Abstract

We present a method for formal program development, designed specifically for use at an introductory level. It formalises stepwise refinement in a natural way by regarding specifications as unimplemented program components. A graphical notation is used for specifications and their refinement rules which supports this approach.

1. Introduction

One well-known aspect of the software crisis is the difficulty of producing highly reliable programs, even for applications in which reliability is critical. Although formal methods of program development appear to provide a promising way of attacking this problem, they have been slow in gaining industrial adoption. It is widely accepted that a major reason for this is the poor technical education of software workers. The great majority of software workers have been taught in the conventional manner, which can be characterised as encouraging students (or trainee programmers) to produce, as quickly as possible, large programs which they know will contain serious errors. Later they may be introduced to methods for precise reasoning about programs, but this topic is presented with such a wide separation in time, style of motivation and subject matter from the original introduction to programming that it appears to be a quite separate subject. Clearly it would be much better to teach the logical foundations of programming together with its practical applications, from the very start.

Of the two main approaches to teaching formal methods in imperative programming, neither that following the tradition of [1] nor that of constructive specification have so far achieved this integration. It will be argued in this paper (§6) that the approach of the former is inappropriate to introductory teaching. On the other hand, the pedagogic material presenting constructive specification methods (eg [5], [3]) reflects the primary purpose of these methods, that is their application to industrial software engineering problems. To make them suitable for introductory teaching, whether to novices or experienced programmers, attention must be given to the choice of suitable refinement rules and to matters of presentation (such as an appropriate concrete syntax).

2. The Programming Language

We present a diagram notation (the "box notation") for depicting stepwise refinement, specifications (§3), and program correctness (§5). An algorithm is expressed in this notation by a structure of nested boxes. Each box contains either an informal description of a step of the algorithm or else a statement in a programming language. The language used here, a (slightly modified) subset of Pascal, has the following BNF grammar, in which the box is to be read as a terminal symbol:

\[
\text{BoxedStmt} = [\text{Stmt}]
\]

\[
\text{Stmt} = \text{SimpleStmt} \mid \text{StructuredStmt}
\]

\[
\text{SimpleStmt} = \text{Assignment} \mid \text{EmptyStmt}
\]

\[
\text{EmptyStmt} = \text{donothing}
\]

\[
\text{StructuredStmt} = \text{CompoundStmt} \mid \text{ConditionalStmt} \mid \text{RepetitiveStmt}
\]

\[
\text{CompoundStmt} = \text{BoxedStmt} \{ ; \text{BoxedStmt} \}
\]

\[
\text{ConditionalStmt} = \text{If BooleanExp} \text{then BoxedStmt} \text{else BoxedStmt}
\]

\[
\text{RepetitiveStmt} = \text{while BooleanExp do BoxedStmt}
\]

The box notation allows algorithm refinement to be shown clearly by simply enlarging a box and revealing its detail. For example, suppose that we are demonstrating the development of a program to implement the informal specification "calculate the integer square root of the absolute value of a number" (henceforth referred to as "find square root of modulus"). Figure 1 could serve as a summary of the early parts of such a development (we assume the absence of built-in abs and sqrt functions).

3. Specifications

A key to the refinement method is the ability to regard specifications as unimplemented programs. This means that the same compositional operators (with the same meanings) must be provided for specifications as for programs. The syntax of
It is straightforward to extend the box notation to denote refinement of a specification. Let the specification shown in figure 3(a) be called $S$. To say that some $Q$ (which may be either a program or a specification) refines $S$ we write $Q$ inside $S$, as in figure 3(b).

![Figure 3](image)

For example, the assertion that the as-yet-unimplemented program "find square root of modulus of $I$" satisfies its specification could be written as in figure 4.

![Figure 4](image)

The task of the developer is now to refine the algorithm "find square root of modulus of $I$" simultaneously with its specification in such a way that the truth of this assertion is preserved by each refinement step. The process is illustrated informally for the first step in §4.

4. Justifying Program Development: the Refinement Process

In this section we demonstrate, by an informal treatment of a development of figure 4, how the process of verifiable construction can be explained using the box notation. The development chosen is based on the first-level algorithm refinement shown in figure 1. This refinement, and the corresponding specification refinement, are presented in figures 5(a) and 5(b) respectively. Both are in obvious modifications of their standard forms (as shown in figures 1 and figure 3(b) respectively) in order to emphasise the correspondence between them.

