DEPENDABLE SOFTWARE IN RAILWAY SIGNALLING

Timothy L. Johnson[1], Hunt A. Sutherland[1], Bart Ingleston[1], and Bruce H. Krogh[2]

[1]Computing and Decision Sciences
GE Global Research (K-1, 5C30A)
1 Research Circle
Niskayuna, NY 12309
Corresponding Author: johnsontl@research.ge.com

[2]Chair, Dept. of Electrical & Computer Engineering
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Abstract: Railway signalling software and safety requirements are summarized, and three short examples of the application of new methods to the assurance of dependability are provided. The strengths and shortcomings of existing methods relative to application needs are illustrated. The analogy between railway signalling and other distribution processes in manufacturing and supply chain management is noted.

Keywords: Railways, formal verification, safety analysis, signals, software safety.

1. INTRODUCTION AND BACKGROUND – RAILWAY SIGNALLING DEPENDABILITY

Dependable signalling has been an inherent requirement of safe railway operation since the inception of railways. Some of Thomas Edison's first inventions involved reliable telegraphy for railroad communications (US Patent Office, 1886). As railway signalling devices were digitized in the 1980's and 1990's, some of the burden of dependability shifted from hardware to software: not only did the computational hardware have to meet high reliability standards, but also the logic of the software (formerly contained in railroad relay wiring) also had to be correct under all circumstances. In addition, the scope of safety concerns has increased beyond dependability of the core program logic to its correct operation under a variety of unusual circumstances (e.g., relative timing of events, interoperability) that previously were the responsibility of dispatchers to resolve. Today, a number of dependability and safety standards (e.g., CENELEC, 1997) govern railway signalling equipment. Rigorous safety certification and test procedures exist at many levels. Still, serious accidents, such as a recent collision of two passenger trains in New York City's Penn Station, can occur (CNN, 2004).

The purpose of this paper is to examine some emerging approaches to achieving the higher level of certainty that may be required as more and more signalling operations become partially or fully automated. Present test procedures for executable code are extremely time consuming and expensive, and are themselves subject to error; standard methods of testing executable code require test sets that grow rapidly in size with the complexity of automated logic. Formal methods and other new design process improvements may be applied at many steps during the product development process, from requirements definition through source code logic verification, offering an opportunity to apply high performance computing to improve both design dependability and test coverage (Morel, et al, 2004). They offer the assurance of logical consistency across large, complex signalling applications. Anticipating these needs, GE and Carnegie Mellon

University have explored potential uses of emerging methods for improvement of dependability of future railway signalling systems. Some of initial findings and examples from this work are reported in this paper.

Railway networks exhibit many conceptual analogies with automated material handling systems that are widely used in most advanced manufacturing plants, and more broadly, with other wide area transportation networks that implement supply chains, and to product distribution systems and are becoming more tightly integrated with "just in time" manufacturing systems. This abstraction is considered important in linking this topic to the area of "dependable manufacturing systems".

2. SAFETY REQUIREMENTS FOR RAILWAY SIGNALLING SYSTEMS

Railway signalling is a large, complex, international field, encompassing both private (mostly freight) and public transit systems, as well as specialized applications such as mining and inter-modal transit. Some signalling areas where dependability research has been done at GE include:

- A locomotive controller (braking function)
- A CAB signalling device (speed limit determination)
- An interlocking controller (interlock logic)
- A computer-aided dispatch system (speed profile planning)

The work reported here is preliminary in nature and should not be interpreted as an endorsement of emerging methods, or of the particular toolsets used here.

Some of the safety/dependability standards that apply to these applications include:

- IEEE Std. 1483-2000 (Verification of Vital Functions in Processor-Based Systems)
- CENELEC Std. EN50128 (Software for Railway, Control, and Protection Systems)
- IEC Std. 61508 (Functional Safety of Programmable Electronic Safety Related Systems)

The *software certification process*, in particular, presents good opportunities for emerging verification methods. Software certification has come to be the one of the most expensive and time consuming aspects of new product development in railway signalling, and thus is of great concern to suppliers, as it comes to dominate costs and schedules, and entails significant new product risk. Emerging methods offer the future prospect not only of partially automating verification and validation processes, but also of significantly improving safety, test coverage, and the time taken for software testing.

