Cross-layer Analysis, Testing and Verification of Automotive Control Software

Manfred Broy
Samarjit Chakraborty
Dip Goswami
TU Munich, Germany

S. Ramesh, M. Satpathy
General Motors R&D,
India Science Labs

Stefan Resmerita
Wolfgang Pree
University of Salzburg, Austria

ABSTRACT

Automotive architectures today consist of up to 100 electronic control units (ECUs) that communicate via one or more FlexRay and CAN buses. Multiple control applications – like cruise control, brake control, etc. – are specified as Simulink/Stateflow models, from which code is generated and mapped onto the different ECUs. In addition, scheduling policies and parameters, both for the ECUs and the buses, need to be specified. Code generation/optimization from the Simulink/Stateflow models, task partitioning and mapping decisions, as well as the parameters chosen for the schedulers – all of these impact the execution times and timing behaviour of the control tasks and control messages. These in turn affect control performance, such as stability and steady-/transient-state behaviour. This paper discusses different aspects of this multi-layered design flow and the associated research challenges. The emphasis is on *model-based* code generation, analysis, testing and verification of control software for automotive architectures, as well as on architecture or platform configuration to ensure that the required control performance requirements are satisfied.

Categories and Subject Descriptors

C.3 [**Special-Purpose And Application-Based Systems**]: Real-time and embedded systems

General Terms

Algorithms, Design, Performance

Keywords

Automotive Control Systems, Model-based code generation, Model-based testing and verification

1. INTRODUCTION

Modern automotive architectures support a large number of control functions, some of the more common ones being (a) the engine control unit, which includes electronic throttle Control and transmission control, (b) the body control subsystem, which includes climate control, locking, and mirror control, (c) chassis control, which involves stability control, and (d) safety functions like adaptive cruise control, lane keeping, lane centering and lane departure warning. Most of these functions are closed loop control algorithms involving one or more feedback control loops around the plant being controlled, along with appropriate sensor and actuator setups. As can be seen from the above list of examples, the control features are diverse in nature, and they also differ in terms of functionality and criticality. The body control functions are discrete and reactive, the safety and powertrain control functions are continuous, hard real-time applications, while the telematics applications are discrete and soft real-time in nature.

A feedback control system aims to achieve the desired behavior of a dynamical system by applying appropriate inputs to the system. Currently, automotive control functions are implemented in software organized in a *federated architecture*. Such architectures follow the principle of *one function per ECU* (Electronic Control Unit), leading to a simple developmental model, wherein multiple suppliers deliver independent ECUs with distinct functions and the OEMs assemble and interconnect the ECUs using one or multiple buses. It is simpler to develop a system using this architecture but it results in too many ECUs in an architecture, leading to higher cost, vehicle weight and poor efficiency. An emerging alternative architecture paradigm, called the *integrated architecture* involves multiple functions being integrated into a single ECU and a single function may be distributed over multiple ECUs. The emerging Autosar standard [2] supports such an architecture that helps both the OEMs and the suppliers. An example of such a setup is shown in Fig. 1(a), where the ECUs have dedicated connections to sensors or actuators. In such setups, the response time of a task on an ECU depends on the operating system (OS) running on the ECU, where common automotive OSs are can either be preemptive (e.g., OSEK, OSEKTime) or non-preemptive (e.g., eCos). Similarly, the transmission time of a message over a bus depends on the arbitration policy implemented on the bus. Bus arbitration policies also can either be time-triggered (e.g., the static segment of FlexRay) or event-triggered (e.g., the dynamic segment of FlexRay, or CAN). Fig. 1(b) shows an example schedule and its impact on the timing and performance of control loops running on the architecture, i.e., sensor-to-actuator delay values.

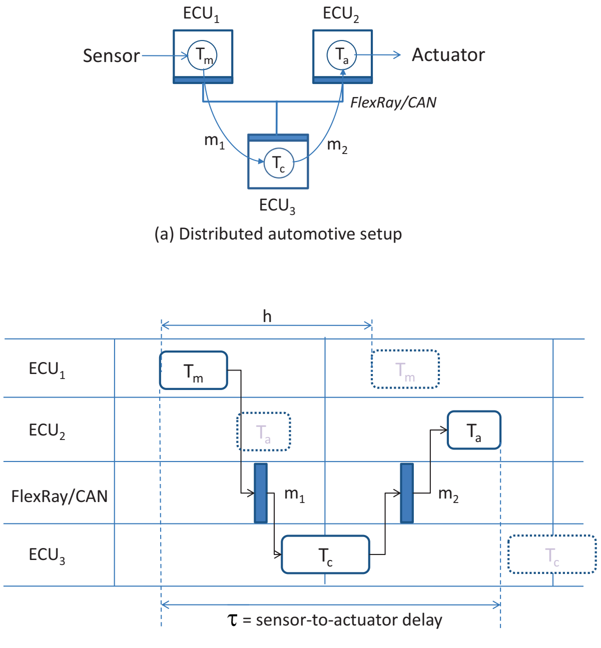
Such distributed architectures result in a significant reduction in cost and a better utilization of hardware resources. The development process starts from high level models usually specified using the Simulink/Stateflow. Next, the model

EMSOFT'11, October 9–14, 2011, Taipei, Taiwan.
Copyright 2011 ACM 978-1-4503-0714-7/11/10 ...$10.00.

