Formal Specification in Metamorphic Programming

David A. Penny, Richard C. Holt, Michael W. Godfrey†

Department of Computer Science
University of Toronto

ABSTRACT

Formal specification methods have not been embraced wholeheartedly by the software development industry. We believe that a large part of industry's reluctance is due to *semantic gaps* that are encountered when attempting to integrate formal specification with other stages of the software development process. Semantic gaps necessitate a dramatic shift in a programmer's mode of thought, and undergoing many such shifts during the development of a software system is inefficient. We identify semantic gaps in the software development process and show how they can be minimized by an approach called *metamorphic programming* that operates in-the-large and in-the-small. The main contribution that metamorphic programming makes to formal specification is to clarify the ways in which specifications can be merged smoothly into the software development lifecycle.

1. Introduction

Many researchers in the field of formal methods would agree that formal specifications are not being used as much as they *ought* to be ([Meyer 85], [Neumann 89], [Wing 90]). Our background is in language design and operating systems construction. One of us has been instrumental in designing and producing a programming language, Turing, that has met with some success [Holt 88a]. Another is the major author of a 60,000 line 4.3BSD UNIX-compatible operating system for large-scale shared-memory multiprocessors written in the Turing language. Being academics, formal methods of all kinds appeal to us. Due to pragmatic reasons, however, we have not been able to apply these methods to our compiler and operating systems programming work as much as we might have liked. In this paper we present our views on why this is so, and put forward a proposal, called *metamorphic programming*, that addresses the problems as we see them.

Excellent work has been done on the foundations of formal specification, and on developing effective notations and tools. Our thesis, however, is that current formal specification methodologies do not fit into the software development cycle very well. We believe this is due to the presence of so-called *semantic gaps*.

The idea of a semantic gap [Holt 91] can be explained by analogy to writing a paper (the foremost topic in our minds at this instant!). Suppose an author were forced by his editor to write the outline in English, the first draft in Cantonese, and the final version in Arabic. Although compelling arguments might be made that English is most suitable for writing outlines, Cantonese an excellent language for first drafts, and Arabic ideal for writing final

† authors' email addresses: ynnep@turing.toronto.edu, holt@turing.toronto.edu, migod@turing.toronto.edu

versions, these arguments are unlikely to sway the practicing author. As preposterous as this situation may seem, we expect the software professional to be able to analyze and design using a CASE tool, specify with VDM, prototype with Smalltalk, and implement in C! Any proposed tool that introduces sizable semantic gaps into the software development process is likely to be rejected by the practitioner as inefficient, and this includes specification languages.

In this paper, we propose a method for minimizing semantic gaps from (our variant of) the software development process. This method allows the formal specification stage to be integrated gracefully with the other stages. We present the method in terms of the concepts and tools with which we are most familiar, but the general approach can be adapted to the work of others.

In the next section, we present our software development model and point out the semantic gaps that can result when existing tools are used. In the following section, we give an overview of how metamorphic programming can minimize these gaps. The subsequent two sections examine the requirements for supporting metamorphic programming, first discussing programming in-the-small and then generalizing to programming in-the-large. We then discuss software development environments, and we conclude by re-stating the ideas most relevant to making formal specification techniques more accessible to the software developer.

2. Semantic Gaps in the Software Development Process

Our model of the software development process, and one we hope will not be too controversial, is denoted by the acronym ASPIT — each letter stands for one of the five general stages in the process.

A) *Analysis* — the process of understanding (in the mind) what needs to be accomplished, and determining the external interfaces. This understanding is most aptly communicated using natural language and diagrams.

S) *Specification* — formally specifying precisely what needs to be accomplished.

P) *Prototyping* — writing a possibly incomplete, and perhaps inefficient, version of the software that can be executed.

I) *Implementation* — writing an efficient version of the software that is well-suited to execution on a standard computer architecture (*i.e.* producing machine-oriented code).

T) *Tuning* — meeting efficiency constraints by profiling the implementation to reveal trouble spots, and re-coding in a manner better adapted to the specific target computer architecture (*i.e.* producing machine-specific code).

Additionally, testing, validation, and verification are considered to occur wherever and as often as needed. When any of the stages proves to be too complex to be held in the mind all at once, decomposition (also called design) must be performed. Figure 1 gives an example of notations commonly used for each ASPIT stage.

Researchers and programmers generally agree that if only a development process such as ASPIT were followed carefully, high-quality software would result. Such a rigid process, however, is considered to be burdensome in practice, and hence is eschewed by many software developers [Parnas 86]. It would be perilous for us, as researchers, to ignore this sentiment, and simply admonish programmers to "get with it". It is far better to seek out the reasons behind the sentiment and address them directly.

Programmers feel they can work more efficiently without having to conform to a strict development process. We believe that, given the state-of-the-art, this observation is accurate. It is supported by anecdotal evidence relating how much faster individuals (who operate as best suit themselves) can work compared to large organizations (where the programmers are constrained to follow a formal development process). A major cause of the inefficiency associated

Figure 1: ASPIT Notations

Typical notations for the ASPIT stages of software development.

with the development process is the cognitive inefficiency of being forced to cross large semantic gaps between development stages over and over again.

There are two kinds of semantic gaps: those concerning programming in-the-small and those concerning programming in-the-large. In addition, there is a cognitive gap between the ideal development process (which operates as a time progression), and the way many programmers prefer to work (which appears more haphazard). We shall first discuss the semantic gaps in-the-small, and then the other gaps.

Figure 2 shows the semantic gaps that operate in-the-small. These gaps occur between the stages of the ASPIT development process.

1) *A-S gap (between analysis and specification)*: Typically, a programmer analyzes a problem by drawing diagrams and contemplating the appropriate sequence of operations. To formally specify a problem, however, the programmer must undergo a *paradigm shift* [Holt 91], and begin thinking in terms of model-oriented methods (or, worse still, property-oriented algebraic methods) that relate inputs to outputs using predicates. We admit that for some problems predicate-oriented specifications are very natural, but for many others they are not. This mismatch between a programmer's familiar thought processes and the thought process needed to formally specify a problem constitutes a semantic gap.

