Verification of BART Controller: An Approach based on Horn Logic and Denotational Semantics*

Lawrence King, Gopal Gupta, Enrico Pontelli
Lab for Logic and Databases
Department of Computer Science
New Mexico State University
{lking,gupta,epontell}@cs.nmsu.edu

Abstract:

We present a semantics-based framework for verifying real-time controllers implemented in Ada. Our semantics based framework uses techniques based on denotational semantics, logic programming, and partial evaluation. We use our framework to verify the correctness of the real-time controller for the *Bay Area Rapid Transit System (BART)*. The main property we wish to verify is that if two consecutive trains start out with some minimum separation distance between them (that is deemed to be safe), then this safe distance is always maintained. This safe distance is dependent on a number of different variables, such as speed, acceleration, track gradient. Our approach allows automatic derivation of a verification system from the Ada implementation of the controller.

1 Introduction

Verification of software controllers is an important problem, since controllers are key components of many critical applications. If the specification of a software controller has an error, then it will result in the controller being incorrectly implemented, which in turn can result in loss of life and property. In this paper, we present an approach to verifying important properties (e.g., safety property) of a controller design.

We assume that the specification for the controller is given informally. This is, in fact, a common occurrence, since specifications are quite frequently written in English. Ideally, a system designer who is responsible for designing and implementing the software controller should first develop a design for the controller based on the informal specification, check that the design is correct, and only then proceed to the implementation.

*This research has been partially supported by NSF grants CCR 99-00320, HRD 96-28450, INT 99-40063, CDA 97-29848, EIA 98-10732, and NSF CAREER Grant CCR 98-75279

In the case that we consider in this paper, namely, a controller for the *Bay Area Rapid Transit (BART)* system, the design process involves selecting the range of velocities and the range of accelerations allowed for the various trains at each time instant, etc., based on the given informal specification. The property that we would like the design to satisfy is that, if there are two trains, one behind the other and at a minimum separation distance that is deemed to be safe (we refer to this distance between two trains as "safe distance" in the rest of the paper), then the controller will never send an instruction to the two trains that will cause the trains to come closer than this safe distance—where the safe distance at any given moment is a function of velocities and accelerations of the two trains and the track conditions. Ideally, one would like the design to be specified precisely using some formal notation. This formally specified design will be then used to automatically verify that the design satisfies the safety properties as well any other property of interest.

However, in most cases the design is not specified using a formal notation, rather it is specified informally—just like the original specification. This informal design is then directly used for implementing the controller. The implementation of the controller is done in a programming language—a formal notation. The implementation formally states how the design decisions are being concretely implemented. Thus, in the whole process the only formal entity that is created is the *implementation*.

We present an approach that takes this formal entity—the implementation—and verifies that the design on which this implementation is based indeed satisfies the safety property and other properties of interest. Our approach is based on denotational semantics [18], logic programming [20, 10], partial evaluation [11], and abstraction. Briefly, our approach works as follows: we start out with the implementation of a controller in *Ada*. We use the denotational semantics of Ada to derive the denotation of the program that represents this implementation. The denotational semantics of Ada is expressed using the declarative notation of Horn-clause logic, also known as *logic programming* [10]. As a result of using logic programming, the denotation of the implementation that we obtain is *executable*. This denotation is a logic program (i.e., a collection of Horn clauses), that can be used to verify interesting properties of the original program. Several abstractions are applied during the verification process to simplify the problem. All the steps are automatic except for the final step of abstraction.

We apply our approach to verify the design of the controllers for the trains of the Bay Area Rapid Transit (BART) system. The informal specification of the BART system has been prepared by a group at Sandia National Labs [23]. We use this informal specification to develop an Ada program implementing the BART train controllers. This Ada program is then used to verify the correctness of the design using the approach described above. The safety property we are interested in is that if the trains in the BART system are in a safe configuration, then after an iteration of the controller they remain in a safe configuration.

2 Semantics-based Verification

We first give a brief introduction to denotational semantics and to logic programming. We next show how by using logic programming as the declarative notation for describing denotational semantics, one can use it for automatic verification.

Denotational Semantics:
Denotational Semantics [21, 18, 5, 19] is a well-established methodology for the design, description, and analysis of programming languages. In the denotational semantics approach, the semantics of a computer language/notation is specified in terms of mathematical objects, such as sets and functions. The denotational semantics of a language $\mathcal{L}$ has three components:

- *syntax specification:* maps sentences of $\mathcal{L}$ to parse trees; it is commonly specified as a grammar in the BNF format;
- *semantic algebra:* represents the mathematical objects used for expressing meaning of a program written in the language $\mathcal{L}$; these mathematical objects typically are sets or domains (partially ordered sets, lattices, etc.) along with associated operations to manipulate the elements of the sets;
- *valuation functions:* these are functions mapping parse trees to elements of the semantic algebras.

In this paper we assume that the reader is familiar with denotational semantics.

Logic Programming:
Logic programming is a declarative programming paradigm based on a subset of first order logic called Horn-clause logic [13]. The most well known logic programming language is Prolog [20].

A logic program is composed of a collection of *clauses*, where each clause has the form

$$Head\ :-\ Body_1, \ldots, Body_n$$

and *Head*, $Body_1, \ldots, Body_n$ are atomic formulae. Each clause represents the logical implication

$$\forall \bar{X}(Body_1 \wedge \ldots \wedge Body_n \rightarrow Head)$$

where $\bar{X}$ are all the variables in the clause. Given a program P, the goal of an execution is to prove whether a given formula G is a logical consequence of the clauses in P, i.e., whether $P \models G$. A goal is typically restricted to a conjunction of atoms.

The execution model used by logic programming traditionally relies on: (i) *SLD-resolution*—intuitively, used to solve each atomic formulae using the clauses in the program; and, (ii) *unification*—intuitively, used to solve equations between logical terms to determine the applicability of a clause [13].

In this paper we assume that the reader has familiarity with logic programming.