is incrementally refined down to software models and then to implementations on execution platforms. A refined model may introduce new behavior that is not accounted for in higher level models specified in the Simulink/Stateflow framework, due to abstraction of execution and communication times. Section 2 discusses various features of such model-based software development, current practices and research challenges. For safety-critical control applications such as those found in the automotive domain, preservation of functional and *timing* properties is crucial when generating software from models. Hence, the timing properties of the software model must be guaranteed to be the same as in the higher level models through the methodology used to generate the code and that used to deploy the code on the platform (consisting of the architecture, OS, etc). In Section 3, we present a survey on currently available tool suites, challenges and opportunities in the area of validation of automotive control software. What comes next is the testing and verification of the generated software models. Testing and validation of automotive control systems are challenging because (a) the control algorithms invariably involve many non-linear computations, (b) in an integrated architecture, components in different ECUs communicate, which may incur non-deterministic and variable delays in message transmission. Section 4 presents current state-of-the-art in testing and verification of automotive control software. Finally, when the control software is implemented on a platform, there is a need to verify whether the control functions meet high level constraints and requirements such as performance and stability. Moreover, certain inherent properties of control functions may be utilized to direct the platform design process and choose platform parameters. Hence, the control algorithms/laws and platform parameters may be co-synthesized to meet certain high-level functional requirements, which satisfying platform constraints like bus bandwidth or those stemming from the bus protocol. Section 5 describes common performance indexes for control functions, how the platform parameters impact control performance and current state-of-the-art on how they can be jointly analyzed and co-designed.

2. MODEL-BASED SOFTWARE DEVELOPMENT

Software based functions not only determine the attractiveness, innovation, enhanced features, differentiation, and speed of realization, but also the complexity and development costs of today's vehicles. With the growing amount of software-based functions the complexity of their development and maintenance in the vehicles also increases. This is accompanied by significant risk of errors in the development process, resulting in ever rising costs for coverage and debugging both in the development phase and in the field of operation. To avoid these risks and the associated spiraling costs for a fast time-to-market, cost-efficient design for an effective development processes (in the sense of optimal results) is needed. This can be achieved through automating the development process with the concepts and principles from the mainstream systems and software engineering domains. In the long term, a comprehensive modular system must be based on an architectural model with high modularity, abstraction and an appropriate system for effective reuse. The long-term goal is to develop flexible, reusable, and modular system components, as well as function-oriented strategies.



(a) Distributed automotive setup

(b) Scheduling

Figure 1: (a) Distributed automotive control application (b) A scheduling example for all the tasks and messages.

The software systems in vehicles have the following characteristics, which make them challenging to design, analyze and debug.

- Multiple, often conflicting and error-prone requirements
- Hard and soft real-time properties
- Stringent and high volume communications requirements
- Multi-functionality with complex dependencies between functions
- Heterogeneity of the application domains

2.1 Approaches

As already mentioned, the growing share of software-based functional features in modern vehicles requires a reorientation of the development process and development paradigms.

- **System orientation** emphasizes the concept of a *system*, with its associated paradigm of *system integration* based on an "architecture" as opposed to a view of a system as an "assembly kit" to assemble largely independent components.
- **Functional orientation** puts emphasis on the functions of a system, in contrast to the subsystems (components, parts) that perform these functions and their interactions. This is in view of the increasingly distributed implementation of functions, as a result of which explicit modeling of functions is essential. The result is a strong emphasis on a systematic requirements engineering and architecture/function development through consistent feature modeling.

- **Systems engineering** refers to a holistic approach to the development of a system. The result is a strong emphasis on requirements management, architecture and the integration phase.

The systematic use of a range of methodological approaches is a must: Consistent model-based development creates the conditions for the required precision and the acquisition of relevant system characteristics. The development of models is often expensive and pays off only if the models are repeatedly and "consistently" used throughout the entire life cycle of a vehicle. What is also important is reuse and product line approaches in other projects as well.

A high level of automation in the development requires the use of formal models and integrated product data models (artifacts and models in the form of back bones that capture all relevant data and models of the systems and are the basis for tool support). The tool support must cover all the usual activities of product development such as model and information collection, analysis (through simulation and verification of properties), transformation, generation, configuration and version control to support the project and its management (tracking project progress).

2.2 Hierarchical Architecture

Model-based development generates artifacts for a comprehensive product data model (the so-called Back Bone) and a durable system for version and configuration management. An essential part of this development process is requirements engineering that is based on a form of functional hierarchy. In such a hierarchy, each function is recorded along with its context that includes its messages, events, and attributes to determine existing dependencies between functions. The logic of each individual function is modeled at an appropriate level of detail. The quality requirements of the overall system and its sub-functions are collected in a standardized quality model, which includes risk and safety analysis and allows a comprehensive architecture review. This evolution is based on predetermined and well-maintained vehicle domain models. These models also capture function hierarchy across projects. Through a comprehensive validation process, the quality of the documented requirements is also ensured. This comprehensive architectural model consists of a number of levels and views, which make the architecture tractable to manage. This requires a systematic approach for a comprehensive description of the architecture of embedded software systems and the functionality provided by them.

- **Conceptual architecture** is related to the functional view and consists of usage level and function hierarchy. It is developed as a result of the requirements engineering and the logical subsystem architecture that describes the interaction logic between local subsystems.

