Proving Refinement Transformations for Deriving High-Assurance Software

Victor L Winter *
Intelligent Systems and Robotics Center
Sandia National Laboratories
Dept 9622, P.O. Box 5800
Albuquerque, NM 87185-0660, U.S.A.
vlwinte@sandia.gov

James M. Boyle †
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, U.S.A.
boyle@mcs.anl.gov

Abstract

The construction of a high-assurance system requires some evidence, ideally a proof, that the system as implemented will behave as required. Direct proofs of implementations do not scale up well as systems become more complex and therefore are of limited value. In recent years, refinement-based approaches have been investigated as a means to manage the complexity inherent in the verification process.

In a refinement-based approach, a high-level specification is converted into an implementation through a number of refinement steps. The hope is that the proofs of the individual refinement steps will be easier than a direct proof of the implementation. However, if stepwise refinement is performed manually, the number of steps is severly limited, implying that the size of each step is large. If refinement steps are large, then proofs of their correctness will not be much easier than a direct proof of the implementation.

We describe an approach to refinement-based software development that is based on *automatic* application of refinements, expressed as program transformations. This automation has the desirable effect that the refinement steps can be extremely small and, thus, easy to prove correct.

We give an overview of the TAMPR transformation system that we use for automated refinement. We then focus on some aspects of the semantic framework that we have been developing to enable proofs that TAMPR transformations are *correctness preserving*. With this framework, proofs of correctness for transformations can be obtained with the assistance of an automated reasoning system.

*This work was supported in part by the United States Department of Energy under Contract DE-AC04-94AL85000, and in part by the BM/C3 directorate, Ballistic Missile Defense Organization, U.S. Department of Defense.

†This work was supported by the BM/C3 directorate, Ballistic Missile Defense Organization, U.S. Department of Defense.

DISCLAIMER

1 Motivation—Need for High-Assurance Systems

Computer systems are increasingly used today in applications where their failure or subversion would threaten the lives or safety of people, or the economic well-being of people or corporations. Such applications require *high-assurance* systems—computer software and hardware systems for which there is strong evidence that failure and subversion cannot occur.

Problems that can be solved by *reactive systems* often have a safety-critical nature. Such systems must (1) sense changes in their physical environment, and (2) generate control actions that enable the system to respond to these changes. Examples of reactive systems include automotive engine, cruise, and anti-lock brake control, cellular telephone control, medical instrument control, fly-by-wire control, robot manufacturing control, and weapon control.

The design of a reactive system is often a difficult task. One reason is that the complexity of such a system grows rapidly, so that even a moderate-sized reactive system can have a finite-state machine representation consisting of billions of states [12]. Another difficulty is that designing a reactive system requires knowledge spanning several domains. For example, the physical capabilities and limitations of the hardware (such as motors and sensors) must be understood, environmental factors must be taken into account, and the interfaces between components in the overall system must be comprehended. All of this information will impact the design and implementation of the software that controls the system.

Reactive control systems define a class of problems that can be easily modeled in terms of finite-state machines. Other safety-critical problem domains exist that are not well suited to a finite-state approach. In fact, many sophisticated control systems require complex scientific computations to process raw sensor data. Obvious examples are fly-by-wire and weapon control systems, where algorithms for solving differential equations may be needed to convert sensor data into the inputs required by the finite-state part of the control system. Again, the design of such software requires the application of expert knowledge from many domains: from mathematics and physics, for example, as well as from computer science.

In the preceding paragraphs, we have discussed two safety-critical problem domains. The first, that of reactive systems, is well suited to finite-state representations and manipulations. The second domain, that of scientific computation, requires more general computation than that of which finite-state automata are capable. Ideally, when constructing high-assurance software to solve hybrid problems belonging to both domains, one would like to use tools that handle software from both domains in a uniform manner.

2 Formal Methods

We argue in the preceding section that the construction of safety-critical systems is a complex task requiring participation by experts from a number of domains. In addition to knowledge, problem-domain experts generally have a good intuition about their domain, which allows them to proceed reasonably in the face of incomplete or conflicting information. But, such diverse knowledge and intuition cannot be applied willy-nilly to the construction of a complex system. If a programmer simply sits down at a terminal and begins typing a program, with problem domain experts kibitzing over his shoulder, chaos will reign, and a low-assurance system is almost certain to result.

One way to reduce chaos is to use sophisticated software engineering tools, which coordinate and structure the software development process and the resulting program. Such tools can improve the software development process *in general*, but they do nothing to give high assurance that a *particular* system design and implementation is correct and safe. Put another way, software engineering tools that address only the structure aspect of software development can provide some *assistance* in writing correct software, but no *assurance* that it is correct.

One way to assure correct and safe operation is to provide a formal proof (verification) that the software constructed correctly implements its specification. Constructing such proofs requires application of a *formal method*, which has a rigorous framework in which both to capture and organize the knowledge required for software development and to carry out proofs of correctness. In this paper we discuss some aspects of proof in a formal method based on automated stepwise-refinement of specifications into programs. This formal method has the advantage of being applicable to both the reactive control systems and the scientific computation software discussed in Section 1. Thus, it provies a uniform approach to constructing complex, high-integrity systems.

2.1 Program Verification vs. Stepwise Refinement

The traditional approach to program verification requires preparation of a formal specification, writing of a program that allegedly implements that specification, and a formal proof that the resulting program does in fact correctly implement the specification. Several formal methods restricted to the domain of reactive control systems have been developed [10][8][16]. Despite the help provided by such systems, proof is generally considered to be the most labor-intensive aspect of rigorous software construction.

Proofs of program correctness are labor-intensive, and hence expensive, for three reasons:

- the difficulty and complexity of proving that a large program meets its specification,
- proofs are often attempted at the wrong level of abstraction, and

- the inability to reuse parts of that proof in other applications.

In general, there is a large gap between a good formal specification for a problem and an efficient program implementing the specification. The final program contains little or no information on how that gap was bridged, making the proof difficult. Moreover, an efficient program is clouded by numerous implementation details specific to the particular problem, which render pieces of the proof difficult to reuse. Thus, a major factor in the difficulty of proof is the size of the gap between the specification and the program.

