An Overview of HATS: A Language Independent High Assurance Transformation System

Victor L. Winter
Sandia National Laboratories
vlwinte@sandia.gov

Abstract
Transformations that are based on syntax directed rewriting systems can have a significant impact on the construction of high assurance systems. However, in order for a transformational approach to be useful to a particular problem domain, a (general) transformation system must be adapted to the notation of that particular domain. A transformation system that can be easily adapted to various domain notations has the potential of having a wide range of applicability.

In this paper we discuss why transformation is attractive from a high assurance perspective, as well as some issues surrounding automated transformation within specific problem domains. We then give an overview of a language independent High Assurance Transformation System (HATS) that is being developed at Sandia National Laboratories.

1 Introduction
1.1 Background
It is recognized that partial “black box” testing, when applied to discontinuous systems, is less effective than when applied to continuous systems [7][1]. The reasoning is that as the functionality of a system becomes more discontinuous, inferring the behavior of input elements that are not tested from “similar” input elements that are tested becomes increasingly problematic. A well known statement reflecting this sentiment is: “Testing can only show the presence of errors, not their absence”.

Given a black box testing environment, it can be argued that as a system becomes more complex it also becomes increasingly difficult to make meaningful inferences concerning portions of its input space that have not been explicitly tested. In turn, this implies that in order to obtain a given level of assurance, a complex system will need to be more extensively tested than a simple system. When high levels of assurance (e.g., less than $10^{-9}$ failures per operational hour) are required for complex (e.g., highly discontinuous) systems with large input spaces, traditional black box testing paradigms become ineffective [7][1].

1.2 Attaining High Assurance in Complex Software
Today we are building systems of unprecedented complexity. In addition, technological advances are also dramatically increasing the rate at which complexity is growing. For example, planes are now being designed that are inherently unstable and require complex control functions in order to fly safely. To date, software systems are at the forefront of technology’s need/demand for complexity.

When developing a software system there are two general sources of error: (1) the initial set of requirements are incorrect, and (2) a correct set of requirements exist, but they are implemented incorrectly. Given this partitioning of error sources, let us consider black box testing of complex software for which we require a high degree of assurance. As we have already mentioned, demonstrating high assurance of a complex (software) system will require the examination of a very large number of test cases. Realistically, the only way that one could hope to examine large numbers of test cases is through automation.

In order to automate testing, a formal (and computable) specification, often called a test oracle, is needed to determine (recognize) whether the input/output pairs generated by the testing process are “correct”.

---

*This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy.

---

1It should be noted that if the test oracle is itself incorrectly
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
ance, when comparing alternative approaches to black box testing we may assume that we are given a correct test oracle. This oracle can also serve as a somewhat nonalgorithmic, formal specification which can be used as the basis for formal software development.

Let $S_0$ denote such a formal specification, $S_n$ denote an implementation of that specification, and $\sqsubseteq$ denote a correctness relation (which is transitive and reflexive). In theory then, if we can show that

$$S_0 \sqsubseteq S_n$$

(automated) black box testing would not be necessary. That is, showing that the implementation $S_0$ is correct with respect to the specification $S_0$ would, by definition of correctness, correspond to exhaustive black box testing using the test oracle, $S_0$. Note that we might however, still want to perform testing to validate $S_0$.

### 1.3 Formal Verification through Transformation

Showing that the relation $S_0 \sqsubseteq S_n$ holds is more commonly referred to as program verification, formal verification, or simply verification. In theory, verification works. In practice however, the calculations needed to directly show that the relation $S_0 \sqsubseteq S_n$ holds are most often overwhelming. Informally, the difficulties encountered here result from the fact that a large part of the verification process is concerned with implementation details and how they interact to solve the desired problem.

Due to the difficulties encountered in directly verifying that a program satisfies a formal specification, a paradigm for obtaining programs from formal specifications is being explored in which the gap between formal specifications and programs is bridged through a sequence of small “steps” or changes. These steps are traditionally called transformations, and their aggregation is called a transformation sequence.

Through a transformation sequence one can transform a specification into an implementation via a sequence of (hopefully small) transformations. This process yields a number of intermediate representations of $S_0$. More specifically, if $n$ transformation steps are performed then we will have the representations: $S_0, S_1, S_2, \ldots, S_n$. Given two representations $S_i$ and $S_j$ in this sequence, it will generally be the case that when $i < j$, $S_i$ will be a representation that “looks” a little more like the initial specification, $S_0$, while $S_j$ will be a representation that “looks” a little more like the final implementation, $S_n$.

