Working Paper Series
ISSN 1170-487X

Towards an integrated refinement environment for formal program development

**by Steve Reeves and
John C. Grundy**

Working Paper 95/26

August 1995

© 1995 Steve Reeves and John C. Grundy
Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Towards an integrated refinement environment for formal program development

Steve Reeves and John C. Grundy

Department of Computer Science
University of Waikato
Private Bag 3105, Hamilton, New Zealand
{stever, jgrundy}@cs.waikato.ac.nz

Abstract

One of the main hurdles to the general adoption of formal program development techniques is a lack of tools to support their use in combination with more traditional development techniques. This paper describes an integrated environment for software development which embodies the aim of formal program development. Multiple levels of refinement of each specification are supported, with associated proof obligations, each of which can be viewed at various levels of detail throughout the development process. All of these formal views are kept consistent with each other and with more traditional design and implementation views. This allows software developers to specify, design, refine, prove, implement and document their software within a single integrated environment.

1. Introduction

Conventional software development uses informal approaches to the analysis, design and implementation of software systems. Software development involves refining an abstract design into a concrete implementation in a programming language. A major problem with this approach is the lack of rigour used when refining these designs and implementations, since errors may be introduced by the refinement process itself.

Formal program specification and development techniques have been introduced to try and alleviate the problems inherent in this informal software development process. Specification languages include Z [1, 2, 3], Object-Z [4] and VDM [5]. Specification by itself is useful for specifying and/or documenting a program, but of limited use when actually refining a specification into an implementation.

Unfortunately most software developers view these formal program specification and refinement techniques as too hard for practical use. In order to make these techniques more readily available, environments supporting them have been developed. These include environments for specification [6] and those for specification refinement [7]. Most existing refinement environments, however, lack integration with commonly-used CASE tools and programming languages.

This paper presents the design for a new refinement environment which extends previous work by the authors in supporting formal Object-Z specification views in an integrated CASE tool environment. Views are provided which represent formal program specifications. These can be refined to more precise, lower-level specifications using the refinement calculus. Complementary proof obligation views, in which work is done to see whether the refinement is valid or not, are generated and maintained also.

2. An Example of Informal Refinement

SPE (Snart Programming Environment) provides an integrated environment for OO analysis, design, and implementation using Snart, an OO Prolog [8]. SPE supports multiple views of a system across multiple phases of development. It has a novel approach to consistency management based on the propagation of discrete change descriptions between views. These descriptions document a change made in one view and are propagated to all other views that could be affected by the change. The receiving views interpret the change and modify their contents appropriately.

We have extended SPE to incorporate Object-Z views, with consistency management between the formal Object-Z views and informal design and code views [9]. Figure one shows a screen dump from SPE with an Object-Z specification for a stack class shown. A high-level graphical analysis view is shown ('window-root class'), together with a lower-level graphical design view ('window-figure stack'). An Object-Z view ('stack-Specification') formally defines the behaviour of the stack class of objects (from [4]). Other views supported are textual class interface and method implementation views, textual documentation views, and textual and graphical debugging and program visualisation views [8].

SPE keeps design, code and Object-Z specification view bi-directionally consistent, i.e. whenever one kind of view is modified, SPE indicates that other views which share the modified information are inconsistent. SPE can automatically resolve some inconsistencies, such as adding or deleting attributes and methods, renaming attributes and methods, and adding, deleting and renaming method arguments. Other inconsistencies cannot be automatically resolved by the environment. These include those that arise when Object-Z operation deltas, pre- or post-conditions are modified or when implementation code is modified (new method calls

might be used, for example). In this situation, SPE indicates the change that has occurred in other affected views and this acts as a cue for programmers to manually resolve outstanding inconsistencies.

