Computer Assistance for Program Refinement

D.A. Carrington
Key Centre for Software Technology
Department of Computer Science
University of Queensland

K.A. Robinson
Department of Computer Science
University of New South Wales

Abstract

This paper explores the role for mechanised support for refining specifications to executable programs. The goal of refinement is to achieve the translation from specification to implementation without the introducing errors. The refinement calculus provides a set of rules for developing procedural programs from abstract specifications. A prototype editor for the refinement calculus is described that was constructed using the Cornell Synthesizer Generator. Based on our experiences, desirable features for future tools are suggested.

1 Introduction

Software development requires a process to transform an abstract specification into an executable substitute, that is, a computer program that exhibits the required behaviour. The transformation process is often referred to as refinement and normally requires a sequence of steps. These steps represent the design decisions leading to the choice of algorithms and of data representations. Formalizing the refinement process is important to establish the correctness of the product. A program is *correct* if it is a satisfactory implementation of the original specification. Correctness may be established by testing (almost always infeasible), by constructing a program proof (usually difficult if attempted after the program has been constructed), or by using a refinement strategy that ensures that each step in the development process maintains the original requirements.

The starting process for refinement is a specification. A major function of a specification is to assist communication between the developer and the client with the objective that both parties understand the intended behaviour of the product to be created.

There is growing acceptance of the benefits of formal specification methods that avoid the ambiguity of natural language. The objective of the specification is to define *what is required* of the product and not *how it is to be achieved.* A specification language should encourage precision, conciseness, and abstraction. It is also desirable to be able to manipulate the specification and prove properties or consequences of the specification. In this way, confidence in the specification as an accurate reflection of the requirements can be established. Unless the specification is expressed in a formal notation, it is not possible to have a systematic method of verifying that an implementation is correct.

Mathematical specification techniques using predicate logic, such as the Vienna Definition Method (VDM) [1,7] and Z [6,16] describe program behaviour as a relation between the input and output states. This allows us to capture the essential behavioural characteristics of a system or algorithm without concern for the implementation. Such specifications have the desirable properties listed above but are not suitable for expressing the concrete algorithms and data structures of the executable program. VDM also incorporates refinement techniques and is one of the few sustained research efforts to tackle this problem.

Having produced a specification that is considered to meet the informal requirements, we want to be able to refine the specification by changing the data structures and introducing programming language constructs until we have an implementation: an executable substitute. The aim of a formal refinement methodology is to ensure that at all times the refinement is in some essential way *consistent* with the initial specification. There are considerable advantages from using uniform notation throughout the refinement process. At the very least, it avoids the difficulties associated with converting between notations. There are two possible approaches: adding programming language elements to a specification language, or adding specification constructs to a programming language. Whichever approach is taken, it is crucial to have a formal semantics that encompasses the extended language. The refinement calculus method adopts the approach of adding specification to a programming language with the extended language defined using Dijkstra's weakest pre-condition semantics [3]. Section Two provides an overview of the refinement calculus and the specification statement on which it is based.

As well as ensuring that the final program is a correct implementation of the initial specification, the refinement steps also provide an excellent design history by documenting the choices made during the development. In that sense, the refinement calculus provides a more rigorous version of the development style that Knuth [8] calls *literate programming*. In section Three, we consider suitable roles for tools to assist in the application of the refinement calculus.

Section Four reviews a prototype refinement editor, a tool for developing programs from specifications based on the rules of the refinement calculus. Based on our experiences with this tool, we make some suggestions for future tools.