Partial Evaluation:
Partial evaluation [11] is a well known technique in the field of programming languages in which instances of a general program are specialized for certain inputs. The specialized program should show "better" behavior than the original program in that it should take less time or take less space, etc. Consider a program P that takes input I and produces output O. Thus, we can write $P(I) = O$. The input usually has two components: the *static input* and the *dynamic input*. The static input, S, is given at the beginning of the program, while the dynamic input, D, is given while the program is executing. We can now write: $P(S)(D) = O$. Let us assume that in a certain situation we know the static input S in advance; then we should be able use this information, to (automatically) obtain a more efficient version of P. Thus, our goal is to transform $P(S)$ (the program P applied to the static input S) to produce a specialized program P_s, which is "more efficient" than P. The process of producing P_s from P and S is termed *partial evaluation*. The process amounts to evaluating the program P partially until the static input is consumed. Note, thus, that for each dynamic input D we require $P(S)(D) = P_s(D) = O$. Automatic partial evaluators have been built for declarative languages—e.g., the *Mixtus* system for the Prolog language [17], and the *Schism* system from INRIA for Scheme [1].

The most significant application of partial evaluation is in generating compilers from interpreters. It is a well known fact [11] that given an interpreter for a language $\mathcal{L}$ and a program P_L written in $\mathcal{L}$, the specialization of

$\mathcal{L}$'s interpreter w.r.t. the program P_L produces compiled code for P_L. This is known as the *first Futamura projection* [11]. As illustrated later, we make use of the first Futamura projection to make our semantics-based verification process more efficient.

Horn Logic Denotational Semantics:
Traditional denotational definitions express syntax as BNF grammars, and the semantic algebras and valuation functions using λ-calculus. Instead, we use Horn-clause Logic (or pure Prolog) to code denotational semantics of programming languages [6]. There are two major advantages of doing this:

1. A Parser can be obtained from the syntax specification with negligible effort: The BNF specification of a language $\mathcal{L}$ can be trivially translated to a *Definite Clause Grammar* (DCG) [20]. As a result, the syntax specification can be loaded in a Prolog system, and a parser automatically obtained. This parser can be used to parse programs written in $\mathcal{L}$ and obtain their parse trees.
2. Denotational Semantics can be used for automatic Verification: The semantic algebra and valuation functions of $\mathcal{L}$ can also be coded in Horn-clause Logic. Since Horn-clause Logic or pure Prolog is a declarative programming notation, just like the λ-calculus, the mathematical properties of denotational semantics are still preserved. This is because logic programming is a declarative programming paradigm based on relations, and relations subsume functions. The relational nature of logic programming allows for the state space of a program written in $\mathcal{L}$ to be explored with ease. We make use of this fact to verify properties of programs.

Since both the syntax and semantics part of the denotational specification are expressed as logic programs, they are both executable. These syntax and semantic specifications can be loaded in a logic programming system and executed, given a program written in $\mathcal{L}$. This provides us with an interpreter for the language $\mathcal{L}$. In other words, the denotation of a program written in $\mathcal{L}$ is executable. This executable denotation can be used for many applications. The denotation can be *partially evaluated* to obtain a more efficient version of the program—this more efficient version is similar to the compiled code that would be generated for the program. The program denotation (or its more efficient, partially evaluated, version) can be used for verification.

In this paper we illustrate the use of logic-based denotations for automatic verification.

3 The BART System

Our goal in this project is to verify the safety properties of the controller for the Bay Area Rapid Transit (BART) system, as specified by the group at Sandia (see www.hcecs.sandia.gov/bart.htm). The controller considered in this project is envisioned as comprising a portion of the Advanced Automatic Train Control (AATC) system being developed for the Bay Area Rapid Transit (BART) system. Specifically, the controller focuses on those aspects of the BART system that are necessary to control the speed and acceleration of the trains, subject to various physical, safety, and policy constraints.

Description of the BART system:
BART provides heavy commuter rail service in the San Francisco Bay Area. On a typical work day it serves approximately 250,000 passengers. During peak hours, more than 50 trains, most consisting of 10 cars, are in service.

Primary system control is automatic, with on-board operators having a limited role in normal operations. For example, operators signal the system when the platforms are clear so a train can depart a station. More importantly, operators trouble-shoot problems and can operate trains manually at low speed in the event of a problem or failure of the automatic control system.

The tracks themselves are partitioned into track segments. The boundary of a segment may represent: (i) a physical change, such as a change in grade or track exposure; (ii) a safety change, such as a change in maximum safe speed; (iii) a gate, which establishes right-of-way where tracks join or merge at switches; and, (iv) a station platform.

Advanced Automatic Train Control (AATC):
BART tracks feed from points north, south, and east into San Francisco. The critical link is the section from Oakland to points west where four lines share one pair of tracks. To meet the ever-increasing demand for commuter-rail service, BART must utilize this section more efficiently or add additional track. Adding new track to this segment, which runs through a tube under the bay and in tunnels under the heart of San Francisco, would be prohibitively expensive. Therefore, BART must provide a higher level of service by running trains more closely spaced and possibly at greater speeds than is currently the case. It is intended that the AATC will enable BART to do this while maintaining existing levels of safety and passenger comfort.

The AATC is intended to replace part of BART's current control system. It will consist of new station controllers, a radio communications network that will link the stations with the trains, and software modifications to the on- board controllers. The station controllers will make control computations and transmit speed and acceleration commands to the trains via the communications network. In addition, each station controller will be required to communicate with its neighbors in order to receive and hand-off trains. In track sections where stations are closely spaced, one station controller may actually control trains over several stations. It is anticipated that each station controller will be capable of handling 20 trains in its assigned track section.

The specific aspects of the station controllers addressed in this paper are the calculation and selection of the speed and acceleration commands sent to the trains. The overall objective of the speed and acceleration commands is to get trains from one point to another as fast and smoothly as possible subject to the following two *safety constraints*:

1. A train should maintain a minimum distance from the train in front of it such that if the lead train stopped suddenly (e.g., because it derailed), the follow train would be able to safely stop without a collision.
2. A train should stay below the maximum safe speed for the current track segment.

The processing which leads to the determination of the speed and acceleration commands is required to meet an additional constraint of terminating within a 1/2 second time period. A fourth constraint, that of trains not entering closed gates, was not addressed in the station controller designed for the study reported in this paper.

Operating environment of the station controllers:
The AATC system operates on 1/2-second cycles. Each half second, the station controller receives position and speed information, along with uncertainty data (mean and standard deviation) and the time the information was sent from each train. For this paper, it is assumed that such data is reliable and accurate.