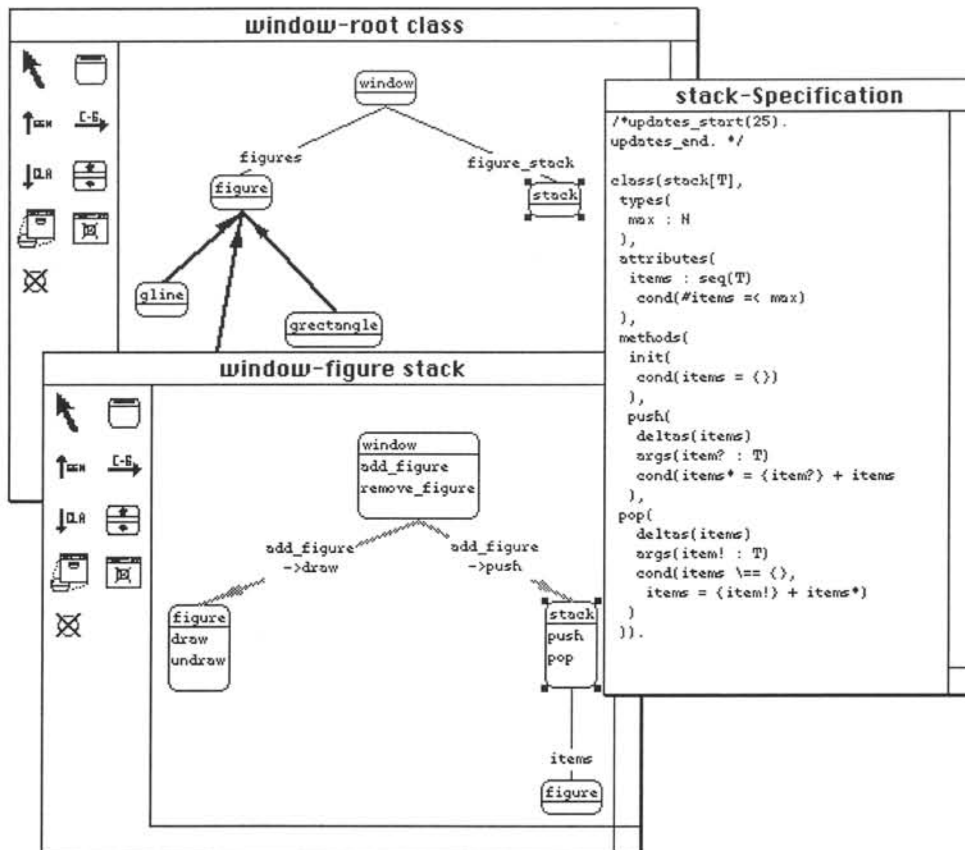

Figure one. A screen dump from SPE showing an Object-Z view and design views.

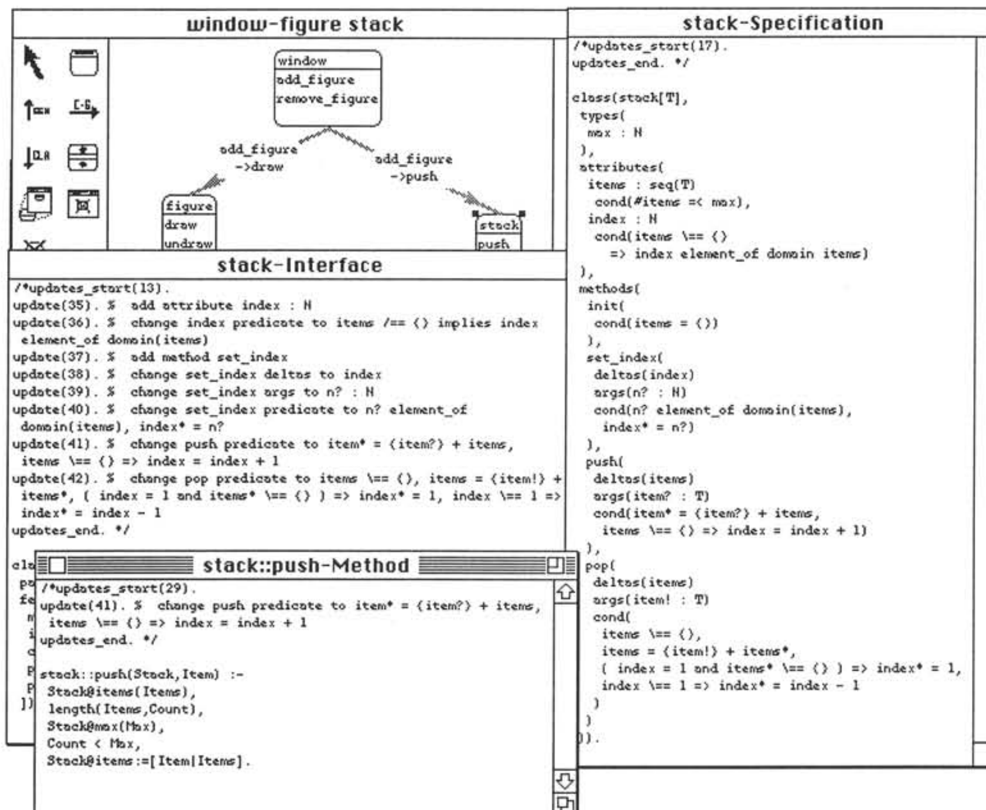

Figure two. Harder view consistency when operation predicates updated.