Stepwise refinement of a specification into a program is an attractive alternative to the traditional approach, especially if the steps in the refinement can be carried out automatically. In automated stepwise refinement, the program evolves from the specification in a series of steps. Stepwise refinement thus trades one large gap and one difficult proof for many smaller gaps and many simpler proofs.

In the extreme, one might make the steps in a refinement very small, so that their proofs are very simple (perhaps almost self-evident). In this approach, the refinement steps become *rewrite rules*, each of which replaces a fragment of a specification or program by another fragment that is just as correct as the original, but more "implemented". In this case, one usually speaks of proving that a rewrite rule *preserves the correctness of* (refines) the specification or program for every instance in which the rule applies.

If the constructs of the specification and programming language are *monotonic*, then refinements can be proved without regard to the context in which they will be applied. Monotonicity greatly simplifies the proof process and makes possibile the reuse of proofs in other contexts (i.e., in other programs). Furthermore, if the refinement relation is transitive, then the correctness of the application of a large number of refinements to a specification to create an implementation follows from the correctness of the individual refinements. As discussed earlier, a formal proof is necessary to provide the high assurance required of implementations for life- and safety-critical systems.

Therefore, if programs could be constructed automatically using simple rewrite rules, a number of advantages would accrue:

- the cost of writing the program by hand would be saved (once the cost of the development of the rules had been paid),
- the one complex proof of correctness for a program would be *factored* into a number of much simpler proofs demonstrating that each rule preserves correctness,
- the rules, and their proofs, could potentially be reused in the implementation of numerous specifications, amortizing the cost of their development over a large number of programs.

In the next section we discuss briefly a program transformation system that is capable of applying correctness-preserving rewrite rules automatically to construct implementations

from specifications. The remainder of the paper discusses part of the verification framework that enables one to prove rigorously that these rules preserve correctness. In this framework, we prove that transformations preserve correctness by mapping them into an axiomatized mathematical domain to which formal (and automated) reasoning can be applied. This mapping is accomplished via an extension to denotational semantics.

3 Automatic Refinement in the TAMPR Program Transformation System

The *Transformation Assisted Multiple Program Realization* (TAMPR) system [1] [5] is distinguished by two important features:

1. the requirement that transformations be pure rewrite rules consisting of a syntactic pattern and a syntactic replacement (in contrast to permitting procedural code in the replacement), and

2. the provision of a limited number of powerful control constructs.

These two features support a philosophy of achieving large refinements through numerous conceptually simple transformations, rather than from a small number of complex transformations. The use of a large number of simple transformations, in turn, facilitates both the proof that transformations preserve correctness and the reuse of transformations and hence their proofs.

The general approach to constructing a derivation in TAMPR is to group refinement transformations into *transformation sets*—several related transformations that achieve a well-defined goal and thus can be viewed as comprehending a single step in the stepwise refinement process. A sequence of several transformation sets, each set applied to the result of its predecessor, can then be composed to achieve the overall goal of a large refinement, such as the construction of an implementation from a specification.

For applying a transformation set, the most frequently used control construct in TAMPR is *application to exhaustion.* That is, the transformations in the set are applied automatically to every fragment in the specification that they match, *including fragments generated by prior applications.* Applying a transformation set to exhaustion (if it terminates) guarantees that the modified (output) specification has a particular *canonical form.* It is this canonical form that embodies the goal of the transformation set and, hence, is a realization of the corresponding refinement step. Perhaps surprisingly, passing a specification through a sequence of suitably chosen canonical forms yields an implementation [1]. The use of canonical forms in automated transformation is discussed further in [2].

TAMPR's ability to apply transformations automatically means that derivations can be constructed from many small transformations rather than a few large ones, as would be

required if manual application were attempted. Typically, a TAMPR derivation requires thousands to hundreds of thousands of rule applications. For example, in bootstrapping the implementation of its own specification, TAMPR applies over 170,000 transformations, in the course of which it investigates some 40 million potential applications. Clearly, such an application strategy could not be carried out manually.

TAMPR has been used to derive programs from specifications in a number of application domains, include solving systems of linear equations [7], bootstrapping the derivation of its own implementation [1], finding eigenvalues [3], and solving hyperbolic partial differential equations [4]. In the latter two examples, the efficiency of the derived programs equals or exceeds that of the corresponding handwritten implementations.

4 TAMPR Transformations

In the TAMPR transformation system and its accompanying wide spectrum language Poly [1] [5], specifications (and programs) and transformations are represented in terms of their syntax derivation trees (SDTs).

The SDTs used in transformations are special, in the following way: In order for a transformation to be useful, it must contain *variables*, so that it can apply to the SDTs of many different programs. The idea of variable is the same as that in algebraic and trigonometric identities. For example, x is a variable in the trigonometric identity

$$sin(x) + cos(x) = 1$$

In TAMPR, nonterminal symbols from the Poly grammar (the grammar of the programs and specifications being transformed) are used, with indices, to represent variables. Thus, the preceding identity (used left to right) would be expressed in TAMPR by the transformation

$$sin(<expr>_1) + cos(<expr>_1) \Rightarrow 1$$

An SDT having nonterminal symbols as variables is called a *schema*; it differs from an ordinary SDT in that not all of its leaves are terminal symbols.

As is evident from the preceding example, the general form of a transformation is

$$t_{pattern} \Rightarrow t_{replacement}$$

where $t_{pattern}$ and $t_{replacement}$ are schema SDTs, both of which have the same root nonterminal symbol. A TAMPR transformation is a rewrite rule stating that if the pattern schema of the transformation matches a subtree of the program SDT, then that subtree should be replaced by another subtree constructed from the replacement schema of the transformation using the values of the matched variables.

Stated formally, transformations are expressed as strings ω, composed of terminals and nonterminals of the wide spectrum language. The presence of nonterminals makes ω a schema. Nonterminals in ω are called *schema variables.* The notation for describing a TAMPR schema is as follows:

1. Let G be a grammar.
2. If $d \stackrel{*}{\Rightarrow} \omega$ in G, then we may write $d\{\omega\}$ in a TAMPR transformation. Here the nonterminal d is referred to as the *dominating symbol* of ω. Note that this notation can be applied recursively. For example, if $d_1 \stackrel{*}{\Rightarrow} \gamma d \beta$ in G, then we may write $d_1\{\gamma d\{\omega\}\beta\}$.