Speed and acceleration commands sent to the trains are time stamped with a Message Oriented Time Tag (MOTT) and become invalid two seconds after the identified time. A train will exercise a command until either a new command arrives or the current command becomes invalid due to the expiration of the two-second validity window. If the current command becomes invalid prior to the receipt of a new command, the train goes into maximum braking. This is to insure that each train will be able to stop safely in the event of a communications failure.

The station controllers also have available to them static data for each track segment in the track section they control. Such data includes:

1. Segment *beginning* and *ending* points.
2. *Grade*: It is assumed that grades are constant over any track segment.
3. *Maximum allowable speed* for each track segment.
4. *Gate locations*: This data was not used in the controller designed for this study.

It is the responsibility of the station controller to produce and transmit a command message for each train in its track section. All command messages must be transmitted by the end of the 1/2-second AATC-system cycle. Each command message must contain the following safety-critical data:

1. *Identification number* of the train for which the command is intended.
2. Commanded *speed*—in the range from 0 to 80 mph.
3. Commanded *acceleration*—in the range from -2 to -0.45 mphps for deceleration (braking), and in the range from 0 to 3 mphps for acceleration.
4. *Message Oriented Time Tag*.

Overall design of the station controller:
It is required that the station controller be divided into two computer systems. First, the Non-Vital Station Computer (NVSC) is a fast, flexible computer system that is reliable enough to meet performance-driven Mean Time Between Failure (MTBF) and Mean Time Between System Shutdown (MTBSSD) goals but not safety-related Mean Time Between Hazard (MTBH) goals specified by BART [23]. It is the task of the NVSC to propose speed and acceleration commands, considering both safety and train-system performance. Second, the Vital Station Computer (VSC), a lower-performance computer system, verifies that the speed and acceleration commands proposed by the NVSC do not violate safety constraints. The VSC then either issues the commands as proposed or modifies them to conform to safety constraints.

Detailed specifications of the station controller designed for this study will be enumerated in the following section of this paper.

Worst-case stopping profile:
The physical performance of the trains in response to commands is highly complex. For the purposes of this study, a worst-case profile approved by the California Public Utilities Commission Interim Order of June 3, 1980 was used. The profile is extremely conservative, meaning that the performance of a train that is worse than this profile (greater maximum stopping distance), which is theoretically possible but extremely unlikely, represents a publicly agreed upon *acceptable risk*.

For example, the profile assumes a six-standard-deviation (6σ) uncertainty in train position, a two-second time-out before maximum braking is commanded, maximum acceleration prior to the braking command, a ten-car train which is 710 feet long, and a twenty percent brake-failure rate.

The inputs to the algorithm that computes the worst-case stopping distance includes both data received from the trains in real time and static track-segment data.

After calculation of the worst-case stopping distance for each train during each 1/2-second cycle, it is required that the station controller issues speed and acceleration commands that preserve the safety constraints mentioned earlier. Also, the computation for all trains in the track segment is required to terminate within the 500ms time frame.

Ada Implementation of BART Controller:
Because the station controller is intended to be a real-time, embedded system, it was designed using the *Hierarchical Object-Oriented Design (HOOD)* technique, with *Ada* as the programming language of choice.

HOOD is based on the software engineering principles of abstraction, encapsulation, and modularity. Top-level objects are identified and then decomposed into lower-level objects—a process called *hierarchical decomposition*. Data, defined as abstract data types, is encapsulated with appropriate access control. Safety-critical processes are similarly encapsulated with well-defined entry and exit points. Both data and processes are resolved into clearly-defined modular objects, related to other objects by well-defined data flows or by decomposition into objects at a different level in the design. As can be seen from the HOOD diagram in Fig. 1, the top level of the controller consists of three entities:

Train Data abstract data type: This is implemented in Ada as a dynamic linked list. Each element in the list is conceptualized as a lower-level object representing a train currently in the controller's track section. As trains enter the track section, objects representing them are added to the list; as they are handed off to another section, their objects are deleted from the list. The objects in the list are identified by a unique number, and contain the position, speed, acceleration, time-tag, and other data transmitted by each train during the 500-millisecond AATC control cycle described in the previous section. Ada procedures in the Train Data object specification allow the trains to update the data in their specific train object and allow the controller object to access this data for purposes of calculating new speed and acceleration commands to be issued during the current cycle. On the other hand, the Train Controller object is unable to modify the data in the Train Data object. However, for simulation purposes, the current train data as well as speed and acceleration commands issued by the Train Controller object were used to estimate new train data which is placed in the Train Data object by a simulation algorithm external to the Train Controller object.

Track Data abstract data type: This is implemented in Ada as a static array. Each element in the array contains the number, beginning and ending points, maximum safe speed, grade, exposure conditions, and gate data for one track segment. The Train Controller object can request data from the Track Data object for two purposes: calculation of the minimum separation distance (worst-case profile) between trains and verification by the Vital Station Computer (VSC) lower-level object that the commands proposed by the Non-Vital Station Computer (NVSC) conform to safety constraints. The Train Controller object cannot modify the data in the Track Data object.

Train Controller: The Train Controller object is the top-level computational engine. It encapsulates three lower-level objects:

- an object that calculates the minimum distance between any two train objects in the Train Data object
- an object representing the NVSC which proposes the speed and acceleration commands to be issued during the current AATC cycle
- an object representing the VSC which insures that the commands proposed by the NVSC object conform to safety constraints

As indicated in the diagram in Figure 1, these objects are linked by a clearly defined data flow. Furthermore, the NVSC object has only one entry point, by which it receives the data it requires to compute proposed commands, and one exit point by which it transmits its proposed commands to the VSC object. Similarly, the VSC object has one entry point, by which it receives the proposed commands from the NVSC object, and one exit point, by which it transmits the final commands to the trains. Such encapsulation is maintained by standard Ada programming techniques.

Design of the NVSC:
The NVSC object is the heart of the top-level Train Controller object—it is here that the commands are computed and issued. After considering several approaches to the design of this object, it was finally determined that a finite-state automata was the design of choice. A fault-tree analysis suggested that violations of the minimum separation distance (worst-case profile) would be more likely to occur because of a "cascading" series of discrete conditions rather than as the interplay of dynamic properties represented by continuous functions. The finite-state automata also enabled decomposition of the NVSC object into a lower-level composed of a series of fault-critical objects related by a clearly defined data flow. We felt that this enhanced the safety-critical effectiveness of this real-time, embedded system. A further advantage of the finite-state automata is that the number of states within each fault-critical object can be significantly expanded without affecting the overall properties of the NVSC.