As an example, consider extending the definition of a stack to include an `index` attribute, acting as a marker to a distinguished element of the stack (this example has been adapted from [4]). The formal view in figure two has been updated to specify the new state and behaviour of stack objects. In this example, the stack specification now includes the new `index` attribute and a state invariant for this attribute. The `set_index` method is used to modify this value, and the `push` and `pop` methods have been updated to ensure the `index` value is updated if items are pushed onto or removed from the stack.

Figure two also illustrates the change descriptions presented to the programmer in the stack implementation views. Note that SPE only displays change descriptions relevant to the implementation view item. Some can be automatically applied, such as adding the new `index` attribute. Others must be implemented by the programmer; the change descriptions serve to inform the programmer such updates are required. This simple example illustrates how SPE supports the evolution of a formal specification and its implementation within an integrated environment.

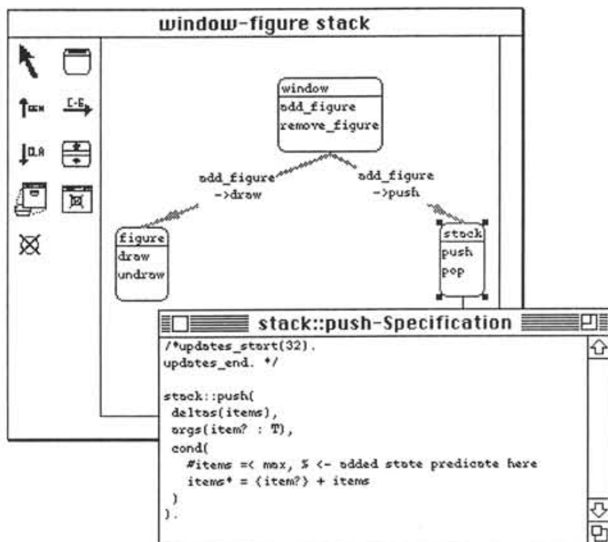

Figure three. State predicate refinement into operation predicates.

Informal specification refinement can be carried out using the Object-Z views in SPE. For example, a designer may wish to refine the stack class specification to incorporate the `items` attribute state predicate into the `push` and `pop` method predicates. The methods will then reflect the state predicate, making translation of the specification into code much simpler [10]. The designer can modify the method specifications to achieve this. Figure three shows an example of refining the `push` method specification to incorporate the `items` attribute's state predicate. SPE's Object-Z views also help to support more complex refinement, such as refining two or more class specifications, by informing the designer of inconsistencies between different views of the specifications.

While this simple refinement of formal specifications is supported by SPE and its Object-Z views, more rigorous specification refinement cannot be supported, as the environment does not ensure the refinement steps used are correct. In order to support checking for correct refinements within a software engineering environment like SPE, we are developing views which utilise the refinement calculus.

3. Refinement

The refinement calculus has its roots in the work of Dijkstra [11] and Floyd [12] on the simultaneous development of programs and their proofs of correctness. This work is based on the simple observation, which underlies the common, informal, technique of stepwise refinement, that the production of a program always goes hand-in-hand with reasoning about why that program is the one that is needed to solve the given problem. They suggested that it would be useful to make both the program and the reasoning about it explicit—the usefulness comes about for many reasons: it gives a permanent record of the development process, which is invaluable for updating and maintaining any program; it allows as watertight as required an argument about the program's correctness relative to its specification; making reasoning explicit allows the programmer to see errors in reasoning more clearly than otherwise and tends to lead to clearer thinking.

The original work referred to above was informal, in the sense that there were equations which gave the meaning of the programming language in terms of pre- and postconditions on the state of the computation (the weakest precondition calculus) but no formalized rules about how to use these equations; the process of using the equations to develop the program was left up to the programmer's knowledge and intuition. The next step, to formalize the development process, had as its outcome (in the independent work of Morgan [13] and Morris [14]) the refinement calculus.

The following sections describe some of the refinement calculus and give an example of its use on a simple problem. We then discuss support tools for refinement, leading to the presentation of a proposed system which, we believe, will be an improvement over existing tools, such as SPE. We also consider the situation where several programmers are working on formal software development in a co-operative fashion. Our proposed tool supports many different views of the development process. This allows one person to concentrate on the reasoning that forms part of the development and another to concentrate on the specification.

4. The Calculus

Refinement takes place in a single language which allows developers to talk about both computation and specification—what might be termed a wide-spectrum language. In this language we have programs which might be 'all specification' or 'all code', or a mixture of both. For example, a <u>program</u> might contain some parts which are executable, the 'code' parts, and some which might not be, the 'specification' parts.