2 The refinement calculus

The refinement calculus developed by Morgan and Robinson [11,12,9] provides a rigorous technique for deriving an executable program from an abstract specification. The essential ingredients of the calculus are

- a programming language (with a formal semantics) that we use as our target language. We presume that this language can be executed on a computer. We follow Dijkstra and use both a guarded command language and a weakest pre-condition semantics. It should be noted that any programming language could be used, provided that it is given a weakest pre-condition semantics. In our work we denote the weakest pre-condition of S with respect to R by the functional application of S to R, *i.e.*

$$S\ R \quad \triangleq \quad \mathrm{wp}(S, R)$$

- a specification construct that is abstract *i.e.* it must allow the specification of programs without being restricted by implementation concerns that are present in any *real* programming language. Our formal semantics must include the specification construct.
- a formal definition of the notion of specification transformation, that we call *refinement.*

2.1 A specification construct

The specification statement [10] captures the essential behaviour of a program or part of a program, without concern for implementation issues. It is denoted

$$\vec{w} : \left[Pre \;/\; Post \right]$$

and denotes a program that, executed in a state satisfying Pre, will terminate in a state satisfying $Post$ while modifying only variables in $\vec{w}$ (where $\vec{w}$ is a subset of the program variables $\vec{v}$). Pre is a predicate in $\vec{v}$ while $Post$ is a predicate in $\vec{v}$ and $\vec{v}_0$ where the decorated variables $\vec{v}_0$ refer to the corresponding values in the initial state.

The weakest pre-condition for $\vec{w} : \left[Pre \;/\; Post \right]$ to terminate in a state satisfying R must be at least as strong as Pre and every program state that satisfies $Post$ must also satisfy R. The formal definition is

$$\vec{w} : \left[Pre \;/\; Post \right] \; R \quad \triangleq \quad Pre \wedge \forall \left(\vec{w} \bullet Post \Rightarrow R \right)_{[\vec{v}_0 \setminus \vec{v}]}$$

where $E_{[\vec{x} \setminus \vec{e}]}$ is the expression E with all free occurrences of each of the variables in $\vec{x}$ replaced by the respective expression in $\vec{e}$. We assume that the $\vec{v}_0$ are local to $Post$ and hence are not contained in R; otherwise, a further renaming is required to avoid confusion.

Two syntactic variations are

- $\vec{w} : \left[Post \right]$ where the implicit pre-condition is $\exists \left(\vec{w} \bullet Post \right)_{[\vec{v}_0 \setminus \vec{v}]}$, the weakest condition for the existence of a program state satisfying $Post$.
- $\vec{w} : \left[Pre \;/\; I \;/\; Post \right]$ where I is a predicate in $\vec{v}$ that is invariant, *i.e.* it is true in both the initial and the final state. It is a shorthand for $\vec{w} : \left[Pre \wedge I \;/\; I \wedge Post \right]$.

We introduce an example that is used subsequently to illustrate some refinement rules. The example is very simple and is chosen to demonstrate the principles involved. We wish to implement $\mathsf{Min}(b, N, m)$ that computes the minimum value in the array $b[0..N-1]$, passing back the result in m. This is represented by the following specification statement:[1]

$$\mathsf{Min}(b, N, m) \quad \triangleq \quad m : \left[N > 0 \;/\; Post \right] \; \textit{where} \; Post \quad \triangleq \quad \begin{array}{l} \exists \left(i : 0..N-1 \bullet m = b[i] \right) \\ \forall \left(i : 0..N-1 \bullet m \leq b[i] \right) \end{array}$$

[1] We use vertical stacking of conjuncts to denote conjunction.

2.2 Refinement

With the refinement calculus, the development of a program is a process of replacing specification statements by other programming constructs using the rules of the calculus to perform the transformations. To ensure that the resulting program is a valid implementation of the initial specification, we require an ordering between specifications that captures the idea that one specification (S_i) may be replaced by another (S_j) in any context. This ordering is denoted $S_i \sqsubseteq S_j$. If in all initial states in which S_i does not abort, the set of final states of S_j is contained in the set of final states of S_i, then we say S_i is refined by S_j. The formal definition uses weakest pre-conditions:

$$S_i \sqsubseteq S_j \text{ iff } (S_i\ R) \Rightarrow (S_j\ R) \text{ for all } R$$

Based on this concept of refinement, we can depict the design process as the sequence

$$S_0 \sqsubseteq S_1 \sqsubseteq \cdots \sqsubseteq S_n$$

where S_0 is the initial specification and S_n is our executable program.

