

# Formal Specification and Refinement of a Safe Train Control Function

Victor L. Winter  
University of Nebraska at Omaha  
Department of Computer Science  
*vwinter@ist.unomaha.edu*

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

Gary Fuehrer  
Altiris  
*gfuehrer@comcast.net*

October 14, 2002

## Abstract

Motivated by the design and development challenges of the BART case study, an approach for developing and analyzing a formal model for reactive systems is presented. The approach makes use of a domain specific language for specifying control algorithms able to satisfy competing properties such as safety and optimality. The domain language, called SPC, offers several key abstractions such as the *state*, the *profile*, and the *constraint* to facilitate problem specification. Using a high-level program transformation system such as *HATS* being developed at the University of Nebraska at Omaha, specifications in this modeling language can be transformed to ML code. The resulting executable specification can be further refined by applying generic transformations to the abstractions provided by the domain language. Problem dependent transformations utilizing the domain specific knowledge and properties may also be applied. The result is a significantly more efficient executable specification which can be used for simulation and gaining deeper insight into design decisions and various control policies. The correctness of transformations can be established using a rewrite-rule based induction theorem prover *Rewrite Rule Laboratory* developed at the University of New Mexico.

## 1 Introduction

Motivated by the BART case study, an approach for developing and analyzing a formal model for reactive systems such as train systems is presented. The methodology presented is suitable for the development of control functions belonging to the class of reactive systems where the ability to predict behavior and deal with behavioral uncertainty is the dominating concern. The BART case study is used for illustrating the approach; the reader may also consult [15] for more details about how the proposed approach has been applied to the BART case study.

Typically, the very first step in domain modeling is the identification of abstractions (both data and algorithmic) relevant to the formulation of a given problem or class of problems. There are several well-known techniques that can be used for such an analysis. For example, Chapter 4 discusses a general purpose analysis method based on the identification of noun and verb phrases in the English description of a problem. The focus of this chapter is not on techniques for identifying abstractions but rather on

how a given set of abstractions can be utilized within a software development methodology that combines formal techniques, such as program transformation and verification, with validation techniques such as simulation and animation.

## 1.1 Some Abstractions for Reactive Systems

In the domain of reactive systems, the *state* and the *transition* are examples of two well-known abstractions. It is well understood that a formal model of any reactive system must include a description of the system's state. In this setting, a transition is an algorithmic abstraction capable of taking the system from one state to another. System behavior is achieved through sequence of transitions, and requirements (e.g., safety requirements, throughput optimization) are generally stated in terms of properties defined on states or state sequences.

In a reactive system with an active controller influencing its behavior, as is the case here for train systems, considerable choices exist how the system evolves from a given current state. Instead of an explicit finite state transition table, there are constraints on state trajectories capturing safety properties, e.g., for a train system, acceleration and speed of a given train should be so chosen as to avoid collision with the lead train, stop at red traffic lights, and obey the speed limits. Only those transitions that keep the system within a safe region of states are thus allowed. Such high-level abstractions appear essential to specifying the behavior of a reactive system.

## 1.2 Proposed Tools and Methodology

The domain modeling language presented introduces a *profile* as the abstraction for modeling state trajectories, and the *constraint* as the abstraction for modeling properties (e.g., safety properties). Relational expressions can be constructed from profiles and constraints describing transition sets whose elements all satisfy the property specified by the constraint. Such expressions can be composed with one another using logical connectives and describe transition sets whose elements satisfy the resulting formula (e.g., transitions that simultaneously satisfy multiple safety properties). From this set of transitions a selection can then be made according to secondary criteria such as throughput optimization.

The domain modeling language introduced is used to define

- states (without taking into consideration any safety requirements),
- safe states,
- next state function that ensures that the system is always in a safe state,
- safety policy, and finally
- a family of controller functions.

The proposed approach is able to select among a family of controller functions, different control policies optimizing different requirements.

Using a high-level program transformation system such as *HATS* [13] being developed at the University of Nebraska at Omaha, descriptions in such a modeling language can be transformed to ML

code. Further optimization of this implementation is possible through generic transformations of the data structures and control structures supported in the domain modeling language as well as problem dependent transformations utilizing domain specific knowledge and properties. These optimization result in a significantly more efficient executable specification. The control function, in its unoptimized or optimized form, can be plugged into a simulator in order to gain a deeper understanding of the consequences of various design decisions and control policies. The correctness of transformations can be established using an rewrite-rule based induction theorem prover *Rewrite Rule Laboratory* [9] developed at the University of New Mexico. The BART case study is used to illustrate how the methodology can be used in practice.

A key advantage of the proposed approach is the use of simulation and animation to debug and validate the formal model. An efficient executable specification obtained from a high-level specification using a series of domain independent and domain specific transformations can be used for simulating a model. The model can be studied concretely, providing insight into its behavior and thus enhancing confidence in its correctness. The simulation can be linked to an animator to view different scenarios. Any problem identified or any anomalies can be traced back to the model and fixed in the model. After extensive simulation, when the designer and user feel confident about the correctness of the model based on extensive simulation scenarios, the model can then be subjected to rigorous formal methods in order to formally verify that desired properties are preserved under the stated assumptions.

## 2 Technical approach and Method

The key steps of the proposed approach are:

- using state and profile abstractions to model various aspects of the system,
- using constraints to model individual safety properties to the extent possible,
- the definition of what it means to be a safe state with respect to a given safety policy,
- the definition of a family of controller functions and of an optimal control policy in the domain modeling language,
- application of transformations (both domain independent as well as domain specific) on the definitions to obtain an efficient executable specification,
- validation through simulation and animation of the above model on scenarios and the animation of the output; revising the above steps to debug the formal model and to ensure correctness, and
- verification (of the model as well as transformations used for the model).

### 2.1 Reactive Systems

A reactive system is typically designed to achieve a desired objective while interacting with its environment. Based on its controlled input and the state of its environment, the system evolves with time. The current state of the system (and the environment) is captured in terms of certain measurements

obtained using a collection of sensors. Among various possible actions, certain actions are chosen by the controller to ensure that the system state never violates any safety policy by entering into a state deemed unsafe. Typically, there is some flexibility in choosing among possible actions. As a result, other considerations can be taken into account for selecting the best possible action in every state.

A reactive system is typically controlled incrementally by placing a control function core within a sense/react loop. In such a paradigm, each iteration of the loop corresponds to an incremental (discrete) control step. The duration for which a reactive system can run unsupervised is generally dependent upon three things: (1) the ability of the sensors to sufficiently capture the state of the system, (2) the fidelity of the system model used by the core, and (3) the ability of actuator commands to realize the desired system behavior. In this chapter, we will use the term *uncertainty* to broadly describe, in temporal terms, the shortcomings present in sensors, system models, and actuator commands. Thus as the *uncertainty* of a reactive system increases, the frequency of the sense/react loop responsible for controlling the system must correspondingly increase. In this framework, a desired system behavior can only be assured when the period of the sense/react loop reduces the uncertainty in the system so that it falls within acceptable limits. The tolerances of models and capabilities of actuators define these limits.

The sensors and actuators in a reactive system can be modeled, respectively, by a vector of monitored variables  $\vec{m} \stackrel{\text{def}}{=} (m_1, m_2, \dots, m_n)$  and controlled variables  $\vec{c} \stackrel{\text{def}}{=} (c_1, c_2, \dots, c_k)$ . For a more detailed treatment, see Janicki et al [7]. We extend the vector  $\vec{m}$  with additional variables describing the information obtained from the trace history of the system. We denote the resulting vector  $\vec{\mathcal{M}}$ . We say that  $\vec{\mathcal{M}}$  describes the monitored state of the system. The combination of  $\vec{\mathcal{M}}$  and  $\vec{c}$  describes the observable state of the system. Let  $M$  and  $C$ , respectively, denote the sets of all possible configurations of the extended monitored and controlled variables that are allowed by the environmental constraints. The set  $S \stackrel{\text{def}}{=} \{(\vec{\mathcal{M}}, \vec{c}) \mid \vec{\mathcal{M}} \in M \wedge \vec{c} \in C\}$  then describes the observable state space of the system. Given such a definition of a state, a *system model* can be constructed by defining the effect of a control value on any observable system state (i.e., how a control value can cause the system to transition from one state to the next).

## 2.2 A Domain Modeling Language

Our development makes use of a domain specific modeling language called SPC, an acronym standing for **s**tate, **p**rofile, and **c**onstraint. SPC is not a full-blown specification language; instead it simply provides primitives in a functional and relational framework enabling control function developers to construct abstractions suited for concisely modeling reactive systems and formally specifying their control functions. SPC encourages a relational perspective of computation[2]. For example, rather than returning a single control vector as the result of its computation, a relational control function might return the set of all control vectors satisfying the specification (e.g., all acceleration values for a train that meet the safety properties specified). The resulting control vector set is then passed to a filter function which selects an appropriate vector according to some other competing criteria (e.g., the maximum acceleration, a train acceleration which best maintains passenger comfort, etc.). When considered in the context of the BART case study, a relational design allows a clean separation of safety related constraints from non-safety related constraints. The decoupling of these constraints enables separation of concerns allowing the specification of train behavior to be decomposed into two smaller/simpler specifications, and whose

composition can easily be shown to satisfy safety properties.