This notation for schemas makes explicit both the leaves of a subtree (some of which may be nonterminals) and its root (the dominating symbol). For more information on TAMPR transformations see [1] [5].

5 Verification Framework

As discussed in section 3, a TAMPR derivation consists of a transformation sequence consisting of a sequence of transformation sets. Let $\mathcal{T}_{1,n}$ denote such a sequence. Given a specification s, the application of $\mathcal{T}_{1,n}$ to s is denoted by the expression $\mathcal{T}_{1,n}(s)$. Let p denote the result of this expression. Our objective is to construct $\mathcal{T}_{1,n}$ so that p is a fully implemented program that can be compiled and executed by a computer.

If we can formally prove that p is correct with respect to s, then we know with mathematical certainty that no behavior or execution of p can ever be found to contradict the behavior of s [6]. In contrast, "verification" by testing only shows that none of the *tested* behaviors of p contradicts a behavior of s. A formal verification thus provides a high degree of assurance, indeed, that a program is correct, subject to the assumptions that the specification is correct and that the proof is carried out correctly.

To demonstrate correctness mathematically, we need to prove that p is at least as defined as s, $s \sqsubseteq p$, which is usually read "s is less defined than p". Such a proof can be accomplished by proving that $\mathcal{T}_{1,n}$ is *correctness preserving*, which in turn can be accomplished by showing that the individual transformations, $\mathcal{T}_i$ that make up $\mathcal{T}_{1,n}$ are correctness preserving. Thus, being able to prove that a transformation is correctness preserving is the key component of our verification framework.

To prove that a transformation of the form

$$t_{pattern} \Rightarrow t_{replacement}$$

preserves correctness we need to prove that the relation $t_{pattern} \sqsubseteq t_{replacement}$ holds. Such a proof implies that any instance of $t_{pattern}$ can be replaced by the corresponding instance of

$t_{replacement}$. This proof can be accomplished by mapping the transformation into an axiomatized mathematical domain where formal (and automated) reasoning can be performed. To perform this mapping, we use an extended denotational semantics. In this approach, the conceptual notion of *program state*, which forms the basis for human reasoning, is represented by the cross product of two functions, an environment function and a store function. This "distributed" representation of state introduces properties that go beyond those of which one is conscious when thinking in terms of the conceptual state. The reasoning framework needs to be aware of these additional properties to obtain a correctness proof. These additional properties of the computational state will be discussed further in section 5.2.

In the following sections, we discuss the notion of refinement in denotational semantics and how one can go about proving that a transformation is correctness preserving.

5.1 The Semantics of Schemas

In denotational semantics [15], programs are viewed in terms of their SDTs, and the semantics consists of valuation functions that map SDTs into expressions in some mathematical domain. These valuation functions assign a meaning to each terminal symbol in the grammar directly, and they assign a meaning to each nonterminal symbol indirectly, based on a composition of the meanings of its subnodes. Thus, in denotational semantics, the meaning of every symbol is derived ultimately from a mathematical expression involving the meanings of terminal symbols.

As mentioned in Section 4, TAMPR transformation schemas generally contain one or more nonterminal leaves. A nonterminal leaf has no terminal symbols from which to compose a meaning, so standard denotational semantics is unable to assign it a meaning. Thus, if one wishes to use the denotational semantics of a language as a basis for reasoning about the correctness of transformation schemas, the denotational semantic valuation functions must be extended to enable them to assign meanings to nonterminal leaves.

Fortunately, the denotational semantics for a language provides enough information to allow an extended semantics for a nonterminal to be determined. One should think of this extended semantics as being the "most general meaning" of the nonterminal; that is, the meaning common to all possible instantiations of that nonterminal.

For example, consider a nonterminal <*expr*> denoting the set of expressions. In general, the *meaning* of any syntax derivation tree having <*expr*> as its root will be an element belonging to the set of *denotable values* (i.e., values, such as integer, list, etc., that the programming language supports). Executing any instantiation of <*expr*> with respect to a specific environment and store will result in a denotable value (or possibly an undefined value) that is the value of the <*expr*>. Thus one thing that we know about the class of SDT's having <*expr*> as its root is that the semantics of every SDT in this class is of the

form:

$$\lambda\ (\varepsilon, s).\ \Delta_v(\varepsilon, s)$$

Where Δ_v is a semantic function having the signature

$$\Delta_v :\ \mathit{environment} \times\ \mathit{store} \rightarrow\ \mathit{denotable\ value}.$$

As another example, consider the nonterminal <*assign*> denoting the set of assignment statements. In general, for a sequential side-effect free language, executing an instantiation of <*assign*> will result in a (single) change to the store. If the denotational semantics of the language under consideration defines assignments as "commands that take an environment and a store as input and produce a store as output", then the corresponding meaning for <*assign*> will be:

$$\lambda\ (\varepsilon, s).\ \Delta_s(\varepsilon, s)$$

Where Δ_s is a semantic function having the signature

$$\Delta_s :\ \mathit{environment} \times\ \mathit{store} \rightarrow\ \mathit{store}.$$

Similarly, execution of an arbitrary declaration will result in a change to the environment (i.e., Δ_ε).

When considered in isolation, the extended semantics of a nonterminal like <*assign*> can be described as a function that takes an environment and a store as input and returns a new store. Because many nonterminals have an extended semantics that can be described in terms of such a "change", we have coined the term *delta function* to describe an abstract valuation function that gives the extended semantics of a nonterminal. Thus, the role of delta functions is to describe the "change in meaning" across a nonterminal.