3. SAFETY, VERIFICATION, AND VALIDATION AND NEW PRODUCT DEVELOPMENT

The "baseline development process" shown in Figure 1 is typical of new product development processes used in the railway signalling industry. Defects identified in later stages of the process require that the entire design process be re-iterated (ideally, with small changes from the original design); these iterations are very expensive can increase time and cost by 15-100% or more, over the initial iteration.

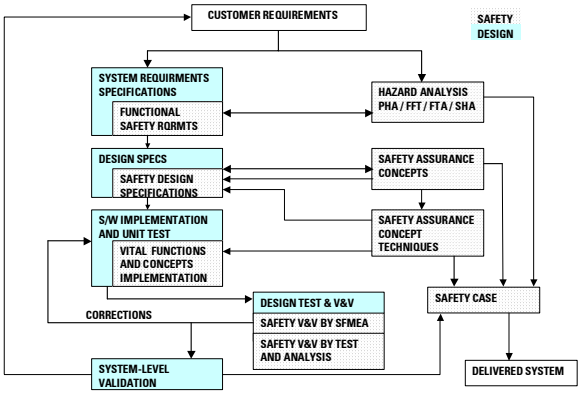

Figure 1: Baseline new product design process (Key to figure abbreviations: PHA = Preliminary Hazard Analysis; FTA= Fault Tree Analysis, FFT=Fast Fourier Transform; SHA = Safety Hazard Analysis; SFMEA = Software Failure Modes & Effects Analysis)

Safety, in particular, is sometimes viewed as a "game against nature" and properly results in detailed safety requirements (based on the most likely forms of unintended use of the system, or accidents) that are expressed at the outset of a design process. In recent upgrades to development processes, these give rise to "safety" test cases that are initiated during the requirements phase and used throughout the test and certification processes. While there is not a consistent definition among practitioners, software verification & validation (V&V) is typically defined in terms of process steps, as in the IEEE standard 1483 (IEEE, 2000). *Verification* is most closely associated with the unit and subsystem test activities concerned with verifying that a design meets stated requirements. *Validation* usually occurs after system integration and certification, and is concerned with whether the formally stated requirements used in the design in fact capture the intended product functions under all conditions, and whether they are complete.

The dependability improvements in this process that are the subject of the case studies used here, are concerned with the early validation of requirements, the verification of correctness of a preliminary design, and with improving automatic testing of logical correctness, consistency, and safety of source code. These all have the benefit of detecting design defects earlier in the product development process and in reducing expensive design iterations.

The following sections provide brief synopses of how emerging methods can be applied to the reduction of test effort and improvement of dependability of railway control product software

systems. *Requirements capture* using an ontology is illustrated for an example drawn from rail yard automation in Section 4. The use of a commercial product, CodeCheck™,[1] is illustrated for an example of *style guide design verification* of a computer-aided dispatch system, in Section 5. Several toolsets developed at Carnegie Mellon University were applied to problems of *model checking, dependency analysis, and automated test generation* for a braking/traction control system of a locomotive, as described in Section 6.

4. SOFTWARE SPECIFICATION CAPTURE & ANALYSIS

Software specification consists of both a description of the desired functionality and often a set of use-cases that describe the operational mission under which the software must perform. If the specification is captured as an executable specification, then it may be analyzed in relation to the chosen application domain through use of an ontology. This kind of analysis can reveal two kinds of errors: (1) design verification errors that can be related to the mathematical language that describes the formal design model, or (2) specification validation errors that can be related to improper description of the application domain itself.

The first type of error is often detectable during initial unit and integration testing through use of unit level testing techniques or through application of design verification methods, as described in other sections in this paper. The second type of specification error is more difficult to detect since it requires subjective and domain specific knowledge about the correct formulation of the system or operational mission as described in the specification. This kind of error may lead to safety risks because it may be propagated from early phases of development and, if undetected, may not be discovered until after the entire functionality of system is operational. Some refer to this error as a conceptual error. Two approaches that may be applied to detect this error that are based upon ontologies, as described in (Kalfoglou and Robertson, 1999): (1) applying ontological constraints that validate specifications against potential conceptual errors through detection of inconsistent or incomplete specification, and (2) augmenting the executable specifications with additional ontological constructs.