2.3 Refinement example

The rules of the refinement calculus express valid transformations of specifications. Many of the rules have an associated applicability condition that specifies when the rule may be used. Rather than attempting to supply a comprehensive set of rules, we give an example refinement. Consult [9] or [11] for exposition of the refinement rules.

A little contemplation of the specification statement for Min suggests that we need an additional variable to control the computation. The construct $[\![\ \mathbf{var}\ \vec{t}\ \bullet\ S\ ]\!]$ introduces new variables $\vec{t}$[2] into the program state inside the block delimited by the symbols $[\![$ and $]\!]$. Applying the appropriate rule to our example, we get

$$m\colon \left[N > 0\ /\ Post\right] \quad \sqsubseteq \quad [\![\mathbf{var}\ j \bullet m,j\colon \left[N > 0\ /\ Post\right]]\!] \quad \left\{\begin{matrix}\textit{Variable}\\ \textit{introduction}\end{matrix}\right\}$$

Introducing an intermediate predicate we refine the inner specification to a sequential composition.

$$m,j\colon \left[N > 0\ /\ Post\right] \quad \sqsubseteq \quad m,j\colon \left[N > 0\ /\ I\right] ; m,j\colon \left[I\ /\ Post\right] \quad \left\{\begin{matrix}\textit{Sequential}\\ \textit{de-composition}\end{matrix}\right\}$$

$$\textit{where } I \quad \triangleq \quad \begin{pmatrix} 0 < j \leq N \\ \exists \left(i\colon 0..j-1 \bullet m = b[i]\right) \\ \forall \left(i\colon 0..j-1 \bullet m \leq b[i]\right) \end{pmatrix}$$

then $m,j\colon \left[N > 0\ /\ I\right] \sqsubseteq m,j := b[0],1 \quad \{\textit{Assignment introduction}\}$.

We intend to refine the second component to a loop, so we have to choose a set of guards $\{B_i\}$ and a variant function. We choose a single guard ($j \neq N$) observing that

$$I \wedge j = N \Rightarrow Post$$

[2]In this example, and in the discussion in general, we ignore the type of variables. Introducing typed variables adds a set membership constraint to the pre- and post-conditions.

and $N - j$ as our variant function to establish termination. Then

$$m,j\colon \left[\, I \;/\; Post \,\right]$$
$$\sqsubseteq \textbf{do}\ j \neq N \rightarrow m,j\colon \left[\, j \neq N \;/\, I \,/\; N - j < N - j_0 \,\right] \ \textbf{od} \quad \{\textit{Do introduction}\}$$

It is obvious that the simplified postcondition, $j > j_0$, of the imbedded specification can be satisfied by incrementing j, so we use the weakest prespecification rule to refine to a sequential composition in which the second component is a specification that increments j.

$$m,j\colon \left[\, j \neq N \;/\, I \,/\; j > j_0 \,\right]$$
$$\sqsubseteq m,j\colon \left[\, j \neq N \wedge I \;/\; \begin{pmatrix} I \\ j > j_0 \end{pmatrix}_{[j \setminus j+1]} \,\right] ; j\colon \left[\, j = j_0 + 1 \,\right] \quad \left\{ \begin{matrix} \textit{Weakest} \\ \textit{prespecification} \end{matrix} \right\}$$

The second component refines to the assignment $j := j + 1$ and the first component is simplified, refined by deleting the variable j from the window and then reorganized to reveal an invariant component as follows:

$$m,j\colon \left[\, j \neq N \wedge I \;/\; \begin{pmatrix} I \\ j > j_0 \end{pmatrix}_{[j \setminus j+1]} \,\right]$$
$$= m,j\colon \left[\, j \neq N \wedge I \;/\; \begin{matrix} 0 < j+1 \leq N \\ \exists \big(i : 0..j \bullet m = b[i]\big) \\ \forall \big(i : 0..j \bullet m \leq b[i]\big) \\ j+1 > j_0 \end{matrix} \,\right]$$
$$\sqsubseteq m\colon \left[\, j \neq N \wedge I \;/\; \begin{matrix} 0 < j+1 \leq N \\ \exists \big(i : 0..j \bullet m = b[i]\big) \\ \forall \big(i : 0..j \bullet m \leq b[i]\big) \end{matrix} \,\right] \quad \{\textit{Restrict variables}\}$$
$$= m\colon \left[\, \exists \big(i : 0..j-1 \bullet m = b[i]\big) \;/\; \begin{matrix} 0 < j < N \\ \forall \big(i : 0..j-1 \bullet m \leq b[i]\big) \end{matrix} \;/\; \begin{matrix} \exists \big(i : 0..j \bullet m = b[i]\big) \\ m \leq b[j] \end{matrix} \,\right]$$