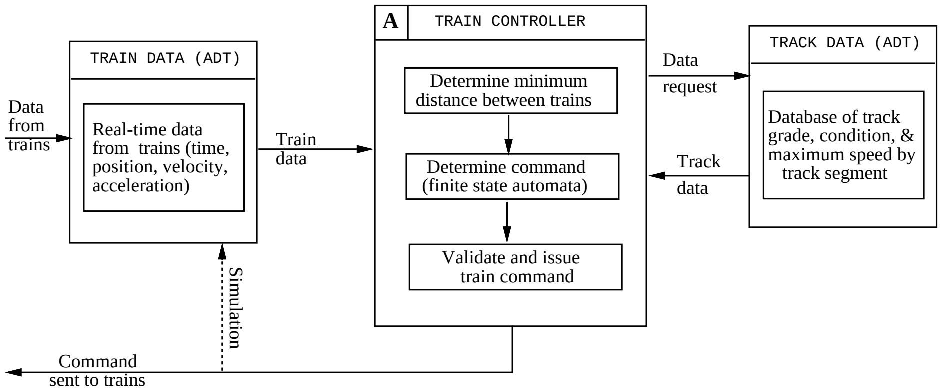

Figure 1: HOOD Specification of BART Controller

The finite state automata was implemented in Ada using standard multi-way selection statements (`case`) with nested function calls. Only the "start" state is visible from outside the NVSC object, and only a final "issue command" state can cause an exit of the NVSC object.

Real-time properties of the Controller:
Because the controller designed for this study was intended as a preliminary model suitable for proving system properties, it is naive. It has no "look ahead" properties that would be present in a fully operational controller, nor does it track any real-time properties of the system. It is only concerned with the two safety-critical issues mentioned in the previous section, maintaining a minimum separation distance (worst-case profile) between trains and preventing the issuance of commands that would violate safety constraints. As the controller evolves, it will become substantially more "intelligent".

However, one concern which was addressed was the issue of response time. By the conditions of the AATC system, the controller had to be capable of computing and issuing commands to the trains within a 500-millisecond cycle. Therefore, repeated simulations were performed to test this property. The controller took on the average only 4.7 milliseconds to compute and issue (print) commands for the maximum 20 trains in its track section. Accounting for the average communication times provided by the communication media used in the BART system, we expect the communication timing constraints to be easily met.

The safety-critical performance of the controller also suggested that the choice of a finite-state automata for the basic computational engine was justified. Testing revealed that the minimum separation distance (worst-case profile) was maintained even under extreme starting conditions, conditions which would never be encountered in an operational environment. Further, it was observed that changes in the finite-state automata to accommodate different fault-trees were relatively simple, and that such changes could be made in the encapsulated NVSC without affecting other objects or violating any properties of the HOOD design. The addition of more complex control structures may lead to violations of the timing constraints—i.e., the system may become unable to provide a response for up to 20 trains within the 500 milliseconds time frame. In this case an additional verification step is needed to ensure the timing properties of the controller (See section 5).

4 Verification of the Ada Implementation

As mentioned earlier, the main aspect of the controller that we wish to verify are the safety properties: if two trains are at a safe distance, then after one iteration of the controller, they will remain at a safe distance; and that the trains never exceed maximum allowable speed for a particular track at any moment. While the Ada implementation was extensively tested, one can only have confidence in its design if it can be formally proven to be error free—i.e., if it satisfies the safety properties.

To perform the verification we used a denotational semantics based approach that is based on logic programming. We first developed the denotational semantics of a subset of the Ada language. This subset is the one that was used to program the NVSC portion of the controller and the main control loop. This denotational semantics was written in Horn-clause Logic (pure Prolog). This semantic specification is executable and acts as an interpreter for the subset of Ada considered. Let us refer to the Ada program that implements BART's controller as $\mathcal{P}_{cntl}$. Then $\mathcal{P}_{cntl}$ can be executed on the interpreter that is obtained when the syntax and semantic specification of Ada are loaded in a Prolog system (let us refer to this interpreter of Ada as $\mathcal{ADA}$). Note that we are not interested in executing $\mathcal{P}_{cntl}$ on $\mathcal{ADA}$, rather we are more interested in the fact that a call to the interpreter $\mathcal{ADA}$ with $\mathcal{P}_{cntl}$ as an argument represents the executable denotation of $\mathcal{P}_{cntl}$. This denotation of $\mathcal{P}_{cntl}$ is a logic program, and will be used by us to verify the safety property of the controller. Let us call this denotation of $\mathcal{P}_{cntl}$ its Horn-clause logic denotation and denote it with $\mathcal{D}_p$. i.e., $\mathcal{ADA}(\mathcal{P}_{cntl}) = \mathcal{D}_p$. Let us denote the preconditions on $\mathcal{P}_{cntl}$ with $\mathcal{I}_p$, and the postconditions with $\mathcal{O}_p$. Preconditions are predicates that hold at the beginning of the program, before the execution. Typically preconditions are constraints on program inputs. Postconditions are predicates that must hold after the program execution is over. Typically postconditions are constraints on program outputs. The problem of verification is to check that if the preconditions hold at the beginning, then the postconditions will hold after the execution of the program is completed.

The general problem of verification is quite hard, as it may require the power of a full first order logic theorem prover to be deployed. To overcome this problem, we tried to restrict the kind of properties that can be verified—mostly by imposing finiteness requirements through the preconditions. This is a commonly adopted practice [15], e.g., in the model checking research, where finiteness is a crucial restriction. These simplifications allows us to perform verification within weaker logics—in this particular case within Horn-clause logic.

