A Refinement-based Approach to Developing Software Controllers for Reactive Systems

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

Deepak Kapur
University of New Mexico
kapur@cs.unm.edu

Ray Berg
Sandia National Laboratories
reberg@sandia.gov

October 27, 1999

Abstract

The purpose of this paper is to demonstrate how transformation can be used to derive a high integrity implementation of a train controller from an algorithmic specification. The paper begins with a general discussion of high consequence systems (e.g., software systems) and describes how rewrite-based transformation systems can be used in the development of such systems. We then discuss how such transformations can be used to derive a high assurance controller for the Bay Area Rapid Transit (BART) system from an algorithmic specification.

1 Transformation and High Integrity Software Development

When developing systems, software-based or otherwise, for high consequence applications, it is crucial to provide sufficiently convincing evidence that the behavior of the system will not lead to a high consequence failure. In fact, one can argue that it is this need to be able to provide strong evidence regarding a system's behavior prior to its actual operation, that distinguishes high integrity (software) system development from other forms of system engineering.

In this paper, we assume that a correct specification, $S_0$, of the system to be developed exists. Such a specification will, by definition, disallow behaviors that lead the system into high consequence failure states. Given these assumptions, the objective in high integrity software development then is to (1) use $S_0$ as the basis for developing an implementation, $S_n$, and (2) provide strong evidence that the implementation $S_n$ satisfies $S_0$. As one might guess, the real difficulty lies in addressing the second point mentioned in the previous sentence.

Methods for providing evidence that an implementation satisfies a specification have been broadly classified as belonging either to Validation or Verification. Validation methods generally provide probabilistic evidence of a system's correctness, which is often described in terms of reliability. For example, one can validate that a system responds correctly to an input test set. In contrast, verification methods make statements covering the entire input space. So the verification that a system's behavior possesses a property, $P$, corresponds to exhaustive testing of the system's input space.

As the input space of a system increases, validation methods are faced with significant problems. These problems are compounded when the level of probabilistic evidence, that a system operates correctly, approaches 1.0 (i.e., the likelihood of a failure approaches 0.0).

High consequence systems generally require strong evidence of correctness and often have large input spaces. This makes them resistant to validation methods. Over the years, convincing arguments have been made that, in the high consequence realm, one generally cannot provide sufficient evidence about the intrinsic behavior of a system and must therefore rely on providing convincing evidence based on an analysis of a predictive model of the system [1][2][3]. Furthermore, it is also widely accepted that testing (model-based or otherwise) alone will not be sufficient to provide the level of assurance required. Thus, other analysis techniques, such as formal verification, must be brought to bear in order to provide a sufficient level of
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.
assurance that the system will not experience a failure.

1.1 Verification

In a formal framework, the notion of an implementation satisfying a specification is defined as a relation denoted by $\subseteq$. The expression $S_0 \subseteq S_n$ asserts that $S_n$ satisfies $S_0$. Showing that $S_0 \subseteq S_n$ 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 $S_0 \subseteq S_n$ 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 transformations. The objective here is to construct a transformation sequence that (1) is capable of producing an implementation, and (2) whose correctness can be proved. The transformation process yields a number of intermediate representations of $S_0$. More specifically, if $n$ transformation steps are performed then we will have the representations: $S_0, S_1, S_2, ..., S_n$. 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, $S_n$.

Empirical evidence suggests that transformations which produce small changes are generally easier to prove correct, than transformations producing large changes. Intuitively, the motivation for having small changes is that as $S_i$ and $S_{i+1}$ become increasingly similar to one another, $S_i \subseteq S_{i+1}$ should become easier to prove.\(^1\)

And finally, since $\subseteq$ is transitive, we can calculate $S_0 \subseteq S_n$ by showing that $S_i \subseteq S_{i+1}$ holds. In this case, we say that the transformation sequence $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow ... \rightarrow S_n$ 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 $S_0 \subseteq S_n$. 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 [9].

2 Background

Rewrite systems, which form the underpinnings of transformation systems, provide an excellent framework for transforming (or rewriting) objects, be they strings without variables, expressions (terms) with variables, trees, graphs or pictures. A language $L$ of objects must be specified, and a rewrite relation among objects is defined. It is worth mentioning that the language of objects as well as the rewrite relation can be infinite. However, such a rewrite relation can often be compactly specified using a finite set of rewrite (transformation) rules. Given a rewrite rule $l \rightarrow r$, the rewrite relation $\rightarrow$ induced by it must be defined, i.e., the objects that the rewrite rule can transform must be defined as well as the result of rewriting must be precisely given.

The relation $\rightarrow$ is typically defined as follows:

- for every uniform substitution $\sigma$ of variables in $l, r$, $<\sigma(l), \sigma(r)>$ is in $\rightarrow$ (stability property).

- given an object $o$ that has an occurrence of $l$ at position $p$ (written as $o[p = l, <o, o'>]$, where $o' = o[p \rightarrow r]$, the object obtained by replacing $l$ at position $p$ by $r$, is also in $\rightarrow$.

The rewrite relation induced by a system of rewrite rules is just the union of the rewrite relation induced by each rewrite rule in the system.