Intuitively, the motivation for having small transformations is that as $S_i$ and $S_{i+1}$ become increasingly similar to one another, $S_i \sqsubseteq S_{i+1}$ should become easier to demonstrate.\(^2\)

And finally, since $\sqsubseteq$ is transitive, we can calculate $S_0 \sqsubseteq S_n$ by showing that

$$\forall i : 0 \leq i < n \rightarrow S_{i-1} \sqsubseteq S_i$$

holds. In this case, we say that the transformation sequence $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n$ is correctness preserving.

Under the right circumstances and with careful planning, calculating that a transformation sequence is correctness preserving is significantly easier (to the point of being practical) than a direct calculation of $S_0 \sqsubseteq S_n$. Thus, when handled properly, the approach to program verification offered by transformation can make a substantial contribution towards the construction of high assurance software. For a nontrivial example demonstrating the benefits of a transformation-based approach to software construction see [8].

### 2 Non-procedural Transformation

Historically, transformations have been categorized as being either procedural or schematic [5]. A distinguishing feature of procedural transformation is that it tends to make use of global (contextual) information (e.g., flow analysis, consistency checks, and global cleanup operations). A standard implementation of a compiler is an example of a procedural transformation system. In contrast, schematic transformations are syntax driven and focus on making local changes in a manner that has its foundations in “term rewriting”.

Recently, the notion of a “match” has become quite involved for many schematic transformation systems. Because of this, we feel it is more appropriate to refer to these systems as non-procedural transformation systems. For the remainder of this article, we will use the term transformation to denote non-procedural transformation.

When viewed externally, non-procedural transformations are simply string substitutions. Their origins lie in directed equalities (i.e., rewrites that are applied in a left to right manner) and have the form:

$$T \overset{\text{def}}{=} \text{pattern} \Rightarrow \text{replacement}$$

\(^2\)Consider the case in the limit where we want to show that $S_i \sqsubseteq S_i$. 

then the value of oracle-based testing is questionable at best.
The semantics is as follows: If $T$ is applied to a string, $s$, that "matches" (i.e., satisfies the property described in) the expression denoted by pattern, then the information gathered from the match (e.g., a substitution list) is used to instantiate the replacement term which is then substituted for $s$.

Traditionally, the entire string of symbols that is being manipulated is referred to as a program [4] even though its relationship to an actual computer program might be very remote (e.g., a program might describe a non-computable function). We will refer to such programs as target programs in order to distinguish them from programs belonging to other languages such as a transformation language (see Section 2.2).

Abstractly, there are two aspects of a non-procedural transformation: (1) determining whether a string satisfies a particular property (i.e., a "match" occurs), and (2) the control mechanism behind the transformation process (e.g., the order and frequency in which strings are examined and transformations applied).

2.1 Matching

In order to achieve general matching objectives, virtually all transformation systems support the notion of a variable which can take on the value of elements belonging to a particular domain. Because of their wide range of applicability (i.e., its matching potential), a transformation pattern containing such variables is referred to as a template or schema. For the sake of clarity, we will refer to variables that occur in schemas as schema-variables.

A distinguishing feature of non-procedural transformation is that the matching process binds schema-variables to values and then uses these bindings to instantiate the replacement expression.

Intimately tied in with the notion of schema-variables and domains is the notion of unification. It is worth noting that in the context of transformation, unification often goes far beyond the traditional kind of unification (e.g., where a variable is unified with a term). For an example of unification in a transformational context see Section 4.2.1.

2.2 Control

How transformation sequences are described is also another fundamental aspect of a transformation system. At one end of the spectrum a user is required to interactively apply transformations to specific points in the target program. In such an approach, the transformation sequence is constructed manually. At the other end of the spectrum transformation sequences are described in a special transformation language much in the same way that calculation sequences are described in a traditional programming language such as C or FORTRAN. We refer to such a formal description of a transformation sequence as a transformation program.

In summary, a transformation program takes a target program as its input, is executed by a transformation system, and produces a transformed target program as its output.