Another goal behind the development of SPC was to construct a framework in which formal specifications could be automatically manipulated in *problem specific* ways. Problem specific manipulation prohibits the application of rigid compiler techniques to carry out the translation from specification to implementation. On the other hand, problem specific manipulation can be readily performed by a program transformation system.

From the perspective of modeling and specification, SPC can be seen as a small programming language whose syntax and semantics has a functional flavor with mathematical overtones. From the perspective of implementation, SPC can be seen as a wide-spectrum language with strong emphasis on referential transparency. A BNF grammar for the entire wide-spectrum language SCP can be found in Appendix D.

An SPC specification is typically a set of definitions where each definition has one of the following forms:

```
define data_abstraction = expression
define data_abstraction = expression where definition-list
```

In the second form, the keyword *where* has the same semantics as it does in standard mathematics. The left-hand side of the equality in a definition can be a single symbol or it may be a tuple. The expression on the right side of the equality operator can be a number of things including an anonymous function, a constant, a variable, or a data\_abstraction.

A control function in a typical reactive system has the following form:

```
define control_function_core =
    ( fn system_state =>
      select( [ actuator_values : constraints ] )
      where (
        define constraints = ...
        define select = ...
      )
    )
```

The above takes a *state* as its input and returns an actuator value that has been *selected* from a set of actuator values satisfying the given *constraints*. At the specification level, a key difficulty is the definition of the system constraints. How constraints are specified is heavily effected by the system model. As a result, their correctness is dependent upon the correctness of the model.

### 2.2.1 The State, the Next State Function, and the Profile

A *state* is simply a tuple of variables where each variable is quantified over a corresponding range of sensor values, trace information, or actuator values.

$$state = (x_1 \in D_1, x_2 \in D_2, \dots, x_m \in D_m)$$

where  $D_i =$  a range of sensor values, a range of trace information values, or a range of actuator values.

The *next state* function calculates the next state after every sense/react step of the controller. The sense/react period precisely defines at which times, the control of the system is possible. Because of this limitation on control, it is not particularly useful to construct a system model in which actuator should be output at times that are inconsistent with (i.e., do not align with) the sense/react cycle.

A *profile* is a finite sequence of states that either can be constructed explicitly or iteratively via the next state function. It is specified using an ML like list notation:

$$profile = [s_1, \dots, s_k]$$

An important yet subtle aspect of profiles is their indexing mechanism can be overloaded with respect to time. Specifically, models can be constructed in which profiles are accessed by indexes that correspond to discrete clock ticks as defined by a universal clock. In the BART case study, this temporal indexing allows the position of two trains to be compared in a meaningful way – It is not as important to know **where** a train is as it is to know **when** it is there (e.g., two trains can only collide if they are in the same position at the same time).

A profile can be iteratively defined by passing a triple of the form: *state*  $\times$  *next\_state*  $\times$  *pred* to an SPC function called *list\_mu*. The result of this evaluation ( $s, f, Q$ ) is a profile, starting with  $s$  as the initial state of the profile, the function  $f$ , applied to any state, produces the next state according to the state transition relation, and  $Q$  is a first-order formula indicating when a desired state is reached. In particular,

$$list\_mu(s, f, Q) = [f^0(s), \dots, f^n(s)]$$

where  $f^0(s) = s \wedge f^{i+1}(s) = f(f^i(s)) \wedge \forall i : 0 \leq i < n \rightarrow \neg Q(f^i(s)) \wedge Q(f^n(s))$ . In the BART case study, various stopping behaviors that a train might have (e.g., emergency stop or normal stop) are computed using this construct.

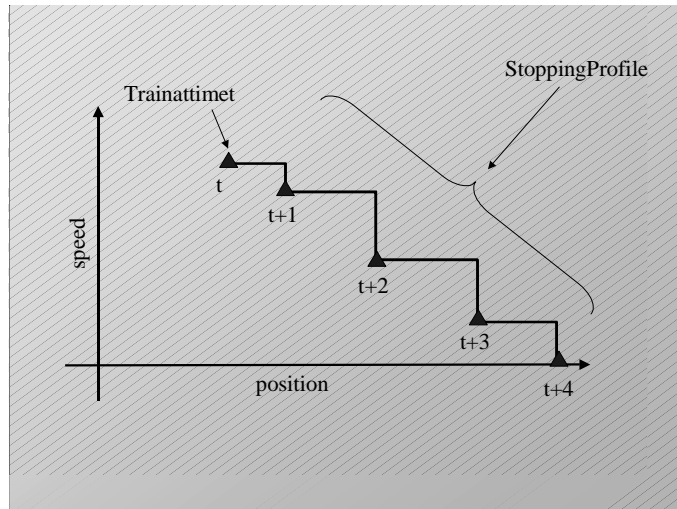


Figure 1: A stopping profile for a train

See the Appendix A for the semantics of *list\_mu*.

## 2.2.2 Constraints

A *constraint* is an expression comparing two profiles. If the expression evaluates to true, the constraint is satisfied, otherwise the constraint is not satisfied. A special-purpose operator,  $\ll$ , is used to compare profiles. A fully general parameterization of this operator is still under development. At present, two useful instances of  $\ll$  have been identified. The first instance, which will be denoted  $\ll_{\mathcal{T}}$  can be used to compare profiles sharing a common time frame (e.g., two train profiles). The second instance, which will be denoted  $\ll_{\mathcal{S}}$  can be used to compare profiles in a temporally independent fashion.

### (Temporally) Dependent Constraint

$$tp_{ot} \ll_{\mathcal{T}} tp_{lt} \stackrel{def}{=} \forall t \in \text{int}(\text{size}(tp_{ot})) : p_1 < p_2 \text{ where } (p_1, s_1, a_1) = tp_{ot}[t] \text{ and } (p_2, s_2, a_2) = tp_{lt}[t]$$

In the definition given,  $tp_{ot}$  and  $tp_{lt}$  denote profiles whose states share a common time frame. The expression  $\text{int}(\text{size}(tp_{ot}))$  calculates the cardinality  $n$  of  $tp_{ot}$  and then creates a the set  $\{1, 2, \dots, n\}$  over which the variable  $t$  is then universally quantified.

Let  $r$  denote a profile,  $i$  an index, and  $r(i)$  the  $i^{\text{th}}$  element in  $r$ . Given these assumptions, the semantics of profile access by  $[ ]$  can be defined as follows:

$$r[t] = \begin{cases} r(t) & \text{when } \text{size}(r) \geq t \\ r(\text{max\_index}) & \text{when } \text{size}(r) < t \text{ and } \text{max\_index} = \text{size}(r) \end{cases}$$

In the definition of  $tp_{ot} \ll_{\mathcal{T}} tp_{lt}$ , the expression  $tp_{ot}[t]$  simply denotes the  $t^{\text{th}}$  element of  $tp_{ot}$ . Potential problems exist only when using  $t$  to access elements of  $tp_{lt}$ . The definition of  $[ ]$  allows for a meaningful comparison between elements in  $tp_{ot}[t]$  and elements in  $tp_{lt}[t]$  even in cases where  $\text{size}(tp_{lt}) < \text{size}(tp_{ot})$ . Similar examples of such *normalizing* extensions can be found in SequenceL [4].

The intuition behind the extension is that the temporally dependent profiles considered are generated using a next function and have a last state satisfying a particular predicate. Such a “satisfying” state may be repeated without changing the semantics of the profile.

The key characteristic of a dependent constraint is that multiple profiles (e.g.,  $tp_{ot}$  and  $tp_{lt}$ ) are accessed by a single **shared** variable (e.g.,  $t$ ).