- **Technical architecture** consists of the software, i.e., the software architecture that describes the code architecture, the task architecture of executable and schedulable units and the scheduling and runtime platform. It also consists of the communication architecture including bus scheduling, the platform hardware and the mapping of the software tasks onto the hardware platform.)

Having the above two levels decouples the specification or requirements from the development. The goal is to enable specification and high-level modeling of control algorithms, as well as a verification of the application, irrespective of the final technical architecture. Furthermore, the application-independent components of the technical architecture (i.e., the hardware platform) may also be modeled, verified and refined independent of the final application to be mapped on the platform.

3. VALIDATION AND VERIFICATION OF AUTOMOTIVE CONTROL SOFTWARE

Modern approaches to software design, such as the one described above, and embedded systems design such as Model-Driven Engineering (MDE) [34] and Platform-Based Design (PBD) [43] advocate a top-down approach for application development. Preservation of timing properties from higher level models to the software level model necessitates the methodologies and tools for obtaining correct-by-construction software applications, where timing properties of the software model are guaranteed to be the same as in the higher level model. Examples in this respect are tools based on synchronous languages [20] such as Scade [14] and tools based on the logical execution time (LET) concept [22] such as the Timing Definition Language (TDL) [36]. In this paper, the latter is described in more detail.

3.1 Achieving correct-by-construction timing behavior with TDL

TDL allows the LET-based specification of timing properties of hard real-time applications. The LET of a computational unit, or task, represents a fixed logical duration between the time instant when the task becomes ready for execution and the instant when the execution finishes. A task's LET is specified at the model level, independently of the task's functionality. When deploying the model on a platform, the LET specification is satisfied if the total physical execution time of the task is within the LET interval for every task invocation, and an appropriate runtime system ensures that task inputs are read at the beginning of the LET interval (the release time) and task outputs are made available at the end of the LET interval (the termination time). This is illustrated in Fig. 2. Between release and termination points, the output values are those calculated in the previous execution. Default or specified initial values are used in the first execution of a task.

Tasks can receive information from the environment via sensors and act on the environment via actuators. A task has input ports, output ports, and state ports. State ports keep state information between different executions of the same task. Tasks that are executed concurrently are grouped in modes. In TDL, a mode is a set of periodically executed activities: task invocations, actuator updates, and mode switches. Such a mode activity has a specified execution rate and may be carried out conditionally. The LET of a task is expressed as the mode period divided by the frequency of the task invocation. Note that the time steps of all activities in a mode period can be statically determined.

Mode activities are carried out by a runtime system which performs the following operations at every time step: a) Update output ports of tasks whose LET end at the current time step. At time 0, the ports are initialized rather than

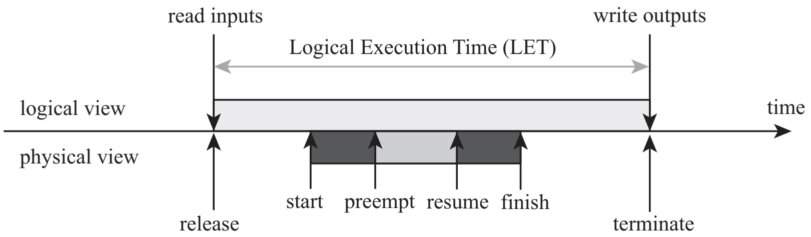


Figure 2: The Logical Execution Time (LET).

updated. b) Update actuators. c) Test for mode switches. If a mode switch is enabled, switch to the target mode. d) Update input ports of the tasks whose LET start at the current time step. e) Trigger the execution of the tasks whose LETs start at the current time step.

TDL provides a top level structuring unit called a module, which groups sensors, actuators, tasks, and modes that belong together. The module concept serves multiple purposes: 1) a module provides a name space and an export/import mechanism and thereby supports decomposition of large systems, 2) modules allow the parallel composition of real-time applications, 3) modules serve as units of loading, that is, a runtime system may support dynamic loading and unloading of modules, and 4) modules are the natural choice as unit of distribution because dataflow within a module (cohesion) will most probably be much larger than dataflow across module boundaries (adhesion).

A commercially available tool suite deals with modeling and deployment of TDL components [11]. TDL components can be written directly in textual form (TDL source code) or designed graphically by using the TDL:VisualCreator tool. A TDL compiler is provided, which targets a real-time virtual machine, called the TDL:E-Machine. To deploy the TDL model on a platform, an implementation of TDL:E-Machine is needed for the platform. The TDL:VisualDistributor can be used to assign TDL modules to a single specified computational node or a distributed system of nodes. Also, the TDL:Scheduler is employed to generate the necessary node and communication schedules. The tools also check for the schedulability of the system, based on provided worst case execution times for the tasks, under the assumption that the periodically time-triggered TDL tasks are the only significant computations competing for the platform resources. The TDL tools have been integrated in Matlab/Simulink. Figure 3 depicts the commercial TDL tool chain. TDL has also been experimentally integrated in the modeling and simulation framework Ptolemy II [6, 39].