Program development, i.e. the act of making progress from a specification to code which meets it, is modelled by a sequence of programs related by a <u>refinement relation</u>. This is denoted by $\leq$. The first program in the sequence is the specification and the last

is the code—the first will typically not be executable and the last will be. Intermediate programs will be a mixture of executable and non-executable parts. Refinement is thus the production of a sequence of programs $p_0,...,p_n$ such that:

$$p_0 \leq p_1 \leq p_2 \leq ... \leq p_{n-1} \leq p_n$$

where p_0 is the initial specification and p_n is the final code.

The relation $\leq$, pronounced 'is refined by', holds between two programs p and p' if someone who would accept p as a solution to their problem would necessarily also accept p' too (ignoring, for the moment, the fact that some parts may not be executable on their target machine or in their target language).

$\leq$ is formally defined in terms of weakest preconditions, i.e. the least that needs to be true of a program's starting state in order that if it terminates it does so in some required final state which satisfies some postcondition ([13]). It turns out that $\leq$ has a very rich and interesting algebraic structure [15, 13, 14].

We will rely on examples and intuition in this paper to convey the meaning of $\leq$, including examples of $\leq$ holding between programs which are obviously executable. $\leq$ also holds equally well between non-executable programs (example adapted from [13]):

220V outlet ≤ 220/110V outlet
safe working load 1000kg ≤ safe working load 2000kg
needs at least 4Mb ≤ needs at least 2 Mb

In each of these cases the relation holds because anyone who would accept something on the left would accept the thing on the right.

5. Programs

A program will include many of the usual executable constructs (assignments, sequencing, conditionals and loops) as well as non-executable constructs, the most common of which is the specification statement, of the form:

$$\underline{v} : [P, Q]$$

where v is a sequence of variables, called a frame, whose values may be changed by the program. P is a precondition that must be true of any initial state in order for the program to start. Q is a postcondition which must be true for any state the program is in if it terminates. A state is defined by the values of the variables at any stage of the computation.

Typically the first program in a sequence of refinement steps will consist of a single specification statement. For example (taken from [13]), consider the program which may alter the value of the variable y, which has precondition $x \geq 0$ and postcondition $y^2 = x$:

$$y : [x \geq 0, y^2 = x] \qquad (p_0)$$

This program can be refined (into one that is equally acceptable to someone who would accept p_0) by strengthening its postcondition, i.e. by making a program which has more things true of it than p_0. An example of this for p_0 would be to add $y \geq 0$ to the postcondition to get:

$$y : [x \geq 0, y^2 = x \wedge y \geq 0] \qquad (p_1)$$

which is acceptable since $(y^2 = x \wedge y \geq 0) \rightarrow y^2 = x$ (N.B. $\rightarrow$ is implication). Then we have:

$$y : [x \geq 0, y^2 = x] \;\leq\; y : [x \geq 0, y^2 = x \wedge y \geq 0]$$

Alternatively, we could weaken the precondition. For example, since anything implies the always true proposition True, we have $x \geq 0 \rightarrow$ True, so:

$$y : [x \geq 0, y^2 = x] \leq y : [\text{True}, y^2 = x]$$

As a precondition True describes any state. In the specification on the right we are asking for a program which given any value for x makes $y^2 = x$. This is (assuming x and y are real) impossible, however, when x is negative. Some refinement steps thus lead us to programs which cannot be refined to code. Such programs are called infeasible.

To show that a refinement step is correct, each time we take a step certain proof obligations arise (for example, $(y^2 = x \wedge y \geq 0) \rightarrow y^2 = x$ above). These obligations require us to show that the program on the right of the refinement really does refine the program on the left. We give some examples of such steps and their proof obligations below. Note that, in a sense, the semantics of the wide-spectrum language are being given as we describe refinement steps and their associated proof obligations.

The style of the rules given below is a natural deduction style: if the statements above the horizontal line in the rule, the premises, have been proved then, in an extra step which uses the rule concerned, the statement below the line, the conclusion, has been proved.

6. Refinement Rules

assignment:

$$\frac{P \rightarrow Q[\underline{v}/\underline{e}]}{\underline{v} : [P\ ,\ Q[\underline{v}/\underline{e}]] \leq \underline{v} := \underline{e}}$$

$\underline{e}$ is a sequence of expressions of the same length as $\underline{v}$, $Q[\underline{v}/\underline{e}]$ is the sentence Q with all free occurrences of any variable in $\underline{v}$ replaced by the corresponding expression in $\underline{e}$.