It is usually argued, with strong methodological justification, that specification should precede and motivate implementation. It may well be easier, however, for a developer with limited experience of formal reasoning to construct figure 5(a) first, using operational reasoning, and then derive figure 5(b) from it. A good method for this developer will be one which by accommodating a mixture of operational and formal reasoning permits such an approach without compromising the logical rigour of the development. Of course, some formal reasoning is essential: at the least, the specification refinement step must be proved correct as soon as it is complete.
Such a specification refinement proof has two parts. These are represented by arrows superimposed on the two parts of figure 6, which is figure 5(b) in standard form. First, the truth of the pre-condition of each component box must be a consequence of known facts about the state at the start of execution of the whole. These known facts are encapsulated in the overall pre-condition. So the truth of this, taken together with changes to the state made by other components, must be enough to ensure the truth of the component pre-conditions. This is shown in figure 6(a).

The second aspect of refinement correctness is that the effect of the component boxes, taken together, should be to guarantee the truth of the post-condition of the original specification. This is illustrated in figure 6(b)*.

Having argued the correctness of a single refinement step, we now show how to extend the argument to the entire refinement process. Consider figure 7, which is figure 5 relabelled. We want to prove that \( P \) satisfies \( S \). First we refined the program \( P \) to its two components \( P_1 \) and \( P_2 \), and the specification \( S \) to its components \( S_1 \) and \( S_2 \). We then showed that the specification refinement was correct. To justify the program refinement, we now have to show that \( P_1 \) satisfies \( S_1 \) and that \( P_2 \) satisfies \( S_2 \).

Figure 8 shows the two boxes that need to be proved correct. Since neither \( P_1 \) nor \( P_2 \) is a single instruction, the boxes \( P_1/S_1 \) (figure 8(a)) and \( P_2/S_2 \) (figure 8(b)) must be further developed by the same method as was used for the original program. This procedure continues until the program has been refined into individual instructions each of which has been shown to satisfy its correctly derived specification. When this has been done, the program as a whole has been shown to satisfy the original specification.

5. Justifying Program Development: the Proof Rules

This section presents the rules for justifying refinement by the compositional operators. The rules are given in a diagrammatic form based on the box notation. (Formal justification of these rules, in their predicate calculus form, is given in [6]). The use of each operator is illustrated by a step in the development of the square root program.

Refinement by Sequence

The starting point for the development is the specification shown in figure 9, from which the square root and modulus symbols have been eliminated. Clauses in the post-condition are "stacked" to denote conjunction.
Refinement of this specification by the first step shown earlier gives figure 10(a). The rule for proving the correctness of a refinement by sequence is shown schematically in figure 10(b). Clearly, to prove figure 10(a) correct it is necessary to invent a suitable predicate on the intermediate state. In other cases (e.g. in a refinement by repetition) this intermediate predicate may already be known; the requirement is then to invent a suitable specification for one of the sequence components.

The arrows on figure 10(b) represent implications according to the following conventions:
- A single unbranched arrow (from source-pred to dest-pred, say) represents the implication source-pred ⇒ dest-pred
- "fan-in" represents the conjunction of antecedents
- "fan-out" represents the conjunction of consequents

Proving the implications represented by the arrows on figure 10(b) establishes the correctness of the refinement.

The correctness conditions for the refinement may now be read off from figure 11. They are

\[
\begin{align*}
\text{True} & \Rightarrow \text{True} \\
\text{True} & \land l_1 \geq 0 & \land (l_1 = l_0 \lor l_1 = -l_0) & \Rightarrow \\
& l_1 \geq 0 & \land (l_1 = l_0 \lor l_1 = -l_0) & \Rightarrow \\
& l_2 \geq 0 & \land (l_1 = l_0 \lor l_1 = -l_0) & \land \text{Intsqrt}(J_2, l_1) & \Rightarrow \\
& (l_2 \geq 0 \Rightarrow \text{Intsqrt}(J_2, l_0)) & \land (l_1 < 0 \Rightarrow \text{Intsqrt}(J_2, -l_0)) & \\
\end{align*}
\]

The proof of these is omitted.

Refinement by Selection

The component specification can now be refined by a step corresponding to the procedural refinement by selection shown in figures 1 and 2. If this procedural refinement is in fact carried out first, it will then be necessary to " reconstruct" the specification refinement. This is shown, together with an assertion of its correctness, in figure 12(a). The rule for justifying such an assertion is shown in fig-
ure 12(b). ("~" inserted in an arrow signifies the negation of the corresponding predicate).

---

The correctness conditions for the refinement of figure 12(a) are:

- \( \text{True} \wedge \sim I < 0 \Rightarrow \text{True} \)
- \( \text{True} \wedge \sim I < 0 \wedge I = \sim \Rightarrow I \geq 0 \wedge (\sim \vee I = \sim) \)
- \( \text{True} \wedge \sim I < 0 \Rightarrow \text{True} \)
- \( \text{True} \wedge \sim I < 0 \wedge I = \sim \Rightarrow I \geq 0 \wedge (\sim \vee I = \sim) \)

Proofs are again omitted. Note that the full details of a diagram like figure 12(a) (in particular the post-condition of the empty statement) might well be discovered only as a result of attempting to justify the refinement.