### (Temporally) Independent Constraint

$$tp_{ot} \ll_{\mathcal{S}} pf \stackrel{def}{=} Q_0(tp_{ot}, pf) \wedge Q_1(tp_{ot}, pf) \wedge Q_2(tp_{ot}, pf)$$

**where**

$$Q_0(tp_{ot}, pf) \stackrel{def}{=} \forall t \in \text{int}(\text{size}(tp_{ot})) \forall j \in \text{int}(\text{size}(pf)) : (p_1 = p_3) \rightarrow (s_1 < s_3)$$

**where**  $(p_1, s_1, a_1) = tp_{ot}[t]$  and  $(p_3, s_3) = pf[j]$

$$Q_1(tp_{ot}, pf) \stackrel{def}{=} \forall t \in \text{int}(\text{size}(tp_{ot}) - 1) \forall j \in \text{int}(\text{size}(pf)) : (p_1 < p_3 < p_2) \rightarrow (\max(s_1, s_2) < s_3)$$

**where**  $(p_1, s_1, a_1) = tp_{ot}[t]$  and  $(p_2, s_2, a_2) = tp_{ot}[t + 1]$  and  $(p_3, s_3) = pf[j]$

$$Q_2(tp_{ot}, pf) \stackrel{def}{=} \forall t \in \text{int}(\text{size}(tp_{ot}) - 1) \forall j \in \text{int}(\text{size}(pf) - 1) : (p_3 < p_1 < p_4) \rightarrow \max(s_1, s_2) < s_3$$

**where**  $(p_1, s_1, a_1) = tp_{ot}[t]$  and  $(p_2, s_2, a_2) = tp_{ot}[t + 1]$  and  $(p_3, s_3) = pf[j]$  and  $(p_4, s_4) = pf[j + 1]$

The key characteristic of an independent constraint is that each profile is accessed by its own **private** variable.

### 2.2.3 Revisiting a Control function

The reader would notice that each of the constructs needed to define a control function is now in place. For a specific reactive system, it becomes necessary to define states, constraints, and the next state function. Bear in mind that a typical reactive system may transition to several states at any given time. Different transitions generally are the result of actuator values that can be selected by the control function or are due to environmental conditions beyond the control of the system.

```
define control_function_core =  
    ( fn system_state =>  
      select( [ actuator_values : constraints ] )  
      where (  
          define constraints = ...  
          define select = ...  
      )  
    )
```

## 2.3 Executable Specification through Transformations

After a system model has been developed and a control function specified, it is transformed into an executable specification (program) that can be executed. Such programs should not be expected to be very efficient since they are the product of a direct translation from a specification whose primary design goal was clarity (not efficiency). For efficiency reasons, it is therefore necessary to restructure a specification during the course of its translation into a program. Furthermore, in a typical development cycle program execution may produce insights that are fed back to the specification, requiring revision of the specification and the (re)generation of another program. Because of such development cycles, it is desirable to automate the program generation phase to the extent possible. Many factors must be considered when determining how to automate and what to automate in the program generation phase.

When developing software for high consequence systems (such as the BART system), the ability to provide a convincing argument that the software has been built correctly is of utmost importance. Generally, such an argument is broken down into two smaller arguments. The focus of the first argument is to provide convincing evidence that the software system has been specified correctly. The goal of the second argument is to provide convincing evidence that a software implementation satisfies its specification. If both arguments can be made, then the software system is certified to be correct.

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,  $\mathcal{P}$ , corresponds to exhaustive testing of the systems 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 [3][6][12]. 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 assurance that the system will not experience a failure.

### 2.3.1 Transformation

A syntax-based (program) transformation system can be viewed as a rewrite system with *refinement* [10][11], denoted by  $\sqsubseteq$ , as the rewrite relation. The expression  $s \sqsubseteq t$  asserts that  $s$  is *refined* by  $t$ , or  $t$  is *correct* with respect to  $s$ . In an automated transformation framework, deriving an implementation,  $\mathcal{S}_n$ , from a specification,  $\mathcal{S}_0$ , proceeds in five steps:

1. An initial refinement relation,  $\rightarrow_{\sqsubseteq}$ , is defined that is based on general refinement knowledge combined with basic (fundamental) problem domain knowledge. Generally,  $\rightarrow_{\sqsubseteq}$  can be (re)used 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 function,  $\mathcal{T}$ .
5.  $\mathcal{T}$  is applied to  $\mathcal{S}_0$  yielding  $\mathcal{S}_n$  as its result. The program  $\mathcal{S}_n$  is the “most refined” object that can be obtained from  $\mathcal{S}_0$  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  $\sqsubseteq$ , it follows that  $\mathcal{S}_0 \sqsubseteq \mathcal{S}_n$ . Under the assumption that  $\mathcal{T}$  is executed correctly, simply proving the correctness of the individual transformations in  $\mathcal{T}$  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  $\mathcal{T}$  to  $\mathcal{S}_0$ , as well as a correspondence between intermediate forms,  $\mathcal{S}_{i+1}$ , and which transformation, in  $\mathcal{T}$ , was applied to  $\mathcal{S}_i$  in order to obtain  $\mathcal{S}_{i+1}$ .

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.

### 2.3.2 HATS

The High-Assurance Transformation System (HATS) [13] is a language-independent program transformation system whose development began in the late 1990s at Sandia National Laboratories and continues at the University of Nebraska at Omaha as well as the University of Texas at El Paso. The goal of HATS is to provide a transformation-oriented programming environment, facilitating the development of transformation rules and strategies whose correctness can be formally verified. The Figure 2 shows the HATS architecture.

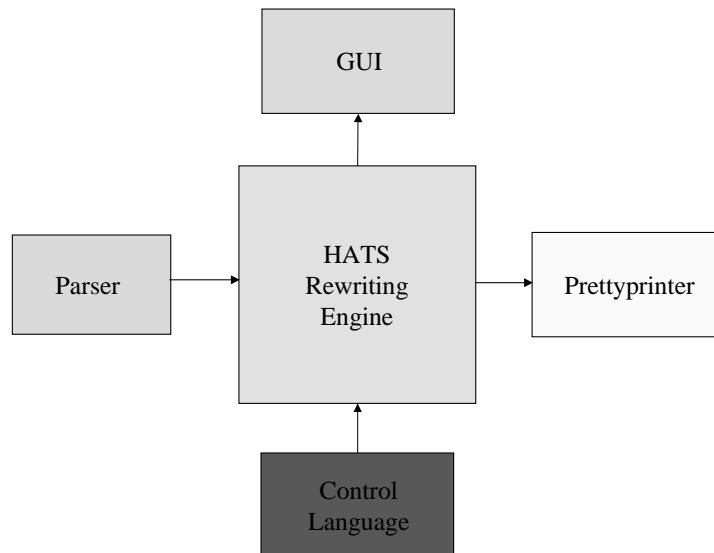


Figure 2: Hats Architecture

In HATS, programs belonging to a particular problem domain are defined by a context-free grammar. Internally, HATS stores and manipulates these programs as syntax derivation trees. HATS provides a special purpose language for defining transformation rules and strategies. This language enables the description of the domain language as a context-free grammar.

## 3 Inputs taken from the BART case study

### 3.1 Inputs Taken

- Trains within a track region are controlled by a centrally located computer.
- Environment may introduce non-deterministic behaviors into the system in the form of derailments.

- A track region is cycle-free.
- A track region is uni-directional.
- Trains may not move backwards.
- The control function core will be embedded in a sense/react loop whose period is 0.5 seconds.
- A track region may contain multiple trains.
- Trains may have different views of signal states (e.g., red vs green).
- Trains have two types of stopping modes, normal and emergency.
- A track segment is a fixed length of track whose position and speed are also fixed.
- Track consists of a collection of track segments.

### 3.2 Safety Policy

- Failures
  - Trains should not collide with one another.
  - Trains must stop at all signals (and stations) when required to do so.
- Hazards
  - An object train should not get so close to its lead train that it would be unable to stop should the lead train unexpectedly derail.
  - At no time should a train exceed the speed limit on the track segment on which it is traveling.
  - A train may only issue an emergency stop command when (1) the system is in an abnormal state, and (2) at no time upon entry of the system into the abnormal system state, was it possible for the train to stop using a normal stopping profile.
- Other
  - In an abnormal state, a train should advance as far as possible.

**Remark 1** *It is worth mentioning that the term **track segment** as it is used in the BART case study has a different semantics from how it is sometimes used in the context of other train systems. In other systems, “track segments” are part of the interlocking system, and it is the interlocking system that is charged with enforcing many of the systems safety properties. For example, the interlocking system is oftentimes charged with the responsibility of preventing a train from entering a fixed “track segment” in situations where the segment is occupied by another train. In such a framework, the control function for a train need not worry about getting too close to its lead train, as this aspect of the safety policy is entrusted to the interlocking system. The drawback of such a solution is that it may be far from optimal, since the “track segments” are fixed regions on the track. In contrast, the control function presented does not rely on the interlocking system to assure such safety properties. The control function specified and implemented dynamically computes how close a train can be to a leading train while still preserving the properties set forth in the safety policy. Conceptually, one can think the control function as computing a dynamically changing virtual segment that exists between a train and the train in front of it. This virtual segment then defines a hazardous region which if entered, would constitute a violation of the safety policy.*

### 3.3 Assumptions

- System
  - The system provides every object train with access to the state of its lead train.
  - If a train derails, then the system is in an abnormal state, otherwise the system is in a normal state.
  - The positions of all trains are known at all times, even in the case of a derailment.
  - Acceleration commands are given to trains in a synchronous fashion. The assumption that trains are controlled in a synchronous fashion is central to correctness of the control function. Several safety properties are specified in terms of temporally dependent constraints (discussed in Section 2.2.2) involving the stopping profiles of an object train and its corresponding lead train. It is only through synchronous control that such constraints can be evaluated in a manner that is meaningful to physical reality.
- Environment
  - A derailment will cause a train to come to an instaneous halt.
  - No assumptions are made regarding when a derailment might occur.
  - No assumptions are made regarding interlocking system’s decision procedure for changing the state of a signal.
  - The interlocking system will only require a train to stop at a signal when, in fact, the train is able to do so.
  - Our interpretation of the BART case study allows the interlocking control system to transmit different signal states to different trains. For example, if a train is too close to a signal to

stop, the interlocking control system will tell the train that the signal is green. However, at the same time, if another train is able to stop for the signal, the interlocking system may tell this train that the signal is red (i.e., the gate is closed). This implies that each train will receive its own (possibly distinct) description of signal states.