If the rewrite rules are derived from equations (i.e., $l \rightarrow r$ is made from the equation $l = r$ by using some measure determining which side is more complex), then an equational theory $E$ (which is also a binary relation) can be associated with the equations corresponding to the rewrite rules. Typically, $E$ is the reflexive, symmetric and transitive closure of $\rightarrow$ corresponding to the rewrite rules obtained from $E$.\(^2\)
The connection between equational theories and rewrite systems is important, because rewrite systems have been found useful in doing equational reasoning and proofs of new equations from the original set of equations. The challenge in this case is to construct from \(\rightarrow_c\), another rewrite relation \(\rightarrow_c\) such that (i) \(E\) is the reflexive, symmetric and transitive closure of \(\rightarrow_c\), and (ii) \(\rightarrow_c\) is canonical, i.e., terminating and confluent. If \(\rightarrow_c\) can be constructed, it becomes possible to determine whether an equation \(s = t\) logically follows from a given set of equations \(\{s_i = r_i | 1 \leq i \leq n\}\), i.e., \(<s, t> \in E\) by just computing the canonical forms of \(s\) and \(t\) using \(\rightarrow_c\), and checking whether these canonical forms are identical. The equation \(s = t\) indeed follows from \(E\) if and only if their canonical forms with respect to \(\rightarrow_c\) are identical. For certain classes of equational theories, the relation \(\rightarrow_c\) can often be generated from \(\rightarrow\) using completion procedures.

It should be noted that the generation of the relation \(\rightarrow_c\) from \(\rightarrow\) is done based on sound inference rules. Thus, from the computation of canonical forms of \(s\) and \(t\) and the generation of \(\rightarrow_c\) from \(\rightarrow\) which in turn is generated from equations in \(E\), a proof of \(s = t\) from the original equation set \(\{s_i = r_i | 1 \leq i \leq n\}\) can be generated.

In the context of high integrity software development, the framework described above is attractive because (1) the correctness (i.e., the equivalence between the left and right hand sides of a rewrite) of rewrites can be determined in isolation, (2) rewriting is an incremental process, and (3) the potential exists for substantial automated assistance.

Roughly speaking, given a set of transformations \(\mathcal{R}\), if \(s\) is a specification, and \(t\) is an implementation, then the calculations which show that \(s\) rewrites to (transforms to) \(t\) using \(\mathcal{R}\) corresponds to a verification that \(t\) correctly implements \(s\), insofar as every transformation has been proved to be correctness-preserving.

2.1 Transformation

A syntax-based (program) transformation system can be viewed as a rewrite system where the relation upon which the rewrite system is built is refinement which is denoted by \(\subseteq^2\). The expression \(s \subseteq t\) asserts that \(s\) is refined by \(t\), or \(t\) is correct with respect to \(s\).

From the perspective of rewriting, in an automated transformation framework, deriving an implementation, \(S_n\), from a specification, \(S_0\), proceeds in five steps:

1. An initial refinement relation, \(\rightarrow_{\mathcal{C}}\), is defined that is based on general refinement knowledge combined with basic (fundamental) problem domain knowledge. Generally, \(\rightarrow_{\mathcal{C}}\) can be reused as the initial relation for all problems belonging to the problem domain.

2. A domain theory, \(\mathcal{D}\), is constructed reflecting specialized domain knowledge and problem specific knowledge such as optimizations.

3. Using the knowledge from 1 and 2, a problem specific refinement relation \(\rightarrow_{\mathcal{R}}\) is designed.

4. This relation, \(\rightarrow_{\mathcal{R}}\), is then realized in the form of a transformation program, \(T\).

5. \(T\) is applied to \(S_0\) yielding \(S_n\) as its result. The program \(S_n\) is the "most refined" object that can be obtained from \(S_0\) using \(\rightarrow_{\mathcal{R}}\).

In this approach, if all of the transformations in \(\rightarrow_{\mathcal{R}}\) have been proved to be correctness preserving (i.e., they really are refinements) then, by the transitivity of \(\subseteq\), it follows that \(S_0 \subseteq S_n\). Under the assumption that \(T\) is executed correctly, simply proving the correctness of the individual transformations in \(T\) is sufficient. However, if this assumption is not made, then a correctness proof would also include a trace of the intermediate forms produced during the application of \(T\) to \(S_0\), as well as a correspondence between intermediate forms, \(S_{i+1}\), and which transformation, in \(T\), was applied to \(S_i\) in order to obtain \(S_{i+1}\).

It should be noted that initially, considerable effort may be required to construct \(\rightarrow_{\mathcal{R}}\). From an economic standpoint, this approach to software development becomes attractive when a refinement theory, \(\mathcal{D}\), is defined for a problem domain for which many problems need to be solved. If this is the case, much of \(\rightarrow_{\mathcal{R}}\) can be reused and the cost of its development can be amortized over the problem space.

2.1.1 Distinctions between Rewriting and Transformation

The major difference between a general rewriting system and a transformation system is that a transformation system is typically not confluent. In fact, it is not easy to generate an equivalent confluent system from a given transformation system using heuristics and techniques such as completion procedures. It is even unclear whether a confluent system is likely to be useful. This means that the order in which transformations
are applied can be crucial. Also, the main objective is to transform a program, which is a ground object for a transformation system, i.e., an object without any variables. It is thus the normalization, that is crucial for this application.