**Refinement by Repetition**

We turn to the refinement of the second component of the sequence, the specification of figure 13.

---

Now the loop properties given above lead directly to figure 15(a) as a refinement by repetition of the second component of the sequence.

---

We intend to use the method of successive approximations to find \( J \). For the purpose of refining the specification, it is in principle necessary to know only the following facts about the algorithm:

- a range of values is to be searched for the desired value of \( J \);
- this range is to be decreased in some way by each iteration;
- the repetition is to terminate when the range has length 1;
- at termination the lower limit of the range has the desired value.

One can envisage a variety of ways in which these facts could be established (not excluding the examination of a known algorithm); we are not however concerned with that question here. Once established, they may be formalised as the following loop properties (in which the range limits are called \( \text{Low} \) and \( \text{Up} \)):

- **Invariant:** \( \text{Low}^2 \leq I < \text{Up}^2 \)
- **Terminating condition:** \( \text{Up} = \text{Low} + 1 \)
- **Variant:** \( \text{Up} - \text{Low} \)

The invariant must first be established. We omit the reasoning which would lead to refining figure 13 by sequence to figure 14.
Figure 15(b) shows the correctness conditions for a refinement by repetition. As this diagram is a direct translation of the rule given in [5] it is not explained in detail here. The identifiers \textit{inv} and \textit{rel} have been placed appropriately on the diagram to aid identification with the original rule. "\textasciitilde" inserted in an arrow signifies reflexive closure of the corresponding condition. The dotted lines connecting the different occurrences of the invariant and the loop condition signify syntactic identity.

The correctness conditions for figure 15(a) can now be read off from figure 15(b). They are:

\begin{equation}
\text{Low}^2 \leq I \leq \text{Up}^2 \Rightarrow \text{Low} \leq I \leq \text{Up}^2
\end{equation}

\begin{equation}
-(I = \text{Low} + 1) \land \text{Low} \leq I \leq \text{Up}^2
\end{equation}

\begin{equation}
\land \text{Up} - \text{Low} \leq \text{Up} - \text{Low} \land I = \text{Ints}\text{qrt}(\text{Low}, I)
\end{equation}

and in addition it must be confirmed that the relation \text{Up} - \text{Low} \leq \text{Up} - \text{Low} \land I = \text{Ints}\text{qrt}(\text{Low}, I) is well-founded. The proofs of these conditions are omitted.

6. Discussion

This paper presents the view that formal and operational reasoning should complement each other in program development. It presents a variant of the Vienna Development Method (VDM) as a method which accommodates this view. The method has two particularly significant characteristics making it suitable for this purpose.

First, it does not force the programmer to use a refinement calculus, as in the classical approach (e.g. [1], [2]). The essence of a refinement calculus is that refinement steps are formulated in the specification language. The disadvantage of this approach is that it makes no use of the developer's operational intuition, which will often be much better-developed than the corresponding formal skills. In such a case, it would be better to permit the developer to postulate a refinement from operational intuition and then to check it using a formal method. The proof rules and notation presented here are designed to accommodate this approach. For a developer more confident in using formal techniques, it is possible to make use of a refinement calculus within the same notation. For example, the \textit{weakest precondition} calculus of [4] fits in the same theoretical framework as the style proposed here.

The second useful characteristic is that the method places specifications and programs within the same semantic framework and gives the same meaning to the standard compositional operators (sequence, selection and repetition) when applied to either. This allows specifications, viewed as unimplemented programs, to be embedded in a program under development, and progressively replaced, initially by compositions of other specifications, and eventually by single program instructions. Provided that each replacement is provably correct with respect to an appropriately defined refinement relation, then the whole process cannot change the meaning from that of the original specification.

We conclude with a brief discussion of the practical usefulness of this work. The rather inexpressive nature of the specification language used here renders the method as presented unsuitable for the development of large programs, in which the amount of information contained in a typical specification is much greater than in the example in this paper. A compensating advantage of the use of this low-level language, however, is a very simple and clear development method, well-suited to introductory work in verifiable program construction. Further work is required to add the new language facilities and mechanical support system which would make this system suitable for industrial use.

7. Acknowledgements

Alan Hamilton and Lynn Marshall have made helpful comments on earlier drafts of this paper. Simon Jones and Muffy Thomas have given useful advice on points of presentation.
8. References