During the construction of an ontology, domain-specific constraints may be built-in that subsequently are used to test automatically whether parts of an executable specification are inconsistent. The structure of the specification may then be further augmented in those sections where a completeness or inconsistency check is required. Additional domain specific error conditions can be defined using special editors, such as in the Protégé system (Musen, et al, 2000), which facilitates customized error checking in conjunction with a reasoner. Methods are needed that automatically associate ontologies with the knowledge domains implicit in the specifications. Culley and McMahon (2002), for example, are investigating use of ontologies in support of requirements capture. Detection of a conceptual specification error can be illustrated using a simple example from rail-yard automation. One automation function is a Yard Flow Analyzer that classifies railcars relative to a route plan. For illustration, consider the software logic specified in the hump computer that routes cars as they arrive on the track. The hump computer, normally, forms bad railcar blocks from cars that must be repaired before leaving the yard. A rule that determines whether a railcar is bad might be:

$$badRailcar(A) \leftarrow (consistType(C) \vee mission(C, M)) \wedge highSpeed(M).$$

This rule classifies bad railcars depending upon the mission and the type of consist that the railcar is to join. The demands of the mission along with the speed that railcars must travel are considered. The rule is correct as specified but may be incomplete. Completeness and consistency depend upon correct specification of the mission as well. For example, a mission that is a local delivery could improperly be assumed to be low speed travel when in fact depending upon the route taken high-speed travel still may be required, as for example may occur over main line track. Ontological axioms can be used to verify the correct use of ontological constructs throughout the specification. Whenever a statement in the specification will not satisfy the ontological axioms an error is reported. In this case, the analysis might raise an inconstancy error since high-speed operation may be permitted when the mission is local.

An adjunct to the use of ontology for detecting conceptual error can be the use of temporal logic. Ontologies are poorly suited to represent behavioral and timing characteristics of software. For that reason, a second approach can exploit the use-case scenarios for overlay of temporal logic, possibly with specific timing characteristics, within the specification. Where an executable specification has been captured then a-priori constraints on the time-ordering may again augment the original specification, as has been described for example by Grabisch (2003) in use-case scenario analysis.

Primary impediments to the use of ontologies and temporal logic are presently the lack of standard ontologies for railway signalling, and the significant manual effort to needed to develop such ontologies due to the large size of the set of relationships that may exist, even for very restricted domains. Still, investment in developing standard railway signalling ontologies is becoming feasible through the use of a common ontology language, such as the Web Ontology Language (OWL), that promotes sharing of knowledge domains and thus facilitates building application-specific ontologies.

[1] CodeCheck™ is a trademark of Abraxis Software, Inc.

5. EXAMPLE OF CODECHECK™ APPLICATION

Application of formal methods normally presumes that code is syntactically correct, or executable, but this is often not the case in practice. This tool was used to inspect a repository of source code for adherence to style and standards guidelines as a means of standardizing inputs for formal analysis. The tool was required to capture a set of style rules and to then identify, capture, and report instances where these style and standards rules were violated in the source code. CodeCheck™ was one such tool that was evaluated.

CodeCheck™ runs with a user-written control program defining the verification rules. It is executed via command-line and takes a source file as a command line argument to run the rule file against, but all files that are used by (included by) the one file are also inspected during the execution. and will have an associated ***listing file (.lst)*** created during the run. CodeCheck™ is executed by a format such as,

check <filename> *-options*

where <filename> is the name of the source file that is to be inspected, and *–options* are any number of command-line options that specify, but are not limited to, the rules file, output directory, the dialect of C/C++ (ANCI C/C++, Microsoft C++, etc…), and/or designate the system to append all CodeCheck™ output to the ***stderr.out*** file (see below).

The environment in which CodeCheck™ is to be executed needs to be same as the environment used to compile the source code that is being inspected. If the environment is not the same, errors may occur that will result in a non-complete run of the CodeCheck™ tool. After running CodeCheck™ on a file, it will produce a number of files with information concerning the inspection of the source code. These files include:

- **stderr.out** - output from CodeCheck™ detailing the file, line number, rule number, and description of each found rule violation.
- **<filename>.lst** - is a copy of the source code file that was inspected during execution, with inserted comments from CodeCheck™ denoting where the rule violation occurred.

The following coding style guideline and its respective rule written in the CodeCheck™ language is shown. The rule uses the aforementioned global variables and helper functions.

Style Rule: File names shall not be longer than 32 characters including the file's extension.