- Train Behavior

- An emergency stop is considerably quicker than normal stop. (See Appendix B for a discussion on this.)
- The control function has access to auxiliary functions capable of generating both normal as well as an emergency stopping profiles. These functions make use of a next-state function whose period corresponds to the sense/react cycle of the system.
- In any given state, there exists a (possibly empty) set of normal as well as emergency acceleration values that are physically possible. In general, only a subset of the physically possible accelerations will satisfy the safety policy.
- The control function, not the interlocking system, is responsible for
  - \* ensuring that trains do not collide with one another
  - \* ensuring that trains do not get so close to one another that hazardous conditions exist.

**Corollary 1** *Train behaviors need only be compared in a pairwise fashion from the perspective of the safety policy as well as the optimization policy.*

**Corollary 2** *In order to satisfy the safety policy, a train does not need to know when the system is in an abnormal state.*

### 3.4 Abstractions and generalizations

- Train has no length. It is worth mentioning that only a minor extension to the model presented would be required to capture the length of a train.
- The state of a train at a given moment in time is modeled by a triple consisting of its position, speed, and acceleration.
- Normal acceleration values are modeled as a set of typed values.  $Normal = \{a_1, a_2, ..a_n\}$
- Emergency acceleration values are modeled as a singleton set of typed values.  $Emergency = \{e\}$
- Access to the normal acceleration values is provided by the function:  $\mathcal{N\_A} : speed \times acceleration \rightarrow set\ of\ acceleration\ values$ . Access to the emergency acceleration value is provided by the function  $\mathcal{ES\_A} : unit \rightarrow e$ .
- Track segments can be modeled by a (position, speed, length) tuple.

- Track consists a sequence of (position, speed) tuples monotonically increasing on position and whose final tuple is  $(p,0)$  where  $p$  is the last position on the track. In this model the length component is redundant and is therefore dropped.
- The set of physically possible train accelerations is a function of the train's current speed and acceleration.
- The track is modeled as a profile of position-speed tuples, where each tuple describes a track segment. In order for a track profile to be well-formed, its track segments must be monotonically increasing on position. Consider two adjacent tuples  $(p_1, s_1)(p_2, s_2)$  belonging to a track profile. We use the term *track segment* to denote the region of the track that begins at  $p_1$  and goes up to (but does not include)  $p_2$ . All positions on the track belonging to the track segment beginning at  $p_1$  have a speed limit of  $s_1$ .
- Signals are modeled in a fashion similar to track segments. In fact, one can think of a signal as defining a speed limit, that varies over time for a particular region of the track. When the signal is red the speed limit for the corresponding track region is 0 mph, and when the signal is green, the speed limit for the corresponding track region is unconstrained and can be modeled by the maximum speed limit for the track (e.g., 80 mph in the case of the BART case study).
- The system modeled consists of three components, a track, a collection of signals, and a collection of trains. A track is modeled as sequence of track segments that are monotonically increasing on position. In turn, track segments are described by position-speed tuples. For the sake of completeness, the track model requires that the final track segment in a profile have a speed limit of 0 mph. This final track segment serves as an end of track marker and when taken together with the track speed limit safety constraint will ensure that a train control function never try to drive a train past the end of the track.

## 4 Applying the approach to the case study

### 4.1 Modeling the Track and Signals

Below is a fragment of the track layout table taken from the BART case study. Following this table is a profile describing the track and a profile describing the current state of the gate signals. Recall that the state of the gate signals is relative to a particular train.

Notice that the track model and the signals model has the final segment  $(8722,0)$ . As discussed earlier, such a segment should be added to the end of the track and signals for safety reasons. In this small example, the assumption is that the physical track ends at position 8722.

Figure 4 shows graphical representation of the track model as seen from our simulator. The region below the track speed limit is considered safe with respect to the track speed limit safety property. Conversely, the region above the track speed limit is unsafe.

Begin (feet)	Segment		Comment	Civil Speed (mph)	Grade	Exposure
	End (feet)	Length (feet)				
5940	6640	700	Daly City Station Platform	36	-0.80%	Open
6640	6741	101		36	-0.80%	Open
	6741		Gate - currently displaying GREEN			
6741	7588	847	Switches to Crossover and Spur	36	-0.80%	Open
	7588		Gate - currently displaying RED			
7588	8522	934		36	-0.80%	Open
8522	8722	200	Parabolic grade transition	80	2.75%	Open

Track Model = [(5940, 36), (6640, 36), (6741, 36), (7588, 36), (8522, 80), (8722, 0)]

Signals Model = [(6741, 80), (7588, 0)(8722, 0)]



Figure 4: Display of Track Model

## 4.2 Model of Train Behavior

$next(f) \stackrel{def}{=} \text{fn } (p, s, a) \Rightarrow (p + s + \frac{1}{2}a', s + a', a') \textbf{ where } a' = f(p, s, a)$ $Accel_1 \stackrel{def}{=} \text{fn } (p, s, a) \Rightarrow \min_{set}(\mathcal{N}_{\mathcal{A}}(s, a))$ $Accel_2 \stackrel{def}{=} \text{fn } (p, s, a) \Rightarrow \mathcal{ES}_{\mathcal{A}}()$ $Q((p, s, a)) \stackrel{def}{=} s = 0 \wedge a = 0$ $nStop(state) \stackrel{def}{=} list\_mu(state, next(Accel_1), Q)$ $eStop(state) \stackrel{def}{=} list\_mu(state, next(Accel_2), Q)$ $dStop((p, s, a)) \stackrel{def}{=} [(p, 0, 0)]$
---

In the above table, the *next* function defines how a train transitions from one state to another. Two acceleration functions, *Accel*<sub>1</sub> and *Accel*<sub>2</sub> are then defined that respectively select minimum normal and emergency accelerations values. The predicate *Q* defines what it means for a train to come to a halt, and the functions *nStop*, *eStop*, and *dStop*, respectively define a normal stopping profile, an emergency stopping profile, and the profile resulting from a derailment.

## 4.3 Specification of Safety Policy

From the perspective of the safety policy, the fundamental question that must be repeatedly asked of a train is: “How long will it take to stop or slow down in a particular mode?” Given a train state and a model describing train behavior, it is possible to answer this type of question by calculating a *stopping profile*. A stopping profile consists of a sequence of train states in which the final state is of the form  $(p, 0, 0)$ . That is, a behavior that results in the train coming to a halt. Additionally, we require that a stopping profile be minimal in the sense that, for a given mode, it is not possible to select alternate acceleration values that would cause the train to stop (or slow down) in a shorter distance.

Given this definition of a stopping profile, the strategy for the control function can be simply stated:

<i>Continue accelerating so long as the stopping profiles satisfy the corresponding safety constraints</i>
--

Clearly, as long as this property holds the train’s behavior will be safe. The question that arises at this time is: “What does it mean for a train to satisfy the safety constraints?”

Analysis of the safety property that the *object train* under consideration should not collide with its *lead train* (i.e., the train immediately in front of it) concludes that the stopping profiles of trains may be compared locally in a pairwise fashion. In particular, global state is not considered when determining whether this safety property holds. Global considerations are generally not useful due to the fact that limited assumptions can be made regarding the future behavior of any train. First of all, the environment can cause a train to derail at any moment. If in addition, an emergency stop takes longer than a normal stop, then no further analysis need be performed since the emergency stop distance is the limiting factor. However, as the length emergency stop gets smaller and hence not longer is the limiting factor, additional limiting factors must be considered. For example, it is not unreasonable to assume that a train may slow down for a variety of unanticipated reasons. An example of this can be found in the BART system where a train will initiate an emergency stop if it fails to receive a valid command.