Because of the lack of confluence of transformational systems, extensive control must be given to the user regarding the application of refinements. Historically, two approaches have been taken to address this problem. In the first approach, the application of refinements is controlled manually by the user. In this approach mechanisms must be provided for (1) selecting subobjects to be refined (i.e., rewritten through substitutions based on refinements), and (2) selecting the refinement to be applied.

A key objection to manual control is that the number of refinements that can be applied is directly limited by human endurance. Thus, in this framework, refinement steps tend to become more ambitious which generally makes them more difficult to verify and often makes their reuse problematic.

Another approach to dealing with the control issue is to develop a transformation control language in which intricate transformation sequences can be expressed. Transformation programs can then be written and their application can be performed automatically. In this approach, the application of refinements is not limited by human endurance and it is relatively commonplace for a transformation program to apply $10^4$ refinements while considering $10^8$ potential application points. These numbers are worth mentioning because of the empirical evidence linking the size of a transformational step to its provability. That is, smaller transformational steps are generally easier to prove than larger transformational steps. In general, it is also the case that as the size of a transformational step goes down, the number of transformational steps need (to achieve the same objective) goes up. Paraphrasing then, if we want to prove the correctness of our transformation steps, we will have to apply a lot of them in order to derive an implementation.

### 2.2 Formal Transformation

One way to view formal program transformation is as follows:

- $\mathcal{L}$ — a wide spectrum language whose syntax is described by a context-free grammar, and whose language constructs are monotonic with respect to the relation $\subseteq$. In this context, language elements (i.e., specifications and programs) are syntax derivation trees (SDT’s) rather than flat strings.

- $\rightarrow_{\mathcal{L}}$ — a set of refinements having the form: $t_1 \Rightarrow t_2$ if application-condition. An actual application of such a refinement to an SDT $x$ proceeds as follows: (1) a substitution $\sigma$ resulting from the match of $x$ with $t_1$, and the evaluation of application-condition is produced, and (2) the resultant (output) SDT is obtained by applying $\sigma$ to $t_2$. In addition, reflexivity and transitivity are also properties that are possessed by

### 3 An Overview of HATS

HATS is a transformation system that has been developed within the High Integrity Software (HIS) program at Sandia National Laboratories. It is freely available and can be downloaded from http://www.sandia.gov/ast/downloads.html.

In HATS, program transformation takes place in a wide-spectrum language, $\mathcal{L}$, that is defined by a context-free grammar and corresponding lexer. An element of $\mathcal{L}$ (i.e., a specification) to which transformations are to be applied is abstractly referred to as a source program, and is represented in terms of its syntax derivation tree (SDT). It is these SDT's that are manipulated through transformation.

To enable the automated application of transformations, HATS provides a special purpose language, called the transformation language. This language contains both functional and imperative constructs and provides a rich environment for describing transformation sequences and controlling the application of transformations.

One distinguishing feature of this language is that transformations are functions that are parameterized on SDT’s. In turn, this requires that unification be an explicit operation, which in the HATS transformation language is denoted by $|=|$. For example, if $s_1$ and $s_2$ denote two schemas or templates, then the expression $s_1 |=| s_2$ is a unification expression whose value is true if $s_1$ and $s_2$ unify and false otherwise. In program transformation, a special case of unification, called matching, is generally used. The distinction between a match and a unification is that a match is a unification where one expression (e.g., term) is ground (i.e., has no variables). While HATS does support unification, the transformation of programs generally only involves matches.

Another distinguishing feature of the transformation language is that, with respect to a given applica-
3.1 Applications

To date HATS has been used to implement:

- An optimizer for a specific class of reactive systems (of which the Production Cell[5] is a member).
- A unit resolution propositional theorem prover. Mainly, this was an experiment to test the capabilities of the control paradigm supported in HATS.
- Martelli-Montanari’s unification algorithm for a set of equations.
- A translator from arbitrary first order expressions into disjunctive normal form. This example required variable renaming and scope expansion of quantified variables.

Later this year, HATS will be used to:

- Implement a C to SPARC compiler.
- Explore the feasibility of creating a transformation-based class loader that will generate an executable ROM image from a Java class file hierarchy, performing the symbolic resolution of the constant pool entries, link editing, and loading of each class file in the hierarchy. The input to the class loader will include the name of the “main” class, I/O specifications, run-time support specifications, and hardware specifications (processor version, memory sizes, etc.). The class loader output will be a ROM image in a format compatible with PROM programmers (e.g., Motorola S Records) and will support the architecture of the Sandia Secure Processor.

4 The Specification of BART

BART is a heavily used train system that provides commuter rail service in the San Francisco Bay Area. Trains in this system must satisfy the following constraints, which are stated here informally. For a more formal discussion of these constraints see [3].

- A train should never get so close to its leading train (i.e., the train in front of it) that an abrupt stop of the leading train results in a collision.
- A train should stop at track signals when told (and able) to do so.
- A train should not exceed the speed limit of the track segment on which it is traveling.
- A train should stop at designated stations.

There are various other objectives that trains must satisfy. For example, trains should eventually reach stations (i.e., liveness). One would like to optimize the behavior of trains thereby increasing the throughput of the system. Trains should also run smoothly—for the comfort of the passengers, and so on and so forth. For an in-depth discussion of the BART case study see [11].

