A Survey: Applying Formal Methods to a Software Intensive System

Adriaan de Groot* and Jozef Hooman
KU Nijmegen

Fabrice Kordon, Emmanuel Paviot-Adet, and Isabelle Mounier
Laboratoire d'Informatique de Paris 6/SRC

Michel Lemoine and Gervais Gaudiere
ONERA Cdt/DPRS/SAE and ENAC - DMI

Victor L. Winter†‡ and Deepak Kapur§
Sandia National Laboratories

Abstract

This paper surveys various formal approaches that could be used to facilitate the development of the train control system described in BART case study. This system is interesting because train control must take into account complex behaviors, positional uncertainties, noise, continuous aspects, and a predefined computational architecture. The approaches discussed are works in progress and are not complete at the time of the writing of this paper.

1 Motivation

The rapid advance of computer technology has lead to increasingly high consequence applications across the technology spectrum. More and more, systems are falling into two distinct categories (1) those which are driven by time-to-market considerations, and (2) those having dependability as a dominating concern. This paper focuses on systems belonging to the latter category.

When developing a high consequence system, significant *a priori* evidence must be provided that the system will meet its dependability requirements. For complex systems having high (or ultra-high) dependability requirements (e.g., the likelihood of a failure occurring per operational hour $\leq 10^{-9}$) testing alone is generally insufficient to provide the necessary level of assurance. For such applications, formal methods are considered to be the most promising approach for providing the (ultra)high assurance necessary to confidently field the developed system.

Because formal methods offer such promise, governments in a growing number of countries explicitly require some degree of formal methods to be applied to the development of high consequence systems (especially safety critical systems whose failure will result in the loss of human life).

At present, formal methods have not gained the level of use that many had hoped for. There are several reasons for this: (1) programming languages and tools supporting traditional software engineering practices have lead to significant improvements in software development, (2) formal methods-based development approaches typically require a deep understanding of various types of logics, related notations, and calculations (e.g. unification, inference, etc.), thus making them somewhat inaccessible to the general programmer/developer, (3) it is not uncommon for a formal specification to approach or even exceed the size of the program that implements it, and (4) in practice, it has been very difficult to scale formal methods to large systems.

Many people in industry view formal methods as still being in the research stage and are therefore not convinced that the cost of their application to a real project can be justified. In practice, testing can often demonstrate reliability levels acceptable to short-term industrial goals (e.g., a short-term view of system reliability may conclude that it is "unlikely" for a system having a reliability of 10^{-6} to experience a failure in the near future). However, as the consequence of failure increases, companies are increasingly finding themselves developing products having properties that are in conflict with such "short-term" corporate

*Supported by NWO-GBE/SION project 612.062.000

†Contact Author

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

§Deepak Kapur was also partially supported by NSF grant nos. CCR-9996150 and CDA-9503064.

philosophies.

At present, institutions conducting research in formal methods (typically universities) have been largely restricted to using "toy problems" or case studies to demonstrate the maturity and/or appropriateness of a formal method to a particular aspect of a problem or problem domain. Toy problems are problems whose primary objective is to demonstrate the capability of a method. They typically represent highly idealized problems and lack the complexity that would allow them to demonstrate that a method can be scaled to real world problems. In contrast, case studies are derived from real world problems and to a reasonable extent reflect the complexity and messiness present in problems being addressed by industry. Research in the application of formal methods to case studies represents a logical next-step in the development of an industrial strength formal method. Unfortunately, only a handful of case studies have been made available to the formal methods community. Two of the most well-known case studies are the Production Cell [7] and the Steam Boiler [1], however, case studies of this caliber are rare.

There are numerous reasons why producing a case study is unattractive to industry. First, the feedback cycle (i.e., the time from when the case study is written to when the research community presents results) is typically on the order of a year or more. This is simply too slow for the timetables of many industries. Additionally, there are often proprietary and legal issues surrounding the development of case studies (e.g., the release of a product specification to a competitor). Given such constraints, it is rare for a company to consent to the development of a case study based on a system that is currently under development. This leaves case studies for systems that have already been developed. While such case studies are of interest to the research community, they are typically not of immediate interest to industry. This is evidenced by the difficulties encountered when trying get industry to devote the necessary resources in order to develop such a case study.