The final specification above is now refined four ways:

1. to the guarded command $m \leq b[j] \rightarrow \textbf{skip}$ $\{\textit{Skip introduction}\}$;
2. to the guarded command $m \geq b[j] \rightarrow m := b[j]$ $\{\textit{Guarded command introduction}\}$;
3. to the union of the two guarded commands $\{\textit{Union introduction}\}$;
4. to an if-statement containing that union $\{\textit{If introduction}\}$.

$$\underbrace{m\colon \left[\, \exists \big(i : 0..j-1 \bullet m = b[i]\big) \;/\; \begin{matrix} 0 < j < N \\ \forall \big(i : 0..j-1 \bullet m \leq b[i]\big) \end{matrix} \;/\; \begin{matrix} \exists \big(i : 0..j \bullet m = b[i]\big) \\ m \leq b[j] \end{matrix} \,\right]}$$

$$\underbrace{m \leq b[j] \rightarrow \textbf{skip}}_{(1)} \quad \underbrace{m \geq b[j] \rightarrow m := b[j]}_{(2)} \quad \underbrace{\begin{matrix} m \leq b[j] \rightarrow \textbf{skip} \\ [\!] \\ m \geq b[j] \rightarrow m := b[j] \end{matrix}}_{(3)} \quad \underbrace{\begin{matrix} \textbf{if} \\ \quad m \leq b[j] \rightarrow \textbf{skip} \\ \quad [\!] \\ \quad m \geq b[j] \rightarrow m := b[j] \\ \textbf{fi} \end{matrix}}_{(4)}$$

Assembling the refinements we get the following program:

$$
\begin{array}{l}
[\![\textbf{var } j \bullet m, j := b[0], 1; \\
\quad \textbf{do } j \neq N \rightarrow \\
\qquad \textbf{if } m \leq b[j] \rightarrow \textbf{skip} \\
\qquad [\!] \ m \geq b[j] \rightarrow m := b[j] \\
\qquad \textbf{fi}; \\
\qquad j := j + 1 \\
\quad \textbf{od} \\
]\!]
\end{array}
$$

The preceding example dealt exclusively with *procedural* refinement. The refinement calculus also contains rules for *data* refinement where abstract types in a program are replaced by more concrete types. For more details see [12].

3 Tool roles

The refinement calculus is intended to improve the quality of the software development process. However applying the calculus involves many small steps where each step may require the computation of a new specification and/or the checking of an applicability condition associated with a rule. While most of these are very straight-forward, there is scope for errors to be introduced by the number of steps.

From this observation we envisage two major roles for refinement calculus tools:

- correctly applying the rules of the refinement calculus as directed by the tool user, and
- recording the sequence of refinement steps in a software development.

The first role requires establishing that the applicability condition associated with each rule is satisfied whenever it is used and performing the computations of the calculus. We do not expect to automate software development; our aim is to provide systematic support. Thus the human designer retains the initiative in the design process with the tool playing the *careful assistant* role with responsibility for confirming the viability of each refinement step. Since the proof obligation of each step is a lemma in the first order predicate calculus, we cannot expect automatic confirmation of all steps. While some are easy to prove, many rely on domain knowledge. Some form of interactive proof editor seems necessary. This approach to tools for software development is not new; Floyd [4] predicted it and many people since have adopted this style for co-operative interaction. Perhaps the best known work is the programmer's apprentice project at MIT [15,17].