5.1.1 The Importance of Delta Functions

Delta functions, in the semantics of schemas, play a role similar to that played by *variables* in standard algebraic expressions. However, it would be incorrect simply to use generic *variables* in place of delta functions. Consider the following transformation which replaces "if <*be*> then <*stmt*>$_1$ else <*stmt*>$_1$;<*stmt_tail*>$_1$" with "<*stmt*>$_1$;<*stmt_tail*>$_1$".

$$\mathcal{T}_1 \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \text{<}\mathit{stmt\_tail}\text{>}\{\ \text{if <}\mathit{be}\text{> then <}\mathit{stmt}\text{>}_1 \text{ else <}\mathit{stmt}\text{>}_1; \\ \qquad\qquad \text{<}\mathit{stmt\_tail}\text{>}_1 \\ \qquad\qquad \} \\ \Rightarrow \\ \text{<}\mathit{stmt\_tail}\text{>}\{\text{<}\mathit{stmt}\text{>}_1; \\ \qquad\qquad \text{<}\mathit{stmt\_tail}\text{>}_1 \\ \qquad\qquad \} \end{array} \right.$$

Given the standard semantics for the if-then-else construct, one might conclude that this transformation is correct. However, the correctness of this transformation not only depends upon the semantics of the if-then-else construct, but also upon whether the evaluation of boolean expressions, in the language under consideration, can cause side-effects. When using delta function semantics for the nonterminal <*be*>, this constraint becomes explicit, and a correctness proof will not "go through" for languages where such side-effects are possible. In contrast, when using a generic *variable* in place of <*be*> this information will not be present and must be accounted for by some other means.

5.1.2 Theoretical Considerations

There are many factors that determine just how specific a delta function can be. One such factor is the language itself. For example, in a language that supports parallel assignments, the most general delta function for a <*parallel_assign*> can only say that one or more identifiers will be assigned new values. Contrast this with an assignment statement in a sequential language where a side-effect free assignment will change the value of exactly one identifier.

Also, complexity plays a role. The preceding examples are quite simple. Nontrivial valuation functions and continuations can exist within a denotational semantics. For some of these situations, it is not immediately obvious what the appropriate and relevant delta functions are.

Finally, in addition to inherent properties of the language, properties established by preceding transformations can also have an effect on delta functions. Applying a sequence of transformations to a specification or program s will result in a program p having certain syntactic and semantic properties deriving from the canonical forms achieved by the transformations in the sequence. For example, a program can be transformed into a canonical form where evaluation of boolean expressions in conditional statements will **not** cause side-effects regardless of the general policy regarding side-effects that is supported by the language. To see this consider the following transformation:

$$\mathcal{T}_2 \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \text{<}stmt\_tail\text{>}\{\text{<}var\text{>}_1 := \text{<}be\text{>}\ ; \\ \qquad\qquad \text{if } \text{<}var\text{>}_1 \text{ then } \text{<}stmt\text{>}_1 \text{ else } \text{<}stmt\text{>}_1; \\ \qquad\qquad \text{<}stmt\_tail\text{>}_1 \\ \qquad\qquad \} \\ \Rightarrow \\ \text{<}stmt\_tail\text{>}\{\text{<}var\text{>}_1 := \text{<}be\text{>}\ ; \\ \qquad\qquad \text{<}stmt\text{>}_1; \\ \qquad\qquad \text{<}stmt\_tail\text{>}_1 \\ \qquad\qquad \} \end{array} \right.$$

This transformation can be applied in general, because it provides the context for its application—namely that the boolean expression of a conditional test consist of a single variable. However, suppose a transformation sequence has been applied to a program so that this property holds for all conditional statements within the program. For such a program the transformation $\mathcal{T}_1$ given earlier is correct! The transformation is correct in this context because transformation sequences can alter the semantics of delta functions.

In general, the properties established by preceding transformations can impact the semantics of delta-functions of future transformations that are used to further refine p. In the presence of such properties, one can think of a nonterminal as having a family of delta functions: a most general delta function which results from the semantics of the language, and other more specific ones that incorporate properties established by prior transformations.

We have found that for many transformation proofs, the most general delta function, which can usually be determined by inspection of the grammar, is sufficient to permit a proof to be obtained. However, because of the potential subtleties in determining the extended semantics of delta functions, and in order to accrue the correctness benefits of automation, we are developing an automated procedure for determining the semantics of delta functions given a set of denotational semantic definitions. We are also examining in detail how application of prior transformations can affect delta functions.

5.2 The Refinement Relation in $\mathcal{M}$

The objective of applying TAMPR transformations is to introduce and restructure computation in a manner consistent with the notion of refinement. To prove the correctness of a TAMPR transformation, we must prove that *any* program fragment that matches the pattern schema of the transformation is *refined* by a program fragment that is the correspondingly instantiated replacement schema. Consequently, to prove that a transformation performs a refinement, one must demonstrate that, for all possible instantiations of the schema variables in the pattern and replacement, the state of the program fragment produced from the instantiated replacement schema is a refinement of the state of the program fragment produced from the instantiated pattern schema. To carry out such proofs, we need to be able to reason about refinement relationships between states.

5.2.1 The State Space of a Denotationally Defined Computation

To give a full and correct description of the scope of identifiers, the *computational state space*, $\mathcal{M}$, for most denotationally defined languages is represented by the cross product of an environment function, ε, and a store function, s (and possibly some additional constructs such as counters). In this representation, obtaining the value corresponding to an identifier requires two steps: the environment function maps the identifier to a storage location, and the store function maps that storage location to a denotable value (i.e., a member of the set

of values that an identifier can denote, such as a number or a logical value). For example, suppose that in a program, the identifier x is assigned the value 5. After the assignment has taken place, the following facts will be true of the state:

$\varepsilon(x) = \alpha$ where α is a storage location;
"which" particular location x gets mapped to is not of interest
$s(\alpha) = 5$

From this example, one can see that, taken together, the environment and store functions are the abstract representation in the denotational semantics of the state information of a concrete program; we call the combination of these two functions the *abstract state.* The abstract state is important because it provides a basis for verification.

In traditional verification of programs, a program fragment is proved to be correct by showing that, if the execution of the program fragment is begun in an abstract state satisfying a given precondition, then it will terminate in an abstract state satisfying a given postcondition.