Additionally, it may be prudent to allow a train operator to initiate a braking sequence (e.g., if a car is stalled on the track, etc.). Due to this uncertainty, an object train must conservatively assume that at any given time step, its lead train will do one of three things: (1) initiate a normal stop, (2) initiate an emergency stop, and (3) derail. Failure to make such assumptions open the door to worst case scenarios. A conservative view of the kind described does not benefit from additional information present in the global state. The reason being that the behavior of a lead train, when considered independently from the remaining trains in the system, is the limiting factor for every corresponding object train.

Formally, we write

$$\text{object\_train\_profile} \ll_{\mathcal{T}} \text{lead\_train\_profile}$$

to denote that the given object train profile satisfies the constraints placed upon it by the given lead train profile. The  $\ll_{\mathcal{T}}$  operator denotes a position-time comparison and is defined in Section 2.2.2.

To capture safety constraints with respect to the track and signals, we write

$$\text{object\_train\_profile} \ll_{\mathcal{S}} \text{track\_or\_signals}$$

to denote that the given object train profile satisfies the constraints of the given track or signals profile. In this equation, the  $\ll_{\mathcal{S}}$  denotes a position-speed constraint and is defined in Section 2.2.2.

#### 4.3.1 Model of Safe State

$\text{SafeState}(\text{state}_{ot}, \text{state}_{lt}, \text{track}, \text{signals}) \stackrel{def}{=} \\ n\text{Stop}(\text{state}_{ot}) \ll_{\mathcal{S}} \text{track} \wedge \\ n\text{Stop}(\text{state}_{ot}) \ll_{\mathcal{S}} \text{signals} \wedge \\ n\text{Stop}(\text{state}_{ot}) \ll_{\mathcal{T}} \text{currentStop}(\text{state}_{lt}) \wedge \\ e\text{Stop}(\text{state}_{ot}) \ll_{\mathcal{T}} e\text{Stop}(\text{state}_{lt}) \wedge \\ e\text{Stop}(\text{state}_{ot}) \ll_{\mathcal{T}} d\text{Stop}(\text{state}_{lt})$
---

The safety policy is defined in terms of a *SafeState* and *SafeTransition* predicate. The *SafeState* predicate can be used to determine if a given train, the *object train*, is presently in a safe state. The *SafeState* predicate holds for a given object train when (1) its normal stop profile satisfies the track speed limits, (2) its normal stop profile satisfies the states of the signals presented to it by the interlocking system, (3) its normal stop profile satisfies the current stopping profile of the train immediately in front of it (i.e., its *lead train*), (4) its emergency stop profile satisfies the emergency stop profile of its lead train, and (5) its emergency stop profile satisfies the derail stop profile of its lead train.

Intuitively, if the normal stop profile of the object train satisfies the track profile, this means that normal acceleration values exist which the control function for the object train can use to assure that the speed limit of the track will not be violated. A similar explanation describes what it means for the object train to satisfy the signals profile.

The third constraint in the *SafeState* predicate is the most interesting. It is this constraint that when taken together with the *SafeTransition* predicate assures that a train never needlessly issue an emergency stop command and that the train advance as far as possible in an abnormal environment.

In a normal environment, the stop behavior for the lead train will be a normal stop. In an abnormal environment, the stopping behavior for the lead train may be either a normal stop, an emergency stop, or a derail stop. It is when this constraint get tripped that the object train will issue an emergency stop.

The fourth and fifth constraints state that the object train emergency stop profile should always satisfy the emergency stop and derail profiles of the lead train.

### 4.3.2 Model of Safe Transition

$$\begin{aligned}
 & \text{SafeTransition}(state1_{ot}, state2_{ot}, state1_{lt}, track, signals) \stackrel{def}{=} \\
 & \quad \text{SafeState}(state1_{ot}, state1_{lt}, track, signals) \rightarrow \text{SafeState}(state2_{ot}, state1_{lt}, track, signals) \wedge \\
 & \quad \neg \text{SafeState}(state1_{ot}, state1_{lt}, track, signals) \rightarrow (is\_type(a') = emergency\_stop) \\
 & \quad \textbf{where } state2_{ot} = (p', s', a')
 \end{aligned}$$

The *SafeTransition* predicate defines the conditions under which an object train may transition to its next state. The parameter  $state1_{ot}$  denotes the current state of the object train, and the parameter  $state2_{ot}$  denotes the state to which the object train may wish to transition. If  $state1_{ot}$  is a safe state, this means that the object train can satisfy all of the safety constraints without needlessly availing itself of its emergency stop. Thus there exists at least one transition out of  $state1_{ot}$  satisfying this condition. Now the transition to  $state2_{ot}$  may or may not be such a transition. If it is, then it will also satisfy the *SafeState* predicate.

On the other hand, if  $state1_{ot}$  does not satisfy the *SafeState* predicate, then the transition to  $state2_{ot}$  must be the result of an emergency stop acceleration. That is, in this situation the object train is forced to issue an emergency stop.

### 4.4 Specification of Controller

$$\begin{aligned}
 & \text{Control}_i(state_{ot}, state_{lt}, track, signals) = select( [\mathcal{N\_A}(s, a) : Acc] ) \\
 & \textbf{where } ( \\
 & \quad (p, s, a) = state_{ot} \\
 & \quad ) \\
 & \text{select} \stackrel{def}{=} \text{if } S = \text{SET\_EMPTY} \text{ then } \mathcal{E}\mathcal{S\_A}() \text{ else } \text{SET\_MAX}(S) \\
 & \text{Acc} = \text{fn } a' \Rightarrow \text{SafeTransition}(state_{ot}, next\_accel(state_{ot}), s_{lt}, track, signals) \\
 & \quad \textbf{where } ( \\
 & \quad \quad next\_accel = next(\text{fn } state \Rightarrow a') \\
 & \quad )
 \end{aligned}$$

Give a train model and the definition of *SafeState* and *SafeTransition*, the specification of the control function is quite simple. The control function calculates all of the normal stop acceleration

values capable of satisfying the safety constraints, and the select function then selects the maximum value from this set. However, if none of the normal accelerations are safe, the select function will select an emergency stop acceleration value.

## 4.5 Transformation

In this section, highlights are given of various transformation steps used to derive an ML program from the given specification. Transformation will be discussed at the conceptual level to the extent possible. That is, rather than inundate the reader with the syntactic details of a particular transformation system (such as HATS), transformational ideas are discussed at the abstract level. Those interested in the details surrounding the implementation of such transformational ideas in HATS are encouraged to read [13][16].

Strategy for obtaining an executable specification:

1. Unfold all definitions.
2. Apply transformations that lower the level of abstraction of various domain constructs.
3. Apply transformations that replace primitive constructs and data with equivalent ML constructs and data.

### 4.5.1 Unfolding Definitions in the Specification

Unfolding definitions within the specification, in effect, requires the interpretation of such definitions as transformations in their own right. This can present somewhat of a challenge for domain independent transformation systems as it is somewhat of a meta-operation. In HATS, this type of meta-operation is supported via a construct known as a *dynamic transformation*. Dynamic transformations are transformations with free variables that can be instantiated with respect to a specific problem instance. The most common use of this type of transformation is for variable renaming. However, in the context of the train control problem it is used to perform macro-unfolding. For a more detailed discussion of dynamic transformations see Winter et al [16].

### 4.5.2 Lowering the Level of Abstraction

Transformation 1 takes advantage of the fact that all sets in SPC are finite.

Transformation 1:

$$\forall x \in S : P(x) \implies [S : P] = S$$

Note that for finite sets, an expression such as  $[S : P] = S$  can be effectively computed.

Transformations 2 through 5 build upon one another for the purpose of achieving a specific objective. Their combined application places any program they are applied to (e.g., the control function specification) into a *normal form*. In the transformation community, *normal form* and *canonical form* are terms frequently used when referring to intermediate stages in a transformation sequence that establish interesting properties. In the case of transformations 2 through 5, the interesting property established

is that the profile accessing construct  $[ ]$  defined in Section 2.2.2 has been removed from all filter expressions. Note that the transformations shown assume that a single profile is being accessed. In such cases, the normalization problems discussed in Section 2.2.2 are not encountered.

Transformation 2 simultaneously factors the *int* and *size* functions out of filter expressions. This transformation is used to change from a data representation where elements of profiles are accessed by indexes to a representation where element access will ultimately be realized by the list accessor function *first*. For example,  $S[1] = \text{first}(S)$ .

Transformation 2:

$$\begin{aligned} & [\text{int}(\text{size}(S)) : (\text{fn } t \Rightarrow F(S[t]))] = \text{int}(\text{size}(S)) \\ & \implies \\ & \text{int}(\text{size}([S : (\text{fn } x \Rightarrow F(x))])) = \text{int}(\text{size}(S)) \text{ when all occurrences of } t \text{ in } F \text{ are in subexpressions of} \\ & \text{the form } S[t] \end{aligned}$$

It is important to note, that Transformation 2 should be applied only when  $t$  is used as an index within  $F$ . In HATS, such checks can be described by boolean expressions that specify the application condition for a transformation. In general, a HATS transformation may only be applied when its application condition evaluates to true. For complex conditions of the kind required by Transformation 2, dynamic transformations have proven themselves to be quite useful.