In the verification process we take advantage of the *reversibility property* of logic programming: a logic programming predicate can be thought of as just performing a test if all its arguments are instantiated, if some of the arguments are not instantiated, then the execution mechanism of logic programming systematically generates successive values for these uninstantiated variables. Since the preconditions are really conditions on inputs to the program, and the inputs are usually not known in advance, if we code the precondition in Horn-clause logic, then they can be turned into generators of inputs. The inputs are successively fed to $\mathcal{D}_p$, the denotation of $\mathcal{P}_{cntl}$, which produces the output. Once the output is produced, and the denotation of the program has finished execution, the postcondition can be subsequently evaluated on the specific output. If the postcondition succeeds for every possible output that the precondition generates, then we can infer that the program $\mathcal{P}_{cntl}$ is correct. Alternatively, if we evaluate the negated postcondition after the execution of $\mathcal{D}_p$, then if the negated postcondition always fails, then we can infer that $\mathcal{P}_{cntl}$ is correct, subject to the preconditions. We take this second approach. Thus, we construct the logic program goal:

$$\text{Goal 1}: \qquad \mathcal{I}_p, \mathcal{D}_p, not(\mathcal{O}_p) \tag{1}$$

where $\mathcal{I}_p$ and $\mathcal{O}_p$ are the Horn-clause logic representations of the preconditions and postconditions. We run this goal on a Horn-clause logic interpreter (i.e., a Prolog system) loaded with the denotational specification of the Ada subset. If this goal fails, then we can infer that the program is correct. Essentially, if the program is correct, i.e., it computes its outputs correctly, then the negated postcondition forces a failure causing the logic programming system to explore the entire search space corresponding to all the inputs generated by the preconditions. Thus, if the goal above fails, it means that there is no input generated by the precondition for which the denotation of the program does not produce correct output.

Horn Logical Denotational Semantics of Ada:
The denotational semantics of the subset of Ada used for the NVSC controller and main loop was defined using logic programming. The specification essentially contains the Definite Clause Grammar for this subset (which yields a parser) and the valuation functions for the main constructs, e.g., the while loop, the if-then-else, and procedure definitions. For instance, consider the `while` construct. The syntax rule (in the DCG format) and the semantic rule are given below.

```
comm_x(while(B,C)) --> [while], bool(B),
     [loop], comm_x(C), [end, loop, ;].

comm_m(while(B,C),Store,NStore) :-
   (bool(B,Store) -> comm_m(C,Store,TStore),
       comm_m(while(B,C),TStore,NStore);
       NStore=Store).
```

The syntax rule `comm_x` states that the while statement consists of the keyword `while`, followed by a boolean expression, followed by the keyword `loop`, followed by the body of the loop, and then the keywords `end loop;`. The syntax rule also synthesizes the parse tree during parsing: the Prolog term `while(B,C)` is created, where `B` is the parse tree that corresponds to the boolean expression (and is produced by the rule `bool_x(B)` that parses that expression), and `C` is the parse that corresponds to the body of the loop (and is produced by the rule `comm_x(C)` that parses the body).

The semantic rule takes the parse tree for the `while` loop and a memory store (`Store`) as an argument, and produces the final memory store (`NStore`) that results after the while loop finishes execution. The rule is defined recursively and is self-explanatory. Similar syntax and semantic rules are given for the other constructs. Note that it took us only a couple of days to specify the semantics of the subset of Ada we needed.

Partially Evaluating the Denotation:
To make the process of verification more efficient, we partially evaluate the denotation $\mathcal{D}_p$, to obtain a more efficient denotation $\mathcal{D}^e_p$. We use the Mixtus [17] partial evaluator from the Swedish Institute of Computer Science. Let us denote the Mixtus system by $\mathcal{MIX}$. During partial evaluation definitions of certain predicates are hidden so that the original denotation is simplified to the point where these predicates are called. Then:

$\mathcal{MIX}(\mathcal{D}_p) = \mathcal{D}_p^e$. Now to perform verification, the goal

$$\text{Goal 2}: \qquad \mathcal{I}_p, \mathcal{D}_p^e, not(\mathcal{O}_p) \tag{2}$$

is now used instead.

Once the semantics of the Ada subset is defined, the denotation of the program $\mathcal{P}_{cntl}$, $\mathcal{D}_p$, is defined and partially evaluated to obtain $\mathcal{D}_p^e$. In fact, the main component of the actual partially evaluated program obtained ($\mathcal{D}_p^e$) is shown in Appendix I.

Notice how partial evaluation has resulted in the original denotation being "compiled" in terms of `build_control` operations (that build the control automata), `access_control` operations (that use the automata to compute the next command to be issued), and `access` and `update` operations (that access and update the memory store). The operations that perform the numerical computations are left as black-boxes during partial evaluation (their definitions are removed, but the calls to these computations are left in the denotation). This technique allows us to select the desired level of abstraction for the result of the partial evaluation.

Abstract Verification of the Controller:
Once the partial evaluation is complete, the definitions of `build_control`, `access_control`, `access` and `update` as well as the black-boxes are reinserted. The precondition and the negation of the postcondition are next added to obtain the Goal 2. Together, this whole thing can be run on a Prolog system, to verify the correctness of the original implementation. Running Goal 2 is considerably more efficient than Goal 1.

In case of the BART controller, the preconditions consists of the range of values that the input parameters can take. The controller implementation considers two consecutive trains at a time, and issues the command for the train that is behind (follow train). From a verification point of view we are interested only in proving the correctness of the actions taken by the controller when two arbitrary consecutive trains are considered. To calculate the next command for the follow train in a pair of consecutive trains, the parameters needed are: (i) a measure of the lead train acceleration, (ii) the relative velocities of the two trains, and (iii) the relative position of the follow train w.r.t. the lead train.

According to the specification of the problem developed by the group in Sandia [23], the preconditions are the following:

- $0 \leq LV \leq 80$
- $0 \leq FV \leq 80$
- $-2.0 \leq LA \leq 3.0$

where LV is the lead train velocity, FV is the follow train velocity, and LA is the lead train acceleration. Since we are primarily interested in verification, the controller code is written assuming that the follow train is just entering the track segment as the execution begins. Thus, the initial position of the follow train is assumed to be 0. The lead train is assumed to lie at the minimum safe distance from the follow train for the given values of LV, FV, and LA. Thus, we start out assuming that the train pair has been *correctly handed off* by the preceding station.

Verification involves showing that the minimum safe distance constraint will not be violated at any moment when the two trains are in the current track-segment, and that maximum allowable speed for the track-segment is never exceeded. This has to be shown for every possible velocity for the lead and the follow train, and every possible acceleration for the lead train. According to the specification, the follow train is always assumed to be at maximum acceleration at the beginning of the track for the calculation of the minimum safe distance. It should be noted that the minimum safe distance between two trains is a function of their velocities and accelerations and the track conditions (e.g., if the track is exposed or covered).