The BART Case Study [17] focuses on the acceleration and speed control for the commuter trains running in the San Francisco bay area. It resulted from a unique set of circumstances in which Sandia National Laboratories was engaged to evaluate safety criteria for the new train control system that was being developed for the Bay Area Rapid Transit (BART) system. At the corporate level, Sandia recognizes the need for research in high consequence software and system development and was therefore willing to devote the necessary resources in order to produce the case study. At the time this case study was developed, the contract for BART had already been awarded (minimizing though not eliminating the concern regarding proprietary information) and the system was about 8 - 12 months away from being implemented.

2 Overview of the BART Case Study

BART is a train system that runs through the San Francisco Bay Area. The use of the BART system was beginning to exceed its capacity and a new system was needed to address this problem. Because BART runs through downtown San Francisco and under the bay, solving the capacity problem by laying additional track was cost prohibitive. The solution that was accepted addressed the capacity problem by spacing the trains closer together. In the old system, trains were typically spaced about 2 minutes and 30 seconds apart. In the new system trains would be spaced around 2 minutes apart. Because train accidents (e.g., collisions or derailments) can result in loss of life, the BART controller is considered to be a high consequence system.

There are many constraints that need to be considered when controlling a train. Trains can take a long time to stop, track segments have different speed limits associated with them, signals (set by the environment) can require a train to stop, a train must respond to the behavior of the train ahead of it, the environment can cause a train to derail, and so on and so forth. A train control system must take all of these constraints into account in the context of achieving various system objectives. The primary objective is safety, followed closely by various other objectives such as increased capacity, smoothness (e.g., the train should not oscillate), energy efficiency, and passenger comfort. The interaction of these system objectives and constraints results in a complex high consequence system.

3 Challenges Posed by the BART Case Study

Many of today's formal methods are based on the ability to construct a system model in which *transitions from*, or *relations between* one state and another can be expressed in a simple fashion. The complexity in such systems arises from the interaction between these

relations or transitions as well as their ability to satisfy (long-range) system goals. In such a framework, tools (e.g., model checkers, theorem provers, etc.) can be effectively used to verify properties such as reachability, consistency, liveness, deadlock and so on.

For example, the Production Cell Case Study[7], describes a robotic system in which simple commands are given to the various components in the Cell (e.g., robot arm, conveyor belt, crane, elevating-rotating table, etc.) in order to control the behavior of the system. The analysis of the suitability of a command can be largely accomplished via static reasoning (e.g., if the robot is turned to the left, it will collide with the elevating-rotating table).

The BART Case Study differs from case studies like the Production Cell because the suitability of a transition (e.g., commanding a particular acceleration) must be reasoned about with respect to long range objectives. For example, in the worst case, a train traveling at 80 m.p.h. on a continuous 4% downgrade will take over 3 miles and 4 minutes to stop! Thus, selection of an acceleration value can have far reaching consequences. Simple models of a train's state space can be constructed, but these models tend to be extremely large. Further increasing the size of such state spaces is the presence of noisy transmission of commands (i.e., an acceleration command may not be received by a train).

Additional aspects of BART that contribute to the complexity of models are positional uncertainties which are defined in probabilistic terms and the presence of an existing infrastructure (including computers) that restricts the solution space, giving the case study an "evolutionary" dimension.

Most of the widely used formal methods approaches have difficulty dealing with (1) continuity, (2) behavior, and (3) probability. The BART case study contains all of these aspects and is therefore an interesting case study to benchmark the effectiveness of various formal approaches to software development.

4 Analysis of the BART system

We believe that effectiveness of formal methods can be dramatically increased by "tuning" them to a specific domain. The objective is to create a formal framework which makes explicit use of domain knowledge at the semantic level that is used by the domain experts themselves. This approach is based on the premise that domain experts in mature fields have developed abstractions and operations that enable them to effectively develop the systems in which they are interested. Thus a formal approach to software development in such domains should try to take advantage of this knowledge and experience to the extent that is possible.

In following subsections, four formal approaches to this case study are surveyed. This is followed by a final section in which the appropriateness of formal methods to the development of such systems is discussed.

4.1 Formal Methods as a Means to Construct a Correct SRD

As mentioned in previous sections, formal methods have not shown their ability to tackle the inherent complexity of real systems. In this section, we demonstrate that semi-formal and formal methods can be fruitfully used, at a real scale, to produce a System Requirements Document (SRD) that meets the following expected qualities:

- lack of ambiguities – every piece of information used should be precisely defined in order to get a unique and shared semantics.
- completeness – the guarantee that all the needed information, for instance all the critical events, are referenced and specified.
- consistency – no contradiction still remains between specified elements.

Indeed the success key of the development of any safety critical system is mainly based on a clear and shared understanding of what the systems should do, and of what are the constrains it must satisfy.

