

Program Transformation Using HATS 1.84

Victor Winter and Jason Beranek

Department of Computer Science, University of Nebraska at Omaha

Abstract. This article gives an overview of a transformation system called HATS – a freely available platform independent IDE facilitating experimentation in transformation-oriented software development. Examples are discussed highlighting how the transformational abstractions provided by HATS can be used to solve various problems.

1 Introduction

Interest in program transformation is driven by the idea that, through their repeated application, a set of simple rewrite rules can affect a major change in a software artifact. From the perspective of dependability, the explicit nature of transformation exposes the software development process to various forms of analysis that would otherwise not be possible.

Within the scope of this article we will use the term *program transformation* (or *transformation*) in a general sense to refer to software manipulation processes that are restricted to the fully automatic application of rewrite rules. We will also predominantly refer to the objects that are the subject of transformation as *terms* or *trees* rather than specifications, programs, code fragments, or the variety of other artifacts over which transformation is possible.

This article gives an overview of a transformation system called HATS. HATS is freely available and provides a platform independent IDE facilitating experimentation in transformation-oriented software development [24, 23]. HATS, an acronym for High Assurance Transformation System, is a program transformation system whose continuing development is inspired by the potential benefits that *transformation-oriented programming* can offer software assurance efforts. As a result, the primitives and abstractions in HATS have been designed with thought given to verification. From a more technical perspective, HATS can be viewed as a higher-order transformation system for manipulating parse trees. Tree structures are defined using an extended-BNF notation supporting precedence and associativity. Transformations are written in TL, a language that supports first-order as well as higher-order transformation. TL also supports standard one-layer traversal constructs together with recursive definition of traversals allowing a variety of generic traversals, both first-order as well as higher-order, to be defined by the user. HATS supports several feedback mechanisms including pretty-printed text, graphical display of parse trees, and a rudimentary trace facility¹ for debugging.

¹ The trace facility is presently under development and is in a somewhat experimental stage.

The rest of this article is structured as follows: Section 2 gives an overview of HATS. Section 3 gives an overview of the HATS parser generator. Section 4 gives an overview of the transformation language TL. Section 5 presents an example showing how three algebraic laws forming part of a Verilog synthesis system can be effectively implemented in TL. Section 6 presents an example showing how the *hide* combinator has been effectively used to realize a transformational step in a class loader for an instance of the JVM. Section 7 discusses related work, and section 8 concludes.

2 An Overview of HATS (Version 1.84)

HATS is a transformation system whose development has been underway for a number of years. During this time, the design of HATS has gone through a number of changes. The earliest version of HATS [25] was influenced primarily by the TAMPR transformation system [5, 4]. In this early version of HATS, transformation was accomplished by applying collections of first-order rewrite rules to terms using a small library of traversals provided by the HATS system. Later versions of HATS drew inspiration from systems like ELAN [3], Stratego [21], ASF+SDF [2], and Strafunski [13].

In its present form, HATS (version 1.84) is a system in which transformation is realized through the execution of special purpose programs written in a language called TL [24]. The language TL provides a computational framework where transformational ideas are expressed in terms of conditional rewrite rules whose application is controlled by a variety of standard strategic combinators and traversals [12] (e.g., sequential composition, biased choice, top-down and bottom-up traversals, etc.). Distinguishing features of TL include: (1) the ability to express transformational ideas in terms of higher-order conditional rewrite rules, (2) the ability to control rule application through a variety of unique combinators including the *transient* combinator and the *hide* combinator, and (3) a semantic foundation where the failure of rule application behaves like the identity transformation (in contrast to strategic systems in which rule failure yields the strategic constant FAIL).

2.1 The Taxonomy of a Domain.

In HATS, the term *domain* is used to refer to the collection of files that support the transformation of parse trees belonging to a given language. Example domains can be downloaded from the HATS homepage [9]. Figure 1 shows an example of what a user would see in a typical HATS session. The pane on the upper-left partitions the files in the domain into a variety of file types: (1) grammar files, (2) lexer files, (3) target files, (4) transformation files, (5) user-defined function files, (6) style files, (7) pretty-printed text files, and (8) parse tree files. The pane on the upper-right serves a variety of purposes including (1) a special purpose text editor, (2) a graphical tree display, and (3) an execution trace display. The pane on the bottom provides error feedback as well as a variety of execution metrics.

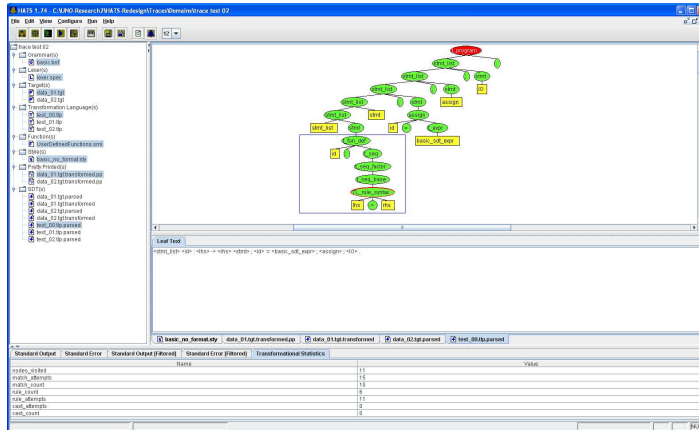



Fig. 1. Using the HATS GUI to View a Domain

2.2 Running HATS

Perhaps the most natural way to think of HATS is in terms of a collection of functions whose execution can be orchestrated from the HATS GUI. Functions that are of central importance to transformation include: (1) a parser generator, (2) a transformation engine, (3) a pretty printer, (4) a tree viewer, and (5) a trace viewer. Figure 2 gives an overview of HATS from the perspective of dataflow. In the figure, icons² are used to denote various functions that are available to the user, source nodes represent user input files, and sink nodes represent system feedback.