The transformational perspective is somewhat different, but nevertheless related. In the application of a transformation, the fragment of program corresponding to the pattern is replaced with the fragment of program corresponding to the replacement. If the semantics of the programming language allows us to conclude that, for any such pair of program fragments, the execution of the fragment corresponding to the replacement results in an abstract state that is a refinement of the abstract state produced by executing the fragment of program matching the pattern, then we can conclude that the substitution (i.e., the transformation) is *correctness preserving.* It is easy to show that correctness preservation is simply an adaptation of the traditional notion of program correctness to program substitution.

5.2.2 Refinement Properties within the State Space

Technically, the domain $\mathcal{M}$ forms a refinement lattice with $m_\perp \stackrel{\text{def}}{=} (\varepsilon_\perp, s_\perp)$ being the bottom element and $m_\top \stackrel{\text{def}}{=} (\varepsilon_\top, s_\top)$ denoting the top element. The components of $m_\perp$ and $m_\top$ are defined as follows:

$$\varepsilon_\perp \stackrel{\text{def}}{=} (\lambda\ x.\ \perp)$$
$$s_\perp \stackrel{\text{def}}{=} (\lambda\ x.\ \perp)$$
$$\varepsilon_\top \stackrel{\text{def}}{=} (\lambda\ x.\ \top)$$
$$s_\top \stackrel{\text{def}}{=} (\lambda\ x.\ \top)$$

$(\lambda\ x.\ \perp)$ is the constant function with value "bottom", and $(\lambda\ x.\ \top)$ that with value "top".

Before we consider refinement in $\mathcal{M}$ we define a notation for *function alteration.* After this, we give some standard definitions of refinement [13]. Then we consider some aspects of how the definition of refinement in $\mathcal{M}$ differs from these standard definitions.

Definition 1 ***(Function Alteration.)*** *Let ε denote an arbitrary environment function. The notation $[x \mapsto \alpha]\varepsilon$ denotes an environment having the same mapping as ε for all identifiers except x. For $[x \mapsto \alpha]\varepsilon$ the storage location (i.e., the value of the function) associated with x is α.*

For more discussion of this notation see [15].

Definition 2 *General refinement on functions. Given any two functions f and g such that $f : D_1 \to D_2$ and $g : D_1 \to D_2$.*

$$f \sqsubseteq g \stackrel{def}{=} \forall x \in D_1, f(x) \sqsubseteq g(x)$$

Definition 3 $f \equiv g \Leftrightarrow (f \sqsubseteq g \wedge g \sqsubseteq f)$

Definition 4 *General refinement on tuples.*

$$(f_1, g_1) \sqsubseteq (f_2, g_2) \stackrel{\text{def}}{=} (f_1 \sqsubseteq f_2) \wedge (g_1 \sqsubseteq g_2).$$

The preceding definition is the standard definition of refinement for tuples [13], applicable to all tuples. In contrast, environment and store functions enjoy additional properties with respect to refinement that are not shared by general functions. These additional properties are important in constructing proofs of correctness-preservation for transformations involving environment and store functions, because these properties enable proofs in cases that could not be proved from the general definition of refinement alone. To emphasize the difference between general refinement for functions and refinement for the domain $\mathcal{M}$, we introduce the symbol, $\sqsubseteq^{\mathcal{M}}$ to denote the refinement relation as it manifests itself for states in $\mathcal{M}$. The semantics of $\sqsubseteq^{\mathcal{M}}$ is given below.

For states, definition 4 can be weakened from an equality to an implication as stated in Axiom 1.

Axiom 1 $(\varepsilon_1 \sqsubseteq \varepsilon_2) \wedge (s_1 \sqsubseteq s_2) \Rightarrow (\varepsilon_1, s_1) \sqsubseteq^{\mathcal{M}} (\varepsilon_2, s_2)$

Axiom 2 *Refinement within $\mathcal{M}$*

$$(\varepsilon_1, s_1) \sqsubseteq^{\mathcal{M}} (\varepsilon_2, s_2) \stackrel{\text{def}}{=} \quad (\forall x \in id, \quad ((\varepsilon_2(x) = \perp) \Rightarrow (\varepsilon_1(x) = \perp)) \wedge (s_1(\varepsilon_1(x)) \sqsubseteq s_2(\varepsilon_2(x))).$$

Intuitively, we know that the particular memory address of an identifier is not important with respect to the conceptual notion of state presented here. This axiom expresses that property for the abstract state; it enables one state to be proved a refinement of another independently of the particular value output by the environment function ε. Note that $((\varepsilon_2(x) = \perp) \Rightarrow (\varepsilon_1(x) = \perp))$ is critical for most imperative languages. This expression distinguishes the case in which an identifier is undefined because it has not been declared from the case in which the identifier is undefined because it has not been assigned a value.

Axiom 3 *For a given* α. $(\neg\exists x \in id, \varepsilon(x) = \alpha) \Rightarrow (\varepsilon, s) \sqsubseteq^{\mathcal{M}} (\varepsilon, [\alpha \mapsto \perp]s)$.

This axiom states that the value of any location in the store that does not have a corresponding identifier is irrelevant. This axiom is included largely for convenience, because it allows the denotational semantics to omit "storage cleanup" operations between scope boundaries.

5.3 Refinement of schemas

We can extend the above definition of refinement of computational states to define refinement for (transformation) schemas. Given a transformation schema t (a syntactic object), we use the symbol $\widehat{t}$ to denote the expression in the mathematical domain (the semantic object) assigned to t by our extended denotational semantics.

Definition 5 *(general refinement—unconditional correctness)*

$$t_1 \sqsubseteq t_2 \stackrel{\text{def}}{=} \forall state_i \in states,\ \widehat{t}_1(state_i) \sqsubseteq^{\mathcal{M}} \widehat{t}_2(state_i)$$

This is the most general form of refinement on schemas. Note that $\widehat{t}_1(state_i) \sqsubseteq \widehat{t}_2(state_i) \Rightarrow \widehat{t}_1(state_i) \sqsubseteq^{\mathcal{M}} \widehat{t}_2(state_i)$, but the implication generally does not hold in the other direction. Also note that this definition extends the definition of refinement from the semantic domain into the syntactic domain. From this point on, it makes sense to talk about "refinement of schemas".