CodeCheck™ syntax:

```
if (mod_begin)
{
    if (strlen(file_name()) > 32)
    {
        warn( 9013, "%s File names shall not
be longer than 32 characters including the
file\'s extension",
                file_name() );
    }
}
```

Though the ***CodeCheck***™ tool provides rule-sets that partially implement certain railway coding standards, its use on a large system proved problematic. The need to execute ***CodeCheck***™ tests in the target environment prevents its effective use on small pieces of a large system when (during development) it may be undergoing frequent revisions. The user needs to manually verify the coding rules in a low level language (C/C++), which is itself an error-prone process. Finally, the error messages (.lst file) are interspersed with the code, and the lack of prioritization of error messages makes debugging very time consuming. Nevertheless, the results provide a rigorous and consistent style verification of source code, comparable in quality to the verification tests of a first-pass compiler. This illustrates one dilemma of all verification tools: Is the extra time and effort required to initially apply the tool larger than the subsequent savings or quality improvement in the code?

6. MODEL CHECKING, FORMAL VERIFICATION, AND OPTIMAL TEST SET GENERATION

6.1 Model Checking

Traditionally, validation and verification of embedded control logic is accomplished by simulation. Simulation can be very time consuming because a given simulation run represents only one particular trajectory and many simulations must be performed to explore the control logic. Moreover, it is usually not possible to know *a priori* the length of time a simulation should run, or the number of simulations that should be run to explore all of the controller behaviors. Consequently, simulation studies are typically based on the designer's knowledge of the system and possible errors that could arise. To cover the cases that a designer did not consider, additional simulation trajectories might be generated randomly. Even with these two strategies combined, it is impossible to guarantee complete coverage of all critical situations.

Model checking is a technique for verifying properties of finite-state systems (Clarke, 1999). In contrast to simulation, model checking considers *every* trajectory of *arbitrary* length. If a specified property is false, the model checker returns a counterexample, that is, a specific sequence of state transitions that violates the property. One popular model-checking tool is SMV, which was originally developed at Carnegie Mellon University (http://www.cs.cmu.edu/afs/cs.cmu.edu/project/modck/pub/www/). SMV uses a textual language to describe a system composed of concurrent, interacting finite state machines and uses fixed point algorithms to determine whether the system satisfies properties given by the user. In SMV, properties, or specifications, are given in a computation tree logic (CTL) syntax. The state variables that are input to

the model checker must be integer-only, and the state machines can only have expressions that contain the state variables, '&', '|', '*', '+','=','>=','<=','>','<', '/', and '-'. A prototype tool called SF2SMV was developed for applying the model checking capability of SMV to the state chart structures in Stateflow™,[2] a MATLAB™ toolbox for implementing discrete-state transition systems within Simulink™ models of continuous-state systems. Stateflow™ diagrams (SFDs) are more complicated than standard statecharts (Harel, 1998). SFDs can have floating point numbers, arbitrary expressions, hierarchy, events, and parallel states. In addition to these extra elements, the execution rules for SFDs are complicated. Testing a SFD by exhaustive simulation is a difficult, if not impossible, task.

Figure 2 shows a basic SFD and Figure 3 shows the associated SMV model generated by SF2SMV. These figures illustrate how the hierarchy of the SFD is reflected in the SMV modules. The SF2SVM user interface supports the construction of CTL specifications to be checked for the SMV model, and makes it possible to play back counter examples on the original SFD when the specification is not satisfied. Our experience with SF2SMV in a locomotive example exposed the difficulty of creating a tool that will be able to perform model checking on large-scale designs. One of the principal sources of difficulty is the integration of model checking capabilities with the existing commercial design tools such as Stateflow™. It is necessary to capture the semantics of the commercial tool, often through reverse engineering. If this is not done correctly, the value of the model checking results will be compromised.

Nevertheless, even for small portions of the control logic, model checking can expose problems the designer has missed because it is not easy for the designer to anticipate all possible combinations of events and signal values. As with other control applications, designers of software for railway signalling typically think of normal operating conditions when constructing the control logic. Exceptions and safety conditions are often added to this code, and it only takes a few new cases to make the possible number of execution sequences explode beyond the designer's capacity for thinking through everything that might happen. Consequently, model checking is an attractive tool for evaluating control modules before they are composed into the complete control package.

[2] StateFlow™, Simulink™, and MATLAB™ are Trademarks of The Mathworks, Inc.