In the context of this paper, a specification is a description of the controller function capable of satisfying the above constraints. Some of the reasons the construction of such a specification is challenging are:

- Trains can take a long time to stop.
- Acceleration values must be ramped up over time (e.g., one cannot simply go to a maximum negative acceleration).
- Limited assumptions can be made regarding the behavior of a leading train (e.g., it may derail).

Generally, what distinguishes an algorithmic specification from an implementation is that the algorithmic specification is written in a domain language having data structures and operators that are not directly supported in the implementation language. The purpose of a domain language is to provide a suitable framework for modeling, constraining, and describing system (e.g., train) behaviors in a natural manner. While designing an algorithmic specification, the emphasis is on ease of expression, elegance, correctness, scalability and adaptability. An algorithmic specification can often serve the role of a formal documentation that is expected to be consulted for any modifications as well as disagreements about how the system should behave. Even though an algorithmic specification includes high-level design decisions about allowable implementations, many low-level decisions are still left
4.1 An Overview of a Domain Language

From our experience with BART we have developed a domain language based on two fundamental constructs: profiles and constraints. A profile describes a relation between position and speed, where position can be quantified over a discrete or continuous domain. Profiles are used to model trajectories in the position-speed plane.

The second construct we call a constraint. A constraint defines a region with respect to a given profile. In particular, a constraint can define the area beneath a profile. Next, a “satisfies constraint” operator, \( \preceq \), is defined enabling the specification that a trajectory satisfy a particular constraint or set of constraints.

In addition, the domain language provides vectors as the data structure for modeling reactive systems of which BART is an instance. Systems are modeled in terms of a vector of monitored and controlled variables. For example, the BART system is modeled in terms of a vector of monitored variables describing the position and speed of the trains in the system, and controlled variables describing the acceleration of each train.

A useful operation on vectors is the projection of subvectors. In the domain language, projections are defined in terms of index lists. For example, given a vector \( v = (x_1, x_2, x_3, x_4) \), the expression \( v[1,4] \) denotes the subvector \( (x_1, x_4) \).

In addition, the domain language provides vectors as the data structure for modeling reactive systems of which BART is an instance. Systems are modeled in terms of a vector of monitored and controlled variables. For example, the BART system is modeled in terms of a vector of monitored variables describing the position and speed of the trains in the system, and controlled variables describing the acceleration of each train.

The behavior (i.e., its state over time) of a train can be modeled by a profile. This profile can also be used as the basis for defining a constraint. For example, with respect to a given object train profile, \( t_0 \), its leading train profile, \( t_1 \), defines a constraint that \( t_0 \) must satisfy.

It is important to mention that the constructs in the domain language are given a formal operational definition. Thus specifications are computable, though the computation sequences they define can be very inefficient. For a more detailed discussion of this domain language see[3].

4.2 An Algorithmic Specification of a Simplified Controller for BART

In this section, we give an algorithmic specification of a controller for a subset of the actual BART system. The system for which the specification below is written is a simplified version of BART that consists of only two trains: a lead train, and an object train. The specification gives an abstract algorithm describing a control function for the object train. It is assumed that the lead train, which is ahead of the object train, is controlled in some suitable manner (e.g., by another control function).

In the specification below, it is assumed that the physical characteristics of both trains (e.g., their ability to accelerate) is the same. Thus a single model is used, by the controller of the object train, to predict the behavior of both trains. The specification begins by defining the set of acceleration values that the control variable for a train is quantified over. This includes minimum and maximum acceleration values, jerk rates, and the resolution of acceleration values (i.e., the next acceleration value that can be realized). Next, successor and predecessor functions are defined allowing enumeration of acceleration ranges in a manner that is consistent with the min and max accelerations, jerk rates, and resolutions. After this, delta and gamma functions are defined which model how trains change state with respect to time. This is followed by an abstraction, created for human readability, that constructs the set of accelerations that are reachable from a given acceleration value (e.g., from the current acceleration of a train).

The constraints that the object train must satisfy are defined next. In particular, the object train must not violate the constraints defined by the stopping profile of the lead train and the track profile. Specifically, an acceleration should not alter the speed and position of the object train in such a way that it can no longer brake in time to avoid the lead train or to avoid exceeding the speed limit of a track segment. And finally, the control function for the object train, when given the state of the system as input (this allows us to avoid dealing with I/O issues), first computes the set of all acceleration values satisfying the constraints and then selects a desired value from this set (e.g., the maximum value). Note that for smoothness purposes, values other than the maximum acceleration may be selected, provided, of course, that the corresponding selection function is defined in the specification.

Notable aspects of BART that are simplified or omitted from this specification are:
The actual model of acceleration. In BART, a change from positive to negative acceleration (and vice versa) must be preceded by a mode change. Furthermore, the maximum acceleration is dependent upon the speed the train is traveling. This acceleration function can easily be specified as a large case statement defined on an enumerated set having “mode change” elements at the appropriate positions.

The position and speed computations have been simplified in our specification.

Our specification does not consider the impact of noisy/lossy transmissions (e.g., the object train gets a garbled acceleration command). In our specification, noise can be accounted for by properly defining how it impacts the stopping profile of the object train. The rest of the specification then remains unchanged.

Given the proper models (defined by domain experts) the specification below can easily be altered to allow them to be included. The objective of this research is not to demonstrate our domain knowledge, but rather addresses the software issues surrounding how specifications built using specific domain models can be transformed into efficient high integrity implementations.