Elicitation of users' needs: As advocated by the EAI-632 [15] standard, the first step of software or system development consists of eliciting the needs of the user. This elicitation can be accomplished using a variety of methods. In the case of the BART system, we have used an object-oriented set of notations provided by the UML [12]. Using this notation, needs of the user have been captured by a large complex Class Diagrams that includes both structural aspects and most of the constraints that the BART system must satisfy.

As usual with UML, the validation phase is a 2-step one, made of a reverse engineering step transforming the semi-formal diagram into informal texts that should then be validated by an inspection from the end user. Not being able to interact with end users, we have validated the Class Diagram by "cross-reading"

(i.e., comparing the Class Diagram to the case study text). Note that such cross-reading does not guarantee that the developers have a sufficient understanding of the system.

In order to overcome this problem, we have given a clear semantics to each operation. These definitions were written in Z [14], a formal language based on the ZF set theory and first order predicate calculus. Z was chosen because it supports the object-oriented aspects we have emphasized in our approach. We have exhibited a set of transformation rules enabling automatic generation of state and operation schemas. Of course, the operational semantics were produced manually. During this manual step we have been faced with 2 main problems:

- Giving a semantics to each operation.
- Verifying the operation properties.

The latter is supported by the Z method. It is not too difficult to check that the specified operations verify (or not) properties such as satisfiability, completeness and consistency.

The former is much more interesting. Indeed, giving a semantics to each semi-formal operation has lead us to either a formal text, and thus its formal verification, or to the discovery of some mistakes in the source class diagram. What kind of mistakes have been discovered? Most of them are related either to a misunderstanding of the intuitive semantics of the operation, or to an incorrect interpretation of relationships between classes.

In other words, giving a formal semantics to semi-formal models (e.g., UML class diagrams) is a process that leads to the discovery of many errors. With this, we concluded the first phase. In the second phase, we focused on: how to use this information to build an SRD having the expected properties?

Rigorous Construction of an SRD: In this phase, we used the evolutionary method presented at [6]. Here, the process of developing an SRD consists of three major steps:

1. The formal specification and validation of a stable kernel derived from a subset of our large class diagram - we begin with a semi-formal model that is considered as being representative of the kernel of the system.
2. The formal specification and validation of increments - we first of all extend the semi-formal model by adding as many attributes, operations, classes and relationships as necessary, we then formalize them, and validate them. Whenever the validation is not reached, we decrement - erase - the current increment and move in another direction.
3. Finally, as soon as we consider the model as complete, we translate both the semi-formal and formal models into an SRD.

The lessons we have learned when using such a way of elicitating end users' need and their formalization are rather obvious. First of all it is mandatory to develop some semi-formal models that can be easily understood and validated by end users. This step also makes the development team more knowledgeable with the system domain. The second step that consists of giving the semantics of operations allows to discover the underlying mistakes the semi-formal models might have. The third step is a total rebuild of the SRD because we proceed in a bottom up manner, starting from a stable kernel and arriving to a large model, which is incrementally produced and validated. Finally, the System Requirements Document is direct translation of the validated semi-formal and formal models.

It is important to consider that with such a way of producing SRD, the development team must have an answer to any question it can ask the end users. Indeed, at this point we can guarantee that the SRD is satisfiable, complete, and consistent.

4.2 Discovering the Environment

From a requirements engineering point of view it is essential to know what parts of the system are already given and what they do. This is very similar to the SRD's of Section 4.1, but since we are describing what is *there*, not what is *desired*, we proceed slightly differently. To this end we concentrate on the trains used by the BART system. In modeling the train there are no properties to be verified: the essential question is one of *validation*: does the train model we produce match the actual train in all essential properties? What *is* an essential property?

This phase of modeling and validation plays a role in the entire software process. The models derived here serve as a context within which the software artifacts — the software for the station controller — is expected to perform. Without a valid model of the train, it is impossible to draw conclusions about the behavior of the combination of trains and controller. We contend that a good model of the train helps in

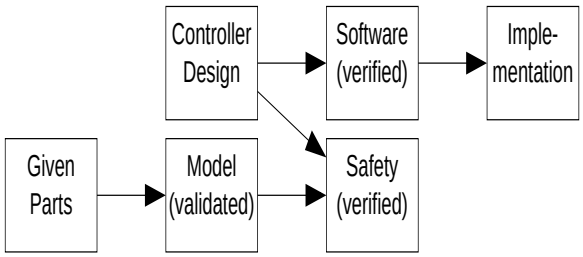

Figure 1: Requirements engineering and the Software process

the actual design of the controller as well due to the increased understanding of the train caused by making the model.