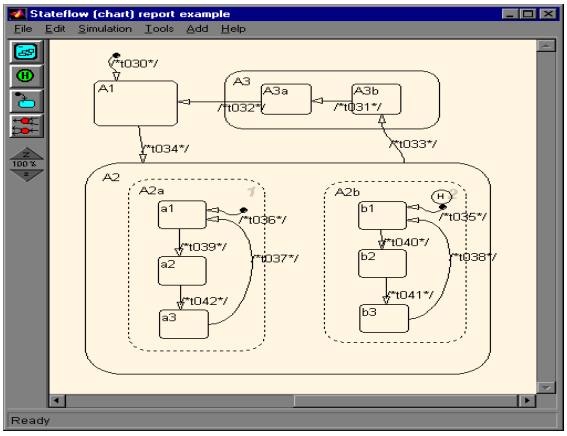

Figure 2. A basic hierarchical Stateflow™ diagram.

6.2 Test case Generation

An essential aspect of a high-quality design process is the development of test patterns, or sets of test inputs, that can be applied to the final product to identify faults and confirm correct system behavior. In the design of embedded control systems, in particular, the growing use of tools for computer-aided design and simulation increases the prospects for performing extensive testing of the control logic before it is realized in software and implemented on the target processor. The rapid growth in complexity of embedded control systems and the demand for short design cycles has increased the interest in effective methods for automatic test generation.

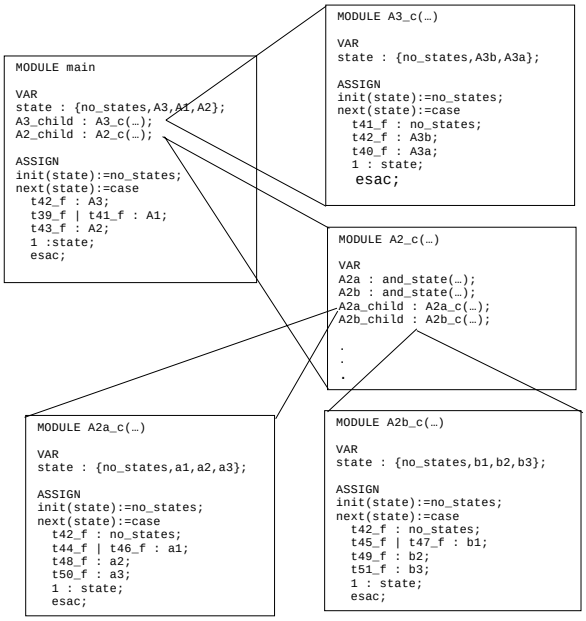

Figure 3. Basic SMV modules for the Stateflow™ diagram in Figure 2.

Automatic test generation for embedded control systems is challenging for two major reasons. First, the problem is complex due to the hybrid nature of the controller and plant composition, which contains both logic (discrete states) and components with continuous dynamics (continuous states). Un-decidability results for hybrid systems indicate that one should not expect to discover completely

algorithmic procedures for test-vector generation for hybrid systems (Henzinger, 1998). Hence, global search methods that use domain-specific heuristics are used to produce test inputs for mixed-signal circuits (Vinnakota, 2001; Tofte 2000, Gupta, 2001). Similar methods are needed to generate test vectors for embedded control systems. The second challenge is practicality: To develop automatic methods, the design must first be captured in a computer model that includes a representation of the plant as well as an embedded control algorithm. In contrast to integrated circuits for which very structured models using standard languages are developed during the design process, it is extremely difficult to extract formal models (such as automata) from existing simulation models of embedded control systems.

To address these problems, a test case generation tool was developed at CMU that provides (i) the formulation of a coverage problem for hybrid systems with discrete and continuous inputs and outputs; (ii) a pragmatic solution to the problem that uses a GA-based algorithm to produce a test input with the specified coverage; and (iii) implementation and demonstration of the method for industrial-sized MATLAB™ Simulink™/Stateflow™ simulation models. This work is described Zhao, et al (2003). Figure 4 illustrates the basic structure of the approach. Experimental results are shown in Tables 1 and 2. For the small applications shown in Table 1, the genetic algorithm always found a test input that gave responses with the desired properties. The results in Table 2 are for a locomotive model consisting of 1478 Simulink™ blocks and 50 Stateflow™ states and 15 input signals. In this case, the genetic algorithm did not always find an appropriate test vector, but the maximum number of model evaluations for each experiment was set to 3000. With a fixed limit on the number of simulations, Table 2 shows that the population size should not be too small or too large. If the population size is too small, the power of the GA to explore several possible locations in the search space in parallel is not fully exploited. If the population size is too large, the relative number of crossover and mutation operations that can be performed is limited because of the large number of chromosomes. At the extreme case in which the population size equals the model execution limit, GA will degenerate to pure random search. Random search *never* found a suitable test vector for this model.