2) *S-P and P-I gaps (between specification, prototyping, and implementation)*: If a distinct prototyping language, such as Smalltalk, is used, it is typically radically different in syntax and concept from both the specification language and the implementation language. Having to shift frameworks when moving among these stages is inefficient.

3) *S-I gap (between specification and implementation)*: If a distinct prototyping language is not used, the gap between specification and implementation becomes very large. An implementation language, being by definition efficiently implementable on standard computer architectures, forces premature optimization, since only efficiently executable types and statements are legal. To implement a formal specification, a programmer must not only switch notations, but also "shift gears" mentally from the abstract mathematical world of sets, sequences, and predicates, to the concrete operational world of arrays, pointers, and statements. The absence of intermediate stages to smooth this transition entails a major semantic gap.†

† The two-tiered approach employed by Larch [Guttag 85] attempts to minimize the S-I gap by relating a (purely mathematical) specification with an implementation via a special interface language.

Figure 2: Interstage Semantic Gaps

Each gap incurs a paradigm shift and a need to validate consistency.

4) *I-T gap (between implementation and tuning)*: High-level programming languages featuring formal definitions, strong typing, and plenty of compile-time and run-time checking, such as Pascal or Turing, are natural initial targets when transforming specifications into programs. Such languages, however, are not generally suitable for systems programming, where machine-specific efficiency and access to the underlying implementation (such as the CPU registers) are important considerations. Unless two distinct programming languages are used (and this introduces another semantic gap), the gap between specification and tuned implementation becomes large.

Each of these *interstage* semantic gaps creates a barrier to the use of formal specification methods in the software development process, as well as creating a need to validate consistency. Because these gaps cause problems even when the size of the software to be developed is small, they are said to operate in-the-small. A semantic gap that shows up only when developing large software systems will now be considered.

In a large project the various stages of ASPIT, including specification, need to be decomposed into manageable chunks and these chunks glued together. There exist CASE tools that aid in software design, but they are typically not well-integrated with other tools. Moreover, there is usually no mechanism for helping the programmer keep the structure consistent between stages. For example, if a specification is decomposed in one way, and an implementation in another, then learning how the specification is structured is of no help in learning how the implementation is structured. This structural discontinuity between the stages of ASPIT is called the *design gap*. This gap is especially important because much of the time spent by programmers new to a project is devoted to understanding the overall structure of the system. If this has to be done more than once, a great deal of effort is wasted.

The design gap is an example of a semantic gap that operates only in-the-large. A problem with the development process that exists both in-the-large and in-the-small, however, is the idea of the development process as a time progression.

Large organizations often force programmers to follow the stages of ASPIT as a time progression in what is known as the *waterfall* model. The waterfall model allows feedback from a later stage to an earlier one, but in practice this is not encouraged ("You want to change the requirements NOW?"). The idea of the development process as a time progression is inconsistent with the way many programmers would operate by choice. For example, when creating a specification some programmers feel it is helpful to write some code first and then do the specification. Constraining programmers to progress linearly through the stages of ASPIT (including formal specification), whether or not they feel that is appropriate for their current project, creates another barrier to the use of formal specification.

By eliminating these problems, the cognitive cost of using formal specifications in the development process drops dramatically and formal specification can be integrated into the overall software development process.

3. Metamorphic Programming

In the remainder of the paper we discuss an approach that minimizes semantic gaps called *metamorphic programming.* In this section we explain some general concepts of metamorphic programming, and show how problems associated with viewing the development cycle as a time progression may be solved. In the following section, on metamorphic programming in-the-small, we show how interstage semantic gaps can be minimized by using a notation that operates over the entire range of expressibility, from analysis to formal specification to eventual tuned implementation. In the subsequent section, we reveal metamorphic programming in-the-small as the degenerate case of metamorphic programming in-the-large, and show how the design gap, which only operates in-the-large, may be minimized.

As the name implies, metamorphic programming strives to keep the outside view (the cocoon) of a software system complete and consistent despite gradual, but eventually radical, changes occurring within. Thus, all interfaces to the software under development remain constant after analysis has determined what they should be. After prototyping, when executable versions of the software become available, this "completeness" implies that all stages of the executable program implement the entire specification, albeit with vastly different performance characteristics.

Within the cocoon, gradual metamorphosis from any one ASPIT stage to an adjacent one is facilitated by minimizing interstage semantic gaps. Unlike a cocoon, however, all stages in the metamorphosis are considered to exist simultaneously. Work on the various stages may proceed sporadically and in any order. To complete an ASPIT version, all stages must be completed and be mutually consistent. This may be contrasted with the traditional view that the various stages of software development have as a goal the production of the final tuned implementation. Here the goal is to produce an ASPIT *version*, which records the entire development process as well as the final product.

Operating in this manner produces a development trail that can be followed by maintenance programmers, and aids the initial programmer in completely thinking out all aspects of the problem being solved. In terms of specification, this view holds that the benefit of producing a specification prior to coding is modest compared to the benefit of having a specification that is consistent with the documentation and code.

The idea that after analysis all interfaces to the code remain constant does not imply that changes to the requirements are not allowed. Rather, a requirements change is considered to create a new *configuration* of the system. Old, and possibly incomplete, configurations may be maintained, kept as historical records, or discarded. Related ideas will be discussed in more detail in the section on metamorphic programming in-the-large. First, however, we shall define metamorphic programming in-the-small, and show how it minimizes interstage semantic gaps.