Our ideal software process is shown in figure 1. We have performed here only the first step — modeling the given components. The remaining steps are imagined as follows: We model BART as the parallel composition of two parts: the trains and the station controllers. We then state safety properties in terms of the (un)reachable states of that parallel automaton. In particular, the safety property stated in the informal specification could be translated (very) roughly as

$$\forall t.\mathrm{pos}(\mathrm{Train2}) > \mathrm{pos}(\mathrm{Train1}) + 700\mathrm{ft}$$

The exact formulation of these properties depends on the formalization we choose; we are inclined to translate a labelled transition system into a set of PVS theories [9] and use an inductive assertion method [11] to verify the properties of the parallel concurrent composition of the labelled transition systems.

Modeling objects: To identify what is given we use a fairly straightforward noun-identification technique common in UML [12]. This gives us a class diagram with classes TrainPhysical (to model the physical behavior of the train) and TrainController (controlling the motors and brakes of the train). We also attempt to assign the actions taken by the train to these classes. This leads to the creation of additional classes TrainLimiter (to model the way a train accelerates smoothly up to 2 mph below the commanded speed) and TrainSensor (for the communication back to the station). A class Train is invented to aggregate objects of these four classes and to handle communications received from the station. This yields the class diagram as shown in figure 2.

We aim to model the behavior of the train as a whole by attaching to each class in the class diagram one or more statecharts that express how the objects

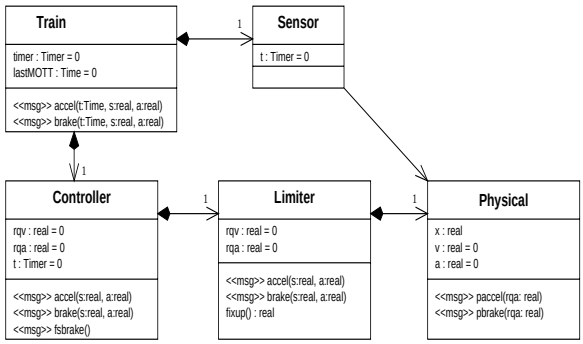

Figure 2: Class diagram for BART trains

react to various messages and the passage of time. The criteria for the class diagram are that each aspect of the train's behavior must fit sensibly into one of the classes. As we examine the train's behavior we return to the class diagram and can add classes, methods, or attributes as needed to be able to describe that behavior.

This process led to multiple versions of the class diagram. For example, only when we considered the way a train smoothly accelerates up to 2mph below the commanded speed did it become apparent that we might need the TrainLimiter class. When the statecharts required to capture the behavior of the limiter became too complex, we moved that complexity into an additional method (called fixup) of the class.

Modeling behavior: Including continuous real-time phenomena in UML diagrams is not covered by the UML standard at all as yet. Even including timers is problematical. However, we can draw upon the theory of (hybrid) timed automata to assure ourselves that such extensions are possible, even if their semantics in the context of UML is unclear.

It seems natural to model the physical behavior of the train as a continuous real-time process. This introduces additional complications into the statecharts. We could also have taken the statement from the informal specification that "the system operates at half-second cycles" at face value to assume a synchronous system where commands come into effect only at the half-second "tick" of the system. This would replace continuous variables by variables whose values change according to a definite integral every half second. Since this synchronous approach seems implausible, we remain with the asynchronous continuous real-time system, which yields the hybrid real-time statechart shown in figure 3.

The extensions in figure 3 are the real-time actions

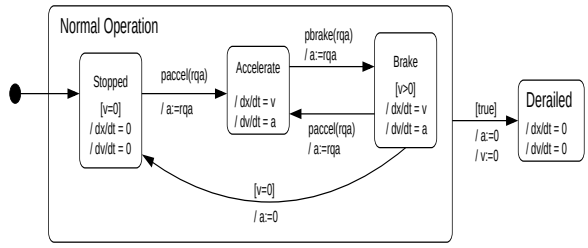

Figure 3: Statechart for physical behavior

with differential equations (shown as actions within the states) and state invariants (shown as guards within the states).

Validation results: This approach — finding classes and assigning behavior to them — leads us to a collection of diagrams. The validation of these diagrams is a matter of insight and careful consideration of the statechart diagrams. For example, in our original statecharts for the TrainController, there was no transition for the command accel for every state. Actually writing the statecharts makes it clear that the informal specification is incomplete with respect to this transition, and we need to ask a domain expert what is intended. This kind of completeness checking is easy to automate and helps remove ambiguities from the informal specification.