This rule specifies that $\underline{v} : [P\ ,\ Q[\underline{v}/\underline{e}]]$ can be refined to $\underline{v} := \underline{e}$ if we can show that $P \rightarrow Q[\underline{v}/\underline{e}]$. In other words, we can perform the refinement step that gets us closer to an executable program as long as the premise to the rule is proved; the premise is an example of a proof obligation. Having proved this obligation we can be certain that the program on the right of the refinement really does refine the program on the left.

Just this one rule allows us to refine p_0 above into an executable program. The refinement step:

$$y : [x \geq 0, y^2 = x] \leq y := \sqrt{x} \qquad \text{(assignment)}$$

is justified by the rule 'assignment' (as recorded against the step) because the proof obligation:

$$x \geq 0 \rightarrow (y^2 = x)[y/\sqrt{x}]$$

can be proved, as in:

$x \geq 0 \quad \rightarrow$ (by definition of $\sqrt{\ }$)

$(\sqrt{x})^2 = x \quad \rightarrow$ (by definition of substitution)

$(y^2 = x)[y/\sqrt{x}]$

sequence:

$$\frac{}{\underline{v} : [P\ ,\ Q] \leq \underline{v} : [P\ ,R]\ ;\ \underline{v} : [R\ ,Q]}$$

This rule allows us to break large programs into sequences of smaller ones as long as the sequencing will work, which is expressed by saying that the postcondition of the first part is the same as the precondition of the second. Note that this refinement step involves us in no new proof obligations.

The next refinement rule allows us to define local variables (introduced by **var**) within a new scope (denote by pairs of braces {...}).

vars:

$$\frac{\text{the variables in } \underline{w} \text{ are new}}{\underline{v} : [P,Q] \leq \{\mathbf{var}\ \underline{w} \mid \underline{w}, \underline{v} : [P,Q]\}}$$

Another example, which uses all the rules given so far: let c and d be some real numbers and consider the specification

$$x,y : [x = c \wedge y = d,\ x = d \wedge y = c]$$

This can be refined to an executable form as follows:

$x,y : [x = c \wedge y = d,\ x = d \wedge y = c]\ \leq$ (vars)

$\{\mathbf{var}\ t \mid t,x,y : [x = c \wedge y = d,\ x = d \wedge y = c]\} \leq$
(sequence)

$\{\mathbf{var}\ t \mid t,x,y : [x = c \wedge y = d,\ t = c \wedge y = d]\ ;$
$t,x,y : [t = c \wedge y = d,\ x = d \wedge y = c]\} \leq$
(assignment and ob1)

$\{\mathbf{var}\ t \mid t := x\ ;$
$t,x,y : [t = c \wedge y = d,\ x = d \wedge y = c]\} \leq$
(sequence)

$\{\mathbf{var}\ t \mid t := x\ ;$
$t,x,y : [t = c \wedge y = d,\ t = c \wedge x = d]\ ;$
$[t = c \wedge x = d,\ y = c \wedge x = d]\} \leq$
(assignment and ob2)

$\{\mathbf{var}\ t \mid t := x\ ;\ x := y\ ;$
$[t = c \wedge x = d,\ y = c \wedge x = d]\} \leq$
(assignment and ob3)

$\{\mathbf{var}\ t \mid t := x\ ;\ x := y\ ;\ y := t\}$

where ob1 is

$$(x = c \wedge y = d) \rightarrow (t = c \wedge y = d)[t/x]$$

ob2 is

$$(t = c \wedge x = d) \rightarrow (t = c \wedge y = d)[x/y]$$

and ob3 is

$$(t = c \wedge x = d) \rightarrow (y = c \wedge x = d)[y/t]$$

all of which are trivial to prove.

7. Refinement tools

There are several tools either under development or being used experimentally which support the building of refinements (for example see [7]). These tools tend to support single, textual views of program refinement, with no preservation of old refinement steps.

The tool we are proposing is unusual in that it will support several views: first, there will be the refinement world view and the proof world view; second, within the refinement world there will be views (linked by refinement steps) of the refinement sequences; third: within the proof world (and linked to the refinement world by the proof obligations) there will be the separate proofs of each obligation. The refinement views will be used both by one person or by several working on different parts of the development or cooperating on a single part.

Being able to move between parts of the development and having the parts structured in this way would prove very useful, both for forming a reliable model of a complicated process and for organizing the mass of partial refinements, partial proofs and the relationships between them. Figure four shows an example of these refinement views.