Transformation 3 is based on a simple observation that  $\text{zip}(S, \text{tail}(S))$  produces a list of tuples whose size (i.e., length) is one less than the size of  $S$ .

Transformation 3:

$$\text{int}(\text{size}(S) - 1) \implies \text{int}(\text{size}(\text{zip}(S, \text{tail}(S))))$$

Transformation 4 is similar to Transformation 2. The difference is that Transformation 4 allows  $t$  to occur as an accessor to  $S$  in two different ways:  $S[t]$  and  $S[t - 1]$ . This type of transformation is useful for lowering the level of abstraction for the temporally independent constraints discussed in Section 2.2.2.

Transformation 4:

$$\begin{aligned} & [\text{int}(\text{size}(\text{zip}(S, \text{tail}(S)))) : (\text{fn } t \Rightarrow F(S[t], S[t - 1]))] = \text{int}(\text{size}(\text{zip}(S, \text{tail}(S)))) \\ & \implies \\ & \text{int}(\text{size}([\text{zip}(S, \text{tail}(S)) : (\text{fn } (x1, x2) \Rightarrow F(x1, x2))])) \text{ when all occurrences of } t \text{ and } t - 1 \text{ in } F \text{ are in} \\ & \text{subexpressions of the form } S[t] \text{ and } S[t - 1]. \end{aligned}$$

And finally, Transformation 5 simply removes any unnecessary *int* and *size* function applications.

Transformation 5:

$$\text{int}(\text{size}([S : F])) = \text{int}(\text{size}(S)) \implies [S : F] = S$$

### 4.5.3 Implement base functions and data in ML

The transformations in this phase are quite straightforward. A primary reason for this is the conceptual compatibility between SPC and ML. Examples of various syntactic driven transformations are shown below.

1.  $F \rightarrow G \implies \text{if } F \text{ then } G \text{ else true}$
2.  $F \text{ where } \textit{definition\_list} \implies \text{let } \textit{definition\_list} \text{ in } F \text{ end}$
3.  $\text{and} \implies \text{andalso}$
4.  $\text{MIN} \implies \text{Real.min}$
5.  $\text{MAX} \implies \text{Real.max}$
6.  $\text{SET\_EMPTY} \implies \text{nil}$
7.  $\text{SET\_MAX}( S ) \implies \text{List.foldl Real.max Real.negInf } S$
8.  $\neg \implies \text{not}$
9.  $[S : F] = S \implies \text{List.all}(F,S)$
10.  $\text{zip}(S1,S2) \implies \text{ListPair.zip}(S1,S2)$

It should be noted that it is in this stage where the choice of the implementation language becomes fixed. For example, if a decision was made to transform the control function specification to C instead of ML, then only the transformations in this stage would need to be changed.

## 4.6 Simulation and Animation

### 4.6.1 Additional Assumptions

- There exist no speed or length dependencies between track segments. For example, it is realistic for a simulator to use a random function to generate the speed and length of track segments.
- Signal states may be randomly generated, provided that trains are only asked to stop for a signal when they are able to do so.

### 4.6.2 The Simulator

Initially, in order to validate the control function as well as try out different multi-train scenarios, a simulator was built. The simulator was written (by hand) in ML and models the track, the interlocking system, the environment, as well as unpredictability of the lead train. The simulator is initialized by:

- generating  $n$  track segments having a length and speed that randomly falls between given parameters,

- composing the track segments to form a track,
- generating  $m$  signals spaced at random intervals on the track,
- selecting the number of trains to be controlled by the control function, and
- selecting various randomness parameters for influencing the behavior of the lead train of the system.

The aggregation of these components forms the system state, which is given as input to the sense-react loop. The sense-react loop proceeds by calling the control function with the appropriate state information for each train. The lead train of the entire system is controlled by a special control function which differs from the developed control function two ways:

1. Where the developed control function selects the maximum acceleration from the computed set of safe acceleration values, the system lead train pseudo-randomly selects an acceleration value from this set.
2. The system lead train will occasionally derail thereby occasionally forcing the following trains to switch from a normal stopping mode to an emergency stopping mode.

A parameter,  $k$ , is used to control the period in which signals can be changed. That is, every  $k$  sense-react cycles, the signal states are randomly changed. And finally, after every sense-react cycle, various static properties of the resulting state are checked by an oracle. The properties checked are:

1. Whether a train collision occurred.
2. Whether a track speed-limit violation occurred.
3. Whether a signal violation occurred.

If a static property is violated, the oracle raises an exception and outputs an appropriate error message. On the other hand, states that satisfy the test oracle are appropriately filtered and output to a file in the form of a vector of numbers. Initially, the position and speed of every train was written to a file. The table below is an example of the output of a 4 step (i.e., 2 second) simulator run for a system containing eight trains.

0 20	200 20	400 20	600 20	800 20	1000 20	1200 20	1400 20
23 26	223 26	423 26	623 26	823 26	1023 26	1223 26	1423 26
52 32	252 32	452 32	652 32	852 32	1052 32	1252 32	1452 32
86 36	286 36	486 36	686 36	886 36	1086 36	1286 36	1486 36
122 37	322 37	522 37	722 37	922 37	1122 37	1322 37	1522 37

It quickly became clear that such data files were not particularly enlightening. Therefore an effort was undertaken to construct a graphical display capable of animating the state changes that were being

described in the data file. In addition to displaying the position and speed of every train, the track segment and signal states are also displayed. This required a slight modification to the data file. The structure of the data file can be formally described by the following regular expression:

$$track\_seg^+ \text{ newline } display\_position^+ \text{ newline } (train^+ \text{ newline } display\_state^+)^+$$

where  $track\_seg = position\ speed$ ,  $display\_position = number$ ,  $train = position\ speed$ , and  $display\_state = [0-2]$ . An example of how this information is displayed is shown in Figure 4. The display is in color. Track segments shown in maroon with the starting position of each segment indicated by a rectangle. Trains are displayed as blue triangles and are connected by a blue line to enhance the users ability to see (dis)continuities. The line connecting two trains is broken if the distance between two trains becomes so large that the line connecting them does not help the user to process the visual information. Signals are displayed as vertical bars and are shown in three colors: green, yellow, and red. A signal whose state is displayed as yellow indicates that at least one train currently behind the signal sees the signal as being in a green state and that at least one train behind the signal sees the signal as being in a red state. (Recall that the interlocking system can assign different signal states to different trains.) A signal whose state is displayed as red indicates that all trains behind the signal have a consistent view of the signal as being in a red state. Similarly, a signal whose state is displayed as green indicates that all trains have a consistent view of the signal as being in a a green state.

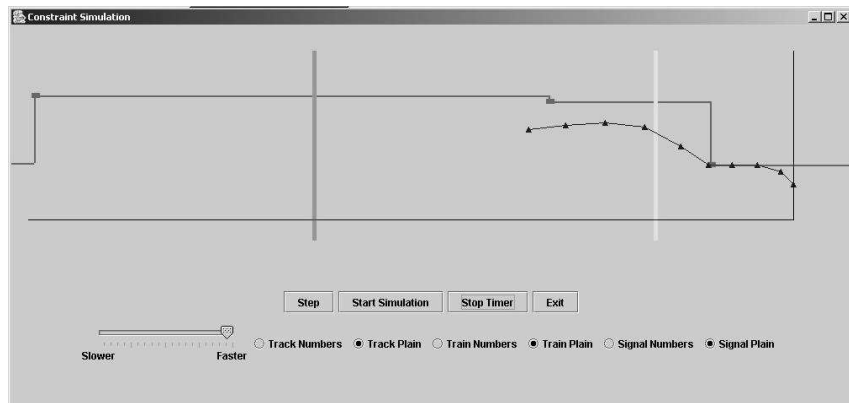


Figure 4: The Graphical Display

## 4.7 Optimization

Simulator runs using the executable specification raised concerns regarding the ability of the control function to satisfy real-time constraints. This led to an optimization cycle where transformations were

developed that significantly reduced the computation sequences needed evaluate various expressions within the control function. Many opportunities for optimization were identified, with several of the more interesting optimizations presented below.

#### 4.7.1 Fusing properties and constraints

Within the context of the train control specification, safety properties are expressed as the conjunction of constraints. The unfolding of these constraints gives rise to a lengthy expression involving a conjunction of first-order formulas. In this conjunction, each formula has its own quantified variables. Frequently variables in different formulas are quantified over the same domain (i.e., profile). In these situations, the redundant generation of profiles can be avoided by the fusing formulas. Recall that stopping profiles are generated by a computation involving *list\_mu*.