5.4 Semantic Properties

In section 5.2.2 we discussed (semantic) properties of the state space. Additional state properties and functions are often useful for showing that one schema is a refinement of another. The most common such property is *uniqueness* (for identifiers) and the most common function is *new* (for addresses). Their definitions are

Definition 6 $unique(x, (\varepsilon, s)) \stackrel{\text{def}}{=} (\varepsilon(x) = \perp)$

Definition 7 $new \stackrel{def}{=} (\lambda\ \varepsilon.\ \alpha)$ such that $\neg(\exists x \in id,\ \varepsilon(x) = \alpha)$ holds.

The reason *new* is a function and not a predicate is that refinements may substitute one state for another. In such cases it is important for *new* to have the desired properties with respect to the substituted state (i.e., *new* is a function on environments). Also note that the definition of *new* places a requirement on the storage allocation and management strategy that it be able to generate an α with respect to a specific ε in accordance with the definition of *new*.

5.5 Syntactic Properties of Fragments Matched by Schemas

In contrast to state properties, which are defined in the semantic domain, there exist properties that are defined directly on the syntactic structure of a fragment that is matched by a schema (i.e., an instantiation of a schema). Schema instantiations possessing certain properties can be correctly transformed in nongeneral ways. Such properties are expressed as predicates defined on the syntactic representation of an instantiation.

Despite their syntactic nature, syntactic properties do have semantic implications that can be utilized in the course of a refinement proof (i.e., a correctness proof). The following is an informal definition for one syntactic property:

Definition 8 $occurs(x, f(t))$—this predicate is *true* if and only if the variable x occurs in $f(t)$,where $f(t)$ denotes the program fragment matching the schema t.

A theorem that describes some of the semantic implications of this definition is

Theorem 1 A semantic consequence of the *occurs* property.

$$\neg occurs(x,t) \Rightarrow \forall(\varepsilon_i, s_i) \in states,\ (\varepsilon_i', s_i') = ([x \mapsto \alpha]\varepsilon_i'', [\alpha \mapsto s_i(\varepsilon_i(x))]s_i'')$$

- where $(\varepsilon_i', s_i') \stackrel{\text{def}}{=} \hat{t}((\varepsilon_i, s_i))$,
- $(\varepsilon_i'', s_i'') \stackrel{\text{def}}{=} \hat{t}(([x \mapsto \perp]\varepsilon_i, s_i))$, and
- $new(\varepsilon_i'') = \alpha$

This theorem states that when executing a program fragment in which the identifier x does not occur, one may create a new environment that does not contain x, execute the program fragment with respect to this new environment and then reinsert x and the value to which it was originally bound (i.e., $s_i(\varepsilon_i(x))$) in the resulting (final) state. Care must be taken that, when x is reinserted, it is mapped to a "new" location in the store (i.e., α must be a "new" location. with respect to ε_i'', not ε). In order to deal with cases such as this, *new* needs to be a function on environments (recall the discussion in Section 5.4).

5.6 Correctness Proofs

In this section we consider two TAMPR transformations that are used to transform Poly specifications into programs. For a partial grammar of Poly and its denotational semantics, see [18]. For more information on TAMPR and the syntax of transformations, see [1].

5.6.1 Two example transformations.

Two TAMPR transformations whose proofs of correctness preservation depend on properties of the environment and store are Declaration Order Interchange and Assignment Distribution for Lambda Expressions:

- Declaration Order Interchange. If two variables are declared in the same statement in a Poly program, interchanging the order in which the two variables are declared is a refinement.

$$\mathcal{T}_1 \stackrel{\text{def}}{=} \begin{cases} \textit{<spec stmt>}\ \{\textit{<standard type>}_1\textit{<ident>}_1,\textit{<ident>}_2\} \\ \Rightarrow \\ \textit{<spec stmt>}\ \{\textit{<standard type>}_1\textit{<ident>}_2,\textit{<ident>}_1\ \} \end{cases}$$

- Assignment Distribution for Lambda Expressions. If a program variable (in procedural code) is assigned the value of a lambda expression (in functional code), then it is a refinement to replace this assignment by a declaration of the lambda variable enclosing the sequence: assign the lambda variable the value of the lambda argument expression followed by assign the procedural variable the value of the lambda body expression.

if (*<ident>*$_2$ does not occur in *<expr>*$_2$) then

$$\mathcal{T}_2 \stackrel{\text{def}}{=} \begin{cases} \textit{<stmt tail>}\ \{ \\ \quad \textit{<ident>}_1 = \textit{lambda <ident>}_2\ @\textit{<expr>}_1\ \textit{end (<expr>}_2\textit{);} \\ \quad \textit{<stmt tail>}_1\textit{;} \\ \} \\ \Rightarrow \\ \textit{<stmt tail>}\ \{ \\ \quad \textit{block;} \\ \quad \textit{declare cell <ident>}_2\textit{; enddeclare;} \\ \quad \textit{<ident>}_2 = \textit{<expr>}_2\textit{;} \\ \quad \textit{<ident>}_1 = \textit{<expr>}_1\textit{;} \\ \quad \textit{end;} \\ \quad \textit{<stmt tail>}_1\textit{;} \\ \} \end{cases}$$

In Assignment Distribution for Lambda Expressions ($\mathcal{T}_2$), the pattern consists of two portions. The first portion is an assignment statement in which the identifier $<ident>_1$ is assigned the value resulting from the application of a lambda function to the argument $(<expr>_2)$. The lambda function has $<ident>_2$ as its formal parameter and $<expr>_1$ as its body. The second portion of the pattern consists of $<stmt\ tail>_1$ which denotes the portion of the program that follows the assignment statement.

The replacement of $\mathcal{T}_2$ also consists of two portions. The first portion is a block (delimited by *end*) in which the identifier $<ident>_2$ is declared. After its declaration, $<ident>_2$ is assigned the value of the expression $<expr>_2$, then $<ident>_1$ is assigned the value $<expr>_1$. The second portion of the replacement consists of $<stmt\ tail>_1$ which denotes the portion of the program that follows the block.