By constructing possible traces of the statecharts we can examine the possible behavior of the system and compare this with our intuitions. At the moment this remains a purely intuitive game.

Conclusion: Using a formal notation helps us order our thoughts, find ambiguities and incompleteness, and provides a secure footing to proceed with the application of formal methods to the *other* part of the BART system, namely the station controllers.

4.3 Formal verification of speed control strategy with Petri nets

The train speed controller system has some interesting features we wanted to deal with:

- *time* – In our framework (that of Petri nets) time cannot be modeled as a continuous flow. Since the train speed controller is structured on cycles, we discretize time following this structure. The computer receives trains position and computes new instructions to be sent to trains.
- *physical properties of the system* involve complex phenomena like gravity, friction, etc. Introducing them in our model disables structural verification capabilities and dramatically increases the complexity of the underlying state space. For this reason, we used computation tables stored as composed tokens in dedicated places. These complex behaviors are then pre-computed and integrated in the specification.

Properties to be verified: The properties we want to verify deal with stops and collisions:

P_1 trains stop at stations and go from one station to the next one as fast as they can,

P_2 trains never collide, even if one train derails.

Modeling: Our modeling procedure relies on **L***f***P**, a UML-like semi-formal notation strongly related to well-formed Petri nets [2]. **L***f***P** is dedicated to code generation for distributed embedded systems [10] and formal verification of behavioral aspects in such systems by means of automatically generated Petri nets [5].

L*f***P** is a first step to the definition of a CASE environment dedicated to the modeling, verification and code generation for distributed embedded systems. Since the approach is not yet implemented, we designed our Petri nets "as if" they were produced from **L***f***P** specifications.

We structured the model in three components following a **L***f***P** like strategy: c_1 (the speed controller), c_2 (the environment) and c_3 (train constraints). The components c_1 and c_2 are related using a shared memory like mechanism: tables representing the environment behavior can be read by the speed controller. The components c_1 and c_3 are related using a synchronous mechanism: the train behavior introduces constraints that must be satisfied by the speed controller (e.g., a train cannot stop instantaneously).

Since time has been discretized, distance and speed must also be discretized. A distance unit is used, corresponding to the shortest distance covered by the train when at slowest (but non-zero) speed. All environmental interactions can be modeled using calculus tables.

Verification strategy: To complete the proof, two Petri net models must be derived from the **L***f***P** specification.

Figure 4 shows the Petri net produced to verify P_1. It corresponds to the behavior of a train going

from a station to another one. Some transitions are fired when the train misses a station (*MissStation* and *TooEarly*). The track is "circular" and thus, a train with a correct behavior should never stop.

Variables are defined as follows:

- *tsp*: current train speed,
- *db*: distance to station before the move,
- *da*: distance to station after the move,
- *da2*: distance to station after the move if the train accelerates,
- *ds*: distance to stop at current speed,
- *ds2*: distance to stop if speed is increased,
- *tid*: the train identifier.

Some places model the environment: *NewDistTable* computes a new distance after a move at a given speed, *StopTable* computes the distance to stop from a current speed[1] and *DistStation* corresponds to a set of possible distances between stations. Place *TrainState* stores a set of trains (to validate P_1, we only need one train). There is a train per token (here, the only train has identity 1, speed 0 and is in a station (distance to station = 0).

Some transitions model the following strategy:

- *AtStation*: When a train is at a station, it accelerates and receives the distance to the next station.
- *TrainAcc*: If the distance is sufficient not to miss the station, the train can accelerate.
- *TrainStable*: The train keeps its speed if the distance is not sufficient to accelerate, but is sufficient if speed remain constant.
- *TrainDec*: If the distance is not sufficient to maintain a constant speed, the train must decelerate.
- *TrainStop*: When train speed is one and distance to the station is one, the train has to stop.

Validation for this model was done using CPN-AMI [8], our Petri-net based CASE environment. First we generate the state space using the PROD model checker [16] in CPN-AMI. We then observed that a terminal state was not reached. It means that transitions *MissStation* and *TooEarly* are never fired (they generate a terminal state).

[1] Using more parameters such as grade is easy. It would add more input entries to the table (only *tsp* is used in Figure 4).

