] David Whitgift. *Methods and Tools for Software Configuration Management*. John Wiley and Sons, 1991.