4. Metamorphic Programming In-The-Small

When a problem is small enough that no explicit decomposition need be done to complete the ASPIT stages, we are considered to be operating in-the-small. Not being *explicitly decomposed* means that the programmer neither needs nor wants aid for recalling or describing the structure of the decomposition. For example, structure diagrams may be essential for understanding how the modules of a complex software system fit together. Similar diagrams showing how statements within a procedure fit together (such as *flowcharts*) are considered burdensome. When the capacity of the programmer's head is exceeded, we move to programming in-the-large.

Metamorphic programming in-the-small has as its goal the elimination of the semantic gaps between the stages of ASPIT. To achieve this, we introduce the idea of *notational continuity*, which means that the languages used to express each of the stages of ASPIT are conceptually and syntactically consistent.

The detailed design of languages satisfying notational consistency is considered to be of secondary importance to the prime objective. We propose a particular notation based on the Turing language, but the reader is encouraged to consider other possibilities, notably those based on VDM or Z [Di Giovanni 90]. Figure 3 illustrates notational consistency using our notation.

We consider that the best way to achieve notational consistency is to design dialects of a single language that blend together seamlessly [Goldberg 89]. The language, however, must achieve certain other goals which we shall now explain.

An overriding goal is that the language be capable of expressing concepts in the entire range from abstract mathematical specifications, to concrete machine-specific code. To bridge the initial gap between analysis (that uses natural language) and specification (that uses formal notation), the specification language must not stray too far from the proficient thought processes used by the programmer during analysis. Existing specification languages, such as VDM [Jones 90] and Z [Spivey 89], go a long way towards achieving this goal. Especially important in these languages is the ease with which predicates can be stated, and their use of high-level, abstract mathematical types such as sets sequences and maps, that more closely match a programmer's cognition than do machine-oriented data types.

To further minimize the gap between analysis and specification, formal specification must be made more *accessible* to the practitioner. One way to enhance accessibility is to introduce operational specifications. Sometimes it is easier to specify a problem as a predicate relating the input state to the output state, and sometimes it is easier to specify a problem as a sequence of sub-problems to be solved in a specified order. We do not argue that one approach is superior to the other, we simply recognize the fact that depending upon what is being specified, one may be more natural than the other, and therefore both should be allowed.

One aspect of many existing formal methodologies that limits accessibility is the need to include in predicates the fact that large parts of the state space remain unchanged. The mind naturally dwells on those things that change. To have to state explicitly that a thing remains unchanged may be mathematically convenient, but is contrary to natural thought. We shall propose a notation to solve this problem.

To bridge the gap between specification and prototyping, it is helpful for the specification language to support operational specification. An operational specification has the advantage that it is executable as a prototype. Thus, in metamorphic programming the prototyping language should be the same as the operational specification language, and should support the same data types, operations, and syntax as the specification language. In this way, the sole semantic gap between specification and prototype is the gap between predicate and operational specifications. To close this gap, the language should support a smooth transformation

Figure 3: Notational Consistency
The ASPIT stages using a stack as an example.

between these two types of specifications. The closely related ideas of transformational programming [CIP 87], operational refinement ([Jones 90], [Morgan 90]) and predicative programming [Hehner 84], have been explored by others.

The smooth transformation from predicate to operational specification should be followed by a smooth transformation from prototype to efficient machine-oriented implementation. Machine-oriented in this context does not mean machine-dependent, but rather means formally defined but designed to be efficiently implementable on standard computer architectures.

In cases where machine-independent features are not efficient enough (as for parts of systems programs), the transformation must be allowed to continue to machine-specific code. Machine-specific language features typically have semantics that can be stated only with reference to a specific compiler implementation and target architecture.

By satisfying these goals, a single metamorphic programming language can support notational continuity in the full range of expressibility, from abstract mathematical specifications to machine-specific final code, yet remain accessible. The rest of the section discusses a notation designed to achieve these goals.

4.1. A Metamorphic Language: Dialects of Turing

Our specific example of designing a metamorphic notation concentrates on evolving a formally defined, machine-oriented language called Turing in two separate directions: towards more abstract mathematical features on one hand, and towards more machine specificity on the other. In this section, we introduce the notation and discuss the design of the existing dialects of Turing suitable for machine-oriented and machine-specific implementations. In the following subsections, we discuss the extension towards specification and prototyping. Figure 4 illustrates the evolution of *Turing* in the two directions.

Figure 4: Notational Continuity
Semantic gaps are minimized using dialects of Turing in each ASPIT stage.

For machine specificity, we have taken as our goal to support the same expressibility, and run with the same efficiency, as the C language. For abstract mathematical features, we have taken as our goal to support the same level of expressibility as the VDM and Z notations. We start with Turing as the base, rather than VDM or Z, because we have a large community of Turing users and because we have expertise with Turing implementations. Thus our syntax is similar to Turing's syntax, and may not appeal to an expert in VDM or Z. The reader should bear in mind, however, that the ideas may be applied equally starting with VDM or Z as the base language.

The dialect of Turing called *Turing Plus* [Holt 88a] supports expressibility equivalent to that of the C language. The dialect called *Spectur* (Specification language for Turing), supports the same level of expressibility as VDM [Godfrey 90]. The *Abstur* dialect (Abstract

Turing) is the operational version of *Spectur*, and supports both operational specifications and prototyping at the same conceptual level as VDM. The dialect called *E/R* (Entity/Relation) *Turing* is used to record the major entities of a software system along with the relations between them and to provide a framework for recording the analysis (written in natural language) of these entities.

The base language, Turing (a direct descendant of the Pascal and Euclid languages), has a formally defined syntax and semantics [Holt 88b] — a complete formal specification of the Turing language has been published. The decomposition and writing of this large specification has given us some insights into the specification process. In addition to its formal definition, Turing supports complete and efficient legality checks at compile-time and run-time (this is called *faithful execution*) with descriptive error messages to indicate violations. Turing is designed to be efficiently implementable on standard computer architectures (*i.e.* runs as fast as C when checking is disabled), and at the same time be easy to learn and use (simple and consistent syntax and concepts). Turing implementations are available for many computers [Stephenson 90].