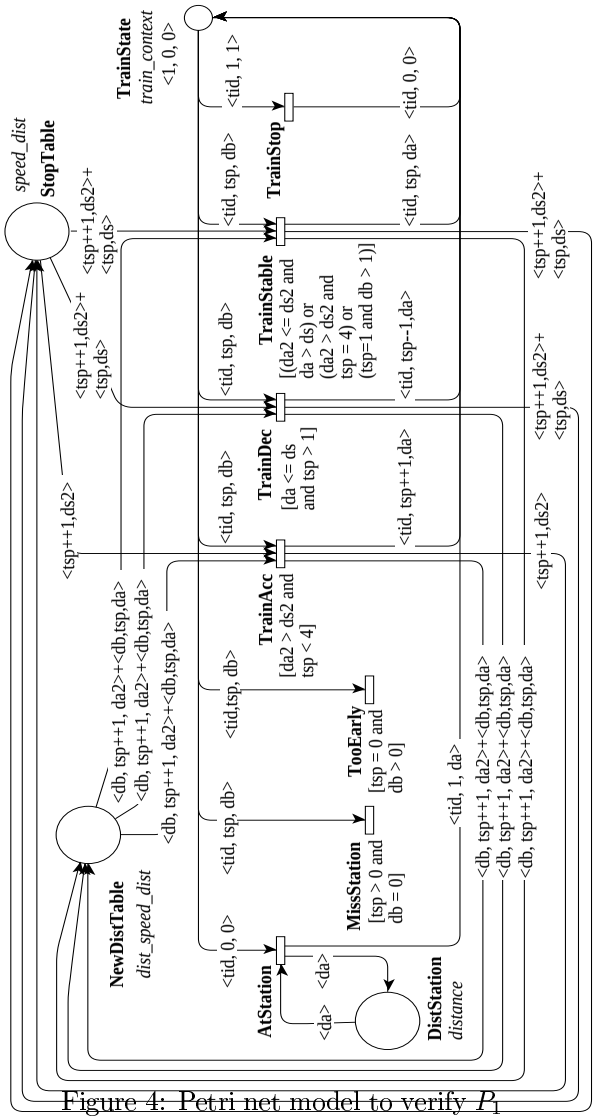

Figure 4: Petri net model to verify P_1

We verified P_2 using a similar modeling strategy. The main difference is that the environment contains a "crazy train" moving randomly (moving backwards is not allowed). The studied train then has to never crash into the "crazy" one.

Due to space constraints, we are unable to show this model. It models a section of track. When the "crazy" train reaches the end of the path, we artificially block the model. Similarly to the model defined for P_1, an *accident* transition blocks the system when the distance between the "crazy" train and the studied one is below a predefined limit.

We used the same strategy to validate the second model. Model checking shows that the only terminal state in the state space occurs when the "crazy" train reaches the end of the modeled path.

Lessons Learned: Our validation strategy is thus composed of two different levels: an environmental description with its own validation procedure and validation of the speed command strategy correctness. The environmental description is compiled in the calculus tables (component c_2) that are used in the Petri net model describing the speed control strategy.

Our two levels validation strategy keeps the obtained model within Petri nets' expression power bounds. Therefore all kind of analysis can be performed: model-checking, but also structural analysis. Furthermore, the resulting model focuses on the main problem (the speed control strategy) and leaves to another level of verification the environmental part of the problem. As a consequence, the resulting model has a small reachability graph. Finally, this validation strategy is general enough to allow different level of specification of the environmental part, and thus an "easy" incremental construction of the final model.

4.4 A Refinement-based Development Approach

The High Integrity Software (HIS) group at Sandia National Laboratories, is exploring the potential offered by the application of highly domain specific formal methods to the software development process. To date, the primary tool that we have developed to assist us in the exploration of this type of development is a general purpose program transformation system called HATS [18]. HATS is a language independent rewrite-based transformation system. Software development takes place within a context-free wide-spectrum language containing both the domain language and the implementation language.

Properties to be Verified: Our verification effort was focused on correctly defining the safety properties of the BART system and then formally showing that our implementation satisfied these safety properties. We focused on the following safety properties: (S1) an object train should never exceed the speed limit of the track segment on which it is currently traveling, (S2) an object train should never get so close to its lead train that an unexpected stop or derailment of the lead train would result in a collision between the object train and the lead train, and (S3) the object train should stop at signals and stations when told to do so. Our verification of property (S3) is based on the assumption that the environment will tell a train to stop at a signal or station only when it is possible for the train to satisfy this request.