In the discussion that follows, we assume a domain whose grammar, lexer, source, transformation program, and style files are respectively named `grammar.bnf`, `lexer.spec`, `source.tgt`, `transform.tlp`, `UserDefinedFunctions.sml`, and `style.sty`.

In HATS, the transformation process is decomposed into the following phases:

1. **Parsing Phase** – The goal of this phase is to produce suitable input files for the transformation phase.
 - (a) The *parser generator*  is invoked to produce a *source parser* for the language defined by `grammar.bnf` and `lexer.spec`. The *source parser* is also used by the *TL parser*, the parser for transformation programs, when parsing the portions of rewrite rules containing code fragments belonging to the source language. Thus, invoking the parser generator will in fact produce two parsers – a parser for the source language and a parser for the transformation language.

² The icons used in the figure are taken directly from the icons that are used by the HATS GUI.

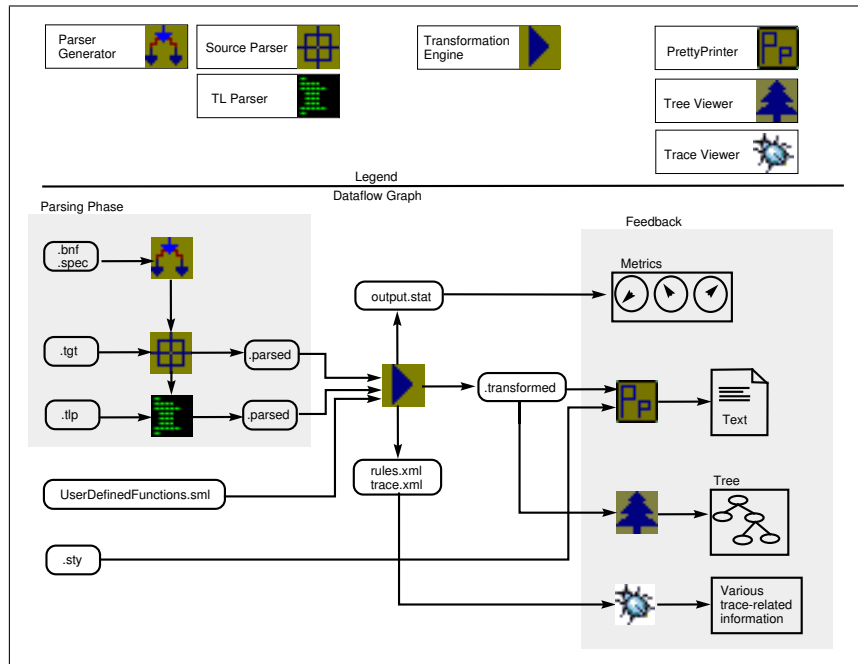




Fig. 2. The Architecture of HATS from a Dataflow Perspective

(b) Creation of `.parsed` files


- i. The *source parser*  can be invoked to parse the file `source.tgt`.

Any errors that arise during parsing will be displayed in the bottom pane of the HATS GUI under the **Standard Error** tab. A successful parse of `source.tgt` will result in the creation of the file `source.tgt.parsed`.

- ii. The *TL parser*  can be invoked to parse the file `transform.tlp`.

Any errors that arise during parsing will be displayed in the bottom pane of the HATS GUI under the **Standard Error** tab. A successful parse of `transform.tlp` will result in the creation of the file `transform.tlp.parsed`.

2. **Transformation Phase** – After completion of the parsing phase transformation

is accomplished by invoking the transformation engine .


The transformation engine accepts as input the files `source.tgt.parsed`, `transform.tlp.parsed`, and `UserDefinedFunctions.sml`. Invocation will cause `transform.tlp.parsed` to be applied to `source.tgt.parsed` using the additional functionality defined in `UserDefinedFunctions.sml`. Runtime errors resulting from semantically ill-formed transformation programs


are displayed in the bottom pane of the HATS GUI. A successful transformation of `source.tgt` will produce the files: `output.stat`, `rules.xml`, `trace.xml`, and `source.tgt.parsed.transformed`. We would like to point out that transformation in HATS is accomplished in an interpretive fashion. Specifically, a denotational semantics for TL has been implemented in ML, and it is this semantics that is used to interpret the `transform.tlp.parsed` tree.

3. **Feedback Phase** – In the feedback phase, various kinds of information can be displayed relating to the computation that has been performed in the transformation phase.

(a) Invoking the *pretty-printer*  will produce the file

`source.tgt.transformed.pp`. The contents of this file is formatted text and can be displayed in the upper-right pane of the GUI. The pretty-printer for HATS is essentially an abstract pretty-printer in which formatting is essentially inserted into BNF-style grammar productions and some control over formatting rule selection is provided. Other transformation systems also provide this kind of functionality. For example, the XT bundle includes a generic pretty-printing tool (GPP) in which a language called Box is used to describe the intended layout of text [8].

(b) Invoking the tree viewer  will display the parse tree of the selected file (e.g., `source.tgt.parsed.transformed`) in graphical form in the upper-right pane of the GUI. Portions of the parse tree can be collapsed, expanded, and selected. Simultaneously, the leaves of the expanded portion of the tree are displayed in textual form with highlighted text denoting selected portions of the tree.

(c) Invoking the *trace viewer*  enables the user to step through the execution of transformations that have been designated to be traced. The designation of which rules to trace can be specified within a TL program.

3 The HATS Parser Generator

HATS provides users with a GLR-style parser generator. From a technical standpoint, the HATS parser generator is not a true GLR parser because it cannot detect (and handle) nonterminating derivation sequences. Section 7.1 gives a more detailed discussion of parsing technology and its role in support of transformation.