Turing, while useful as a general-purpose language, is not usable by itself for systems programming. Therefore, Turing has been extended by adding exception handlers, concurrency, and machine-dependent access to the underlying machine architecture. This extension to Turing, called Turing Plus, has been implemented on many different computers, in some cases by emitting C as the target language. So as not to confuse the formally-defined and machine-dependent parts of the language, the syntax is designed so that a programmer never gets machine-dependent features by default, but must explicitly write them in each case. Thus, a lexic scan of the program text is sufficient for determining if machine-dependent features are being used. One of the authors has used Turing Plus to write a 60,000 line 4.3BSD UNIX-compatible operating system for a large-scale multiprocessor. This operating system was developed by starting with an initial implementation written entirely with formally defined language features, and adapting (metamorphosing) it through the use of machine-specific features to the target architecture.

Thus, a consistent evolution of the Turing base language into one that supports machine-specific constructs has been completed successfully, with conceptual and syntactic consistency throughout. Our success in this direction encouraged us to try designing extensions in the opposite direction, towards the more abstract mathematical concepts used in VDM and Z, suitable for specification and prototyping.

This design has, in many cases, turned out to be more an exercise in removing mathematically arbitrary restrictions from Turing than in adding extra features.† We shall now describe the essential concepts of this design. Unlike the Turing and Turing Plus dialects, this evolution remains a paper exercise. First, we shall describe the Spectur dialect that is notationally continuous with respect to Turing and Turing Plus, and that supports model-oriented VDM-like specification. Following this we shall describe the Abstur dialect, which adds operational specification and prototyping to Spectur.

4.1.1. Spectur: Specification Language for Turing

To minimize semantic gaps between specification and implementation, we designed Spectur as a dialect of Turing [Godfrey 90]. Turing already supports **pre**, **post**, and **invariant** statements in modules and procedures, and a syntax for non-quantified boolean expressions. To evolve Turing towards a more VDM-like specification language, we have added quantification and abstract mathematical data types together with their constructors.

† De Man has described a similar experience [De Man 90].

The Turing language, which the reader may take to be a variant of Pascal, supports arrays, sets, and strings (character sequences). To provide Spectur with abstract types, we relax the restrictions that Turing makes on its types. To provide sequences of arbitrary types, we relax the restriction that a string is necessarily a sequence of characters. To provide sets of arbitrary types, we relax the Turing restriction that elements of sets must be index types (subranges and enumerated types). To provide maps, we relax the restriction that array subscripts must be index types. With these relaxations, the practitioner moves easily between the worlds of abstract specifications and efficient implementations.

To illustrate this concept, we give data types that can be used to specify an operating system disk cache. This example uses a map, which is effectively an array indexed by an arbitrary type. A disk cache keeps in-core copies of recently accessed disk blocks, so that disk requests can often avoid using the physical disk, which is comparatively slow. A particular disk block in a UNIX-like operating system is identified by a *DeviceAddress*, which lists the *major device identifier* (which disk driver to use), the *minor device identifier* (which disk controlled by that driver to use), and the *block address* (the block number on that disk). The in-core version of a disk block is kept in an array of bytes. With our extensions to allow maps and arbitrary sets, the cache's data can be declared as follows:

```
type Byte: 0..255

const BlockSize := 4096
type Block: array 0..BlockSize-1 of Byte

type DeviceAddress:
   record
      majorDeviceID: nat      /* A "nat" is a natural number */
      minorDeviceID: nat
      blockAddress : nat
   end record

/* Set of blocks currently controlled by the cache */
var cached: set of DeviceAddress

/* Block contents */
var cache: map DeviceAddress to Block
```

This disk cache, which the programmer may eventually implement by hashing *DeviceAddress* to find the associated 4K buffer, is specified easily and powerfully by allowing *DeviceAddress* to be the index type for the *cache* map, and by specifying currently active cache blocks using the *cached* set.

As another example, we shall use Spectur data types to specify a binary tree of integers. Each node of the tree is defined by its position relative to the root. A node's position is specified in terms of directions from the root of the tree to the node (*e.g.* go left, go right, go left). The nodes currently in the tree are given by the set *nodes*, the values of those nodes are given by the *data* map.

```
type Direction    : enum(left,right)
type Path         : sequence of Direction

var nodes         : set of Path
var data          : map Path to int
```

Since Spectur maps, sets, and sequences are generalizations of Turing arrays, sets, and strings, notational consistency is achieved by using the operators of the latter with the former.

A further restriction that may be removed from Turing to provide a more powerful language is the distinction between sets and types. If types are viewed as sets, and sets as types, this allows us to express the idea of variables only taking on values from a given set, as illustrated in this example (Hehner has expressed similar ideas [Hehner 84]).

```
type EvenInt: set of int := {i: int | i mod 2 = 0}
var x: EvenInt
```

In this section, we have shown how a specification language, Spectur, at the conceptual level of VDM or Z may be designed by extending a machine-oriented base language. The extensions consist of a handful of new features and a relaxation of Turing's existing context rules for types. These modifications result in a specification language which provides continuity with the programmer's thought processes used during analysis (through the use of data types such as those in VDM or Z), and with machine-oriented and machine-specific implementation languages (because the syntax and operators are all derived from the same base language). Spectur omits the operational parts of Turing, but the Abstur dialect, which will now be presented, includes them.

4.1.2. Abstur: For Operational Specifications and Prototyping

The *Abstur* (Abstract Turing) dialect of Turing is a prototyping language and a language used for operational specification. It is notationally consistent with the predicate specification language (Spectur) and the implementation languages (Turing and Turing Plus).

We mentioned earlier that one of our goals for a metamorphic programming language is the ability to merge operational specifications with predicate (pre/post) specifications. This is considered necessary to close the gap between analysis, where the programmer often thinks operationally, and specification. Furthermore, an operational specification language can also serve as a prototyping language.