Notice that there are three views needed here:

- the sequence of refinement sequences, which forms a history of the development
- the current refinement step, leading to a relation between the most recent refinement sequence and a new refinement sequence
- the proof obligations, arising from each refinement step, and their proofs.

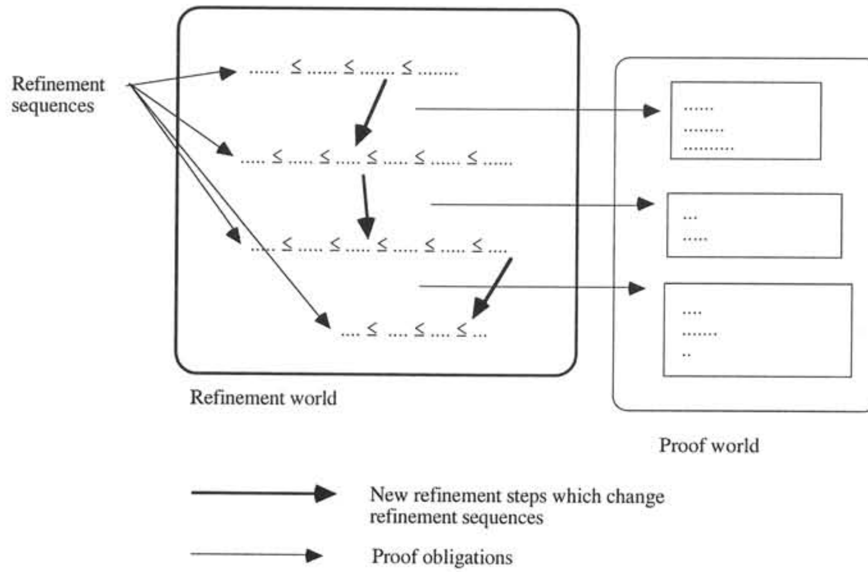

Figure four. Refinement views and proof obligation views.