The HATS parser generator allows users to describe context-free grammars using an extended-BNF notation. The extended-BNF notation supported can be thought of as a merging of BNF notation with regular expression notation. Figure 3 gives an overview of the meta-symbols supported by the HATS parser generator.

| Symbol | Description |
|-------------|--|
| ::= | The nonterminal definition operator. |
| . | The production terminator. |
| <id> | Nonterminal symbols should be enclosed in pointy-brackets. |
| “token” | Tokens should be decorated with (i.e., enclosed in) double quotes. |
| domain_vars | Terminals denoting domain variables (e.g., id) are not decorated. |
| | The alternate choice operator. |
| [] | Portions of a production enclosed in square brackets are optional. |
| () | The constant epsilon. |
| () | Used only for grouping just like in regular expression notation. |
| ()* | The Kleene-closure of the portion enclosed in parens. Note that this will create trees of varying degree and is not particularly useful when using first-order matching. |
| { } | Equivalent to (...)*. |
| (* *) | Comments are similar to ML and may span multiple lines. However, comments may not be nested. |

Fig. 3. Meta-symbols of the extended-BNF syntax supported by the HATS parser

3.1 Precedence and Associativity

The HATS parser generator also supports precedence and associativity rules as a mechanism for disambiguating two or more productions. The *dangling else*³ problem is a classic example of a situation that is typically resolved using precedence rules.

There are two types of associativity: LEFT_ASSOC and RIGHT_ASSOC. LEFT_ASSOC indicates that the operator is left-associative. RIGHT_ASSOC indicates that the operator is right-associative. A BNF grammar can begin with zero or more associativity rules. An associativity rule is of the form:

$$\%assoc_type\ quoted_token_list. \quad (1)$$

where *assoc_type* is one of the associativity types mentioned in the previous paragraph, and *quoted_token_list* is a list of one or more tokens separated by blanks. In this context, a token is a string enclosed in quotes. Within an associativity rule all tokens have the same precedence.

Given two associative rule declarations r1 and r2, if r1 lexically occurs before r2, then the tokens in r1 will have a lower precedence than the tokens in r2. All tokens within a given rule have the same precedence.

```
%LEFT_ASSOC "+" "-" "L1" . Lower precedence
%LEFT_ASSOC "*" "/" "L2" . Higher precedence
```

Each grammar production can have an optional precedence attribute as a suffix. The general form is as follows:

³ The *dangling else* problem concerns itself with determining with which *if* construct an *else* fragment should be associated.

$$\text{nonterm} ::= \text{alpha}[\%PRECtoken]. \quad (2)$$

where *token* belongs to the *quoted_token_list* of some associative rule declaration. For a detailed example of how to construct grammars having associativity and precedence rules download the type checking demo domain from the HATS homepage [9].

4 TL: The Basics

This section gives a brief overview of TL, a labelled conditional (higher-order) rewriting language supporting a variety of strategic operators and generic traversals. For a more detailed discussion of TL see [24]. In TL, parse trees are the “objects” that TL programs transform. Rewrite rules have the following form:

$$r : lhs \rightarrow s^n \text{ [if condition]} \quad (3)$$

In this example, r denotes the label of the rule, lhs denotes a *pattern* describing a tree structure, s^n denotes a *strategic expression* whose evaluation yields a strategy of order n , and *if condition* denotes an optional Boolean-valued condition consisting of one or more *match expressions* constructed using Boolean connectives.

A *pattern* is a notation for describing the parse tree structures that are being manipulated. This notation includes typed *variables* that are quantified over specific tree structure domains, e.g., $stmt[id_1 = 5]$ is a tree with root $stmt$ and leaves id_1 , $=$, and 5 . In this context, the subscripted variable id_1 denotes a typed variable quantified over the domain of all trees having id as their root node. In general, a pattern of the form $A[\alpha']$ is structurally valid if and only if the derivation $A \stackrel{\pm}{\Rightarrow} \alpha$ is possible according to the grammar and α' is obtained from α by subscripting all nonterminals occurring in α .

A *strategic expression* is an expression whose evaluation yields a *strategy* having a particular order. In the framework of TL, a *pattern* is considered to be a strategy of order 0. A rewrite rule that transforms its input tree into another tree is considered to be a strategy of order 1 (i.e., a first-order rule). Let s^1 denote a first-order strategy. Then the rule $lhs \rightarrow s^1$ denotes a second-order strategy (e.g., s^2), and so on.

A *match expression* is a first-order match between two patterns. Let t_1 denote a pattern, possibly non-ground, and let t_2 denote a ground pattern. The expression $t_1 \ll t_2$ denotes a match expression and evaluates to *true* if and only if a substitution σ can be constructed so that $\sigma(t_1) = t_2$. One or more match expressions can be combined using the Boolean connectives $\{ \text{and, or, not} \}$ to form the *condition* of a rewrite rule.

4.1 Combinators

In TL, a variety of *combinators* can be used to compose conditional rewrite rules into strategies. First-order strategies define controlled sequences of rewrites and

can be applied to tree structures to produce other tree structures. Thus, a first-order strategy can be viewed as a function that rewrites or *transforms* one tree into another. Because of the important role played by strategies, transformation languages of this kind are also referred to as *strategic programming languages*. In this article we will use the terms *strategy* and *transformation* interchangeably. Figure 4 gives an overview of some of the combinator primitives provided by TL.