The Ada implementation is written so that if at any moment a violation of the safety conditions is detected, then a flag is raised, and execution stopped. The postcondition will check that the flag is never raised when the end of the execution is reached.

The precondition and the program denotation in conjunction with the negated postcondition should produce an inconsistency, if the design is correct. This is indeed what an ideal logic programming system should produce, when Goal (2) is run on it, with the precondition, program denotation and the postcondition appropriately replaced.

However, because of the involvement of non-linear equations in the computation of the safe distances between two trains, a Prolog system will not be able to deduce this inconsistency due to the lack of powerful enough constraint solvers for non-linear equations. To get around this problem, we make some approximations. We discretize the ranges in which the input variables lie. Thus, the precondition is transformed to the following:

```
member(LV, [1..80]),
member(FV, [1..80]),
member(LA, [3.0,2.0,1.89,1.26,1.0,0.78,
      0.63,0.52,0.26,0.0,-1.0,-1.5,-2.0]).
```

where `member(X,L)` is the standard Prolog program for checking if `X` is present in list `L`. Note that if `X` is uninstantiated and `L` is a ground list, then the Prolog interpreter will instantiate `X` to successive values from the list. Thus, `LV` and `FV` will be successively bound to integers from 1 through 80. The possible values for `LV` and `FV` are discretized in this way to limit the search space. Also, it should be noted that the minimum safe distance is not affected by small changes in velocities. For the lead train acceleration, `LA`, more specific possible values (shown above in the precondition) have been chosen. This is because the BART case study specified that acceleration rate depends on train speed. Therefore, three levels of acceleration were identified for each of the acceleration commands, representing estimated achievable accelerations at low, medium, and high train speed. These nine acceleration levels combined with three levels of deceleration and zero acceleration (propulsion) comprise the thirteen possible acceleration values for the lead train that were used for instantiating `LA` in the precondition above.

The precondition will thus act as a generator for values of LV, FV, LA. These variables will be successively instantiated to values in the list, and then substituted in the partially evaluated denotation. The negated post-condition (that the flag is not raised) will fail for every combination of values, if the program is correct.

We ran the above computation for verification on a Pentium II 233MHz workstation running Linux. The whole verification procedure took about 4 hours to complete. The design was indeed proved to be correct, which was our original goal.

Advantages of our Approach:
There are two major advantages of our approach: Firstly, given that our approach is based on logic programming and denotational semantics, we can do more interesting things with the denotation. We can take advantage of the fact that logic programs are reversible to derive interesting information about safety properties of the system. Suppose we fix the initial values for all but one parameter, then logic programming will allow us to derive the admissible values for the remaining parameters. Thus, for instance, suppose the lead train derails at position 10,000 feet from the beginning of the track (i.e., lead train velocity and acceleration are 0, and lead train position is 10000). Then, running the Goal 2, will deduce that the follow train velocity should be between 1 and 7 mph to avoid collision. Likewise, if we assume that the lead train is derailed at some position `X` feet from the beginning of the track, and the follow train enters the track segment at maximum speed, then the denotation can deduce that `X` must be greater than or equal to 16,557 feet to avoid collision. This can be generalized just as easily to cases where more than one parameter are unknown. In such a case, all admissible combinations for the unknown parameters will be generated. This is a distinct advantage of our approach, that is not present in other approaches known to us.

Secondly, note that this black-box approach coupled with partial evaluation allows us to perform verification incrementally. Thus, one would first verify that the design of the the black-boxes is provably correct. Once this correctness is proven, these black-boxes can be assumed to work correctly in the partially evaluated code. Breaking up the verification in several such steps can drastically reduce the state space to be explored during verification. Note also that this approach based on denotational semantics, partial evaluation, and black-boxes, also lets us verify very large programs. Essentially, unimportant parts of the code can be turned into black boxes, the code partially evaluated, and then verification performed on this simpler program obtained.

5 Verifying the Timing Properties

The model proposed so far relies on the fact that the control structure is simple and efficient, and commands are issued and delivered to the trains within the 500ms time frame. The use of different decision structures—e.g., use of more complex decision trees in the controller—may possibly violate this property.

The denotational semantics and Horn logic framework presented earlier can be extended to allow verification of timing properties as well. Let us focus, for the sake of simplicity, directly on the logic programming code produced from the Ada program denotation. This code can be easily extended to keep track of the time at which the different events take place. If the estimate of the time taken by the BART controller to perform one step in the decision automata (which depends on many factors, e.g., CPU speed of the VSC/NVSC) is given, then the timing properties can also be verified. This is done by extending the postcondition with appropriate values for the time (e.g., the final execution time should meet the constraint $T \leq 500$). Details are not given due to lack of space, however, this approach has been applied by the authors to verify timing properties of other real-time systems—a description can be found elsewhere [8].

6 Conclusion and Related Work

In this paper we presented a semantics-based approach to verify controller programs written in Ada. We make synergistic use of many techniques that have been developed in the programming languages community to aid in automatic verification: denotational semantics, logic programming, and partial evaluation.

Our approach is novel in that it uses logic-based denotational semantics for automatic verification. Denotational semantics has not been widely used for automatic verification; it has been mostly used for manually analyzing programming language features and for manually proving equivalence between programs [18]. The main reason we were able to use denotational semantics for automatic verification is that we chose to write denotational definitions using logic programming.

We believe that an advantage of our approach is that it is easy to use by non-expert programmers, since the Horn logical semantics of Ada needs to be written only once. The programmer needs to specify the pre-conditions on the inputs, namely, the range of values they can take, and the postcondition that needs to be checked. The programmer is also responsible for abstracting the range of values for inputs sufficiently, so that the verification process will finish in reasonable amount of time. Another advantage of our approach is that the use of partial evaluation allows us to prove the correctness of different subparts of the program in an incremental way. Also, our approach allows the programmer to "play" with the denotation of the program (by making simple changes to the precondition and the postcondition), and deduce interesting information. We illustrated our approach on a sufficiently large example (the controller of the Bay Area Rapid Transit system) provided by others. Thus, the main contribution of this paper is to illustrate our logic programming, denotational semantics, and partial evaluation based framework, and how it can be applied to solve large, real-life problems. We have also successfully applied our framework to other scenarios: provably correct parallel compilation [7], verification of real-time systems [8], domain specific languages [9], etc.