The specification of BART given below is with respect to the following system model:

- The system state is defined in terms of a vector of monitored and controlled variables. For more on this see [3]. In this example, the state of the system can be modeled by \( s = (p_1, s_1, p_2, s_2, t, a_1, a_2) \), where \((p_1, s_1, p_2, s_2, t)\) denotes the vector of monitored variables and \((a_1, a_2)\) denotes the vector of controlled variables.

- A vector describing the state of a train. Given the state of the system, \( s \), the state of the object train is described by the projection \( s[1] = s[1, 2, 6] = (p_1, s_1, a_1) \), and the lead train is described by the projection \( s[2] = s[3, 4, 7] = (p_2, s_2, a_2) \).

- A vector describing the profile elements of a trace (i.e., its position and speed variables). Given the state of the system, \( s \), the track profile is described by the projection \( s[3] = s[5] = (t) \).

- A collection of functions capable of enumerating the set of acceleration values according to the physical limitations of the system.

\[ \text{monitored variables} = (p_1, s_1, p_2, s_2, t, a_1, a_2) \]

\[ \text{controlled variables} = (a_1, a_2) \]

\[ \text{system state} = (p_1, s_1, p_2, s_2, t, a_1, a_2) \]

\( \text{object train} = (p_1, s_1, a_1) \)

\( \text{lead train} = (p_2, s_2, a_2) \)

\( \text{track} = \text{profile} \)

\( \text{train state} = (p, s, a) \)

\( \text{position} = (p) \)

\( \text{acceleration} = (a) \)

\( \text{speed} = (s) \)

\[ \text{define } \max\text{\_suc}\text{\_a} = (\text{lambda } a. \min(a + \text{MAX\_POS\_JERK\_RATE, MAX\_A} ) ); \]
define max_pred_a = \( \lambda \, a. \max(a - \text{MAX\_NEG\_JERK\_RATE}, \text{MIN\_A}) \); 
define min_pred_a = \( \lambda \, a. \max(-\text{MAX\_NEG\_JERK\_RATE}, \text{MIN\_A}) \);

\[
\begin{align*}
\text{delta_t_step} &= (\lambda \, \text{p, s, a}. \, [\text{p} + \text{s}, \max(\text{s} + \text{a}, 0), \text{a}]); \\
\text{gamma_t_step} &= (\lambda \, \text{p, s, a}. \, [\text{p}, \text{s}, \max(\text{a}, \text{s} + \text{a})]);
\end{align*}
\]

\(\text{EMPTY} = []\);

\[
\begin{align*}
\text{define acceleration_range} &= (\lambda \, a. \\
& \quad \text{let} \\
& \quad \text{define start} = \max(\text{succ_a}(a)); \\
& \quad \text{define finish} = \max(\text{pred_a}(a)) \\
& \quad \text{in} \\
& \quad \text{list\_mu(start,} \\
& \quad \text{min\_pred_a,} \\
& \quad \text{(\lambda \, a. \, (min\_pred(a) < finish)))} \\
& \quad \text{end};)
\end{align*}
\]

\[
\begin{align*}
\text{define constraints} &= (\lambda \, \text{object\_train\_profile,} \\
& \quad \text{lead\_train\_profile,} \\
& \quad \text{track\_profile.} \\
& \quad \text{in} \\
& \quad \text{select(} \\
& \quad \text{\{ range_a :} \\
& \quad \text{\lambda \, \text{a.}} \\
& \quad \text{constraints(} \\
& \quad \text{get\_object\_train\_profile(object\_train, a)} \\
& \quad \text{)}} \\
& \quad \text{lead\_train\_profile,} \\
& \quad \text{track\_profile} \\
& \quad \text{)} \\
& \quad \text{)} \\
& \quad \text{)} \\
& \quad \text{)}
\end{align*}
\]

\[
\begin{align*}
\text{define controller} &= \\
& \quad (\lambda \, \text{system\_state.} \\
& \quad \text{let} \\
& \quad \text{define object\_train\_state} = \text{system\_state}[1]; \\
& \quad \text{define lead\_train\_state} = \text{system\_state}[2]; \\
& \quad \text{define track\_profile} = \text{system\_state}[3] \\
& \quad \text{in} \\
& \quad \text{react(object\_train\_state,} \\
& \quad \text{acceleration\_range(object\_train\_state[g3],} \\
& \quad \text{lead\_train\_state,} \\
& \quad \text{track\_profile)} \\
& \quad \text{end} \\
& \quad \text{);}
\end{align*}
\]

In [3], we argue that the specification for the simplified version of BART given above can be easily extended to the complete version of BART. In BART, signals and stations can be easily modeled in terms of profile-based constraints. The presence of these constraints slightly change the system state model (i.e., the vector of monitored and controlled variables) and change the constraints equation from:

\[
\begin{align*}
\text{object\_train\_profile} \\
\triangleleft \\
\text{[ lead\_train\_profile, track\_profile ]}
\end{align*}
\]


to

\[
\begin{align*}
\text{object\_train\_profile} \\
\triangleleft \\
\text{[ lead\_train\_profile, track\_profile, signals, stations ]}
\end{align*}
\]

Additionally, specifications for multi-train systems simply extend the above expression so that each
train (except for the very first train) plays the role of the object train with respect to the train immediately in front of it (i.e., its lead train).