Abstur inherits all the abstract data types of Spectur. The Spectur language does not allow procedures to have *bodies*, only pre and post-conditions. We could have designed Abstur to be the opposite, namely, to omit pre/post and to allow only bodies. Instead, to minimize the semantic gap between pre/post specification and prototyping, Abstur supports both pre/post and bodies. In Abstur, such a body can serve both as an operational specification and as a prototype implementation. Abstur is so closely blended with Turing on the one hand, and Spectur on the other, that the distinction is somewhat blurred. We consider this an advantage.

For example, to specify that the disk cache described above is to be flushed (written back to disk and invalidated), the following Abstur code could serve both as an operational specification and as a prototype implementation.

```
for deviceAddress: cached
   Device.WriteBlock(deviceAddress, cache(deviceAddress))
end for
cached := {}
```

In Turing proper, only index types are allowed as the bounds in **for** loops. However, when type restrictions of this sort are relaxed, and when types and sets are considered equivalent, this usage, which non-deterministically iterates through the members of the *cached* set, falls out naturally.

4.1.2.1. Handling Non-Determinism with the Pick Statement

One of the common criticisms levelled against operational specifications is that they overconstrain. To solve this difficulty, the **pick** statement, a non-deterministic assignment statement, is added to Abstur. This statement is used to specify non-determinism in operational specifications. The syntax of **pick** is:

> **pick** *variableReference* [**:** *typeOrSet*]

For example, in specifying operating systems procedures, we found it common that successful completion is non-deterministic as it depends on unknown resource limitations [Godfrey 88]. This is specified as follows:

> *var success: boolean*
> *pick success*

The **pick** statement non-deterministically chooses a value for *success* from that variable's base type (**boolean**, in this case). The range of allowed values may be narrowed by specifying a particular set to pick from:

> *var j: int*
> *pick j: {i: int* **|** *i mod 2 = 0}*

This example non-deterministically chooses an even integer to assign to variable *i*. Note that **pick** is *not* the same as a random number generator — no particular distribution of results is guaranteed or even suggested. Using a **pick** in an operational specification specifies that any of the values in the given set is an acceptable solution to the problem being specified.

4.1.2.2. Mixing Predicates and Statements with the Attain Statement

So far the Abstur language allows operational specifications to be mixed with predicate specifications only at the level of a procedure. A more finely grained mixing is desirable, however, and so Abstur also includes the **attain** statement for inserting predicate specifications in-line with an operational specification.† The full syntax of **attain** is:

> **attain** [**import** *importedVariables*]
>   [**pre** *preCondition*]
>   *postCondition*
>   [**by**
>   [**pre** *preConditionForBy*]
>   *Statements*
>   {**elsby**
>   [**pre** *preConditionForElsby*]
>   *Statements*}
>   **end by**]

The simplest form of this statement gives nothing but the post-condition. First we shall discuss this post-condition then the optional parts of the statement.

The post-condition specifies a relationship between "before" and "after" values of the state. This does not have an immediate operational meaning (and hence cannot be used for prototyping) unless a **by** clause accompanies it (more on this later).

The values of variables after the **attain** are denoted using a prime ('), the values before are written without a prime. The prime may be attached to any reference that may be assigned to. The following example specifies a search of an integer array for an element with the value 12:

---

† The specification statement of the Refinement Calculus [Morgan 90] is similar to Abstur's **attain** statement, but does not have an analogue of the **by** clause.

```
var a: array 1..5 of int := init(14,12,15,43,2)
var i: 1..5
attain a(i') = 12 ∧ a'=a
```

This **attain** statement specifies that after completion, the new value of *i* should be set to the index of an element of *a* containing the value 12, and that the array *a* should not be changed.

An optional pre-condition may be attached to the **attain** statement to indicate explicitly the requirements for implementability of the predicate, and to record guaranteed knowledge of the state that an eventual implementation may assume. For example, the above predicate is not attainable unless a 12 exists in the array:

```
attain   pre ∃ i: 1..5 • a(i) = 12
   a(i') = 12  ∧  a'=a
```

Note that if the pre-condition is not actually met, an **attain** statement can be thought of as a *miracle* statement [Hehner 84] that attains the impossible.

4.1.2.3. The Frame Problem

Repeatedly stating those things that must remain unchanged, for example *a'=a*, becomes a significant nuisance when many large and complex data structures are involved. This is called the *frame problem*, and results from an insufficient narrowing of the scope of interest [Genesereth 87]. This problem can make pre/post specifications clumsier than the programmer wants to write or decipher.

To illustrate the frame problem, we present an example that we encountered when specifying a virtual memory manager. The major data types for this specification are:

```
type Descriptor: nat                  /* Each memory space has a unique Descriptor */
type VirtualAddress: nat

type SpaceRecord:
   record
       file  : File.Descriptor            /* File containing the initial memory image */
       bytes : map VirtualAddress to Byte  /* Contents of virtual memory */
   end record

var space: map Descriptor to SpaceRecord
```

The predicate used to state the simple concept of a given byte in a virtual memory space changing value (*i.e.* the *poke* operation) turns out to be rather tedious:

```
procedure poke (s: Descriptor, va: VirtualAddress, b: Byte)
    attain space(s).data(va)' = b ∧
        ∀ a: VirtualAddress  • ( a ≠ va  ⇒  space(s).bytes(a)' = space(s).bytes(a) ) ∧
        space(s).file' = space(s).file ∧
        ∀ d: Descriptor  • ( d ≠ s  ⇒  space(d)' = space(d) )
end poke
```

The first line of this post-condition is obvious and deals with change. The remaining lines (the confusing part) state simply that all the rest of the *space* data remains unchanged.