While the benefits of approaches such as MDE and PBD are well-understood, their full adoption in the established embedded systems industry is rather slow. One of the main factors responsible for this is the large base of legacy applications, which have been traditionally developed at the programming language level, are usually highly optimized and thoroughly tested. In this substantial part of the embedded system industry, model-driven engineering is employed only partially: typically, for developing new functionality up to the software model, which is then manually merged with the existing legacy code. This poses new challenges to top-down approaches such as TDL. Some examples in this respect are: dealing with high-priority event-driven tasks, usage of shared memory, and the requirement to minimize changes to the legacy code. To achieve robust timing behavior of legacy applications, TDL has been enhanced [39] and the tool-suite has been expanded to deal with legacy tasks. The approach presented in [38] entails a minimal instrumentation of the original code combined with an automatically generated runtime system, which ensures that the executions of designated periodic computations in the legacy software satisfy the logical execution time specifications of the TDL model. Code instrumentation and delaying of task execution to obtain a certain behavior is also used in Wang et al. [55]. In this approach the authors use code instrumentation to generate deadlock free code for multi-core architectures. Timed Petri nets are generated from (legacy) code by instrumenting the code at points where locks to shared resources are accessed in order to model blocking behavior of software. A controller is synthesized from the code and used at run-time to ensure deadlock-free behavior of the software on multi-core platforms by delaying task executions which would lead to deadlocks.

3.2 Verification and Validation of Legacy Automotive Software

Formal methods for V&V of control applications are usually employed at higher level models (e.g., Simulink/Stateflow, timed automata, Petri nets). This motivates an increasing effort for modeling legacy software at higher levels of abstraction.

There are various approaches that generate models from legacy code, but only a few of them include the timing aspect in the modeling. Some software reverse engineering methods find equivalent modeling constructs in a modeling language to reconstruct the same behavior as exhibited by the software. An example is the work reported in [42], where C programs are reverse engineered to Simulink models. In [49], a formal framework is described for building timed models of real-time systems in order to verify functional and timing correctness. Software and environment models are considered to operate in different timing domains which are carefully related at input and output operations. A timed automaton of the software is created by annotating code with execution time information. The common challenge that is facing both model-based V&V and legacy software modeling is scalability. Other technical challenges include floating point and non-linear mathematics, look-up tables, logic with counters and timers [7] as well as indirect referencing in the software (e.g., usage of pointers in C).

State-of-the-art validation of functional and timing properties of embedded software is performed by extensive testing involving hardware in the loop(HiL). Software in the loop (SiL) simulation is mainly used for testing functional properties of applications represented as software or as higher

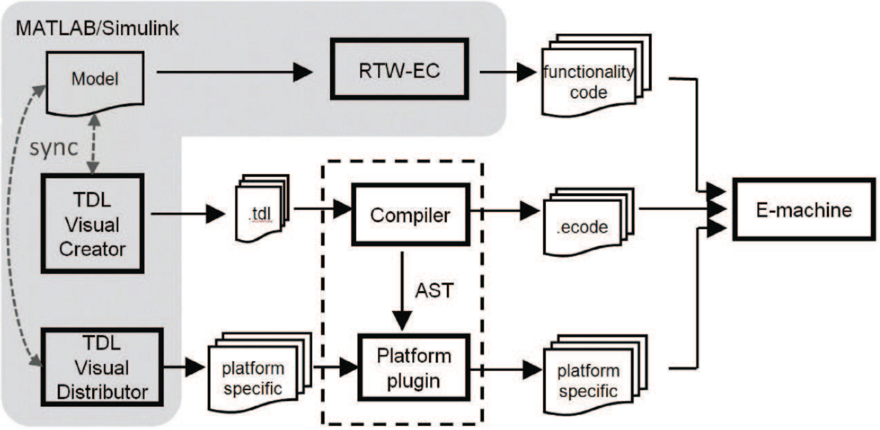


Figure 3: The TDL tool chain with Simulink integration.

level executable models. The costs of HiL testing, plus the increased complexity of distributed embedded applications make the case for shifting the main load of timing-related testing towards SiL setups. Clearly, to simulate the timing behavior of an embedded application, one needs to simulate also the functionality and timing of the execution platform (hardware and operating system), sensors, actuators, and the physical plant under control. An important challenge in this case is finding the right level of abstraction, which determines the modeling effort, the properties that can be tested as well as the simulation speed.

Software in the Loop Validation for Automotive Control Software: In hardware-software co-simulation, the processor can be modeled at the microarchitecture level, which is the most accurate but also the slowest of possible solutions. Faster co-simulation tools avoid modeling the processor in detail but implement a synchronization handshake [33]. Some co-simulation environments also provide a virtual operating system to emulate or simulate the target platform [40]. Some approaches employ instruction set simulators (ISS) in order to obtain correct timing information. However, ISS are slow because of the fine granularity of the simulation. Performance issues are addressed for instance with caching [29] and distributed simulation by applying distributed event-driven simulation techniques. Co-simulation aims at validating functionality of hardware and software components by simulating system parts that can be described at different levels of abstraction. The challenge is in providing the right interface between these levels. Co-simulation as a basis for co-design and system verification can be done in various ways where typically a trade-off between accuracy and performance has to be made [13]. Various commercial and academic co-simulation frameworks have been proposed in literature; more detailed surveys can be found in [13, 24, 5].

The Validator tool from Chrona [12] is based on a systematic way to instrument the application code with execution time information and execution control statements which enables capturing real-time behaviors at a finer time granularity than most of the currently available tools with similar functionality. Validator can operate in closed loop with plant models simulated by a different tool (such as Matlab/Simulink).