There are a number of combinators that have been identified as being generally useful in strategic programming [12]. Two widely used combinators are: (1) left-to-right sequential composition ($<;$), and (2) left-biased conditional composition ($<+$). Let s_1 and s_2 denote two strategies. The expression $s_1 <; s_2$ denotes the left-to-right sequential composition of s_1 and s_2 . When applied to a tree t , this strategy will first apply s_1 to t and then apply s_2 to the result. In contrast, the expression $s_1 <+ s_2$ denotes the left-biased conditional composition of s_1 and s_2 . When applied to a tree t , the application of s_1 to t is attempted, and if that succeeds, the result is returned; otherwise, the result of the application of s_2 to t is returned. In TL, if neither s_1 or s_2 apply then t is returned unchanged.

| | | |
|------------------|--|---------------------------------|
| $s_1^n <; s_2^n$ | Left-to-right sequential composition. | Precedence Associativity |
| $s_1^n ;> s_2^n$ | Right-to-left sequential composition. | |
| $s_1^n <+ s_2^n$ | Left-biased conditional composition. | |
| $s_1^n +> s_2^n$ | Right-biased conditional composition. | |
| $transient(s^n)$ | A unary combinator restricting the application of s^n . | |
| $hide(s^n)$ | A unary combinator that hides the application of s^n from an enclosing conditional composition combinator. | |
| | | (lowest) $+>$ right |
| | | $<+$ left |
| | | $;>$ right |
| | | (highest) $<$ left |

Fig. 4. The basic combinators of TL

The *transient* Combinator. The transient combinator is a very special combinator in TL. This combinator restricts a strategy so that it may be applied *at most once*. The “at most once” property is the hallmark of the *transient* combinator.

Transients open the door to *self-modifying* strategies. When using a traversal to apply a self-modifying strategy to a term, a different strategy may be applied to every term encountered during a traversal. For example, let $int_1 \rightarrow int[2]$ denote a rule that rewrites an arbitrary integer to the value 2. If such a rule is applied to a term in a top-down fashion all of the integers in the term will be rewritten to 2. Now consider the following self-modifying transient strategy:

$$\begin{aligned} &transient(int_1 \rightarrow int[1]) <+ \\ &transient(int_1 \rightarrow int[2]) <+ \\ &transient(int_1 \rightarrow int[3]) \end{aligned}$$

When applied to a term in a top-down fashion, this strategy will rewrite the first integer encountered to 1, the second integer encountered to 2, and the third integer encountered to 3. All other integers will remain unchanged.

The *hide* Combinator. The notion of choosing the application of one rule over another is central to strategic programming. An essential component of both the left-biased and right-biased conditional composition combinators is the ability to “observe” the behavior of strategy application (i.e., whether the application of a strategy to a term has succeeded or failed). Let us consider the introduction of a unary combinator called *hide* into a strategic framework supporting left-biased and right-biased conditional composition combinators. In this context, the purpose of *hide* is to prevent the application of a strategy from being observed. As a consequence a strategy of the form $hide(s_1) <+ s_2$ will always attempt to apply s_1 followed by s_2 , in effect undoing the discriminatory nature of the conditional composition combinator.

The strategic combinator *hide* provides an interesting extension to the framework of TL. This unary combinator restricts the observability of strategy application from the perspective of the conditional composition combinators. In particular, the *hide* combinator satisfies the following properties:

$$\begin{aligned} hide(s_1) <+ s_2 &\equiv s_1 <; s_2 \\ s_1 +> hide(s_2) &\equiv s_1 ;> s_2 \end{aligned}$$

At first glance, it appears as if the *hide* combinator does not add anything new to the standard combinator set. However, section 6 gives an example showing how *hide* can be effectively utilized. We would like to point out that, although it is not shown in the example given in Section 6, the *hide* combinator is also very useful in conjunction with the *transient* combinator and higher-order strategies.

4.2 Traversals

Combinators are a control mechanism that define *how* a collection of rules should be applied to a given tree. They do not define *where* (i.e., to which trees) a collection of rules should be applied. This dimension of transformation can be defined by *traversal* mechanisms.

A *generic traversal* can be thought of as a curried function parameterized on a strategy s and a tree t . As the name suggests, a generic traversal will *traverse* its input tree structure t and apply its input strategy s at one or more points along the traversal. Two common traversals are a top-down left-to-right traversal which TL denotes by the symbol TDL, and a bottom-up left-to-right traversal which TL denotes by the symbol BUL. From a computational perspective, a TDL traversal can be understood as corresponding roughly to non-strict (outside-in) evaluation while the traversal BUL corresponds roughly to strict (inside-out) evaluation.

The *generic tree traversal* is an abstraction that is virtually essential in order for transformational ideas to scale to real-world problems (i.e., the transformation of trees whose structures are derived from large grammars). TL provides a

rich mechanism (not discussed in this paper) for defining generic and pseudo-generic tree traversals. TL also supports the definition and use of higher-order generic traversals. Informally, one can think of a higher-order traversal as mechanism for dynamically collecting a number of strategies and combining them to form a new strategy. A common higher-order traversal is one that traverses a tree in a BUL fashion, applies a higher-order strategy s^{n+1} , and composes the resulting order- n strategies using the $\lt;$ combinator. In TL, this traversal is denoted by the identifier *lseq_bul*. In contrast, *rcond_tdl* denotes a higher-order generic traversal that traverses a tree in a *TDL* fashion, applies a higher-order strategy s^{n+1} and composes the resulting order- n strategies using the $\gt;$ combinator.

5 Example I: A Verilog Synthesis Fragment

In this example, we look at how transformation can be used to realize a fragment of a logic synthesis system for a hardware description language called Verilog, whose syntax has a C-like flavor. We will look at how the three algebraic laws shown in Figure 5 can be effectively implemented in TL. These algebraic laws were initially presented in [10] and have been used by Iyoda et al [11] in the abstract design of a synthesis system for a small subset of Verilog.

| | |
|-----------------------------|---|
| Law 1 (completion). | $(x, y, \dots := e, f, \dots) = (x, y, \dots, z := e, f, \dots, z)$ |
| Law 2 (reordering). | $(\dots y, z \dots := \dots f, g \dots) = (\dots z, y \dots := \dots g, f \dots)$ |
| Law 3 (propagation). | $(\vec{v} := g; \vec{v} := h(\vec{v})) = (\vec{v} := h(g))$ |

Fig. 5. Three Laws of Synthesis

An important thing to note about Laws 2 and 3 is that they cannot be directly implemented as transformations. Law 2 requires that ellipsis be matched so that the **number of variables** matched by ellipsis on the left-hand side of an assignment is equal to the **number of expressions** matched by the corresponding ellipsis on the right-hand side of the assignment. Such matching constraints lie beyond traditional associative matching. Law 3 describes constant propagation over sequentially composed assignment statements. Though its intention is clear, in the form stated, Law 3 does not directly map onto the syntax of parallel assignment statements since the right-hand side of a parallel assignment is a comma-separated list of expressions and not a function (e.g., h) as is suggested in Law 3. In spite of these issues, we will show that in the higher-order framework of TL it is possible to express the intent of Laws 2 and 3 using just two rewrite rules!