Due to the ease in which a specification can be extended to handle additional trains and constraints, the specification given in this paper is a reasonable representative of the type of train controllers required in BART.

5 Transforming the Bart Specification

Having been given an algorithmic specification, $S_0$, we are now in a position to consider constructing an implementation, $S_n$, in such a manner that high assurance can be provided that the implementation is correct with respect to the specification. More formally stated, we require high assurance that the relationship $S_0 \subseteq S_n$ holds.

In this paper, we describe how $S_n$ can be obtained from $S_0$ via a sequence of refinements or correctness preserving transformational steps. In a refinement-based approach, the specification is passed through a number of intermediate steps on its way to becoming an implementation. Thus software development can be abstractly characterized as the creation of the following sequence:

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n$$

where $S_{i+1}$ is obtained from $S_i$ through a transformational step, and $S_n$ is a program that can either be directly executed on a computer (e.g., assembly code) or belongs to a programming language for which a compiler exists. We target ML as our implementation language. The reason for choosing ML is because of its syntactic similarity to our domain language.

Since the domain language has an operational semantics, one way of implementing $S_0$ is to simply realize (i.e., implement) the operational semantics in a programming language (in our case, ML). If this is all that is done, then one has for all practical purposes created a domain library, and the transformation from specification to implementation amounts to little more than "syntactic sugar". Syntactic sugar issues can become interesting as the domain and implementation languages diverge. For example, transforming a functional program to an imperative program can be quite challenging. A compiler is another classic example of the challenges that are faced in this type of transformation[9].

However, due to the flexibility of many of today’s languages in dealing with syntactic issues (e.g., user defined infix operators, enumerated types, etc.), it is not very interesting to simply define (1) the syntax of a "domain language", (2) its operational semantics, and (3) a transformation sequence from the domain language to some "similar" implementation language. If this approach is taken, then strong arguments can be made to simply define a suitable library and data structures directly in the implementation language itself, thereby eliminating the need for transformation altogether.

We believe that a key objective of transformation is optimization, in particular, problem dependent optimization. If this view is adopted, then the purpose of the specification phase of software development is to clearly describe a solution to a problem, and the purpose of the implementation phase is to apply domain and computational knowledge arising from axioms and theorems in order to optimize the abstract algorithm described by the specification. In this context, syntactic issues are considered somewhat incidental.

Another argument against the domain library approach is that it is often the case that operators and constructs that are well suited to clearly describing an abstract algorithm do not yield efficient implementations. Thus while it might be possible to directly implement a specification in terms of a domain library, the resulting code may be very inefficient. Here the term "inefficient" can mean many things. At one extreme, an implementation that is inefficient is an irritation. At the other extreme the inefficiency is so severe that the computation is, for all practical purposes, not viable (e.g., it may take hours or years to compute a answer). In addition, efficiency becomes much more of an issue in the presence of real-time constraints. In such cases, even the delay of a second (or less) can be unacceptable.

Reactive systems having real-time constraints are classic examples of systems in which the efficiency of an implementation is often crucial. BART is an example of such a system. In BART, trains should be given acceleration commands at half second intervals. Furthermore, if a train does not receive an acceleration command within four such intervals (e.g., two seconds), it will go into emergency braking mode. For these reasons, concentrating on optimizing BART specifications is necessary. The specification for BART, given in Section 4.2, is transformed in several stages, of which the optimization stage is of greatest importance. In the first stage, the semantics of the foundational operators constrain, mu, list,mu, and list
are made explicit in the specification. These operators form a computational foundation upon which domain language operators are defined.

Then operators and data structures relating to profiles, constraints, and system models (e.g., vectors) are added. At this point we essentially have an abstract domain library whose functions are used in the specification.

In the next stage, domain knowledge and problem specific optimizations are applied. This knowledge is in the form of slightly altered versions of functions (such as the constrain function) that take advantage of the specific problem described in the specification. These problem specific optimized functions are created by transforming the algorithms that define the original functions. Examples of these optimizations are given in Section 6. The specification is then optimized via transformations incorporating these optimizations where ever possible.

In the following stage, transformations describing datatype translations are applied. These transformations are responsible for shifting the data types used in the specification into representations supported by ML. And finally, in the last stage, transformations are applied that perform the syntactic changes needed to produce an ML implementation.

The figure below graphically depicts the transformation stages that were used to derive an implementation from the specification of BART given in this paper.

5.1 Optimization

Abstractly speaking, a computation sequence is distinguished by the set of operators and constants it contains as well as the order in which operations are applied to constants.

Computation sequences can be viewed as functions. Let \( c_1 \) and \( c_2 \) denote two computation sequences taking input from a domain \( D \). We say \( c_1 \) is refined by \( c_2 \) if \( \forall x \in D : c_1(x) \subseteq c_2(x) \). When discussing refinement between computation sequences we often drop the argument list and simply write \( c_1 \sqsubseteq c_2 \).

In is important to note that the number and types of operations and constants may vary greatly among computation sequences. Let \( f \) denote an abstract function that calculates the time it takes a computation sequence to complete on a given input. For a given input \( x \), \( c_2(x) \) is an optimization of \( c_1(x) \) iff \( c_1 \subseteq c_2 \land f(c_2(x)) < f(c_1(x)) \).

In most cases, real-time information concerning the time it takes to execute computation sequences is sketchy. For this reason, \( f \) is usually an informal calculation that is based on the judgement and experience of the person writing the optimization.