Validator can also simulate preemption at the highest level of abstraction that still allows for capturing the effect of preemption on data values, avoiding at the same time the slow, detailed simulation achieved by instruction set simulators. Validator enables advanced debugging and design space exploration of the entire simulated system (software, hardware configuration, plant). For example, it is possible to step through the execution of application code across preemption points both in forward and reverse directions. Validator also allows one to start a simulation from a previously saved state. Being implemented entirely in C, the simulator can be easily interfaced/integrated with existing simulation tools .

An example of Validator's application is regression testing. The objective is to compare the behaviors of an industrial engine control software (ECS) with a version that had been re-engineered to achieve robust timing behavior based on TDL. Validator was employed to simulate the two embedded systems in parallel, with the original system in control of the plant. The testing revealed several software bugs in the interface between the application code and the TDL runtime system. Simulation tools related to the Validator are the Time Multitasking Ptolemy domain [30] and TrueTime [10] in the academic area, as well as the commercial tool ChronSim from INCHRON [25].

A prerequisite for validation of a control application running on an embedded platform is the availability of execution times of the software for the platform. This is a major requirement and challenge in the case of automotive software, where real-time properties are crucial for a correct operation of the system. Various approaches have been proposed for obtaining conservative estimates (worst-case executions times). Techniques based on abstract interpretation form the basis for commercial tools such as the aiT analyzer from Absint [1], which relies on detailed processor models. The Gametime tool [48] employs path analysis and game-theoretic algorithms to estimate worst-case execution times based on measurements on the target hardware. A survey of techniques and tools for execution time estimation can be found in [57]. Execution time estimation tools are being integrated in larger modeling and simulation tool suites, as in the case of Scade [15] and Chrona's Validator.

4. MODEL BASED TESTING OF AUTOMOTIVE CONTROL SOFTWARE

At present the Simulink/Stateflow (SL/SF) modeling framework is one of the de-facto industry standards for developing automotive controllers. As described in the previous section, various techniques (mostly based on simulation) are available for validating the design models against the requirements. Testing is also one of the primary means employed for validating the controllers implemented in software. It serves multiple purposes like revealing bugs, enhancing the confidence in the implemented functionality and determining the control system's performance. When the system is safety-critical, then much more emphasis is given to the quality of testing. By testing, in general, we mean (a) code execution with respect to the test inputs and oracle matching, and (b) model simulation. Model simulation is included because it reveals the conformance of the model execution with respect to the intentions of the controller requirements.

Testing requires a Test Infrastructure, which involves the executable system under test (SUT), the system's environment, a test suite (timed input-output sequences) and a Test Bench which facilitates test execution, test result matching and test coverage estimation. Testing of an automotive software is quite extensive and a lot of time and efforts are spent on it. A design model and the corresponding code – generated manually or by a code-generator – together is treated as one unit. Unit testing means simulation of the design model or the testing of individual features on a workstation. Automotive control functions in general are subjected to a wide variety of testing possibilities, like, Plant-in-loop testing, HW-in-loop simulation and Vehicle level testing. The Vehicle level testing involves integrating all the features and domain functionality are tested on the vehicle with real execution on the target hardware and software platforms.

The test bench is a platform that enables automatic or manual testing of the Software Under Test (SUT). It prepares the SUT for testing, implements test execution, generates test reports and measures test coverage. An SUT does not run alone and requires supporting HW/SW which are provided (actual or model) by the test bench. They include extensive support for modeling the required plant and environment and one then carries out plant-in-loop or Hardware-in-loop (HIL) Testing. One example of such a test bench is the HIL set up from dSPACE. Elaborate test scripts are written which enable running and collecting of the test results. They (a) initialize the SUT, (b) put the SUT in the required context, (c) create test inputs, (d) pass the inputs to the SUT, (e) record SUT response, and (f) assign verdicts.

A test suite is a finite collection of test cases, each of which is a timed sequence of (input,output) pairs. It is a crucial artifact in testing. The quality of the test suite determines the extent of testing and directly influences the confidence in the SUT. The computation of test suites is mostly manual in practice and is considered one of the difficult steps. Due to the finiteness of testing, the test suite is often incomplete and its effectiveness is measured by some coverage criteria.

4.1 Recent Advances in Code-Based Testing

The classical testing techniques can be broadly classified into two kinds: functionality or black box testing and code-based or white box testing. In black box testing, the test cases are generated based upon requirements. The test cases are chosen so as to cover the functionality or input domain. Equivalence partitioning, domain decomposition are some of the well-known techniques for test case generation. Test case generation is often manual as formal requirements are absent. In code based testing or white box testing, the test cases are generated based upon the code under test. The tests are directed towards covering code elements such as statement, conditions, and branches. Symbolic execution and constraint solving are some of the common techniques used for generating test cases. Code-based testing recently attracted a lot of attention, thanks to the availability of better constraint solvers. In the following, we will outline some of these successful techniques.

Godefroid et al. in [17] discuss the DART (Directed Automated Random Testing) approach for testing of C programs. The power of DART comes from a combination of random inputs, constraint solving and directed coverage. Given a C program, first a random input vector is supplied to execute the program and the symbolic constraint associated with the covered path is extracted. This symbolic constraint is altered – a branch predicate is flipped – to obtain the path constraint of a path adjacent to the earlier path. If this path constraint can be solved, then we have the input vector for the new path. Given a finite execution tree, by systematically altering the branch predicates, the DART approach can cover all the paths in the tree. However, when the path constraints are non-linear or involves library function calls, the above is not a straightforward task. The DART approach uses concrete values from previous executions to address this problem in some specific cases[17]. Concolic Testing [47] is similar to DART but, in addition, it can deal with complex C data structures like structures and pointers.