The recorded design history based on the refinement steps is a valuable artifact of the design process. A tool simplifies the task of capturing this information systematically and accurately. The design history can serve several purposes:

- to explain and document the current design
- for future modification by incremental changes to part of the refinement sequence
- for re-use in different contexts

4 A prototype refinement editor

A prototype refinement editor [2] has been built as an interactive tool for the refinement calculus. It was constructed using the Synthesizer Generator [13,14] which generates language-based editors from descriptions of an abstract syntax and display representation. Attribute grammars are used to incorporate context-sensitive constraints and incremental re-evaluation algorithms to ensure that a consistent document exists after each editing operation.

The development of a prototype was simplified by using parts of Bill Pugh's pv demonstration editor supplied with the Synthesizer Generator distribution. The pv editor is a preliminary program verification editor for Dijkstra's guarded command language. The editor generates verification conditions and incorporates an interactive simplifier for proving them. Using pv, programs are entered statement by statement with the verification conditions confirming the correctness of the program.

Our refinement editor manipulates similar objects but the development style is quite different. From pv we have been able to use the basic structures for predicates, predicate simplification and expression substitution for computing weakest pre-conditions. We also chose the same domain of discourse: integer arithmetic with either simple or array variables. The prototype is restricted to procedural refinement.

The abstract syntax for the refinement editor represents the refinement process as a tree with the initial specification as the root node and each edge corresponding to a refinement step. When the refinement is complete, the leaf nodes correspond to constructs of the executable programming language. Proof subtrees are attached to nodes to verify the applicability condition of each refinement rule. Some of these can be immediately simplified but there is the capability for user interaction for more difficult cases.

The refinement rules of the calculus are provided as transformation commands that act on selected nodes in the tree by extending the tree with an additional refinement step. The Synthesizer Generator displays a sub-window of valid transformation commands whose contents depend on the current edit selection.

The editor uses two display unparsing schemes to present two views of the program under development:

- the primary view is a linear exposition of the refinement steps based on a top-down depth-first traversal of the tree.
- the alternative view is a consolidation of the refinement tree to extract the "final" program. This displays only the leaf nodes. Parts of the program that are not fully refined are displayed as specification statements.

The editor user can dynamically choose either view for independent subtrees so partial compression or expansion is possible.

4.1 Experiences

While our first refinement editor is very rudimentary, it has provided considerable insight into the challenge of building support tools for the refinement calculus. Basing the support tool on the editing paradigm has been a good idea since it encourages an

exploratory style where the user investigates refinements, deleting those that are not successful. We have used it to explore the refinement of many small programming examples and we are enthusiastic about the potential of the approach.

As an illustration, after using the editor to develop the Min example, we investigated how easily it could be modified to compute Max instead. The necessary editing changes were to

1. the initial specification (which then propagated through the complete refinement tree),
2. the guards within the if command(or equivalently the guarded statements), and
3. the intermediate predicate used to create the first sequential composition.

The last step was required because this predicate was entered by hand rather than by computation based on the goal post-condition. It would certainly be feasible to incorporate rules to transform predicates in this manner. This example highlights the benefits of recording the design steps and having the capability to incrementally modify individual steps and observe the consequences. Entering information once and then propagating it wherever it is required is convenient compared to performing refinements by hand where there is potential for transcription errors.

There are, of course, shortcomings with the prototype in the scope of problems that can be tackled and with performance. The Synthesizer Generator provides a realistic prototype very quickly and effectively but we encountered some limitations in the flexibility of the user interface.

5 Future tools

In future research into tools for the refinement calculus we plan to investigate the following issues:

Notation If the use of the predicate calculus is to be successful, we need to abbreviate our pre- and post-conditions. The normal way to do this is by defining names (possibly parameterized) for subcomponents of the predicates. Any more substantial tool needs this capability but must also be able to manipulate predicates that contain such abbreviations without requiring complete expansion. The work of Griffin [5] is of interest here.