2.2.1 The Pros and Cons of Manual and Automated Control

The advantage of a manual transformation system is that complex transformation sequences can be realized. A disadvantage however, is that the number of transformations that one can realistically expect a user to apply is limited (e.g., perhaps to thousands). Hence this approach does not scale very well. Another disadvantage is that manually applied transformation sequences (i.e., those interactively constructed by the user) cannot readily be reused or applied to "similar" problems.

The advantages and disadvantages of automatic application of transformation sequences are essentially the reverse of those that were given for manual transformation. That is, automatic transformation is (1) scalable, (2) reusable, and (3) not as flexible as a manual approach.

3 Practical Issues

3.1 High Assurance Transformation

How does one demonstrate high assurance for an implementation (target) program that is obtained through transformation? As was mentioned in Section 1.3, this can be accomplished by presenting sufficiently convincing evidence that the transformation sequence, used to generate the implementation, is correctness preserving, which in turn consists of showing the correctness of each individual transformation in the sequence.

In practice, verifying the correctness of a single transformation requires (1) that there is a formal semantics for the target language, (2) that the transformations themselves can be given a formal semantics (e.g., schema-variables, etc.) with respect to the semantics of the target language, and (3) that the desired correctness relation (e.g., $\subseteq$) can be axiomatized within this semantic framework.
Given these requirements it should be clear that, a program that is transformed in such a manner that it syntactically falls outside of the target language, will have an undefined semantics and cannot satisfy the correctness relation with respect to the semantic framework that has been constructed. Thus it is essential that transformations rewrite programs in such a way that the resulting programs remain syntactically within the target language. This presents somewhat of a conceptual problem when the objective of transformation sequence is to move a program from one level of abstraction (e.g., a high-level specification) to another (e.g., a compileable implementation). A standard approach taken to address this problem is to create a wide-spectrum (target) language in which it is possible to express (within a single target program instance) constructs at various levels of abstraction.

3.2 Domain Specificity

Ideally, one would like to start a transformation-based approach to software development within the problem domain. This is based on the belief that the concepts, constructs, and notations of a domain language should be better suited to expressing, understanding, and reasoning about a problem indigenous to that domain than some general purpose (possibly even computer oriented) language. It is at this juncture where a transformational approach to software development is faced with two serious challenges. The first challenge is how does one reconcile the notion of a domain specific language with the notion of a wide spectrum language? The second challenge is how, in practice, does a general purpose transformation system support the flexibility necessary to permit a domain specific philosophy?

3.3 Domain Specific and Wide Spectrum Languages

In order for a transformational approach to work, there must exist a wide-spectrum language in some form. In order for a transformational approach to be practical, it must be able to adapt to a specific domain language. Two general approaches address this dichotomy. In the first approach a (new) wide-spectrum language is constructed by merging the given domain language with the original wide-spectrum language.

The second approach is to develop transformational “definitions” (or axioms) that enable two languages (e.g., domain and wide-spectrum) to be linked together. This can be done at both the syntactic and the semantic levels. In effect, this creates a transformational compiler between the domain and wide-spectrum language.

3.4 Internal Representation

When viewed internally, target programs are generally not represented as strings. Depending on the matches and manipulations (e.g., substitutions) a transformation system permits or facilitates, target programs can be internally represented in a number of ways. For example, one transformation system might represent a target program in terms of a data dependency graph, while another will represent it as a syntax derivation tree. As the amount and type of information contained within an internal representation increases, so does the complexity of the representation.

From a practical perspective, internal representations of the target program can significantly impact the amount of effort needed to extend transformation into specific problem domains (e.g., matching is defined with respect to the internal representation). Depending on the complexity of an internal representation, the translation process from a program in the domain language to an internal representation can become quite involved. At one extreme, this translation can become “compiler-like” — with associated correctness issues and concerns.

Interestingly enough, such compilers can be realized through transformations themselves, with corresponding high-assurance benefits. Because of the possibility of transformational compilation, we believe that an internal SDT representation of programs is a very promising paradigm.

In an SDT representation, the transformation system is given a context-free grammar of a wide-spectrum language that contains the domain language as a subset. One very attractive aspect of an SDT representation is that non-terminals in the grammar define typed schema-variables within the transformational framework. Thus, it requires no effort beyond the definition of the grammar and tokens in the language) on the part of the user to tune an SDT-based transformation system to a particular problem domain.