We shall outline three solutions to the frame problem, intended to make the use of pre/post specifications less tedious and hence more acceptable to the programmer. The first is the explicit use of an **import** list, which parallels import lists that Turing uses for procedures and modules. In this example, the **attain** can be simplified to:

attain import var space(s).data(va)
space(s).data(va)' = b

This import list effectively creates the additional predicates that specify that nothing but *space(s).data(va)* is allowed to change.

The second solution is the automatic creation of an import list based on a lexical scan of the predicate. This scan locates all primed (parts of) variables and effectively allows only them to be changed. In this example, the import would be implicitly created exactly as we have just given it explicitly.

The third solution is to implicitly define the import list based on the meaning (not just the lexic form) of the post-condition. This approach avoids a problem with lexically produced import lists, which is that semantically identical predicates may have have different import lists, and hence mean different things. This happens exactly when a primed item in a predicate can be removed by rewriting the predicate to another equivalent predicate.

To define an import list based on the meaning of the post-condition, we start by dividing the state into *atoms* of a language-defined granularity. Given a generic initial state, any atom which can possibly have an effect on the truth or falsity of the post-condition is included in the implicit import list. An atom would be, for example, an element of an array with a primitive type such as an integer.

We define the dependency relationship *dep(Q,a')* to be true when post-condition *Q* may possibly depend on the value of atom *a'*. Let *A'* be the set of all "after" atoms except *a'*.

$$dep(Q,a') \equiv \exists A' \bullet (\exists a' \bullet Q \wedge \exists a' \bullet not\ Q)$$

In other words, *Q* depends on *a'* for a given initial state if there exists a final state in which *Q* is true, and there exists another final state, differing only in the value of *a'*, in which *Q* is false. The implicit import list contains exactly those (parts of) variables that *Q* depends on. This import list can be written as a predicate stating which atoms remain unchanged as a function of the initial state, and this predicate should be considered to be implicitly conjoined with the post-condition. Using this technique, our example can be specified more concisely:

procedure poke (s: Descriptor, va: VirtualAddress, b: Byte)
attain space(s).data(va)' = b
end poke

4.1.2.4. "By" Clauses to Maintain What-How Relationships

To complete the discussion of the **attain** statement, we now consider the use of the **by** clause which assigns an operational meaning to an **attain** statement. This feature is intended to provide a convenient path from predicate specifications to operational ones, and thus also to prototypes. The **by** clause maintains the *what-how* relationship between a predicate specification and an operational implementation. The *what* part is the predicate to be attained. The *how* part consists of the statements within the **by**. An example that uses the **by** clause is:

```
var a: array MIN..MAX of int
var i: MIN..MAX
const v := 12

attain pre ∃ j: MIN..MAX  • a(j) = v
   a(i') = v
by
   i := 1
   loop
        exit when a(i) = 12
        i := i + 1
   end loop
elsby
   pre ∀ j: MIN..MAX-1  • a(j) <= a(j+1)
   binarySearch(a,MIN,MAX,v, i)
end by
```

Here, a predicate specification has been transformed into two alternative operational implementations, the first given in-line, and the second calling a procedure. The first **pre** describes the requirements for implementability of the predicate, and thus any operational specification must assume this. The first **by** clause is guaranteed to set *i* to an index of an element of *a* with value *v*, provided that such an element exists.

In general, *how* a predicate is attained may depend upon the context, and thus the pre-condition within the **elsby** clause in the example given above. The **elsby** clause provides an alternative implementation assuming, in addition to the implementability constraint, that the array is sorted.

The overall pre-condition for an **attain** statement is the conjunction of the explicit pre-condition for the **attain** with that of each of its **by** and **elsby** clauses. If the **attain**'s pre-condition is omitted, it is taken to be **true**.

In this section, we outlined the design of the Abstur operational specification and prototyping language. Abstur supports integrated operational and predicate specification through the use of Spectur data types, Turing statements and operators, and Abstur **pick** and **attain** statements.

5. Metamorphic Programming In-The-Large

Metamorphic programming in-the-small removes the semantic gaps between the stages of the ASPIT development process, serving to blend formal specifications smoothly into the development process. We have previously stated that when none of these ASPIT stages need to be decomposed explicitly, we are operating in-the-small. Conversely, if any of the stages do need to be decomposed, we are operating in-the-large.

All the ASPIT stages, including formal specification, require decomposition into parts when dealing with large systems. For example, the formal specification of the context conditions for the Turing language is based on seven distinct sub-specifications (of such things as type definitions), each of which is formally specified [Holt 88b]. Similarly, in any large system the other ASPIT stages eventually need to be broken into parts.

As we have mentioned before, when the decomposition into parts of one stage (such as specification) differs from the decomposition in another stage, we have a design gap. We call this difference in decomposition a *structural discontinuity*. Structural discontinuities are a particularly pernicious form of semantic gap, and a key goal of metamorphic programming in-the-large is to minimize them. Of the many problems to be solved in programming in-the-large

Figure 5: Design of a Software Object
The decomposition of the MiniTunis operating system.

([ISPW-4-88], [IWSCM-2-89]), we shall concentrate on this particular one, and its relationship to formal specifications.

Figure 5 shows an example of the decomposition of a piece of software. This shows the structure of the MiniTunis operating system, which is a model operating system used primarily for teaching about operating systems. MiniTunis [Penny 88] is essentially a simplified variant of UNIX. Even with simplification, it has an internal structure containing a number of subsystems. The development of the system as a whole goes through the ASPIT stages, and relies in turn on the completion of the ASPIT stages of its parts. This illustrates the fact that the ASPIT process is recursive, namely, a large software object goes through each of the A, S, P, I and T stages, and each of these in turn may require decomposition and further ASPIT development, and so on through the layers of decomposition.