This form of fusing can occur between constraints as well as within a temporally independent constraint. For example, in Section 2.2.2  $Q_0$ ,  $Q_1$ , and  $Q_2$  all can be fused in the manner described here. In addition to the efficiency obtained by avoiding the regeneration of constraints, a savings is also obtained by avoiding the redundant examination of profile elements.

Let  $PF$  denote a profile whose elements must be generated.

When such The goal of this transformation is to avoid redundant generation of profiles as well as redundant examination of profile elements. Thus, if  $PF$  denotes an expression that must be evaluated in order to produce a profile, then expressions such as

$$(\forall x \in PF \forall y \in S : F) \wedge (\forall x \in PF \forall y \in S : G)$$

would require  $PF$  to be evaluated twice. Furthermore, the expression would require the elements in  $PF$  as well as  $S$  to accessed twice. Optimization 1 fuses two formulas sharing the same quantified variables. The resulting expression requires only a single generation of  $PF$  and  $S$ . In addition, further optimizations can be applied to short-circuit the evaluation in cases where values not satisfying  $PF$  or  $S$  are encountered.

Optimization 2 fuses two formulas sharing a single quantified variable.

$$\text{Optimization 1: } (\forall x \in PF \forall y \in S : F) \wedge (\forall x \in PF \forall y \in S : G) \implies (\forall x \in PF \forall y \in S : F \wedge G)$$

$$\begin{aligned} \text{Optimization 2: } & (\forall x \in S1 \forall y \in S2 : F) \wedge (\forall x \in S1 \forall z \in S3 : G) \\ & \implies \\ & (\forall x \in S1 : (\forall y \in S2 : F) \wedge (\forall z \in S3 : G)) \end{aligned}$$

#### 4.7.2 Common subexpression elimination

Even after fusing optimizations have been applied, formulas can exist which require a profile to be regenerated multiple times. For example, a formula like:

$$(\forall x \in expr_1 : (\forall y \in expr_2 : F))$$

will require multiple regenerations of  $expr_2$  in cases where  $expr_2$  is denoted by a formula requiring generation, such as  $nStop(state_{ot})$ . Such regeneration can be avoided by binding the profile expression to a variable and then substituting this variable in place of the original profile expression. Optimization 3, introduces a let construct to achieve such a local binding, thereby moving the calculation of  $set\_expression$  outside of the  $\forall$  construct.

Optimization 4 allows nested let expressions to be collapsed. The assumption here is that name conflicts do not arise, but such properties can be easily assured.

Optimization 3:  $(\forall x \in set\_expression : F) ==> let\ S1 = set\_expression\ in\ (\forall x \in S1 : F)\ end$

Optimization 4:  $let\ S1 = expr_1\ in\ (\forall x \in S1 : let\ S2 = expr_1\ in\ (\forall y \in S2 : F)\ end)\ end$   
 $==>$   
 $let\ S1 = expr_1\ S2 = expr_1\ in\ (\forall x \in S1 : (\forall y \in S2 : F))\ end$

### 4.7.3 Problem Specific Optimizations

All of the safety properties specified for the train control function are monotonic with respect to acceleration. Informally, one can argue that “slower is safer”. Formally, one can prove that for all acceleration values  $a_1$  if an acceleration value  $a_2$  satisfies a safety property  $P$  and if  $a_1 \leq a_2$ , then  $a_1$  will also satisfy  $P$ . This knowledge enables a descending enumeration of the acceleration values to be short-circuited as soon as an acceleration value has been found satisfying the safety policy.

## 5 Results Raised by this Technique

There are many insights gained by the proposed approach. The first and foremost is that before any formal verification of properties of a complex reactive system can be undertaken, it is extremely useful to do simulation runs to gain confidence in the formal model, especially understanding how close it is to the real system being formalized. Animation techniques can be helpful in focusing on key aspects of the system behavior as well as comprehending large amounts of data. Often, it takes many iterations for a formal model to be debugged in order to be consistent with reality. It is essentially impossible to get it right in the first few attempts. Given that formal verification, which attempts to establish that a given system is devoid of any bugs, is extremely expensive, it makes sense to use such techniques only when other techniques to debug the formal model have been exhausted. Otherwise, one could waste significant resources on identifying problems in a model which might have been detected easily using simpler and less expensive techniques.

The success of the proposed approach is based on developing a formal model of a reactive system in a high-level language and then applying problem independent and domain specific transformations to obtain an efficient executable specification from the formal model. Using an executable specification, simulation runs can be easily performed to analyze different scenarios. As the model is debugged, transformations are repeatedly applied so as to generate an executable specification to run simulations. Any changes in the formal model can thus easily get reflected in the executable. High-level transformations play a crucial role in making this approach effective.

The effectiveness of the proposed approach relies on the ability to develop transformations which can be applied on high-level specifications of reactive systems to generate efficient executables. Further, the correctness of these transformations should be relatively easy to establish, preferably using automated techniques. Confidence in the correctness of the transformation process is essential, otherwise an executable specification may produce erroneous results even though the associated formal model is correct.

## 6 Conclusions

An approach for developing a formal model of a complex reactive system as a train system motivated by the BART case study is discussed. The approach relies on defining key abstractions in a reactive system, such as state, next state function, profile, in a high-level functional language; safety policies are specified as constraints on profiles since other components such as signal, station and track can also be described as profiles. A control function can be described algorithmically, taking into consideration, specific physical characteristics of a train system.

A specification is developed in a high-level functional language that supports the key abstractions of reactive systems. The specification is then transformed to an efficient executable using both problem independent as well as domain-dependent transformations. Executable specification can be used to simulate different scenarios; the simulation can be animated to enable the designer focus on important aspects of the system behavior.

The proposed approach provides flexibility to test different control functions, giving the designer a choice based on other consideration such as throughput, passenger comfort (in case of train systems), etc. In this chapter, the primary focus has been on safety and the secondary focus has been on maximizing throughput.

The formal model of a train system discussed in this paper relies on discrete time model. Such a model can sometimes be difficult to work because it can go against the common sense intuition about the behavior of a system that is continuous in reality. In [14], a continuous time model is developed from which the discrete model is derived by proving mathematical properties of the equivalence of the discrete model to the continuous model. The paper did not discuss how the correctness of both problem independent and domain dependent transformations can be established; an interested reader may consult for more details. Also missing is any formal verification of the control algorithm.

## References

- [1] F.B. Bastani, V. Reddy, P. Srigiriraju, and I.-L. Yen. *Systematic Validation of a Relational Control Program for the Bay Area Rapid Transit System*. In: High Integrity Software, Eds. V. L. Winter and S. Bhattacharya, Kluwer Academic Press, 2001.
- [2] Chris Brink, Wolfram Kahl, and Gunther Schmidt (editors). *Relational Methods in Computer Science*. Advances in Computing 1997, Springer-Verlag, Wien 1997.

- [3] Ricky W. Butler and George B. Finelli. *The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software*.
- [4] D. E. Cooke and V. Kreinovich. *Automatic Concurrency in SequenceL*. In *Electronic Notes in Theoretical Computer Science*, 25 (1999).
- [5] H. Gomaa. *Software Design Methods for Concurrent and Real-Time Systems*. Addison Wesley, 1993.
- [6] C. Michael Holloway. *Why Engineers Should Consider Formal Methods*.
- [7] Ryszard Janicki, David Lorge Parnas, and Jeffery Zucker. *Tabular Representations in Relational Documents*. *Relational Methods in Computer Science*, C. Brink & G. (eds.) in cooperation with R. Albrecht, Springer-Verlag, 1996.
- [8] D. Kapur and H. Zhang. *An Overview of Rewrite Rule Laboratory (RRL)*. In *J. of Computer and Mathematics with Applications*, 29, 2, 1995, 91-114.
- [9] Deepak Kapur and Victor Winter. *On the Construction of a Domain Language for a Class of Reactive Systems*. In: *High Integrity Software*, Eds. V. L. Winter and S. Bhattacharya, Kluwer Academic Press, 2001.
- [10] Carroll Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science, 1990.
- [11] H. Partsch and R. Steinbruggen. *Program Transformation Systems*. *ACM Computing Surveys*, Vol. 15, No. 3, pp 199–236, Sept 1983.
- [12] John Rushby. *Formal Methods and their Role in the Certification of Critical Systems*.
- [13] V. Winter. *An Overview of HATS: A Language Independent High Assurance Transformation System*. *Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET)*, March 24-27, 1999.
- [14] V. Winter and D. Kapur. *On the Construction of a Domain Language for a Class of Reactive Systems*. In: *High Integrity Software*, Eds. V. L. Winter and S. Bhattacharya, Kluwer Academic Press, 2001.
- [15] V. L. Winter, D. Kapur, and R.S. Berg. *A Refinement-based Approach to Deriving Train Controllers*. In: *High Integrity Software*, Eds. V. L. Winter and S. Bhattacharya, Kluwer Academic Press, 2001.
- [16] V. L. Winter, S. Roach, and G. Wickstrom. *Transformation-Oriented Programming: A Development Methodology for High Assurance Software*. To appear in *Advances in Computers*.