The approach to logic synthesis described here consists of passing Verilog *modules*, which are roughly the equivalent of C functions, through two canonical

forms. The first canonical form is achieved when all assignments in a module are *total*. Let m denote a Verilog module and let \vec{v} denote a vector consisting of all variables that occur on the left-hand side of an assignment in m . An assignment is *total* if it well-formed and has the form: $\vec{v} = \vec{e}$. A Verilog module is said to be in *Total Form* if all its assignment statements are total.

The second canonical form consists of propagating the bindings resulting from assignments over assignment sequences. This transformation is justified by Law 2 and Law 3. The following example shows how a simple sequence of Verilog assignments can be transformed into parallel normal form.

| Canonical Forms | | | | |
|--|--------------|--|---------------------|-----------------------------|
| <u>Input</u> | \implies | <u>Total Form</u> | \implies | <u>Parallel Normal Form</u> |
| $x = e_1;$ $y = e_2;$ $z = e_3;$ | <i>Law 1</i> | $x,y,z = e_1,y,z;$ $y,x,z = e_2,x,z;$ $z,x,y = e_3,x,y;$ | <i>Laws 2&3</i> | $x,y,z = e_1, e_2, e_3$ |

5.1 Implementation

The rules and strategies needed to place the modules in a Verilog program into parallel normal form (PNF) are shown in Figure 7 and the relevant portion of the Verilog syntax is shown in Figure 6. The strategy that achieves the transformation to parallel normal form is labelled PNF and is defined as follows:

$$\text{PNF: BUL}\{\text{Prep}\} \prec; \text{BUL}\{\text{Law1}\} \prec; \text{BUL}\{\text{Law3}\} \quad (4)$$

When applied to a Verilog program p_{in} the strategy PNF will first traverse p_{in} using a BUL traversal and apply the strategy *Prep*. *Prep* rewrites blocking assignments to fork/join blocks, which is what we use to model parallel assignments. For the remainder of this discussion, we will refer to such fork/join blocks as *multi-assignments* to emphasize how they are being used in this context.

Law1 makes all multi-assignments in a module total. Thus, the result of the traversal $\text{BUL}\{\text{Law1}\}$ is a program p_{total} that is in Total Form. Next p_{total} is traversed, again using a BUL traversal, and the strategy *Law3* is applied. The result is a program p_{pnf} that is in parallel normal form.

In *Law1*, the strategic expression $lseq_bul\{make_total\}[modulee_0]$ will traverse the object tree $modulee_0$, apply the higher-order rule *make_total* to all subtrees and compose the results using the $\prec;$ combinator. The rule *make_total* is a higher-order rule that produces an instance of the strategy $transient(check[id_1] \prec+ add[id_1])$ for each (blocking) assignment encountered in a given module. What distinguishes one instance from another is the value bound to id_1 . For example, if id_1 is instantiated with the variable x then the following strategy is created:

```

transient (
  stmtS[ [ x = E2; stmtS3 ] ] → stmtS[ [ x = E2; stmtS3 ] ]
  <+
  stmtS[ [ ] ] → stmtS[ [ x = x; ] ]
)

```

| | |
|---------------------|---|
| modulee | ::= module module_id “;” module_item_0orMore endmodule |
| module_item_0orMore | ::= module_item module_item_0orMore () |
| module_item | ::= continuous_assign always_stmt ... |
| continuous_assign | ::= “assign” lvalue “=” E “;” |
| always_stmt | ::= “always” stmt |
| stmtS | ::= stmt stmtS () |
| stmt_or_null | ::= stmt “;” |
| stmt | ::= blocking_assignment “;” seq_block par_block ... |
| seq_block | ::= “begin” stmtS “end” |
| par_block | ::= “fork” stmtS “join” |
| blocking_assignment | ::= lvalue “=” E |
| ... | |

Fig. 6. A wide-spectrum language fragment containing a small subset of Verilog

| | |
|--------------|--|
| PNF: | $BUL\{Prep\} <; BUL\{Law1\} <; BUL\{Law3\}$ |
| Prep: | $stmt\llbracket blocking_assignment_1; \rrbracket \rightarrow stmt\llbracket fork\ blocking_assignment_1; join \rrbracket$ |
| Law1: | $modulee_0 \rightarrow Special_TD\{ lseq_bul\{ make_total \}[modulee_0]\}(modulee_0)$ |
| | $make_total: blocking_assignment\llbracket id_1 = E_1 \rrbracket \rightarrow transient(check[id_1] <+ add[id_1])$ $check: id_1 \rightarrow stmtS\llbracket id_1 = E_2; stmtS_3 \rrbracket \rightarrow stmtS\llbracket id_1 = E_2; stmtS_3 \rrbracket$ $add: id_1 \rightarrow stmtS\llbracket \rrbracket \rightarrow stmtS\llbracket id_1 = id_1; \rrbracket$ |
| Law3: | $stmtS\llbracket par_block_1\ par_block_2 \rrbracket$ \rightarrow $BUL\{lseq_tdl\{propagate\}[par_block_1]\}(stmtS\llbracket par_block_2 \rrbracket)$ |
| | $propagate: blocking_assignment\llbracket id_1 = expr_1 \rrbracket \rightarrow expr\llbracket id_1 \rrbracket \rightarrow expr_1$ |