In the fortuitous case, the parts into which one stage is decomposed (the specification, for example) will correspond exactly to the parts into which the other stages are decomposed. In simple systems, we can force this correspondence, but in large software systems we must recognize that complexity and changing requirements will eventually force discontinuities among the stages. Creating an underlying model that can accommodate such discontinuities during development, but which will in the end result in one structurally continuous version of the ASPIT process, is a goal of metamorphic programming in-the-large.

Figure 6 illustrates a particular software system called object *E*. The interface to the object, represented by its methods with their signatures, remains constant. However, there have been three distinct development cycles for the object, with distinct internal structures — version 1 is un-decomposed, version 2 is composed of sub-objects F1 and F2, and version 3 is composed of sub-objects G1, G2 and G3.

If *E* is an operating system, we can imagine that version 1 represents an initial naïve attempt to complete the entire project in-the-small (without decomposition). Versions 2 and 3 are two alternative decompositions. Version 2 is decomposed into a file manager (with embedded device management) and a memory manager. Version 3 is decomposed into a file manager, a distinct device manager, and a memory manager. Within a version, the decomposition of all stages is constrained to be identical to all others. Thus in each version the specification is structured in the same manner as the implementation. Indeed, we can take this to be the definition of a *version*, namely, that all of its ASPIT stages have the same structure.

The two basic reasons for beginning a new version are *structural decomposition* and *structural discontinuity*. In figure 6, versions 2 and 3 are structural decompositions of version 1. Version 2 and version 3 are structurally discontinuous.

Figure 6: Structural Discontinuity
Object E has a particular signature and meaning.
There are three versions of E's internal structure.

In metamorphic programming, we typically start by assuming that any object can be completed in-the-small. Sometimes this works out, but often it does not. In the case of the MiniTunis operating system, it is feasible to perform the analysis entirely in-the-small. We may take version 1 in figure 6 to illustrate such an analysis.

To specify MiniTunis, however, the un-decomposed structure of version 1 is not suitable. Rather, it is necessary to decompose the specification into a *File Manager* object and a *Memory Manager* object.† Version 2 in figure 6 illustrates this case. Since the structure of the specification is more decomposed than that of the analysis, we have a mild form of structural discontinuity. To resolve this discontinuity, and produce a version of the ASPIT development process in which the specification is structurally continuous with respect to the other stages, we must complete version 2 using the same structure throughout all the stages. Completing version 2 entails writing ASPIT for sub-objects F1 and F2, and then writing the ASPIT for object *E* in terms of these sub-objects.

To implement MiniTunis, however, the structure of version 2 turns out to be inherently inefficient, because all access to the disk, including the heavily used paging access from the

† The formal specification of the MiniTunis file manager [Godfrey 88] can be contrasted with the formal specification of the UNIX file system that was done by Morgan and Sufrin using a Z-like notation [Morgan 84].

Memory Manager, is constrained to go through the *File Manager* object. During implementation of version 2 this fact was discovered, and thus version 3 was started. For version 3, the *Device Manager* object is moved up a level in the decomposition. Versions 2 and 3 now exhibit a strong form of structural discontinuity in that the decomposition of the specification from version 2 is different from that of the implementation from version 3. To resolve this discontinuity, and produce a version of the ASPIT development process in which all stages are structurally continuous, we must complete version 3. As for any other version, completing version 3 entails writing ASPIT for all its sub-objects (G1, G2, and G3), and then writing the ASPIT for object *E* in terms of these sub-objects.

Being recursive, whatever applies to an object applies equally to its sub-objects. Thus sub-objects will, in general, consist of several versions in various stages of completion. Note that an object is considered complete once *at least one* complete ASPIT version of that object, and all its sub-objects, has been produced.

From this discussion, the key conclusion with respect to formal specifications is that it is essential for the structure of the specification to match that of the rest of the version. If this is not the case, the programmer attempting to understand the structure of an object will waste time learning, in turn, the structures of each of the stages separately. The implication, therefore, is that the specification must be divided carefully into parts that are suitable for implementation. If this is not the case, the implementation will no doubt adopt a conflicting structure, at considerable cost in terms of the overall development. The usual case, when such a conflict occurs, is to abandon the specification and carry on modifying the implementation. The way to make sure the specification will continue to be of value is to update it repeatedly along with the implementation and all other stages. This in turn demands minimization of semantic gaps throughout all aspects of the development process.

Checking for the structural continuity of a version can be largely mechanized, given notational continuity among the stages. Lacking such support, the headache of repeated manual checking of the information in each stage is likely to be too much for the programmer to manage. In the following section we shall take a look at tools, such as ones to help solve this problem, required to support metamorphic programming.

6. Software Development Environment

Most professional software developers make extensive use of support tools, and there is a large amount of work in progress to integrate tools into software development environments [ISPW-4-88]. For formal specifications to be used widely in software development, they must be supported by tools that are well-integrated into these environments.

The metamorphic approach to programming provides a basis for accomplishing this integration. For development in-the-small, notational consistency immediately implies the possibility of a common computer representation for object interfaces across the ASPIT stages. For example, figure 3 shows that the methods of a object can be represented identically across the stages. This allows the environment to mechanically check consistency among stages, or even to automatically generate the interface part of an incomplete stage based on that of other stages.

Figure 3 gives five distinct views of a software object, corresponding to each of the ASPIT stages. Ideally, the programmer considers the object as a coherent whole that can be viewed in these various ways. The environment should support this concept by presenting a graphic icon representing the object, and allowing the programmer to view or expand it, by selecting, for example, the S (specification) option.

For development in-the-large, the environment should maintain the relationship among the sub-objects of an object, which in turn means there must be a mechanical record of the object's structure. There should be an option to display this structure graphically, in a format such as that shown in figure 5, as is already done in many CASE tools (such as [Coad 90]). Many of the structural relations, such as those given by import lists, have an obvious computerized representation and can be enforced automatically across the stages of a given version of an object. In other words, the environment can automatically prevent or detect structural discontinuities.