Majumdar and Sen [31] have extended the concolic testing concept to cover deep targets in the state space of the program. Initially concolic testing is applied, and when saturation occurs in a region in the program state space as regards to target coverage, a random input sequence takes execution control to a different region. Concolic testing is performed in the new region, and upon coverage saturation, a new random sequence again takes control to another region in the state space. This continues till adequate coverage is achieved.

Mutation testing [26] is an emerging area in the field of code-based testing. Faults are seeded into the original program – which represents the mistakes that programmers are likely to make – and the resulting program is called a mutant. The quality of a test suite can be quantified by measuring to what extent the test suite can distinguish the original program from its (non-equivalent) mutants. It said that mutation testing subsumes many other test coverage criteria like MC/DC [35]

We will now discuss how the above techniques have been adapted in model-based testing of automotive designs specified in Simulink/Stateflow.

4.2 Model-based Testing

A broad definition of model-based testing (MBT) is the use of models in any step of the testing process. We classify MBT into three kinds:

- Weak MBT: Use of models in measuring coverage of testing over the model elements.

- Strong MBT: Models serve as the *golden model* for computing the test results.
- Strongest MBT: Models help in generating test suites. This is what is usually known under the name MBT.

We will focus on the strongest MBT in this paper and assume that the models are for discrete and continuous controllers expressed as SL/SF designs.

Note that the models of the environment and supporting HW/SW are provided by the *Test bench* module. Our test cases are directed towards covering requirement models and we assume that the requirements are properties of the SUT and the generated test cases satisfy/falsify such properties. The SL/SF framework provides techniques like assertion blocks which encode the properties at the requirement level, and can be embedded within the SL/SF models. There can be many test criteria in relation to the test case generation problem from SL/SF designs. The *Structural coverage* test criteria specify coverage over various model elements during model simulation. For a SL/SF model, such criteria could be State Coverage, Transition Coverage, Simulink Block Coverage and Condition Coverage. MC/DC (Modified Condition/Decision Coverage) [53] is also highly recommended as per the automotive standard ISO 26262. The coverage of the input domain is also another test criteria.

There are many test case generation techniques which depend upon the models used. Some of the popular methods are: (a) Random Test case generation, (b) Systematic model exploration, (c) Symbolic execution and Constraint Solving, and (d) test case generation using model checking. Random test case generation is one of the widely used methods as it is simple and quite effective in achieving reasonably high coverage when all the paths in a model or program are equally likely. Often models have paths that require specific input sequences to be chosen. In such cases, the systematic exploration technique is effective. This is also one of the classical techniques, in which all paths up to some finite length are enumerated, the symbolic constraints of the paths are collected and the constraints are solved to identify the inputs. One of the reasons why this method is very successful is its dependence upon constraint solvers. This technique has been very effective in protocol conformance testing where the models are finite state machines.

With the emergence of powerful constraint solvers, recent efforts have tried to combine random methods with systematic exploration techniques. DART and Hybrid concolic testing, the two recent techniques for code based testing, has been also applied to SL/SF model-based testing.

Model checking is a well-known technique for formal verification of finite state models. In model-checking, counter-examples are generated when a given property is not satisfied by the model. Counter-examples can be considered as test cases when the property is appropriately chosen. Given a property that a certain state or transition or condition is not reachable, the model checker can produce a counter-example that actually reaches the state/transition yielding a test case. The advantage of this approach is that design verification as well as automatic test case generation (ATG) can be performed together. The model size can be a limiting factor in the performance of the model checker. For an Automatic Test Case Generation (ATG) tool using model checking, test criteria specification and model requirements are additional inputs. The coverage goals like state, transition, and MC/DC over the SL/SF models can be translated into equivalent formal models like SAL [51]. Counter examples generated by the model checker are turned into input traces, which are in turn used to simulate the model to generate the output sequence and hence the test cases.

4.3 Tools for SL/SF Model based ATG

Many commercial tools have appeared in the market for test case generation from SL/SF designs. Some of the prominent tools are: *Reactis* from Reactis System Inc. [37], *STB* from the TNI Software [41], `BEACON Tester` from Applied Dynamics International [3] and `T-VEC tester` from the T-VEC technologies [52]. All these tools, given a SL/SF design model and a coverage goal over the model elements, generate test vectors meeting the coverage goal. These tools use powerful random or constraint solving techniques or a combination of both.

AutoMOTGen [16] uses model checking for test case generation from SL/SF models. SL/SF models are transformed into a SAL [51] formal model. Next, based on a test criteria, *bounded model checking* (BMC) is applied on the SAL model to derive test cases from counter example traces. Simulink Design Verifier (SDV) from the `Mathworks` uses model checking for proving model properties of SL/SF models and to generate model-based test cases. These tools face scalability problems when a model is large or when the models involve complicated non-linear constraints.