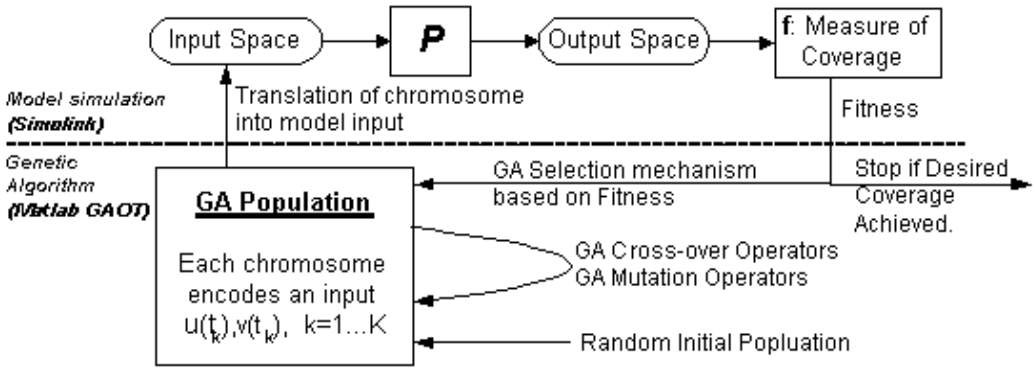

Figure 4. Genetic Algorithm approach to test case generation.

Table 1: Automatic Test Generation results – Embedded real-time systems

Table 1. Results for some small examples run on a 1.7-GHz Pentium 4 computer.					
Example	**Number of Blocks (SF States)**	**Number of Inputs**	**Sample Time (Sim Time)**	**GA Populations (Generations)**	**Model Executions (Compute Time)**
Test drive (illustrative)	78 (7)	1	5.0 (30.0)	20 (5)	63 (45 s)
GE RTC embedded system	24 (4)	4	1.0 (10.0)	20 (2)	37 (30 s)
Ford drive train	232 (11)	6	1.0 (50.0)	40 (10)	372 (552 s)

Table 2: Locomotive Model Test Generation runs, showing GA Population sizes.

Table 2. Results for the GE Traction Pilot model executed on an 800-MHz Pentium 3 computer.		
GA Population	**Number of Successful Runs**	**Average Number of Model Execution (Compute Time)**
4	18/20	1044 (1,000 s)
16	18/20	518 (500 s)
64	10/20	299 (300 s)

Experience with the GA approach to test case generation is very positive. Although the tool is currently only a research prototype, it has demonstrated that test cases can be generated with reasonable computation *using the same models that were developed for simulation and system design.* Such tools can have enormous impact on the overall time can cost of controller design.

7. PRESENT LIMITATIONS AND FUTURE POSSIBILITIES

The development of formal methods for the statement and application of system requirements is still a key bottleneck (Grimson and Kugler, 2000; Jaffe, et al, 1991; Svedung, 2002). The B-formal method (Abrial and Mussat, 1998) provides a potential future means of carrying formal verification through multiple stages of the control system design cycle. A rail-control *ontology* is needed so that the inferred relationships among entities in a requirements document can be explored automatically (e.g., by traversing relationship graphs and inferring implied requirements from them). However, such ontologies may be large, and the problem of determining the transitive closure of statements generated as a subset (e.g., "safety statements") of a formal grammar is yet to be investigated. Use of ontologies for requirements engineering support has been studied. e.g. in the following project: http://www.bath.ac.uk/~ensmjd/ORCS_blurb.html, which confirmed the large size of ontologies needed in problems of practical interest. The successor project:http://www.bath.ac.uk/imrc/mechengineering/Projects/SMIR.htm is also of interest.

Style checking (Section 5) can be used to assure consistency of statements in formal languages such as ANSI C/C++ source code, and might be generalized to cover statements made in natural language requirements documents: this would remove spurious inconsistencies and typographic errors.