The correctness of the transformation $\mathcal{T}_2$ depends on a global assumption that the name of every lambda variable is unique (this name occurs as the name of the lambda variable in no other lambda expression in the program being transformed). This assumption is easily guaranteed by applying an earlier transformation set that renames lambda variables to guarantee uniqueness.

Given the assumption of unique lambda variable names, $<ident>_2$ does not occur in $<expr>_2$. Hence, $<expr>_2$ may be evaluated in an environment in which $<ident>_2$ has been newly declared. Essentially, $\mathcal{T}_2$ describes how function parameters and parameter passing can be implemented by imperative (nonfunctional) commands.

Theorem 2 *(declarations are commutative).*

$$< \ spec\ stmt > \{<standard\ type>_1 <ident>_1, <ident>_2\}$$
$$\sqsubseteq$$
$$< \ spec\ stmt > \{<standard\ type>_1 <ident>_2, <ident>_1\}$$

Proof: see [17]

Theorem 3 $\neg occurs(x, <expr>_2) \Rightarrow$

```
<stmt tail> {
                <ident>1 = lambda <ident>2 @<expr>1 end (<expr>2);
                <stmt tail>1;
              }
⊑
<stmt tail> {
                block;
                declare cell <ident>2; enddeclare;
                <ident>2 = <expr>2;
                <ident>1 = <expr>1;
                end;
                <stmt tail>1;
              }
```

Proof: see Appendix A.

6 Conclusions and Future Work

Correctness proofs are necessary to have high assurance that design and implementation will produce software that satisfies the original specification.

We have argued that using an automated program transformation system to derive programs from specifications is an attractive approach to carrying out such proofs. Automating the derivation enables the use of large numbers of transformations that perform very simple refinements. It is thus relatively easy to prove that these small transformations preserve correctness, that is, that they are indeed refinements. Hence, the key component of our approach is to enable individual transformations to be proved to preserve correctness with the expenditure of a reasonable amount of effort.

In our approach, the semantics of the specification and implementation language is defined using denotational semantics. Traditional denotational semantics does not define the semantics of schema variables. Schema variables occur frequently in TAMPR transformations and the need to assign meanings to them motivated us to extend denotational semantics with delta functions. Delta functions can have a straightforward extended semantics; however, languages and contexts within a transformation sequence can also make the extended semantics of delta functions complex. For these reasons, we are developing an automated procedure for determining the extended semantics of delta functions with respect to a given grammar and its denotational semantics.

In general, the computational state space $\mathcal{M}$, within the denotational semantics of a language consists of an environment and a store function. The (execution) semantics of programs (syntactic objects) are then defined in terms of $\mathcal{M}$. The environment and store functions when considered together capture the notion of the conceptual state of a

computation. But, spreading information about the conceptual state over two functions in the computational state introduces dependencies between the two functions. To allow reasoning about the computational state to proceed "smoothly", these dependencies must be factored out. The axioms, definitions, and lemmas in section 5.2.2 permit "smooth" reasoning with respect to the state space $\mathcal{M}$ that we have chosen.

In conclusion, we believe that an automated refinement-based approach to software design, implementation, and verification within a properly adapted denotational semantic framework can provide high assurance of correctness for software.

A Proof of Theorem 3

In the interests of clarity, the fact that $<ident>_1$ and $<ident>_2$, are actually schema variables (and therefore are semantically denoted by delta functions) is omitted in this proof. Because the delta functions for $<ident>_1$ and $<ident>_2$ do not play a significant role in the proof, we simply treat them as identifiers. Also omitted from the proof is how type information participates in the proof process (namely the relationship between the type of a lambda bound identifier and a declared identifier).

The proof begins by noting that $<expr>_1$, $<expr>_2$, and $<stmt\_tail>_1$ are schema variables whose extended semantics will be denoted respectively by the following delta functions:

1. $delta_{<expr>_1} \stackrel{\text{def}}{=} (\lambda(\varepsilon, s).\ \triangle_{<expr>_1}(\varepsilon, s))$ where $\triangle_{<expr>_1} : \varepsilon \times s \rightarrow$ denotable value
2. $delta_{<expr>_2} \stackrel{\text{def}}{=} (\lambda(\varepsilon, s).\ \triangle_{<expr>_2}(\varepsilon, s))$ where $\triangle_{<expr>_2} : \varepsilon \times s \rightarrow$ denotable value
3. $delta_{<stmt\_tail>_1} \stackrel{\text{def}}{=} (\lambda(\varepsilon, s).\ (\triangle_\varepsilon(\varepsilon), \triangle_s(s)))$
   where $\triangle_\varepsilon : \varepsilon \rightarrow \varepsilon'$ and $\triangle_s : s \rightarrow s'$

The extended denotational semantics maps

<stmt tail> {*<ident>*$_1$ = *lambda <ident>*$_2$ @ *<expr>*$_1$ *end (<expr>*$_2$ *);*
<stmt tail>$_1$
}

to

4. $\lambda(\varepsilon, s).\ (\triangle_\varepsilon(\varepsilon),$
   $\triangle_s([\varepsilon(<ident>_1) \mapsto (\triangle_{<expr>_1}([<ident>_2 \mapsto \alpha_2]\varepsilon,$
   $[\alpha_2 \mapsto \triangle_{<expr>_2}(\varepsilon, s)]s))]s)$

where $new(\varepsilon) = \alpha_2$. Similarly, the schema

$<stmt\ tail>$\{
block;
declare cell $<ident>_2$*; enddeclare;*
$<ident>_2 = <expr>_2$*;*
$<ident>_1 = <expr>_1$*;*
end;
$<stmt\ tail>_1$*;*
\}

is mapped to

$$5.\lambda(\varepsilon, s).\quad (\triangle_\varepsilon(\varepsilon),\ \triangle_s([\varepsilon(<ident>_1) \mapsto (\triangle_{<expr>_1}([<ident>_2 \mapsto \alpha_2]\varepsilon,\ [\alpha_2 \mapsto \triangle_{<expr>_2}([<ident>_2 \mapsto \alpha_2]\varepsilon, s)]s))]s)$$