If we view the software development process as the continuing creation of software objects for installation in a re-use library, we see that the metamorphic programming approach provides considerable advantages beyond those provided to a single project. The new advantage is that we gain a single paradigm by which the software professional browses, understands, updates, etc., the rather large set of software objects available to be used in ongoing projects. This comes about because the library itself is organized as objects, each of which has an ASPIT organization and is in turn composed of more sub-objects of a similar nature.

Returning to the idea of object decomposition, we see that the designer should strive continually to divide an object into parts that already exist in the library, or conversely, to combine existing objects to create new objects of general utility.

The idea of a single development paradigm is one of the "holy grails" of object-oriented computing. What we have tried to clarify is how formal specification can be fitted smoothly and profitably into such a paradigm. In this view, a specification is not thought of as an end unto itself, but rather as a stage that is an integral part of a software development system.

7. Conclusions

In this paper we have argued that *semantic gaps* are a primary reason why formal specification techniques such as VDM or Z have not been accepted more generally by practitioners. These gaps correspond to *paradigm shifts* for the professional programmer, in other words, to significantly different ways of understanding the software under development. After having identified these gaps in the software development process, especially those related to specification, we showed how they can be minimized using an approach called *metamorphic programming* that operates both in-the-large and in-the-small.

The main contribution that metamorphic programming makes to formal specification is to clarify the ways in which specifications can be merged smoothly into the software development lifecycle.

References

[Bjorner90] D. Bjorner, C. A. R. Hoare and H. Langmaack (eds.), *VDM '90: VDM and Z — Formal Methods in Software Development — Proc. of the Third International Symposium of VDM Europe 1990, Kiel FRG*, Springer-Verlag LNCS #428, April 1990.

[CIP-87] *The Munich Project CIP — Volume 2: The Program Transformation System CIP-S*, Springer-Verlag LNCS #292, 1987.

[Coad90] Peter Coad and Edward Yourdon, *Object-Oriented Analysis*, Prentice Hall, Englewood Cliffs NJ, 1990.

[DeMan90] Jozef De Man, "Making Languages More Powerful by Removing Limitations", *Proc. of ACM SIGSOFT International Workshop on Formal Methods in Software Development*, ACM Software Engineering Notes, vol. 15, no. 4, September 1990.

[DiGiovanni90] R. Di Giovanni and P. L. Iachini, "HOOD and Z for the Development of Complex Software Systems", in [Bjorner et al 90].

[Genesereth87] Michael R. Genesereth and N. Nilsson, *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann, Los Altos CA, 1987.

[Godfrey88] Michael W. Godfrey, *Toward Formal Specification of Operating System Modules*, M.Sc. thesis, Dept. of Computer Science, Univ. of Toronto, May 1988.

[Godfrey90] Michael W. Godfrey and Richard C. Holt, "Spectur — A Specification Language for the Programmer", Technical Report CSRI-241, Univ. of Toronto, June 1990.

[Goldberg89] A. Goldberg and D. Robson, *Smalltalk-80: The Language*, Addison-Wesley, Reading MA, 1989.

[Guttag85] J. V. Guttag, J. J. Horning, and J. M. Wing, "Larch in Five Easy Pieces", Technical Report 5, Digital Equipment Corporation Systems Research Center, July 1985.

[Hehner84] E. C. R. Hehner, "Predicative Programming, parts 1 and 2", *Communications of the ACM*, vol. 27, no. 2, February 1984.

[Holt88a] R. C. Holt and J. R. Cordy, "The Turing Programming Language", *Communications of the ACM*, vol. 31, no. 12, December 1988.

[Holt88b] R. C. Holt, P. A. Matthews, J. A. Rosselet, and J. R. Cordy, *The Turing Programming Language: Design and Definition*, Prentice Hall International, 1988.

[Holt91] R. C. Holt, Terry Stanhope and George Lausman, "Object Oriented Computing: Looking Forward to Year 2000", ITRC Tech. Report TR-9101, Information Technology Research Centre, Univ. of Toronto, April 1991.

[ISPW-4-88] *Proc. of the Fourth International Software Process Workshop* ACM SIGSOFT Engineering Notes, vol. 14, no. 4, June 1988.

[IWSCM-2-89] *Proc. of the Second International Workshop on Software Configuration Management*, ACM SIGSOFT Engineering Notes, vol. 17, no. 7, November 1989.

[Jones90] C. B. Jones, *Systematic Software Development Using VDM*, Second Edition, Prentice Hall International, 1990.

[Meyer85] B. Meyer, "On Formalism in Specification", *IEEE Software*, vol. 2, no. 1, January 1985.

[Morgan84] C. Morgan and B. Sufrin, "Specification of the UNIX Filing System", *IEEE Trans. on Software Engineering*, vol. 10, no. 2, March 1984, pp 128-142.

[Morgan90] Carroll Morgan, *Programming from Specifications*, Prentice Hall International, 1990.

[Neumann89] Peter G. Neumann, "Flaws in Specifications and What To Do About Them", *Proc. of the Fifth International Workshop on Software Specification and Design*, ACM SIGSOFT Engineering Notes, vol. 14, no. 3, May 1989.

[Parnas86] David L. Parnas and Paul C. Clements, "A Rational Design Process: How and Why to Fake it", *IEEE Trans. on Software Engineering*, vol. SE-12, no. 2, February 1986.

[Penny88] David A. Penny and Richard C. Holt, "The Concurrent Programming of Operating Systems using the Turing Plus Language", Course Notes, Dept. of Computer Science, Univ. of Toronto, 1988.

[Stephenson90] C. Stephenson and R. C. Holt, "Changing Trends in High School Programming", *Educational Computing Organization of Ontario Output*, vol. 11, no. 2, July 1990.

[Spivey89] J. M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall International, 1989.

[Wing90] Jeannette M. Wing, "A Specifier's Introduction to Formal Methods", *IEEE Computer*, vol. 23, no. 9, September 1990.