There is considerable amount of work in the area of verification and model checking of software systems [15, 3]. Work has also been done on verification of Ada programs [2, 4, 12, 14]. Leveson et al [12] suggest *fault tree analysis* as a technique for finding safety violations in Ada controller programs. However, this analysis has to be done manually. In contrast, while we used fault tree analysis to come up with the design of the BART controller, the verification step in our framework was largely automatic. Gauspari et al [4] present a tool called Penelope for verifying Ada programs. Their approach, while more complete than ours (finiteness restriction need not be imposed on preconditions), requires more information from the user. At the very least the user will have to specify the loop invariants. Thus, Gauspari's approach uses the classic Hoare style verification technique. Dillon considers verification of concurrent Ada programs, however, that is not our central focus. Luckham et al [14] have designed the *Anna* notation. In this approach the user annotates a program with assertions; specialized tools are then used that make use of the annotations to aid in debugging of the program. In some respects their approach is similar to ours, however, extensive annotation is required in their case, whereas in our framework, all that is needed is the precondition and the postcondition. Work has also been done on verification of the BART controller by others. A collection of papers can be found in High Integrity Software Conference proceedings [22].

REFERENCES

[1] C. Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. *Symp. on Partial Eval. and Semantics-Based Program Manipulation*, 1993.

[2] L.K. Dillon. Using symbolic execution for verification of Ada tasking programs. ACM TOPLAS. 12(4), 1990.

[3] J-P Finance (ed). Foundational Aspects of Software Engineering. Springer Verlag. 1999.

[4] D. Guaspari et al. Formal Verification of Ada Programs. TSE 16(9): 1058-1075, 1990.

[5] C. Gunter. Programming Language Semantics. MIT Press. 1992.

[6] G. Gupta "Horn Logic Denotations and Their Applications," *The Logic Programming Paradigm: A 25 year perspective*. Springer Verlag. 1999.

[7] G. Gupta, E. Pontelli, et al. Automatic Derivation of a Provably Correct Parallelizing Compiler, *Int. Conf. on Parallel Processing*, IEEE Press, 1998.

[8] G. Gupta, E. Pontelli. A Constraint-based Denotational Approach to Specification and Verification of Real-time Systems. In *Real-time Systems Symposium*, IEEE, 1997.

[9] G. Gupta, E. Pontelli. A Horn Logic Denotational Framework for Specification, Implementation, and Verification of Domain Specific Languages. Tech. Report, NMSU, 1999.

[10] J. Jaffar, M.J. Maher. Constraint Logic Programming. In *Journal of Logic Programming*, Vol. 19/20, 1994.

[11] N. Jones. Introduction to Partial Evaluation. In *ACM Computing Surveys*. 28(3):480-503.

[12] N. G. Leveson et al. Safety Verification of Ada Programs Using Software Fault Trees. IEEE Software 8(4), 1991.

[13] J.W. Lloyd. Foundations of Logic Programming. Springer Verlag. 2nd ed. 1987.

[14] David C. Luckham, et al. Two-Dimensional Pinpointing: Debugging with Formal Specifications. IEEE Software 8(1): 74-84 (1991)

[15] M. Müller-Olm et al. Model checking: a tutorial introduction. Proc. Static Analysis Symp. Springer, 1999.

[16] W. Polak. Program Verification Based on Denotational Semantics. ACM POPL 1981. pp. 149-158

[17] D. Sahlin. An Automatic Partial Evaluator for Full Prolog. Ph.D. Thesis. 1991. Royal Institute of Technology.

[18] D. Schmidt. *Denotational Semantics: a Methodology for Language Development*. W.C. Brown Pub., 1986.

[19] D. Schmidt. Programming language semantics. In Handbook of Computer Science, CRC Press, 1996.

[20] L. Sterling & S. Shapiro. The Art of Prolog. MIT Press, 1994.

[21] J. Stoy. Denotational Semantics: The Scott-Strachey Approach to Programming Languages. MIT Press. 1977.

[22] V. Winter, R. Berg (eds). High Integrity Software Conference. Sandia National Labs. 1999.

[23] V. Winter et al. Bay Area Rapid Transit District Advance Automated Train Control System: Case Study Description. Technical Report, Sandia National Labs, 1999.