Redirect is another ATG tool [45, 44] which adapts DART [17] and Hybrid Concolic testing [31] techniques for generating test cases from SL/SF models. When a constraint of a certain target is non-linear, then Redirect uses a set of heuristics depending on the pattern of the target to cover the targets. Consider a constraint involving a non-linear sub-constraint like $f(x, y) > g(x)$, f and g being non-linear functions. One heuristic could be to fix the value of x to a constant, and then value of y can be either gradually increased or decreased to cover the desired target. Redirect handles the issue of scalability because, similar to hybrid concolic testing, before a constraint solving or a heuristic application, a random phase can be used. Experiments have shown that the heuristics are able to cover deep non-linear targets.

Mutation testing has also been applied to SL/SF models for the purpose of test case generation [21]. Finding the difference between a Simulink model and its mutant has been cast as a model checking problem, and test cases can be obtained from the counter-example traces. The authors have optimized this process by analysing the structure of the Simulink models.

Alur et al. [28, 27] have presented an analysis technique which combines numerical simulation with symbolic analysis for input domain coverage of SL/SF models. Given a simulation trace of a SL/SF model, using polyhedra based backward symbolic analysis, a region of the initial state space inducing equivalent behaviors in the model is identified. Further simulation is carried out on the model, starting from initial states outside of this region and the above step is repeated a finite number of times to have adequate coverage over the initial states.

4.4 Issues and Challenges in Testing

There are many challenges in generating test cases from SL/SF control designs some of which are listed below:

- **Test cases being too long:** It is sometimes the case that a huge number of simulation cycles are needed to cover a target deep in the state space. This happens in particular when the model involves an integrator block whose output has to attain a high value so that a specific target is satisfied. This target may be too deep for a model checker or the corresponding constraint might be too large. Heuristics may cover such targets but without any guarantee.

- **Distributed control:** In an integrated architecture, components may be across multiple ECUs, connected by buses like CAN, FlexRay and LIN. These networks may cause message delays. It may be necessary to decompose a SL/SF model into component models and the components going to multiple ECUs. In this case, the blocking semantics of the original model may not match the non-blocking semantics of the component models. Therefore the test cases of the original models are in danger of being invalid when we use the distributed design.

- **Testing of non-functional properties:** In a distributed implementation, testing of non-functional properties like the timing properties is a challenge.

5. PERFORMANCE ANALYSIS OF AUTOMOTIVE CONTROL FUNCTIONS

Finally, in this section we discuss issues related to the performance analysis of controllers when they are implemented on a distributed automotive architecture. In general, a feedback control loop performs mainly three operations:

- *Measure*: The output of the dynamical system is measured by one or more measuring devices or *sensors* and thus, the measured signals act as feedback signals.

- *Compute*: The feedback signals are compared with desired output and necessary correction in the input signal is computed by the control algorithm.

- *Actuate*: The correction is incorporated by changing the input signal or actuation.

In an ideal implementation, a feedback loop is executed instantaneously and the resulting system behaves like a continuous-time system. However, performing these operations continuously in any implementation platform requires infinite computational power. Hence, in a digital implementation platform of such feedback loop, these operations are performed only at discrete time intervals (sampling instants). Naturally, the performance of the control functions depends on the choice of sampling period.

Fig. 4 shows an execution sequence of a typical feedback control loop. A control function is mainly *partitioned* into three categories of tasks: T_m (measure), T_c (compute) and T_a (actuate). In a distributed setup, the execution time of a feedback loop consists of (i) finite execution time of the three operations at the ECUs (e_s,e_c,e_a), (ii) the transmission time of the messages over the communication medium (e_{m1},e_{m2}), (iii) the waiting time among various data-dependent tasks and messages (e_w) as shown in Fig. 4. The execution time or sensor-to-actuator delay τ is the summation of all these delay components (Fig. 5).

The sampling period of a control function decides how frequently the tasks need to be executed and how many messages are transmitted over the bus. Intuitively, a shorter sampling period implies more computational demand at the ECUs and a higher bandwidth requirement from the buses. Hence, the possibility of experiencing longer sensor-to-actuator delay is higher in the case of shorter sampling periods.

5.1 Control Performance

Typically, a high level goal of such feedback control aims to achieve desired behavior (i) fast, and (ii) accurately, while stability of the loop must be maintained. The speed and accuracy of the feedback control system quantify the performance of the controller. There is a wide variety of notions to quantify the performance of a control function. The performance notions are broadly classified into two categories: *steady state* and *transient phase* performance. An example of a transient performance index is the *settling time*, i.e., how fast the system responds to some external *disturbances.* One the other hand, a commonly used performance index at steady state is the quadratic cost function which captures a combination of tracking accuracy and input energy,

$$J = \int_0^\infty [u(t)^2 + y(t)^T y(t)]dt, \qquad (1)$$

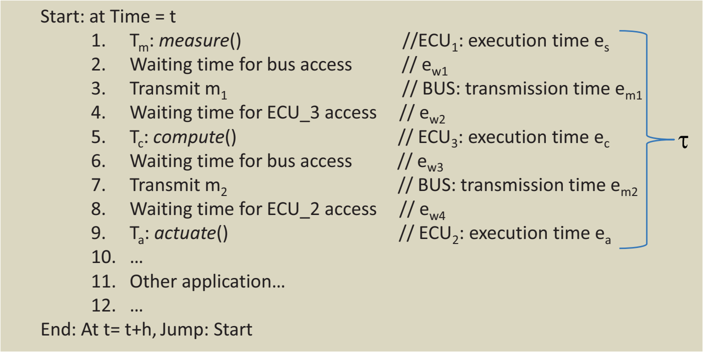
where $u(t)$ and $y(t)$ are the system's input and output respectively. In both cases, the performance depends on (i) the sampling period h, and (ii) the sensor-to-actuator delay τ.