4 An Introduction to HATS

At Sandia National Laboratories, the High Integrity Software (HIS) program is developing a High Assurance Transformation System (HATS) to explore
the potential of transformation-based software development. Figure 1 gives an overview of HATS.

HATS is an SDT-based transformation system that can be adapted to a problem domain in the following manner:

- specify the tokens of the target language,
- provide a grammar of the target language, and
- specify how target program strings should be formatted (e.g., indentation, carriage returns, etc.).

The results of these activities are represented by shaded boxes in Figure 1.

After HATS has been adapted to a problem domain, it can be used as an automatic transformation system by supplying it with (1) a specific target program as input, and (2) a transformation program that describes how the target program should be transformed.

As a first step in the execution of HATS, the transformation program is parsed. During this parse, HATS also checks that all transformations will only produce syntactically valid target programs. After this has been completed, the denotational semantics of the transformation language is used to execute the transformation program.

A transformation program will contain the following sections:

- an input expression that names a specific target program (i.e., a pathname to a file)
- a section where transform functions are defined (see Section 4.2 for a discussion of transform functions)
- a section where transformation sequences are defined (i.e., where transform functions are composed)
- a control section that defines how transformation sequences are applied to the target program.

When the execution of the transformation program encounters the "input target program" expression, a recursive-descent parser is invoked to parse the target program. Because of the desire to (effortlessly) apply to a wide range of target languages, this parser is extended with backtracking capabilities to enable it to resolve local ambiguities that may arise in the parse table. As long as the target grammar is not truly ambiguous, target programs will be able to be parsed. However, it should be noted that in the worst case (e.g., if many parse table locations have multiple entries) the parsing process will be time consuming.

![Figure 1: HATS](image)

### 4.1 Schemas

Given a target grammar, $G$, (SDT) schemas are defined as follows:

- Select a nonterminal symbol, $d$, belonging to $G$. The symbol $d$ will be the root of the SDT that HATS is constructing. In this context, we refer to $d$ as the dominating symbol of the schema.
- Construct a derivation of the form: $d \Rightarrow \alpha$. Note that $\alpha$ may contain nonterminal symbols.
- The expression $d[:\alpha[:]}$ is a schema describing an SDT whose root is $d$ and whose leaves are $\alpha$. Here the string “[:"]" serves as a begin-end

$^3$Exponential with respect to the number of productions in the target grammar.
marker. The difference between $\alpha$ and $\alpha'$ is as follows: Let $C$ denote an arbitrary nonterminal symbol in $\alpha$. The distinction between $\alpha$ and $\alpha'$ is that any nonterminal such as, $C$, in $\alpha$ will be represented by a schema-variable of the form $<C>_{i}$, where $i$ is a non-negative integer, in $\alpha'$. Note that subscripts are used to distinguish instances of schema-variables.

Through recognition of the begin-end markers and by making the assumption that the string $<C>$$_{i}$ will not be a legal token in the target language, HATS is able to parse $\alpha'$ to ensure that the derivation $d \Rightarrow \alpha$ is possible. This assures that schemas are syntactically correct. Furthermore, HATS also requires that tree substitutions made during the transformation process always replace (substitute) trees having the same dominating symbol. This results in transformations that, by definition, produce syntactically legal (though not necessarily semantically legal) programs.

### 4.2 Matching and Control in HATS

As was mentioned in Section 2, transformations have traditionally had the form of rewrite rules such as:

$$T = \text{def} \ schema_{1} \Rightarrow \text{schema}_{2}$$

in which the notion of a match is implicit in the transformation, and the notion of control is external to the transformation. For example, to what SDT's within the target program does one attempt to apply $T'$? Historically, this lack of explicit control was not an issue because rewrites were applied manually (e.g., mathematical expressions were simplified by hand). However, in the context of automatic transformation, such rewrite rules lack expressive power. When limited to basic rewrite rules it becomes difficult to express, within a transformation program, a refined application strategy.