Appendix I: Partially Evaluated Code

Note that the code below corresponds only to the the main loop which computes the commands to be sent to the follow train and the part of the controller that synthesizes the fault tree.

```
main(accel, high, far, A, B) :-
   mainaccel1(A, B).

mainaccel1(A, B) :-
   initialize(C),
   initialize(D),
   update('LA', 3.0, D, E),
   update('LV', 5, E, F),
   update('LP', 10800, F, G),
   update('FV', 5, G, H),
   update('FP', 9003, H, I),
   update('SDV', 3.0, I, J),
   update('A', accel, J, K),
   update('V', high, K, L),
   update('P', far, L, M),
   update('M', 1800, M, N),
   update('ID', b24, N, O),
   build_control(sStart, accel, sLA, C, P),
   build_control(sStart, propl, sLP, P, Q),
   build_control(sStart, decel, sLD, Q, R),
   build_control(sLA, high, sCA, R, S),
   build_control(sLA, equal, sCA, S, T),
   build_control(sLA, low, sCA, T, U),
   build_control(sLP, high, sCA, U, V),
   build_control(sLP, equal, sL1, V, W),
   build_control(sLP, low, sCD, W, X),
   build_control(sLD, high, sL1, X, Y),
   build_control(sLD, equal, sCD, Y, Z),
   build_control(sLD, low, sCD, Z, A1),
   build_control(sCA, far, sCAmax, A1, B1),
   build_control(sCA, medium, sCAmed, B1, C1),
   build_control(sCA, close, sCAmin, C1, D1),
   build_control(sL1, far, sCAmin, D1, E1),
   build_control(sL1, medium, sCP, E1, F1),
   build_control(sL1, close, sCP, F1, G1),
   build_control(sCD, far, sCDmin, G1, H1),
   build_control(sCD, medium, sCDmed, H1, I1),
   build_control(sCD, close, sCDmax, I1, J1),
   commwhilelessthanvarLP1(O, A, J1, B).

commwhilelessthanvarLP1(A, B, C, C) :-
(   access('LP', A, D),
    D<28115 ->
    black_box(get_lead),
    black_box(get_follow),
    black_box(dmin),
    black_box(create_control),
    black_box(lead),
    black_box(velocity),
    black_box(position),
    access('A', A, E),
    access('V', A, F),
    access('P', A, G),
    access_control(E, sStart, C, H),
    access_control(F, H, C, I),
    access_control(G, I, C, J),
    update('CC', J, A, K),
access('CC', K, L),
(   L==sCAmax ->
    (   access('FV', K, M),
        0=<M,
        M<30,
        access('FV', K, N),
        O is N+10,
        update('V_NEW', O, K, P),
        update('A_NEW', 3.0, P, Q)
    ;   access('FV', K, R),
        30=<R,
        R<61,
        access('FV', K, S),
        T is S+7,
        update('V_NEW', T, K, U),
        update('A_NEW', 1.89, U, Q)
    ;   access('FV', K, V),
        61=<V,
       V<80,
        access('FV', K, W),
        X is W+4,
        update('V_NEW', X, K, Y),
        update('A_NEW', 0.78, Y, Q)
    )
;   Q=K
),
access('CC', Q, Z),
(   Z==sCAmed ->
    (   access('FV', Q, A1),
        0=<A1,
        A1<30,
        access('FV', Q, B1),
        C1 is B1+8,
        update('V_NEW', C1, Q, D1),
        update('A_NEW', 2.0, D1, E1)
    ;   access('FV', Q, F1),
        30=<F1,
        F1<61,
        access('FV', Q, G1),
        H1 is G1+6,
        update('V_NEW', H1, Q, I1),
        update('A_NEW', 1.26, I1, E1)
    ;   access('FV', Q, J1),
        61=<J1,
        J1<80,
        access('FV', Q, K1),
        L1 is K1+4,
        update('V_NEW', L1, Q, M1),
        update('A_NEW', 0.52, M1, E1)
    )
;   E1=Q
),
access('CC', E1, N1),
(   N1==sCAmin ->
    (   access('FV', E1, O1),
        0=<O1,
        O1<30,
        access('FV', E1, P1),
        Q1 is P1+6,
        update('V_NEW', Q1, E1, R1),
        update('A_NEW', 1.0, R1, S1)
    ;   access('FV', E1, T1),
        30=<T1,
        T1<61,
        access('FV', E1, U1),
        V1 is U1+5,
```

```
        update('V_NEW', V1, E1, W1),
        update('A_NEW', 0.63, W1, S1)
    ;   access('FV', E1, X1),
        61=<X1,
        X1<80,
        access('FV', E1, Y1),
        Z1 is Y1+4,
        update('V_NEW', Z1, E1, A2),
        update('A_NEW', 0.26, A2, S1)
    )
;   S1=E1
),
access('CC', S1, B2),
(   B2==sCDmax ->
    (   access('FV', S1, C2),
        0=<C2,
        C2<30,
        access('FV', S1, D2),
        E2 is D2-8,
        update('V_NEW', E2, S1, F2),
        update('A_NEW', -2.0, F2, G2)
    ;   access('FV', S1, H2),
        30=<H2,
        H2<61,
        access('FV', S1, I2),
        J2 is I2-8,
        update('V_NEW', J2, S1, K2),
        update('A_NEW', -2.0, K2, G2)
    ;   access('FV', S1, L2),
        61=<L2,
        L2<80,
        access('FV', S1, M2),
        N2 is M2-8,
        update('V_NEW', N2, S1, O2),
        update('A_NEW', -2.0, O2, G2)
    )
;   G2=S1
),
access('CC', G2, P2),
(   P2==sCDmed ->
    (   access('FV', G2, Q2),
        0=<Q2,
        Q2<30,
        access('FV', G2, R2),
        S2 is R2-6,
        update('V_NEW', S2, G2, T2),
        update('A_NEW', -1.5, T2, U2)
    ;   access('FV', G2, V2),
        30=<V2,
        V2<61,
        access('FV', G2, W2),
        X2 is W2-6,
        update('V_NEW', X2, G2, Y2),
        update('A_NEW', -1.5, Y2, U2)
    ;   access('FV', G2, Z2),
        61=<Z2,
        Z2<80,
        access('FV', G2, A3),
        B3 is A3-6,
        update('V_NEW', B3, G2, C3),
        update('A_NEW', -1.5, C3, U2)
    )
;   U2=G2
),
access('CC', U2, D3),
(   D3==sCDmin ->
    (   access('FV', U2, E3),
        0=<E3,
        E3<30,
        access('FV', U2, F3),
        G3 is F3-4,
        update('V_NEW', G3, U2, H3),
        update('A_NEW', -1.0, H3, I3)
    ;   access('FV', U2, J3),
        30=<J3,
        J3<61,
        access('FV', U2, K3),
        L3 is K3-4,
        update('V_NEW', L3, U2, M3),
        update('A_NEW', -1.0, M3, I3)
    ;   access('FV', U2, N3),
        61=<N3,
        N3<80,
        access('FV', U2, O3),
        P3 is O3-4,
        update('V_NEW', P3, U2, Q3),
        update('A_NEW', -1.0, Q3, I3)
    )
;   I3=U2
),
access('CC', I3, R3),
(   R3==sCP ->
    (   access('FV', I3, S3),
        0=<S3,
        S3<30,
        access('FV', I3, T3),
        U3 is T3,
        update('V_NEW', U3, I3, V3),
        update('A_NEW', 0.0, V3, W3)
    ;   access('FV', I3, X3),
        30=<X3,
        X3<61,
        access('FV', I3, Y3),
        Z3 is Y3,
        update('V_NEW', Z3, I3, A4),
        update('A_NEW', 0.0, A4, W3)
    ;   access('FV', I3, B4),
        61=<B4,
        B4<80,
        access('FV', I3, C4),
        D4 is C4,
        update('V_NEW', D4, I3, E4),
        update('A_NEW', 0.0, E4, W3)
    )
;   W3=I3
),
access('LP', W3, F4),
G4 is F4+1000,
update('LP', G4, W3, H4),
access('FV', H4, I4),
access('A_NEW', H4, J4),
K4 is I4+J4/2,
update('FV', K4, H4, L4),
commwhilelessthanvarLP1(L4, B, C, C)
;   B=A
).
```