In Section 4 it has been illustrated how an ontology-based analysis of the software specification can be applied to detect conceptual errors that normally might not be found until after the system is operational. Also proposed is the use of temporal logic for analysis of the use-case scenarios. Often, it is ironic that the controller or controlled object may not be mentioned at all in requirements documents: it is *assumed* to exist! This gap may be filled by the use of standardized simulation languages, practices, or graphic representation paradigms (Vain and Kyttner, 2001). For instance, many libraries of process control primitives, and application-specific modelling packages now exist for steam pipe systems, power plants, motors, batch process operations, circuit design, and signal processing, and these could be generalized to rail applications. In this way, a requirement can be associated with a set of preconditions on a high-level plant model, a set of operations (or controlled modes), and a set of outcomes. By using inference (or perhaps fuzzy inference; Holmes and Ray, 2001) to traverse such graphs or ontologies, a much more complete set of inferred requirements (and also test cases) can be automatically generated.

Formal methods are of interest in the verification of design and simulation models, as well as controllers. Initial feasibility of model checking has been demonstrated for locomotive braking models, for instance. This is a precondition for closed loop testing, and for test generation via open loop models. When a feedback loop is closed via a controller, hybrid modes of behavior can occur, introducing issues of decidability (Section 6). Often, it is difficult to define precise quantitative behaviors that are expected for specific quantified inputs to a system: not only are the test cases difficult to generate, but the expected performance in any specific test case may be difficult to derive. In fact, tracing-through various requirements that apply in a given test case, may allow one to define – by intersecting a number of qualitative performance conditions, each derived from a different path through the requirements network – a much more precise statement of expected behavior.

In conclusion, the use of emerging source-code verification methods to improve dependability of railway signalling logic is still at an early stage of evolution. Many gaps exist in the vision of linking verification at earlier design process stages (requirements definition) to verification at later stages of design (source code). The use of formal verification methods for individual design process steps is expected to proceed when commercial grade tools become available, and when these can be constructed to save time and demonstrably improve software quality.

REFERENCES

Abramovici M., M. Breuer, and A. Friedman (1995). Digital Systems Testing and Testable Design, *IEEE Press*.

Abraxas Software, Inc. (2004a). CodeCheck™ Home page, URL: http://www.abxsoft.com/codchk.htm.

Abraxas Software, Inc. (2004b). CodeCheck™ User's Guide, "C and C++ Source Code Analysis using CodeCheck™".

Abrial J.R., Mussat L., Introducing Dynamic Constraints in B, *Lecture Notes in Computer Science* 1393, Springer, ISBN 3-540-64405-9.

Banphawatthanarak, C., B. H. Krogh and K. Butts (1999). Symbolic verification of executable control specifications. *Proceedings of the 1999 IEEE Conference on Computer Aided Control System Design*, Hawaii, Aug. 22-26.

Bensalem, S., *et al.* (1999). A methodology for proving control systems with Lustre and PVS, in C.B.weinstock and J. Rushby (eds.) *Dependable Computing for Critical Applications,* **7**, pp.89-107.

Bullock, D. and C. Hendrickson (1994) Roadway Traffic Control Software, *IEEE Trans. On Control Systems Technology*, Vo. **2**, No. 3, pp 255-264.

CENELEC (1997). Railway Applications: Software for railway, control, and protections systems, *Standard EN 50128* (June)

Clarke, E. M., O. Grumberg, and D. A. Peled (1999). Model Checking. *The MIT Press*, Cambridge, Massachusetts.

CNN.com (2004). Trains Collide in New York's Penn Station (April 19, 2004). URL: www.cnn.com/2004/US/Northeast/04/19/trains.collie

Culley, S. J. and C.A. McMahon (2002). Investigation into the Use of Ontologies for Requirements Capture Support, URL: http://www.bath.ac.uk/~ensmjd/ORCS_blurb.html.

Grabisch, M. (2003). Temporal scenario modeling and recognition based on possibilistic logic. *IEEE Artificial Intelligence*. **148** , Issue 1-2, pp 261 – 289.

Gorski, J. (1986). Design for Safety Using Temporal Logic. In *IFAC SAFECOMP '86*, pp 149-155, Sarlat, France.

Gupta A., S. Malik, and P. Ashar (2001). Toward formalizing a validation methodology using simulation coverage. *IEEE/ACM International Conference on Computer Aided Design*, pp. 286-292.