We have addressed this problem in HATS by explicitly parameterizing rewrites with respect to the SDT that they are applied to. This is a distinguishing feature of HATS that we believe is extremely powerful. We call these parameterized transformations **transform functions** to distinguish them from the standard unparameterized transformations. Along with the notion of explicit parameterization arises the need for an explicit match operator. In HATS, the explicit match operator is denoted by the symbol: $| = |$. A basic match is then a boolean-valued expression of the form $e_{1} | = | e_{2}$ where $e_{1}$ and $e_{2}$ are schemas, or variables that are or can be bound to schemas. We have also found it useful to introduce a special universal SDT that denotes the empty match. In this paper we denote this SDT by $\phi$, and refer to it as the **null tree**. This SDT is unique (and universal) because it cannot match with anything else (including itself). An interesting property of $\phi$ is that because it cannot participate in a successful match it can be used to terminate a recursive transformation. Abstractly, $\phi$ is the intersection between the control domain and the SDT value domain.

Using the ideas and notation just described, the transformation given earlier would be expressed as the following transform function:

$$T = \text{def} (\lambda \text{sdtd sdtn} | = | \text{schema}_{1} \Rightarrow \text{schema}_{2})$$

Thus the pattern, the expression to the left of the $\Rightarrow$, of the transform function is a boolean expression, whose successful evaluation produces an environment (a generalized substitution list) which is then used to instantiate the replacement (i.e., the expression to the right of the $\Rightarrow$). When viewed from this perspective it becomes natural to consider further extending the pattern expression to include more general boolean expressions containing more general matches. Another extension is to support a Dijkstra-like guarded command construct.

One particularly powerful idea comes from realizing that (1) transform functions produce SDT's as outputs, and (2) match operations bind variables to SDT's. After seeing this connection, it is natural to consider applying transform functions to variables that are bound in match operations and then matching the resulting SDT to a particular schema. In this manner, very refined control can be expressed within a pattern expression. From here on out we will refer to patterns that contain this type of control as **control-patterns**.

With this capability, a transformation system achieves theorem prover-like characteristics. Transformation sequences can be viewed as focused search strategies, and control-patterns can be seen as proving lemmas and providing the transform function with the resulting information.

### 4.2.1 Example: Lambda Expansions

The example given here is a classic example that arises when restructuring functional expressions in the lambda calculus. Here the transformational objective
is to expand the scope of lambda expressions of the form:

\[(\lambda x.\text{expression\_body})\]

In particular, if a function expression contains an argument that is a lambda expression, we want to expand the scope of this lambda expression to contain the entire function expression. The following is a concrete example of such a transformation:

\[f(5, x, (\lambda y.\, g(y)), z)\]
\[==>|\]
\[(\lambda y.\, f(5, x, g(y), z))\]

We will refer to this transformation as a lambda-expansion. Note that in general, there are correctness issues concerning the uniqueness of the lambda bound variable with respect to the arguments in the function expression that the lambda operator is expanded over. However, in this paper we do not go into these issues. Instead, we focus our attention on what kind of matching capabilities are necessary to transformationally express a general lambda-expansion in an SDT framework.

In order to consider lambda-expansions, we need a grammar describing our target language. Without loss of generality, we define a limited target grammar that only describes function expressions.

\[
e\text{-list} ::= e\text{-list}, e \mid e
\]

\[
e ::= id (\ e\text{-list} ) \quad /*\text{a~function~expression}*/
\]
\[\mid (\lambda id.\ e) \quad /*\text{a~lambda~expression}*/
\]
\[\mid id
\]

Given this grammar, one possibility for expressing a lambda-expansion transformation is:

\[
T := \quad <id>1 (\ *1 (\lambda <id>2.\ <expr>1) *2) \\
==>| \\
(\lambda <id>2.\ <id>1 (\ *1 <expr>1 *2))
\]

where *1 and *2 are wildcard symbols. The intent is to have these wildcards match with arguments that precede and follow an argument matching "(\lambda <id>2.\ <expr>1)". However, there are problems with this approach. First, a fully general wildcard symbol can be very inefficient to implement. Second, it becomes difficult to limit or restrict the matching capabilities of a wildcard in such a way that they don’t include expressions within expressions. For example, in a lambda-expansion we might not want a wildcard symbol to match just a portion of an argument to the outermost function expression (i.e., we want the match to be in increments of whole argument expressions).

In general, in order to make wildcard symbols useful (and comprehensible) we need to restrict their scope (i.e., their matching capabilities). Two approaches come to mind: (1) one can create types, with respect to the target grammar, for wildcard symbols, or (2) one can develop a notation for restricting the scope of a wildcard symbol. Both of these approaches are viable when considering a fixed target grammar. However, in a language independent system these approaches can be quite awkward.