Interactive proof techniques A realistic refinement editor will rely heavily on some form of interactive theorem prover for first-order predicates. The ability to manipulate predicates is crucial to the viability of a refinement tool. Also necessary are methods to establish and use theories about

- program objects — both abstract data types used in specifications such as sets and sequences and the concrete data types of our programming language.
- problem domain knowledge

User Interface The user interface for a refinement tool is extremely important. How to present the developing refinement sequence and provide facilities for navigating

though and modifying that sequence is a significant challenge. Some of the problems are similar to those addressed by hypertext systems and we are interested to see if we can use results from that research. The user of a refinement tool needs to focus on different concerns at different stages of the refinement task. Once a refinement step has been established by verifying the applicability of the rule, the proof is normally of little concern and need not be displayed. By comparison, during the proof construction, we are unlikely to be interested in the refinement path leading to this proof.

Data refinement Extending our work to include data refinement will require a slightly different approach. Instead of focusing on the refinement of a single specification, data refinement is distributed over a larger scope and requires the transformation of each component of the scope. The transformations are driven by an abstraction invariant that relates the concrete variables to the abstract ones.

Storage and retrieval We see a need to go beyond a monolithic design document for each program. If the potential for reusing the design history is to be achieved, more flexible methods of storage and (particularly) retrieval are required. The possible application of object-oriented data base technology will be investigated.

Calculus rules The refinement calculus rules are not an inviolate set. There is a common core of rules which will be augmented and extended by new rules as experience with the calculus grows. Future tools should make it easy to add new rules and package derived rules (combinations of more basic rules). This provides potential difficulties as a refinement sequence is dependent on the rules used in its development. It may be necessary to store the rules used with each refinement sequence.

6 Conclusions

The development process is at least as important as the product, the executable software, since it serves to document the design and provides a method for checking the correctness of the program (with respect to the initial specification). The refinement calculus is a formal method for refining specifications. Applying the calculus effectively requires computer-based support to manage the many steps from the initial abstract specification to the final executable program. In the paper we have summarized our experiences with tool support for the refinement calculus and made suggestions about goals for future tools.

7 Acknowledgements

We gratefully acknowledge the influence and inspiration of the research on specification at the Programming Research Group, Oxford University. The refinement calculus was developed during Ken Robinson's study leave at PRG in 1985/86. Special thanks are owed to Carroll Morgan for his collaboration on the development of the refinement calculus.

This research is partially supported by an Australian Research Council grant.

References

[1] D. Bjorner and Jones C.B. *Formal Specification and Software Development.* Prentice-Hall International, 1982.

[2] D.A. Carrington and K.A. Robinson. A prototype program refinement editor. In *Australian Software Engineering Conference*, pages 45–63. ACS, 1988.

[3] Edsgar W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[4] R.W. Floyd. Toward interactive design of correct programs. In *IFIP*, pages 7–10, 1971.

[5] T.G. Griffin. *Notational Definition and Top-Down Refinement for Interactive Proof Development Systems.* PhD thesis, Dept. of Computer Science, Cornell University, 1988.

[6] I.J. Hayes, editor. *Specification Case Studies.* Prentice-Hall International, 1987.

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

[8] D.E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[9] Carroll Morgan. *Programming from Specifications.* International Series in Computer Science. Prentice-Hall, 1990.

[10] C.C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[11] C.C. Morgan and K.A. Robinson. Specification statements and refinement. *IBM Journal of Research and Development*, 31(5):546–555, September 1987.

[12] C.C. Morgan, K.A. Robinson, and P. Gardiner. On the refinement calculus. Technical Report PRG-70, Programming Research Group, 8-11 Keble Road, Oxford OX1 3QD, UK, 1988.

[13] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator.* Springer-Verlag, 1989.

[14] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual.* Springer-Verlag, third edition, 1989.

[15] C. Rich and H.E. Shrobe. Initial report on a LISP programmer's apprentice. *IEEE Transaction on Software Engineering*, 4(6):456–467, 1978.

[16] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International, 1989.

[17] R.C. Waters. The programmer's apprentice: Knowledge based program editing. *IEEE Transaction on Software Engineering*, 8(1):1–11, 1982.