5.2 An Optimization Example

In the domain language, there is a set filter (or constrain) operation that is denoted by the colon symbol (which we refer to as constrain). Let \( S \) denote an enumerated set and let \( P \) denote a predicate on the elements of \( S \), then the expression \( [S : P] \) denotes the subset of \( S \) whose elements satisfy \( P \).

Consider the following operational semantics of the constrain (i.e., “colon”) operator:

\[
[S : P] = \\
\text{if } S = \emptyset \\
\quad \text{then } \emptyset \\
\text{else if } P(\text{first}(S)) \\
\quad \text{then } P(\text{first}(S)) \cup [\text{rest}(S) : P] \\
\text{else } [\text{rest}(S) : P]
\]

Now consider the expression: \( [S : P] \neq \emptyset \). If the constrain operator is simply treated as a library function then \( [S : P] \) will have to be evaluated first followed by a comparison with the empty set. This evaluation sequence can be inefficient when \( S \) is large. However, a new function, constrain_empty, can be created by distributing the comparison with the empty set over the operational definition of constrain. This results in the following:

\[
\text{constrain}_\text{empty}(S, P) = \\
\text{if } S = \emptyset
\]
then $\emptyset \neq \emptyset$
else
  if $P(\text{first}(S))$
    then $(\{P(\text{first}(S))\} \cup \text{constrain.empty}(S,P)) \neq \emptyset$
  else $\text{constrain.empty}(\text{rest}(S),P)$

Using the following facts:

- $(\emptyset \neq \emptyset) = \text{false}$
- $(\{P(\text{first}(S))\} \cup \text{constrain.empty}(S,P)) \neq \emptyset) = \text{true}$

allows the definition of $\text{constrain.empty}$ to be further simplified to:

$$\text{constrain.empty}(S,P) =$$

- if $S = \emptyset$
  then $\text{false}$
  else if $P(\text{first}(S))$
    then $\text{true}$
  else $\text{constrain.empty}(\text{rest}(S),P)$

Thus enabling the context-dependent optimization of the $\text{constrain}$ operator as follows:

$$[S : P] \neq \emptyset$$

Thus enabling the context-dependent optimization of the $\text{constrain}$ operator as follows:

$$\text{constrain}(\text{list}(x,\text{succ},n),P) \neq \emptyset$$

Note that this expression can be optimized by distributing "$\neq \emptyset"$ into the body of $\text{constrain}$, in effect short circuiting the computation when $P(x)$ is true.

Theorem 1: The generation of a set by $\text{list}(x,\text{succ},n)$ can be distributed over the operational definition of $\text{constrain}$.

Let $S = \text{list}(x,\text{succ},n)$.

$$\text{constrain}(S,P) =$$

- if $S = \emptyset$
  then $\emptyset$
  else if $P(\text{first}(S))$
    then $(\{\text{first}(S)\} \cup \text{constrain}($ \text{rest}(S),P))$
  else $\text{constrain}($ \text{rest}(S),P)$

The specification we are given computes the set of all acceleration values that satisfy the given constraints (e.g., the lead train and the track), and only then selects an acceleration value from this set. Rather than checking each possible acceleration independently, the set of accelerations can be obtained more efficiently simply finding the largest acceleration, $a_1$, satisfying the constraints and then including $a_1$ and all accelerations, $a_2$ such that $a_2 < a_1$. Informally the argument is that "slower is safer". Formally, one is required to prove that the system constrains are monotonic with respect to the acceleration of the object train.

The theorem below considers the stopping profile produced after accelerating, to $a_1$, the current object train state, $\text{OT}$. If this profile satisfies the constraint defined by the lead train profile, then all accelerations, $a_2 < a_1$, will also satisfy this constraint.

Theorem 2: The lead train profile constraint is monotonic with respect to the acceleration of the object train.

$$\forall a_1, \forall \text{lead\_train\_profile}:$$

- $(\text{OT}: \text{stop\_profile}(\Delta t(\gamma t(\text{OT},a_1)))) \ll$ \text{lead\_train\_profile}$

- $(\forall a_2 : a_2 < a_1 \rightarrow (\text{OT}: \text{stop\_profile}(\Delta t(\gamma t(\text{OT},a_2)))) \ll$ \text{lead\_train\_profile}$

Theorem 3: The track profile constraint is monotonic with respect to the acceleration of the object train.
$\forall a_1, \forall \text{track-profile:}$

$\begin{align*}
(\text{OT}: \text{stop-profile}(\Delta t(\gamma t(\text{OT}, a_1)))) & \text{[profile]} \\
\subseteq & \text{track-profile}
\end{align*}$

$\forall a_2 : a_2 < a_1 \rightarrow$

$\begin{align*}
(\text{OT}: \text{stop-profile}(\Delta t(\gamma t(\text{OT}, a_2)))) & \text{[profile]} \\
\subseteq & \text{track-profile}
\end{align*}$

Corollary 4 The acceleration of the object train is monotonic with respect to the system constraints.

The theorem below describes how a specific instance of the constrain set operator, “;”, can be refined by the function `constrain_monotonic`.