The control-pattern solution to defining a lambda-expansion consists of three transformations. The first transformation, \(T_1\), traverses an argument list (of a function expression) and returns the smallest SDT having e_list as its dominating symbol (i.e., its root) and containing an argument that is a lambda expression. If no argument in the list is a lambda expression, then \(\phi\) is returned. The traversal described is accomplished by a guarded command construct where control-patterns are guards, replacements are the actions that correspond to guards, and guards are separated from one another by the vertical bar symbol: |.

Note that this transformation is not correctness preserving when considered in isolation.

The second transformation, \(T_2\), traverses the argument list (i.e., the e_list) of the function expression and “erases” the lambda binding of the first lambda expression argument that it encounters. All that remains of the lambda expression after the lambda binding has been erased is the expression_body that is contained within the lambda expression. If no argument is a lambda expression, then \(\phi\) is returned.

The third transformation, lambda_expansion, does the following:

- Require the match:
  \[t ::= e[\ ] <id>1 ( \ <e\text{-list}>1 ) [\ ] . \] This match ensures that the input tree \(t\) is a function expression. If this match fails the transformation does not apply.

- Require the match:
  \[e\text{-list}[\ ] (\lambda <id>2.\ <e>2) [\ ] \models T_1 (<e\text{-list}>1)\.] Note that if this match fails, then the argument list does not contain a lambda expression.

- Require the match:
  \[<e\text{-list}>3 \models T_2 (<e\text{-list}>1).\] If the previously stated match with \(T_1(<e\text{-list}>1)\) succeeds, then this match will be with an argument list that has the first lambda-binding removed.

- Construct the replacement so that the lambda binding precedes the function expression.
\( \mathcal{T}_1 := \lambda t. \)

Base Case Ia: The (remaining) argument is a lambda expression

\[
t \models e_{\mathit{list}[]} (\lambda <\mathit{id}>_1. <e>_1][] \implies e_{\mathit{list}[]} <e>_1[]
\]

Base Case Ib: The (remaining) argument is not a lambda expression

\[
t \models e_{\mathit{list}[]} <e>_1[] \implies \phi
\]

Base Case II: A lambda expression has been found

\[
t \models e_{\mathit{list}[]} <e>_1[], (\lambda <\mathit{id}>_1. <e>_1)[] \implies e_{\mathit{list}[]} <e>_1[]
\]

General Case: Traverse \( e_{\mathit{list}[]} \)

\[
t \models e_{\mathit{list}[]} <e>_1[] \implies \phi
\]

Base Case Ia: The (remaining) argument is a lambda expression

\[
t \models e_{\mathit{list}[]} (\lambda <\mathit{id}>_1. <e>_1][] \implies e_{\mathit{list}[]} <e>_1[]
\]

Base Case Ib: The (remaining) argument is not a lambda expression

\[
t \models e_{\mathit{list}[]} <e>_1[] \implies \phi
\]

Base Case II: A lambda expression has been found

\[
t \models e_{\mathit{list}[]} <e>_1[], (\lambda <\mathit{id}>_1. <e>_1)[] \implies e_{\mathit{list}[]} <e>_1[]
\]

General Case: Traverse \( e_{\mathit{list}[]} \)

\[
t \models e_{\mathit{list}[]} <e>_1[] \implies \phi
\]

\[\lambda \text{e}\_\text{list} := \lambda t.\]

\[\mathcal{T}_2 := \lambda t.\]

Base Case Ia: The (remaining) argument is a lambda expression

\[
t \models e_{\mathit{list}[]} (\lambda <\mathit{id}>_1. <e>_1][] \implies e_{\mathit{list}[]} <e>_1[]
\]

Base Case Ib: The (remaining) argument is not a lambda expression

\[
t \models e_{\mathit{list}[]} <e>_1[] \implies \phi
\]

5 Conclusion

Because of the potential for correctly manipulating programs via small changes, an automated non-procedural transformation system can be a valuable tool for constructing high assurance software. Effective automating of the transformation process requires an expressive control structure. In HATS, this is realized largely through control-patterns. From a practical perspective, the transformation process should begin in the domain language in which a problem is initially expressed. Thus in order for a transformation system to be practical it must be flexible with respect to domain languages. HATS has been designed to facilitate transformation in this framework.

References