Fig. 7. Transformations implementing Parallel Normal Form(PNF)

The semantics of an instance of the strategy $transient(check[id_1] <+ add[id_1])$ can be best understood in the context of its application (via a traversal) to the sequence of statements contained within a multi-assignment. When applied to the assignments in a multi-assignment, the rewrite rule derived from $check[id_1]$ will apply if id_1 occurs on the left-hand side of an assignment within the multi-assignment block. If this happens, the rule will fire and the *transient* combinator will remove the instance of $check[id_1] <+ add[id_1]$. However, if $check[id_1]$ never applies, the traversal will proceed to the end of the multi-assignment at which point $add[id_1]$ will cause the identity assignment $id_1 = id_1$ to be added. This application will again cause the *transient* combinator to remove the instance of $check[id_1] <+ add[id_1]$. Summarizing then, an instance of $transient(check[id_1] <+ add[id_1])$ will leave a multi-assignment unchanged if the multi-assignment

contains an assignment to id_1 ; otherwise $transient(check[id_1] \leftarrow add[id_1])$ will add the assignment $id_1 = id_1$ to the multi-assignment.

In this first transformational phase, what remains is to process each multi-assignment statement within a module in the manner just described. This can be accomplished with the help of a special purpose traversal called `Special_TD` that has been written especially for this problem domain. What makes the traversal special is that it enables each multi-assignment in a module to receive its own individual copy of the strategy resulting from the evaluation of $lseq_bul\{make_total\}[modulee_0]$. This is important because, $transient(check[id_1] \leftarrow add[id_1])$ can only be applied once. However, we want it to be applied once to **every** multi-assignment occurring in a module. The `Special_TD` traversal accomplishes this and, though not shown, can be defined in one line of code using standard TL primitives.

Law3 takes two multi-assignment statements, par_block_1 and par_block_2 , and propagates the assignments from the first multi-assignment to the second. This is accomplished with the help of the higher-order rewrite rule *propagate*. When applied to a blocking assignment of the form $id_1 = expr_1$, *propagate* will produce the first-order rule $expr[id_1] \rightarrow expr_1$ that rewrites occurrences of id_1 in expressions to $expr_1$. The strategic expression $lseq_bul\{propagate\}[par_block_1]$ creates and collects propagating rules for all assignments in par_block_1 which are then applied to par_block_2 using the traversal `BUL`. We would like to conclude this example with the following remarks: (1) the fact that *Law3* is applied in a bottom-up fashion in *PNF* assures that all sequences of multi-assignments will be collapsed into a single multi-assignment statement, and (2) *Law 2* is never needed to transform a Verilog program into parallel normal form.

6 Example II: A Java Class Loader Fragment

In this section, we take a look at how the *hide* combinator can be effectively used to solve a problem encountered in Java class loading. In particular, we will take a closer look at class loading as it relates to the Sandia Secure Processor (SSP) [22]. The SSP is a hardware implementation of a significant subset of the Java Virtual Machine whose application domain extends to embedded high consequence systems. Class loading for the SSP is performed statically (prior to runtime) and preserves the conceptual structure of class files. These properties make the SSP class loading problem well-suited to a transformation-based solution.

To date, several full class loader designs have been implemented for the Sandia Secure Processor (SSP) [26] using the HATS system. In the discussion that follows, we join the transformation process of one such design at a time when a significant portion of the class loading transformation has already been completed. In particular, we assume that symbolic resolution has taken place, and the list of Java class files comprising the application has been partially ordered by subtype as follows: Let $c_1 c_2 \dots c_n$ denote the partially ordered list of class

files. Let \prec denote the subtype relation on classes, and let \Rightarrow denote logical implication⁴. The partial order of class files satisfies the following property:

$$c_j \prec c_i \Rightarrow i < j \quad (5)$$

This implies that, when inspecting the class list from left-to-right (e.g., via a top-down left-to-right traversal), a parent class will always be encountered before any of its children.

The transformational goal at this stage is to further resolve symbolic references to fields so that all symbolic field references satisfy the following property.

Static Binding Property: *The class component in a symbolic field reference corresponds to the class in which the field has been declared.*

The reason why the static binding property is not universally true already is that Java makes one exception to its essentially static binding of fields. This exception arises when a reference is made to an *inherited* field. For example, suppose we have situation where (1) an integer field x is declared in class A , (2) the class B *extends* A , i.e., $B \prec A$, (3) myB is an instance of B , and (4) C is the class containing a reference to $myB.x$. In this case, the Java compiler will produce a class file for C whose constant pool will have a *constant_fieldref_info* entry corresponding to the reference $myB.x$. Standard symbolic resolution of this entry will yield:

$$constant_fieldref_info[[B\ x\ I]] \quad (6)$$

This term represents a symbolic reference to the integer variable x belonging to the class B (even though x is not declared in B) which violates the static binding property. In this example, we would like the symbolic resolution of the constant pool entry corresponding to $myB.x$ to yield:

$$constant_fieldref_info[[A\ x\ I]] \quad (7)$$

For a typical JVM, this extended-resolution step is performed at runtime by a dynamic search up the inheritance chain. However, the operating assumptions of the SSP enables this runtime search to be avoided by an extension to the symbolic resolution algorithm. The strategy shown in Figure 8 implements this extended-resolution step and produces a class file list in which all symbolic references to fields satisfy the static binding property.