Harel, D. (1987). Statecharts: A visual formalism for complex systems, *Science of Computer Programming*, **8**, pp.231-274.

Henzinger, T., P.W. Kopke, A. Puri, and P. Varaiya (1998). What's decidable about hybrid automata. *Journal of Computer and System Sciences*, **57**, no.1, pp. 94-124.

IEEE (2000). Verification of Vital Functions in Processor-Based Systems Used in Rail Transit Systems. *STD 1483-2000.*

International Electro-technical Commission. (1998-2000). *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety related systems*.

Kalfoglou, Y. and D. Robertson (1999). A case study in applying ontologies to augment and reason about the correctness of specifications. *11th International Conference on Software Engineering and Knowledge Engineering (SEKE99)*, Kaiserlauten, Germany, URL:http://www.ecs.soton.ac.uk/~yk1/research.html

Krogh, B. (2003) SliceMDL URL: http://www.ece.cmu.edu/cecs/main/projects.html

The Mathworks, Inc. (1999). Stateflow User's Guide, URL:http://www.mathworks.com/access/helpdesk/help/pdf_doc/stateflow/sf_ug.pdf

Morel G., Mery D., Leger J-B., Lecomte T. (2004). Proof-Oriented Fault-Tolerant Systems Engineering: rationales, experiments and open issues. *7th IFAC Symposium on Cost Oriented Automation*, Gatineau, (Québec), Canada, June 6-9.

Musen, M., R. Fergerson, W. Grosso, N. Noy, M. Crubezy, and J. Gennari (2000). Component-Based Support for Building Knowledge-Acquisition Systems. *Conference on Intelligent Information Processing (IIP 2000) of the International Federation for Information Processing World Computer Congress (WCC 2000)*, Beijing, China.

Nadjm-thrani, S. and J.-E. Stromberg (1999). Formal verification of dynamic properties in an aerospace-application. *Formal Methods In System Design*, **14**, pp.135-169.

Place, P.R.H., and K. C. Kang (1993). Safety-Critical Software: Status Report and Annotated Bibliography. *CMU/SEI-92-TR-5*.

Rausch, M., and B.H. Krogh (1998). Symbolic verification of Stateflow™ logic. *International Workshop on Discrete Event Systems (WODES98)*, Cagliari, Sardinia, August 26-28.

Silva, B. I., O. Stursberg, B. H. Krogh and S. Engell (2001). An assessment of the current status of algorithmic approaches to the verification of hybrid systems. Proc. *40th IEEE Conference on Decision and Control.*

SMV(2004). URL: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/modck/pub/www/

Spencer, C. (2001) SF2SMV GUI Readme. See also URL: http://www.ece.cmu.edu/~webk/sf2smv/

Suyama, K. (2003). Safety integrity analysis framework for a controller according to IEC 61508. Proc. *42nd IEEE Conference on Decision and Control.*

Svedung, I. (2002). Graphic representation of accident scenarios: Mapping system structure and the causation of accidents. *Safety Science*, **40**, Elsevier Science Ltd., pages 397–417.

Tofte, J., C.K. Ong, J.L. Huang, and K.T. Cheng (2000). Characterization of a pseudo-random testing technique for analog and mixed-signal built-in-self-test. *Proceedings. 18th IEEE VLSI Test Symposium*, pp. 237-246.

Vain, J. and R. Kyttner (2001). Model Checking – A New Challenge for Design of Complex Computer-Controlled Systems. *Proc. 5th Int'l Conf. On Engineering Design and Automation*, Las Vegas, pp. 593-598.

Vinnakota B. (2001), Analog and Mixed-Signal Test, *Prentice Hall*, 2001.

World Wide Web Consortium (2004). OWL Web Ontology Language Reference. URL:http://www.w3.org/TR/owl-ref/.

Zhao, Q, and B.H. Krogh (2001) Formal verification of statecharts using finite-state model checkers. *Proceedings of the 2001 American Control Conference*, Vol.**1**, pp. 313- 318.

Zhao, Q., B. H. Krogh and P. Hubbard (2003). Automatic generation of test inputs for embedded control systems. *IEEE Control Systems Magazine*, **23**, no. 4, pp. 49-57.

U.S. Patent Office (1886). A System of Railway Signalling. *Patent 350234*, issued to T. A. Edison and E. T. Gilliland, 10/5/1886.