## A Appendix: Operational Semantics of Various SPC Constructs

*define*  $list\_mu(state, succ, Pred) =$  *if*  $Pred(state)$  *then*  $[state]$   
*else*  $state::list\_mu(succ(state), succ, Pred)$

*define*  $[set : \mathcal{P}] =$  *if*  $isEmpty(set)$  *then*  $SET\_EMPTY$   
*else if*  $\mathcal{P}(first(set))$  *then*  $first(set) :: [set : \mathcal{P}]$   
*else*  $[set : \mathcal{P}]$

Given that SPC restricts its attention to finite sets, filter formulas can also be used to define universal and existential quantification.

- $\forall x \in S, P(x) \equiv ([S : P] = S)$
- $\exists x \in S, P(x) \equiv ([S : P] \neq \emptyset)$

## B Discussion of the Functionality of Emergency Stop

A certain amount of complexity arises from the possible interplay between the two stopping modes considered. A normal stop is accomplished using normal acceleration values, and an emergency stop is accomplished using emergency acceleration values. Differing viewpoints can be presented as to the motivation behind normal and emergency stopping modes. One possibility is to view the emergency stop as a backup braking mechanism. This backup should be used in situations where normal stopping is malfunctioning. Though somewhat counter-intuitive, in this interpretation it may be reasonable to consider a worst-case scenario in which it actually takes longer for a train to perform an emergency stop than it would using a normal stop. In response to such an interpretation, one would have to question why the train could not simply be equipped with a redundant mechanism capable of performing a normal stop. Thus providing a backup system capable of performing a normal stop.

A contrasting interpretation is to view an emergency stop as a “hard stop” of the type seen in movies when the emergency brake is pulled on a train. From this perspective, an emergency stop should bring a train to a halt considerably quicker than a normal stop. However, due to the strain on both the passengers as well as the mechanical components of the train the emergency stop should not be used casually.

Though the control function developed in this chapter can handle both interpretations, the state safety policy assumes that an emergency stop should be avoided if at all possible. Under normal conditions, a control function should only respond to system changes using normal acceleration values. In fact, it is considered a violation of the safety policy for a train to respond to a normal environment using an emergency acceleration value. In contrast, an abnormal environment permits the control function to consider both normal as well as emergency acceleration values. However, in abnormal situations an additional subtle requirement is placed on the control function that it must use normal acceleration values to establish safe behavior if it is possible to do so. For example, suppose a train has

derailed on the track 5 miles ahead. As a result the train immediately behind (i.e., 5 miles behind) the derailed train sees an abnormal environment. Assuming it is possible to do so, this train must come to a halt using normal acceleration values. In particular, it may not continue advancing on the position of the derailed train until it can no longer halt without availing itself of an emergency stop. Consider the cascading effect that would result if a train, upon seeing an abnormal environment, would unnecessarily issue an emergency stop. Such an action would be perceived by the following train as abnormal which, in turn, would cause it to issue an emergency stop, etc., etc.

## C Appendix: Train Motion in an Iterative Feedback System

In an iterative feedback system, the equations of motion of a train must be computed for each time step from the general function that relates position, speed, acceleration, and time. Since the train is confined to the track (at least under normal circumstances), we actually have a one dimensional problem in  $p$ , the position of the train along the track, rather than a three dimensional problem. Issues such as track slope can be taken into account through perturbations on a train's acceleration rather than through an independent acceleration variable.

The following Taylor's series expansion can be used to describe the behavior of a train over a small time interval:

$$p(t) = p(t_0) + s(t_0)(t - t_0) + \frac{1}{2}a(t_0)(t - t_0)^2 + \dots$$

In general, the series can be terminated with those terms that are explicitly shown.

In the train controller, we cannot control the position,  $p$ , or the speed,  $s$ , because these two parameters must be continuous. This means that the time development of the position and speed are also continuous. However, we can set the acceleration,  $a$ , which does not have to be continuous. When we discretize the parameters,  $(p, s, a)$ , we are, in effect, selecting the value at the beginning of each time interval. This is different from saying that the values are constant for the time interval. In such a case, the train would have to violate the laws of physics to move. The Taylor's series can be used, however, to predict the values for the train's motion variables by using the entire time interval in the calculation. In this case, we can say

$$p(\Delta t) = p(0) + s(0)\Delta t + \frac{1}{2}a(0)\Delta t^2$$

Here, we are setting a new value for  $a$  at  $t = 0$ , so the train's parameters will change with time. The speed equation then becomes:

$$s(\Delta t) = s(0) + a(0)\Delta t$$

This projection process results in one of the error terms that can affect the position calculation of the train. Others can include perturbations on the acceleration due to slopes of the track, wind speed, track conditions, and others. These perturbations of the command acceleration have the combined effect of changing the true acceleration from the command acceleration, and therefore, the actual position of the train can deviate from the calculated position. In an iterative feedback system, the train controller

accounts for this deviation by getting a report of the train’s position after every time interval (i.e., before the deviation becomes too great). The report can then be used to correct the commanded acceleration to account for the deviation.

What we have described, is one way to model the motion of a train in the context of an iterative feedback system in which the train’s motion is controlled through its acceleration. Note that the model developed cannot assume constant position, speed and acceleration over a time interval, since this would violate the laws of physics. However, one can assume that the values reported at the beginning of each interval are the only ones that are known, and then use those values, which is considerably different. Note that a similar approach can be taken when modeling the speed limits of track segments.

Due to the “drift” that can occur between the model and the actual train system some provision needs to be made to prevent a train from exceeding a speed limit. One way to do this is to put a *guard band* on the speed. A guard band is essentially a buffer zone that can be used to assure the train’s speed limit isn’t exceeded by any process that increases the speed. In such a case, the speed limit is not exceeded by virtue of the guard band, whether it is the result of under damping in the system, a down slope that the train is on, or some other factor. The train will simply set its speed goal at a few (say 3) miles per hour below the target speed. This means that while the acceleration is used as a control variable, it is the speed that is goal of the command acceleration. Thus one had to know the target speed, at the beginning of the acceleration process, so one could include that in the calculation of the acceleration profile. The size of the guard band would be determined by such things as the expected overshoot. In our model, we use such guard bands. For example, if the speed limit of a track is 53 mph, then we will model the speed limit as being 50 mph. Note that the size of guard band should be determined by a domain expert.

## D Appendix: Extended-BNF Grammar for SPC

Spec ::= Def\_list

Def\_list ::=  $\epsilon$  | Def\_list Def ;

Def ::= define InfixPatr = WhereExpr  
 | val [rec] InfixPatr = WhereExpr

WhereExpr ::= Expr [where ( Def\_list ) ]

Expr\_list ::= [Expr\_list ,] Expr

Expr ::= if Expr then Expr else Expr  
 | case Expr of Match\_list  
 | fn Match\_list  
 | let Def\_list in WhereExpr end  
 | ForAllExpr

Match\_list ::= [Match\_list “|” ] Atomic\_Match

Atomic\_Match ::= InfixPatr => ForAllExpr  
 ForAllExpr\_list ::= [ForAllExpr\_list] FORALL InfixPatr IN Expr  
 ForAllExpr ::= [ForAllExpr\_list :] DisjExpr  
 DisjExpr ::= [DisjExpr or\_op] ConjExpr  
 or\_op ::= OR | orelse  
 ConjExpr ::= [ConjExpr and\_op] NotExpr  
 and\_op ::= AND | andalso  
 NotExpr ::= [not\_op] ImplExpr  
 not\_op ::= ~ | not  
 ImplExpr ::= [ImplExpr ->] RelExpr  
 RelExpr ::= [RelExpr rel\_op] InfixExpr  
 rel\_op ::= = | <= | < | > | >= | != | <>  
 InfixExpr ::= [InfixExpr Symbl] E  
 E ::= [E + | E -] T  
 T ::= [T \* | T /] SubExpr  
 SubExpr ::= AppExpr [ "[" Expr "]" ]  
 AppExpr ::= [AppExpr] TermExpr  
 TermExpr ::= num  
           | Ident  
           | ( [Expr\_list] )  
           | "[" Expr : Expr "]"  
 Patr\_list ::= [Patr\_list ,] InfixPatr  
 InfixPatr ::= [InfixPatr Symbl] AppPatr  
 AppPatr ::= [AppPatr] TermPatr  
 TermPatr ::= wildid

| Ident  
| ( [Patr\_list] )

Ident ::= alphid

Symbl ::= symbid