When applied to a Java application hierarchy (app_0) the strategy x_res first evaluates the strategic expression $rcond_tdl\{sbind\}[app_0]$. This evaluation results in a first-order strategy that is then applied to app_0 , using the traversal TDL , to achieve the static binding property. The workhorse of the extended-resolution transformation is the second-order strategy $sbind$ that, for every class it is applied to, will output a strategy that abstractly has the form:

$$hide(lift-class-associated-with-field) +> check-for-declaration \quad (8)$$

⁴ We do not wish to overload the \rightarrow symbol, whose use we reserve for rewriting.

```

x.res : app0 → TDL{ rcond_tdl{sbind}[app0] }(app0)

sbind: classfile[ cp1 classthis classsuper fields1 mt1 methods1 ]
  →
  ( hide(lift[classthis][classsuper]) +> rcond_tdl{collect_decs[classthis]}[fields1] )

lift: classthis →
  classsuper →

  constant_fieldref_info[ classthis name1 descriptor1 ]
  →
  constant_fieldref_info[ classsuper name1 descriptor1 ]

collect_decs: classthis →
  field_info[ access_flags1 name1 descriptor1 ] →

  constant_fieldref_info[ classthis name1 descriptor1 ]
  →
  constant_fieldref_info[ classthis name1 descriptor1 ]

```

Fig. 8. An extended-resolution step ensuring the static binding property

Note that the control idea in the above strategy is somewhat similar to the *transient-check-add* strategy used in the Verilog synthesis example.

Within $x.res$, the evaluation of the strategic expression $rcond_tdl\{sbind\}[app_0]$ will visit classes in the partially ordered class list in a top-down left-to-right (tdl) fashion, apply the *sbind* strategy to each class encountered, and compose the resulting strategy instances using the $+>$ combinator. The top-down left-to-right nature of the *rcond_tdl* traversal assures that in the resulting strategy, superclass strategies will be placed to the left of subclass strategies. The $+>$ combinator used by *rcond_tdl* will compose (sub)strategies in such a fashion that an overall strategy will be created whose application will proceed from right-to-left, meaning that strategies associated with subclasses will be applied before the strategies of their corresponding superclasses.

In the strategy produced by *sbind*, the portion of the strategy abstractly denoted in (8) by *check-for-declaration* results from the evaluation of the strategic expression:

$$rcond_tdl\{collect_decs[class_{this}]\}[fields_1] \quad (9)$$

In this expression, the strategy *collect_decs* is applied to the term $class_{this}$, which is the symbolic reference of the class that is currently being processed. The strategy resulting from this application is then applied to the fields section ($fields_1$) of the current class file using the traversal *rcond_tdl*. At this stage, terms in the fields section have the form $field_info[access_flags_1 name_1 descriptor_1]$. The result is a strategy that applies an *identity transformation* to constant

pool entries that satisfy the static binding property. Notice that, once such an identity transformation is applied, the application of the strategy to this entry is completed (since the strategy is constructed entirely using the conditional composition combinator $+>$). For entries that do not satisfy the static binding property none of the identity transformations associated with the current class apply so application continues until an appropriate instance of the strategy $hide(lift[class_{this}][class_{super}])$ is encountered. This strategy rewrites the class of the symbolic field reference to its superclass. Since the strategy is enclosed in a *hide* combinator its application does not trigger completion of the overall strategy and application continues. At this time, the symbolic reference may satisfy the static binding property. If this is the case, then an identity transformation (derived from one of the field declarations of the superclass) will apply after which the application terminates. Otherwise another lift is applied and the process repeats.

7 Related Work

This section gives a brief overview of parsing technology and briefly discusses some other approaches to transformation.

7.1 Parsing: From Strings to Terms

Parsing technology is a key enabler of transformation. In this context, a *parser* is seen as a function that is given a flat (i.e., one dimensional) string as its input and returns a (two dimensional) structure called a *term* or *tree* as its output. The structural elements in a *term* provide crucial information against which transformations are written. Without this structure, transformation degenerates to little more than an automated version of the “cut and paste” capability found in text editors.

A *parser generator* is able to automatically generate a parser for given grammar. This capability enables the development of language independent transformation systems. The ability to deal with ambiguity represents a limiting aspect of parsers and parser generators. Typical LALR(1) parser technology is effective for parsing languages provided all derivations can be disambiguated by looking ahead one token in the input. LALR(1) parsers are widely used because of their efficiency and are produced by the YACC and GNU Bison compiler-compilers.

A drawback of LALR(1) parsers [17] is that they can only effectively handle a subset of the set of unambiguous context-free grammars (i.e., not every context-free language has an LALR(1) grammar). It turns out that this restriction is significant in the realm of language independent transformation. As a result, more powerful parsing algorithms are typically employed by transformation systems. At the present time, the generalized LR (GLR) parsing algorithm is almost universally agreed upon as being necessary in order to seriously consider language independent transformation. A number of GLR parser generators

are freely available. Elkhound [16] and Harmonia [1] are two systems that provide GLR parsing capabilities. In both of these systems, a grammar is defined using an EBNF syntax. In the realm of transformation, perhaps the most widely known and freely available GLR-based parser generator is that provided by the Syntax Definition Formalism (SDF) [20] [19]. SDF is a realization of a scannerless generalized LR parser (SGLR) supporting declarative disambiguation rules. Examples of disambiguation rules include rules that resolve ambiguities resulting from associativity and precedence of mathematical operators. SDF also supports modularization of grammars that enable grammars to be factored. Grammar modules can also be composed and thus reused.

7.2 Other Approaches to Transformation

In a classical setting, a strategic programming system can be viewed as a rewriting system in which constructs controlling rule application have first-class status. From a practical perspective, explicit control over rule application is essential when dealing with rule sets that are neither confluent nor terminating. Typically, it is the presence of explicit control constructs (e.g., sequential composition of rules, generic term traversals, etc.) that distinguishes a transformation system from a pure rewriting system.