Modeling: We modeled the state of the system as a vector of monitored and controlled variables quantified over discrete finite domains. In this framework, train behavior at discrete points can be modeled as a sequence of state vectors. Our belief is that this model directly captures the system and is harmonious with the system view presented in the case study document.

At this point, we encountered a fundamental problem when trying to formally define safety properties with respect to our model. For example, what relationships should a sequence of discrete points defining object train behavior and track speed limits have in order to satisfy the property: *the object train should never exceed the speed limit of the track segment on which it is traveling*? The problem here is that safety properties are most directly expressed in a continuous framework and our model is discrete. Some experimentation led us to the conclusion that the most elegant way to address this problem was to lift the discrete model into a continuous framework. This lifting was done in a conservative manner with respect to the safety properties we wanted to verify. Thus conservative lifting would assure that safety properties verified for the model would also extend to our discrete model which reflects our understanding of the physical system.

An important thing to note about safety properties is that they must ultimately be defined, either directly or indirectly, in terms of train behaviors (i.e., state sequences) rather than in terms of train states. We defined safety properties in a recursive manner and explored defining safety from both the negative ("shall not" properties) as well as positive ("shall" properties) manner. For example, in a negative approach one defines a safety property by defining the set of *unsafe states* (i.e., the states that statically represent a safety violation or those states in which one cannot guarantee that the train controller can avoid a transition sequence to a statically unsafe state). In this context, the goal of a train control algorithm is to avoid transitioning from a safe state to an unsafe state. The positive approach to defining safety property is in terms of *safe states*. In this approach, a train's state is safe if the controller can avoid (from this state) a behavior sequence that would force it into an unsafe state. From this perspective, the goal of the train control algorithm is to only execute transitions to other safe states. There difference between the two approaches, while theoretically small, can have a significant impact on the elegance in which special cases can be specified, etc.

Development, Verification Process, and Results: In the development and verification phase, our goals were to (1) specify an abstract train control algorithm, (2) formally show that this algorithm satisfies the stated safety properties, and (3) refine this specification into an implementation using correctness-preserving transformations.

Our abstract train control algorithm is simple:

max_of(all accelerations leading to safe states)

This is computed as follows (1) obtain the set of all possible accelerations for the current state, and (2) select the subset of accelerations which transition to safe states. The main difficulty here, of course, is step 2 which involves the exploration of train behaviors. Analysis of the initial assumptions of the problem leads to the conclusion that, with respect to safety, there are only a few types train behaviors that must be considered: derail, emergency stop, and shortest possible normal stop. It is relatively easy to show that other behaviors need not be considered when considering safety properties. We believe that this type of observation is key in order to make the size of the behavioral state space manageable.

In our approach, we developed/defined a domain specific operator, $\ll$, for comparing continuous object train behaviors with safety properties. We then formally verified two theorems that enabled us to map such continuous comparisons to discrete comparisons. At this point we had an abstract algorithm that was expressed in terms of various operations on finite discrete sets [4]. The next step was to map these operations to a high-level language. Our target language was ML. This transformation resulted in an executable specification, which was not efficient enough to meet the real-time constraints of the system. In order to improve the efficiency, we applied various optimizing transformations. Some of these transformations were formally verified by RRL while others were verified by hand. The result was an implementation that ran significantly faster than the original executable specification.

We would like to point out that while we did not verify the correctness of all transformations that were applied, we did verifiy the correctness of key transformations. Another thing worth mentioning is that we made various simplifications in our system model. For example, the acceleration function and the equations for train motion have been simplified. Refining these aspects of our model will not effect our specification or the definition of safety properties, but will effect the latter stage of the transformational development phase.

5 Conclusions

From a formal standpoint, there are three major areas that must be addressed in order to analyze and develop a system such as BART. First, there is the problem of mapping from an informal system description (e.g., the text of the case study) to a formal one. Under the assumption that the case study text reflected the thoughts of domain experts, one can assume that the text mentions important abstractions, concepts, and operations. Some examples of this from the BART case study are: (1) stopping profile, (2) worst case stopping profile, (3) train speed, (4) track speed limit, (5) acceleration, and (5) equations describing train motion. An important part of model development is the formalization and validation of these terms. In Section 4.1, this issue is addressed by producing a rigorous SRD. In Section 4.2, the mapping issue is addressed by explicitly modeling a physical train which is distinctly different from the abstract train.