**Theorem 5** A constrained set expression can be refined as follows:

```ml
[ list(a, pred, [acceleration])::
  (lambda a.
    (\text{OT}: \text{stop-profile}(\Delta t(\gamma t(\text{OT}, a)))) \text{[profile]} \\
    \triangleq \text{track-profile} \\
  ) \\
]
\triangleq
constrain_monotonic(
  list(a, pred, [acceleration]),
  (lambda a.
    (\text{OT}: \text{stop-profile}(\Delta t(\gamma t(\text{OT}, a)))) \text{[profile]} \\
    \triangleq \text{track-profile} \\
  )
)
```

where

```ml
constrain_monotonic( list(x, f, n), P) =
  if n = 0
  then list(x, f, n)
  else if P(x)
     then {x}U constrain_monotonic(succ(x), succ.n-1, P, Q)
  else if Q(x)
     then @
  else constrain_monotonic(succ(x), succ.n-1, P, Q)
```

The following lemma and theorem take advantage of the fact that tuples in BART profiles are monotonic with respect to their position element. This allows us to more quickly determine that a profile satisfies a constraint. Informally, we don’t need to check positions that fall outside of the range of the stopping profile of the object train.

**Lemma 6** Profiles (in BART) are monotonic with respect to position.


**Theorem 7** Some constrain expressions can be optimized.

Let P =

$\begin{align*}
(\lambda j. \text{object.train}[i][j][g1] \leq \text{track.profile}[j][g1] \\
\wedge \text{track.profile}[j][g1] \leq \text{object.train}[i+1][g1] \\
\wedge \text{object.train}[i][g2] \geq \text{track.profile}[j][g2]
\end{align*}$

\[
[ \text{list}(x, \text{succ}, n) : P ]
\]

constrain_special(x, succ, n, P, Q)

where

```ml
constrain_special =
  (lambda x, succ, n, P.
    if n=0
    then @
    else if P(x)
       then {x}U constrain_special(succ(x), succ.n-1, P, Q)
    else if Q(x)
       then @
    else constrain_special(succ(x), succ.n-1, P, Q)
```

7 Execution Results

In this section we compare the running times of $S_0$ with $S_n$. $S_0$ is an implementation that was obtained from $S_0$ by directly transforming the constructs of the domain language into ML. The program $S_0$ is a representative of the kind of implementation that could be obtained if an ML library of the domain is created and the resulting program is not further optimized by manipulating the source code in the library. In contrast, the program $S_n$ was obtained from $S_0$ by using transformations to optimize the computations defining the domain constructs. These optimizations were performed in the context of $S_0$, which means that the optimizations are problem specific. The optimized program was then transformed into ML yielding $S_n$.
We developed, by hand, a simulator that would allow us to run an object train controller (e.g., $S_0$ or $S_n$). In this simulator, the train system is defined. For a two train system, this includes the position, speed, and acceleration for both the object train and the lead train, as well as the configuration of the track (e.g., the length of various track segments, the speed limits associated with each track segment and how many track segments the track consists of). The lead train is controlled by the simulator and must obey the speed limit of the track segment on which it is traveling. Other than that, it may slow down, even stop, or speed up in any manner it wishes (i.e., as determined by a random number generator) subject to acceleration limitations imposed by the model.

The columns of the table below are to be read as follows: The first column indicates how many simulation steps were performed. The second column indicates how many track segments made up the system state. The third column describes the platform on which $S_0$ and $S_n$ were run, and the last two columns give the running times of $S_0$ and $S_n$ in seconds. One thing we would like to point out is that the running times include the overhead associated with the simulator program. Clearly, this overhead, which is constant between $S_0$ and $S_n$, has a much more significant impact on $S_n$ than it does on $S_0$, making the difference of the actual running time between $S_0$ and $S_n$ even greater.

<table>
<thead>
<tr>
<th>Sim. Steps</th>
<th>Track Seg.</th>
<th>Platform</th>
<th>$S_0$ time in sec</th>
<th>$S_n$ time in sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>PII400/256MB</td>
<td>3.938</td>
<td>0.344</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>PII400/256MB</td>
<td>40.218</td>
<td>4.578</td>
</tr>
<tr>
<td>200</td>
<td>100</td>
<td>PII400/256MB</td>
<td>64.719</td>
<td>8.078</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>500</td>
<td>100</td>
<td>PII400/256MB</td>
<td>154.203</td>
<td>16.906</td>
</tr>
<tr>
<td>10</td>
<td>300</td>
<td>PII400/256MB</td>
<td>24.84</td>
<td>0.734</td>
</tr>
<tr>
<td>100</td>
<td>300</td>
<td>PII400/256MB</td>
<td>252.41</td>
<td>8.5</td>
</tr>
<tr>
<td>200</td>
<td>300</td>
<td>PII400/256MB</td>
<td>401.376</td>
<td>14.312</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>500</td>
<td>300</td>
<td>PII400/256MB</td>
<td>963.782</td>
<td>37.953</td>
</tr>
</tbody>
</table>

Below is a graphical representation of the results from our timing experiment.

8 Conclusions and Future Work

In this paper we argued that rewrite-based transformation is a technique that can be used in the development of high integrity software. We then demonstrated how transformations could be used to derive an efficient implementation, $S_n$, given a suitable specification, $S_0$.

Future work includes exploration of targeting other implementation languages such as C++. as well as investigation of the issues surrounding transforming more complex specifications (e.g., specifications containing complex models for determining speed and accelerations, etc.).

References