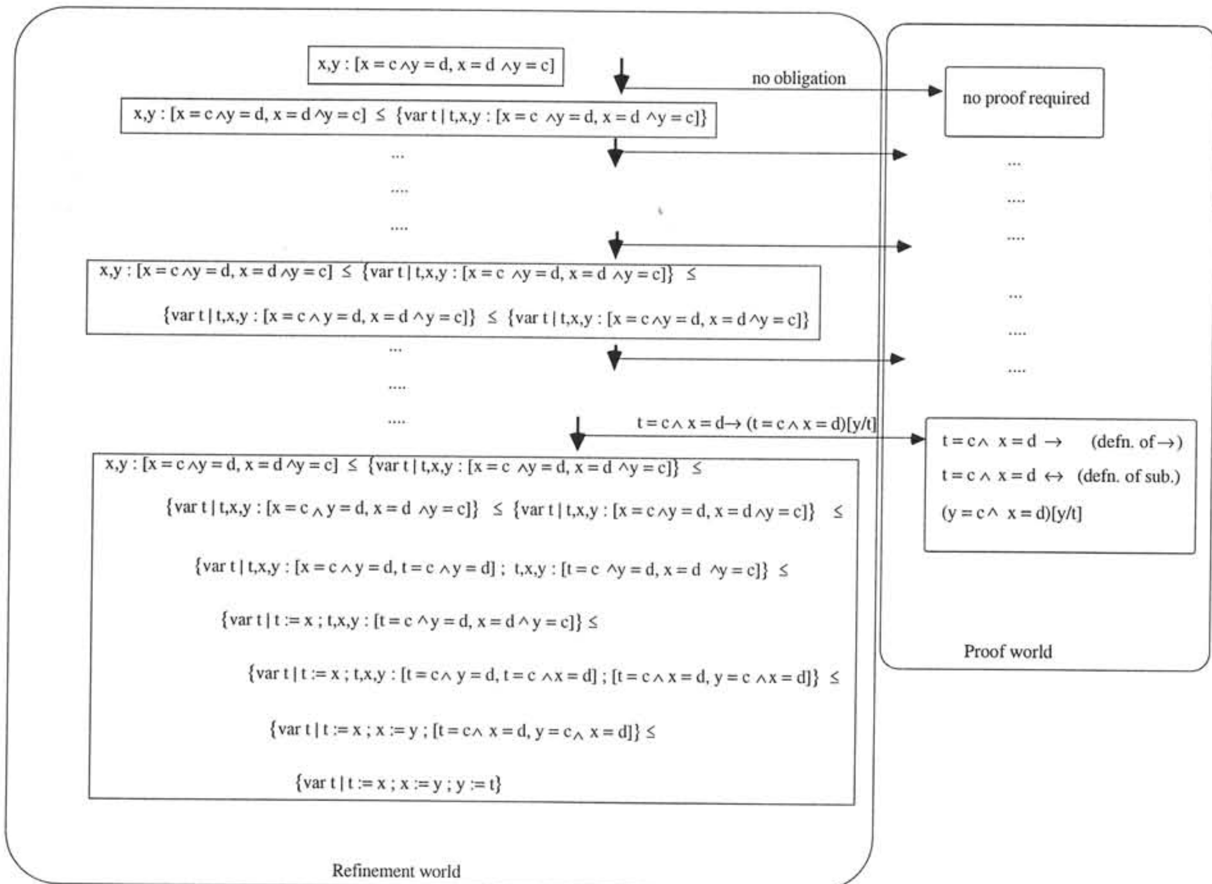

Figure five. Refinement and proof obligation views for the swapping refinement example.

Figure five shows an example of taking the refinement from above and viewing it in this way.

These refinement views must be provided to users of the environment in such a way that they can be browsed and modified as necessary. They must also be integrated with views provided by an environment like SPE, which support informal design and implementation. These informal views can be used to both provide higher-level structuring of refinement views, and to allow a low-level program specification to be translated into an implementation in an executable programming language.

Figure six shows an example of refinement views for the example from above. The first specification in the refinement is shown (Step 1), along with the final specification (Step 6).

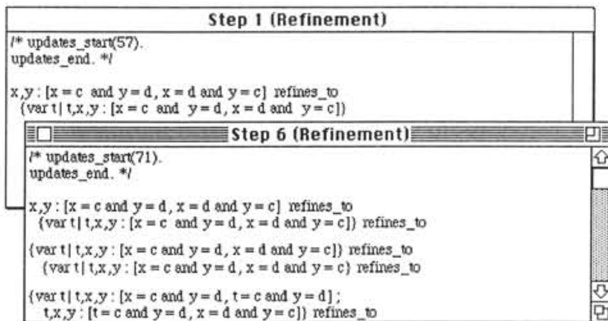

Figure six. Refinement views.

Figure seven shows additional proof obligation views for the example from above.

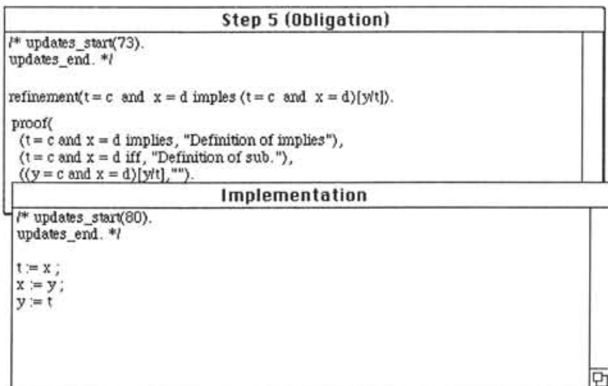

Figure seven. Proof obligation views.

These diverse views of software development must be kept consistent to make the environment truly useful for integrated software development. The SPE approach of expanding change descriptions into views has been used to propagate changes between views. For example, consider the example in figure eight.

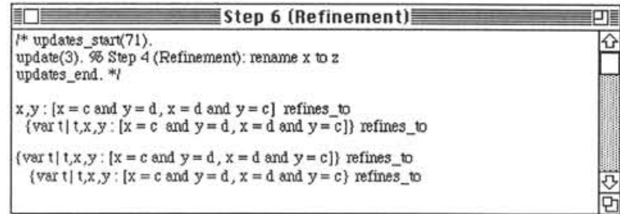

Figure eight. View consistency management.

This shows a very simple example of consistency management where the name of a variable has been changed in an earlier refinement view. All affected refinement and proof obligation views have a change description shown in them to indicate the change. Some of these inconsistencies could be resolved automatically by the environment, such as variable renaming. Others, such as changing a refinement step or adding extra pre- or post-conditions, would normally be manually implemented. The propagated change descriptions in these circumstances provide documentation to the designers that inconsistencies exist and need to be resolved.

8. Architecture and Implementation

SPE with Object-Z views is implemented as a collection of classes, specialized from the MViews framework [16, 17]. MViews supports the construction of Integrated Software Development Environments (ISDEs) by providing a general model for defining software system data structures and tool views, with a flexible mechanism for propagating changes between software components, views and distinct software development tools.

MViews describes ISDE data as components with attributes, linked by a variety of relationships. Multiple views are supported by representing each view as a graph linked to the base software system graph structure. Each view is rendered and edited in either a graphical or textual form. Distinct environment tools can be interfaced at the view level (as editors), via external view translators, or multiple base layers may be connected via inter-view relationships.