The semantic implications of the assumption $\neg occurs(<ident>_2, f(<expr>_2))$ together with Axiom 3 allow us to conclude that

$$6.\ \triangle_{<expr>_2}(\varepsilon, s) \sqsubseteq \triangle_{<expr>_2}([<ident>_2 \mapsto \bot]\varepsilon, [\varepsilon(<ident>_2) \mapsto \bot]s)$$

Combining 6, 4 and the definition of refinement on states gives

$$\begin{aligned} 7.\ \lambda(\varepsilon, s).\quad & (\triangle_\varepsilon(\varepsilon),\ \triangle_s([\varepsilon(<ident>_1) \mapsto (\triangle_{<expr>_1}([<ident>_2 \mapsto \alpha_2]\varepsilon,\ [\alpha_2 \mapsto \triangle_{<expr>_2}(\varepsilon, s)]s))s) \\ \sqsubseteq^{\mathcal{M}} & \\ \lambda(\varepsilon, s).\quad & (\triangle_\varepsilon(\varepsilon),\ \triangle_s([\varepsilon(<ident>_1) \mapsto (\triangle_{<expr>_1}([<ident>_2 \mapsto \alpha_2]\varepsilon,\ [\alpha_2 \mapsto \triangle_{<expr>_2}([<ident>_2 \mapsto \bot]\varepsilon, s)]s))s) \end{aligned}$$

A final application of the definition of refinement on states gives

$$\begin{aligned} 8.\ \lambda(\varepsilon, s).\quad & (\triangle_\varepsilon(\varepsilon),\ \triangle_s([\varepsilon(<ident>_1) \mapsto (\triangle_{<expr>_1}([<ident>_2 \mapsto \alpha_2]\varepsilon,\ [\alpha_2 \mapsto \triangle_{<expr>_2}([<ident>_2 \mapsto \bot]\varepsilon, s)]s))]s) \\ \sqsubseteq^{\mathcal{M}} & \\ \lambda(\varepsilon, s).\quad & (\triangle_\varepsilon(\varepsilon),\ \triangle_s([\varepsilon(<ident>_1) \mapsto (\triangle_{<expr>_1}([<ident>_2 \mapsto \alpha_2]\varepsilon,\ [\alpha_2 \mapsto \triangle_{<expr>_2}([<ident>_2 \mapsto \alpha_2]\varepsilon, s)]s))]s) \end{aligned}$$

Q.E.D.

References

[1] James M. Boyle. Abstract programming and program transformation—an approach to reusing programs. In T. J. Biggerstaff and A. Perlis, editors, *Software Reusability, Vol. 1*, pages 361-413. Addison-Wesley, 1989.

[2] James M. Boyle. *Automatic, Self-adaptive Control of Unfold Transformations.* PROCOMET '94, IFIP Working Conference on Programming Concepts, Methods and Calculi, San Miniato, Italy, June 6-10, 1994. North-Holland/Elsevier, 1994, pages 83-103.

[3] J. M. Boyle, S. M. Fitzpatrick, and T. J. Harmer. *The Construction of Numerical Mathematical Software for the AMP DAP by Program Transformation.* Parallel Processing: CONPAR 92 - VAPP V, Second Joint International Conference on Vector and Parallel Processing, Lyon, France, September 1992, Lecture Notes in Computer Science Vol. 634, pages 761-767, Springer-Verlag, Berlin, 1992.

[4] J. M. Boyle and T. J. Harmer. A Practical Functional Program for the CRAY X-MP. Journal of Functional Programming, Vol. 2, No. 1, January 1992, pp. 81-126.

[5] James M. Boyle and Manohar N. Muralidharan. Program Reusability through program transformation. *IEEE Transactions on Software Engineering*, Vol. SE-10 (5):574-588, September 1984

[6] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, Princeton, New Jersey 1994.

[7] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *Linpack User's Guide.* SIAM Philidelphia, 1979.

[8] Frederic Boussinot and Robert de Simone. *The Esterel Language.* Proceedings of the IEEE, Vol. 79, No. 9, Sep. 1991, 1293-1304.

[9] David Gries. *The Science of Programming.* Springer-Verlag, New York, New York, 1985.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. *The Synchronous Data Flow Programming Language Lustre.* IEEE Special Issue on Real Time Programming, Proceedings of the IEEE 79(9), Sep. 1991, 1305-1320.

[11] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, Inc. San Diego, California, 1973.

[12] Claus Lewerentz and Thomas Lindner (Eds.). Formal Development of Reactive Systems: Case Study Production Cell. Lecture Notes in Computer Science, Vol. 891, Springer-Verlag, Berlin, 1995.

[13] Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill Inc., New York, New York, 1974.

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

[15] David A. Schmidt. *Denotational Semantics.* Wm. C. Brown Publishers, Dubuque, Iowa, 1986.

[16] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. *Programming Real-Time Applications with signal.* Proceedings of the IEEE, Vol. 79, No. 9, Sep. 1991.

[17] Victor L. Winter and James M. Boyle. *Proving Refinement Transformations Using Extended Denotational Semantics.* Proceedings of the '96 Durham Transformation Workshop.

[18] Victor L. Winter. *Proving the Correctness of Program Transformations.* Ph.D. dissertation, University of New Mexico, 1994.

B Biography

Victor L. Winter received his Ph.D. from the University of New Mexico in 1994. His dissertation research focused on proving the correctness of program transformations. Dr. Winter is a member of the High Integrity Software (HIS) group at Sandia National Laboratories. His research interests include trusted software, formal semantic models, theory of computation, automated reasoning and robotics. Dr. Winter can be reached by phone in the United States at (505) 284-2696 or by email at vlwinte@sandia.gov.

James M. Boyle received his Ph.D. from Northwestern University in 1970. He has been active in the field of program transformation since writing his dissertation on the initial design of the TAMPR transformation system. He is a member of the Mathematics and Computer Science Division at Argonne National Laboratory. Dr. Boyle's other research interests include trusted software, parallel processing, and automated reasoning. He is coauthor of the books *Automated Reasoning—Introduction and Applications* and *Portable Programs for Parallel Processors.* He can be reached at +1 708-252-7227 or by email at boyle@mcs.anl.gov