ELAN [3] is a first-order strategic programming system in which rules can be grouped into labelled rule bases. An efficient AC-matching algorithm is used to control the application of such rule bases to terms. The consequence of AC matching and labelled rule bases is that the application of a rule to a specific term may yield multiple results. This form of non-determinism surrounding rule base application is central to ELAN and gives the system a deductive/declarative flavor. The ρ -calculus [7] provides the semantic foundation for ELAN [6].

Stratego [21] is a first-order strategic programming system whose control constructs include a wide range of combinators and one-layer traversals. Stratego further extends this control framework with scoped dynamic rewrite rules [15]. Though their semantics are slightly different, scoped dynamic rewrite rules can be seen as a first-order cousin of the higher-order rules of TL discussed in this article. Stratego also provides access to a number of low-level abstractions (e.g., *match* and *build*) which can be composed to define a variety of transformational ideas. As such, it provides a foundation upon which other kinds of transformation systems can be built.

ASF+SDF [2] is a first-order rewriting framework in which an extended form of matching provides the ability to perform associative matching on list structures. Recently, the ASF+SDF has been extended so that one can combine parameterized rewrite rules with a fixed set of generic traversals [18].

Strafunski is a Haskell-based system in which transformation is approached from a functional perspective. Monads are used to propagate information and traversals are described in terms of catamorphisms such as *fold b* \oplus . This connection between catamorphisms and strategic driven term traversal was first made in [14].

8 Conclusion

The ever increasing complexity of software systems has presented transformation-based approaches to software manipulation with unique challenges. Significant progress remains to be made with respect to problems relating to scale and reuse. Orthogonal to these issues, a ubiquitous theme in transformation concerns itself with bringing together data from unrelated portions of a term. Higher-order transformations provide an abstraction capable of distributing data throughout a term structure. However, we believe the story is far from over. HATS represents an environment in which transformational ideas can be explored. HATS gives plenty of feedback to users and the interpreted nature of the HATS transformation engine lends itself to modification allowing new transformational constructs and abstractions to be added to HATS with relatively reasonable effort. The hope is that this will encourage continued exploration and experimentation in the area of transformation-based design and development.

References

1. A. Begel, M. Boshernitsan, and S. L. Graham. Transformational generation of language plug-ins in the harmonia framework. Technical Report CSD-05-1370, University of California, Berkeley, California, January 2005.
2. J. A. Bergstra. *Algebraic specification*. ACM Press, New York, NY, USA, 1989.
3. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of elan. *Electr. Notes Theor. Comput. Sci.*, 15, 1998.
4. J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR Program Transformation System: Simplifying the Development of Numerical Software. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser Boston, Inc., 1997.
5. J. M. Boyle and M. N. Muralidharan. Program Reusability through Program Transformation. In *IEEE Transactions on Software Engineering*, volume 10:5, pages 574–588, September 1984.
6. Cirstea, Horatiu and Kirchner, Claude. The rewriting calculus as a semantics of ELAN. In J. Hsiang and A. Ohori, editors, *4th Asian Computing Science Conference*, volume 1538 of *Lecture Notes in Computer Science*, pages 8–10, Manila, The Philippines, Dec. 1998. Springer-Verlag.
7. Cirstea, Horatiu and Kirchner, Claude. An introduction to the rewriting calculus. Research Report RR-3818, INRIA, Dec. 1999.
8. M. de Jonge. A pretty-printer for every occasion, 2000.
9. HATS. <http://faculty.ist.unomaha.edu/winter/hats-uno/hatsweb/index.html>.
10. J. He and C. A. R. Hoare. *Unifying theories of programming*. Prentice Hall International Series in Computer Science, 1998.
11. J. Kyoda and H. Jifeng. Towards an Algebraic Synthesis of Verilog. Technical Report UNU/IIST Report No. 218, The United Nations University, July 2001.
12. R. Lämmel, E. Visser, and J. Visser. The Essence of Strategic Programming. 18 p.; Draft; Available at <http://www.cwi.nl/~ralf>, Oct.15 2002.
13. R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, Jan. 2003.

14. R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In J. Jeuring, editor, *Proceedings of WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59, July 2000.
15. K. O. M. Bravenboer, A. van Dam and E. Visser. Program transformation with scoped dynamic rewrite rules. Technical Report UU-CS-2005-005, Institute of Information and Computing Sciences, Utrecht University, 2005.
16. S. McPeak. Elkhound: A fast, practical glr parser generator. Technical Report UCB/CSD-2-1214, University of California, Berkeley, California, December 2002.
17. M. van den Brand, M. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *In Proceedings of the 6th International Workshop on Program Comprehension, IWPC'98*, pages 108–117, Ischia, Italy, 1998.
18. M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.
19. M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.
20. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.
21. E. Visser, Z. el Abidine Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *ICFP '98: Proc. of the third ACM SIGPLAN international conference on Functional programming*, pages 13–26. ACM Press, 1998.
22. G. L. Wickstrom, J. Davis, S. E. Morrison, S. Roach, and V. L. Winter. The SSP: An example of high-assurance system engineering. In *HASE 2004: The 8th IEEE International Symposium on High Assurance Systems Engineering*, 2004.
23. V. Winter. Strategy Construction in the Higher-Order Framework of TL. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 124, 2004.
24. V. Winter and M. Subramaniam. Dynamic Strategies, Transient Strategies, and the Distributed Data Problem. *Science of Computer Programming (Special Issue on Program Transformation)*, 52:165–212, 2004.
25. V. L. Winter. An overview of hats: a language independent high assurance transformation system. In *In Proc. of IEEE Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET)*., pages 222 – 229, March 1999.
26. V. L. Winter, J. Beranek, A. Mametjanov, F. Fraij, S. Roach, and G. Wickstrom. A Transformational Overview of the Core Functionality of an Abstract Class Loader for the SSP. In *Tenth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005)*, 2005.