When a software or view component is updated, a change description is generated. This is of the form `UpdateKind(UpdatedComponent, ...UpdateKind-specific Values...)`. For example, an attribute update on `Comp1` of attribute `Name` is represented as: `update(Comp1,Name,OldValue,NewValue)`. All basic graph editing operations generate change descriptions and pass them to the propagation system. Change descriptions are propagated to all related components that are dependent upon the updated component's state. Dependents interpret these change descriptions and possibly modify their own state, producing further change descriptions. This change description mechanism supports a diverse range of software development environment facilities, including semantic attribute recalculation, multiple views of a component, flexible, bi-directional textual and graphical view consistency management, a generic undo/redo mechanism, and component "modification history" information.

We are currently building our formal refinement environment using MViews. This involves reusing

abstractions provided by an object-oriented framework based on the MViews model. We are specialising MViews classes to define software components, views and editing tools to produce our new refinement environment. A persistent object store is used to store component and view data.

9. Summary

We are currently in the process of building the tool described above. As ever, this is turning out to be an experimental enterprise, but the ideas presented here form a firm foundation on which we are basing our implementation.

We expect that this tool will have advantages over current refinement support tools because we will have several views of the development of a program and each view will be maintained in a consistent way relative to the others. So, experimentation with different refinement steps at a point in the development will be less fraught than it currently tends to be—having done all the work based on one of the alternative steps at a given point, the thought of having to retrace one's steps in order to try another alternative can be a disincentive to bother with the alternatives. This tendency clearly has a negative effect on the exploration of designs at this very high and formal level, which tends to lead into question the point of working in this way in the first place.

References

[1] Diller, A., *Z: An Introduction to Formal Methods* (2nd edition), John Wiley and Sons, 1994.

[2] Ince, D.C., *An Introduction to Discrete Mathematics, Formal System Specification and Z* (2nd edition), Oxford University Press, 1992.

[3] Wordsworth, J.B., *Software Development with Z*, Addison-Wesley, 1992.

[4] Duke, R., King, P., Rose, G., and Smith, G., "The Object-Z Specification Language Version 1", Tech. Rep. TR 91-1, SVRC, University of Queensland, 1991.

[5] Jones, C.B., *Systematic Software Development using VDM*, Prentice-Hall, 1986.

[6] Jia, X. , "ZTC: A Type Checker for Z Notation: User's Guide", version 2.01, May 1995, Division of Software Engineering, DePaul University, Chicago, U.S.A, 1995.

[7] Groves, L, Nickson, R. and Utting, M., "A Tactic Driven Refinement Tool", Technical Report CS-TR-92/5, Department of Computer Science, Victoria University of Wellington, November 1992.

[8] Grundy, J.C., Hosking, J.G., Fenwick, S., and Mugridge, W.B., "Connecting the pieces", Chapter 11 in *Visual Object-Oriented Programming*. Prentice-Hall, 1994.

[9] Grundy, J.C. Hosking, J.G. Software environment support for integrated formal program specification and development, accepted for the 1995 Asia-Pacific Software Engineering Conference, Brisbane, Australia, 7-9 December, 1995, to be published by the IEEE CS Press.

[10] Johnston, W. and Rose, G., "Guidelines for the manual conversion of Object-Z to C++", Tech. Rep. 93-14, SVRC, University of Queensland, 1993.

[11] Dijkstra, E.W., "A Constructive Approach to the Problem of Program Correctness", BIT, vol. 8, no. 3, pp. 181-185, 1968.

[12] Floyd, R.W., "Towards Interactive Design of Correct Programs", technical report CS-235, Stanford University, September 1971.

[13] Morgan, C., *Programming from Specifications*, Prentice-Hall, 1990.

[14] Morris, J.M., "A Theoretical Basis for Stepwise Refinement and the Programming Calculus", Science of Computer Programming, vol. 9, 1987, pp. 287-306.

[15] Back, R.J.R., "A Calculus of Refinements for Program Derivations", Acta Informatica, vol. 25, pp. 593-624, 1988.

[16] Grundy, J.C. and Hosking, J.G., "A framework for building visusal programming environments", in *Proceedings of the 1993 IEEE Symposium on Visual Languages,* IEEE Computer Society Press, 1993, pp. 220-224.

[17] Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Amor, R., "Support for Collaborative, Integrated Software Development", in *Proceedings of the 7th Conference on Software Engineering Environments,* Netherlands, April 5-7 1995, IEEE CS Press, pp. 84-94.