The second major area concerns itself with the incorporation of domain knowledge such as equations describing the behavior of a train over a finite time interval. At some point, the software developer must accept domain knowledge in the form of axioms. These axioms define the ground rules from which model-based system analysis proceeds. For example, in Section 4.3 such axioms may provide the basis for table generation and in Section 4.4 they provide the building blocks for profile calculation. The decision regarding what is a "fair" axiom can have a dramatic impact on the complexity and attendant analyzability of the model. An extreme (i.e., unfair) example in BART would be to have a "safe-acceleration" operation be axiomatic. This example demonstrates that one must be careful regarding the interplay between axiomatic domain knowledge and what insight one hopes to gain from model analysis. A document explicitly describing what domain knowledge is treated as axiomatic and how it impacts model analysis would be helpful to clarify what the benefits of formal analysis would be.

And finally, how complexity is handled is crucial to the success of the approach. An incremental approach can be taken, where a simple model is expanded in manageable increments. A lemma-based approach could be taken in which a growing knowledge base could be used to restrict or focus the anal-

ysis effort. Other approaches are also possible.

In conclusion, an interesting and important area of formal methods research is on how to integrate the above so that the best approach and toolset can be used for the job at hand.

References

[1] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control.* LNCS 1165, Springer-Verlag, October 1996. (ISBN 3-540-61929-1)

[2] G. Chiola, C. Dutheillet, G. Franceschini and S. Haddad. *On Well-Formed Coloured Nets and their Symbolic Reachability Graph.* High Level Petri Nets. Theory and Application. Edited by K. Jensen G.Rozenberg, Springer Verlag 1991.

[3] D. Kapur and H. Zhang. *An overview of Rewrite Rule Laboratory (RRL).* J. of Computer and Mathematics with Applications,} 29, 2, 1995, 91-114.

[4] Deepak Kapur and Victor Winter. *On the Construction of a Domain Language for a Class of Reactive Systems.* In High Integrity Software, Eds. Winter and Bhattacharya, Kluwer Academic Press, 2001.

[5] F. Kordon, I. Mounier, E. Paviot-Adet and D. Regep. *Formal verification of embedded distributed systems in a prototyping approach.* Proceedings of the International Workshop on Engineering Automation for Software Intensive System Integration, June 2001.

[6] M. Lemoine et al. *Validating Requirements: The Evolutionnary Approach.* Computer Software and Application Conference (COMPSAC98), Wien, Austria, 1998.

[7] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell.* Lecture Notes in Computer Science Vol. 891, Springer-Verlag, 1995.

[8] MARS-Team. *A **LfP** :The Mars Project Home Page.* http://www.lip6.fr/mars.

[9] S. Owre, J. Rushby, and N. Shankar. *PVS: A prototype verification system.* 11^{th} Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, Vol. 607, pp 748-752, Springer-Verlag, 1992.

[10] D. Regep and F. Kordon. ***LfP** : A specification language for rapid prototyping of concurrent systems.* Proceedings of the 12th IEEE International Workshop on Rapid System Prototyping, June 2001.

[11] W. P. de Roever et. al. *Concurrency Verification: An Introduction to State-based methods.* Cambridge University Press, Sept. 2001.

[12] J. Rumbaugh, I. Jacobson and G. Booch. *The Unified Modeling Language - Reference Manual.* Addision-Wesley, 1999.

[13] John Rushby. *Formal Methods and their Role in the Certification of Critical Systems.* Technical Report CSL-95-1, March 1995.

[14] M. Spivey. *The Z notation - A reference manual.* Prentice Hall International, 1989.

[15] Standard. *EAI-632: Processes for Engineering a System.* INCOSE 1998.

[16] K. Varpaaniemi, K. Hiekkanen and T. Pyssysalo. *PROD reference manual.* Helsinki University of Technology, Digital Systems Laboratory, 1995.

[17] V.L.Winter, R. S. Berg, and J. Ringland. *Bay Area Rapid Transit District Advance Automated Train Control System Case Study Description.* In High Integrity Software, Eds. Winter and Bhattacharya, Kluwer Academic Press, 2001.

[18] V. L. Winter. *An Overview of HATS: A Lanaugage Independent High Assurance Transformation System.* Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET), March 24-27, 1999.

[19] V. L. Winter, D. Kapur, R. S. Berg. *Formal Specification and Refinement of a Safe Train Control Function.* Submitted to The Computer Journal, draft available at *www.sandia.gov/AST.*

[20] V. L. Winter, D. Kapur, and R.S. Berg. *A Refinement-based Approach to Deriving Train Controllers.* In High Integrity Software, Eds. Winter and Bhattacharya, Kluwer Academic Press, 2001.