Hence, an optimal implementation of a control function often boils down to the problem of finding the optimal tradeoff between the sampling period and the resulting sensor-to-actuator delay in the feedback loop.

5.2 Approaches

The various timing constraints coming from high-level specifications of control functions pose constraints on the scheduling of the buses and the ECUs. Here, *control/scheduling codesign* methods mainly focus on co-synthesis of the control strategy and the scheduling parameters to deal with feedback delays, their variation and their impact on control performance. The feedback delays in the control loop are dealt with by designing appropriate control strategies following research that originated from the Network Control System (NCS) community [56]. In these approaches, the network acts as a black box which injects feedback irregularities such as delay and packet drops. The major difference between the distributed setting in NCS and the automotive setups we are discussing, is that the designer has sufficient handle over the network parameters in our case. There has been some research effort to jointly optimize control functions and their implementation platform over the last decade [50, 9, 58, 54, 46, 23]. Here, a typical setup might consist of multiple control functions running on either multiple ECUs [50, 54, 46] or on a single ECU [4, 9].

Most of the research in the direction of joint control/scheduling optimization relies on the facts that (i) shorter period improves the control performance, and (ii) a shorter sampling period requires higher computational and communication resource usage [9, 46]. The overall goal is to choose sampling periods of the control tasks such that (i) the control functionalities are robust and provide desired performance, and (ii) all the control and other tasks and messages are schedulable, given the resource constraints.



```
Start: at Time = t
    $T_m$: measure()                      //$ECU_1$: execution time $e_s$
    Waiting time for bus access           // $e_{w1}$
    Transmit $m_1$                        // BUS: transmission time $e_{m1}$
    Waiting time for ECU_3 access         // $e_{w2}$
    $T_c$: compute()                      // $ECU_3$: execution time $e_c$
    Waiting time for bus access           // $e_{w3}$
    Transmit $m_2$                        // BUS: transmission time $e_{m2}$
    Waiting time for ECU_2 access         // $e_{w4}$
    $T_a$: actuate()                      // $ECU_2$: execution time $e_a$
    …
    Other application…
    …
End: At t= t+h, Jump: Start
```

Figure 4: Execution sequence of typical feedback control loops.

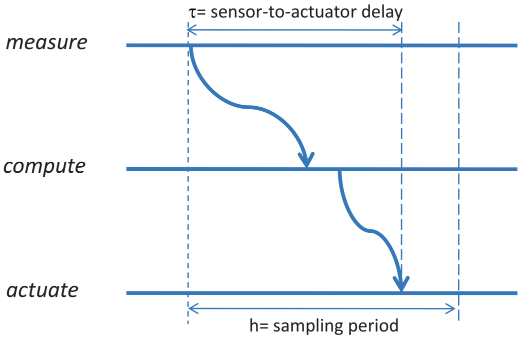


Figure 5: Timing in feedback control loops.

A number of recent research efforts focus on exploiting certain inherent properties of control loops for the co-synthesis of control strategy and schedules. For example, the fact that control loops are generally robust and a fraction of messages may be allowed to miss their deadlines without compromising with the high-level requirements (such as the stability and performance) is utilized for control/scheduling co-design [18]. For example, Fig. 6 shows the effect of feedback (packet) drop or deadline miss in a DC motor speed control application in electric cars. The goal is to achieve a speed reference of 50 units. The result shows the system behavior in the presence of certain packet drops. The system behavior does not deteriorate much with up to 25% of the feedback signals being dropped. Clearly, the control loop is robust enough to accomodate certain irregularities. The challenge is to quantify the amount of allowed irregularity in the feedback loop to guarantee stability and meet the desired performance of the loop.

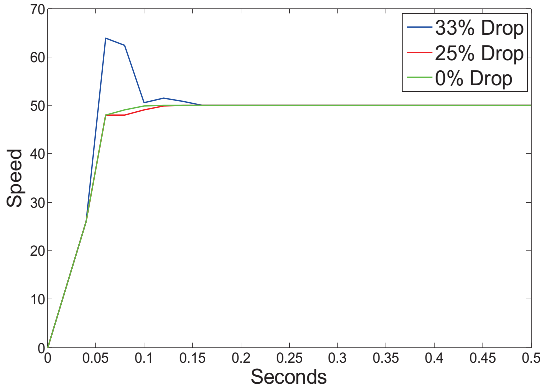


Figure 6: Robustness of a control loop.

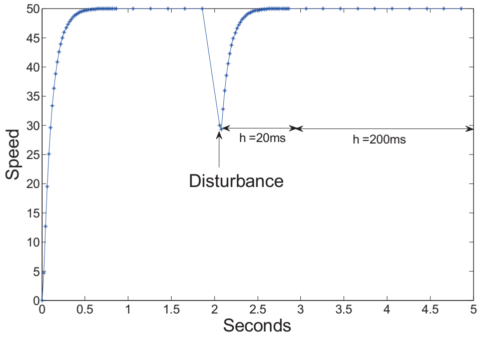


Figure 7: Sensitivity of control performance to sampling period in transient and steady states.

Another key aspect of control functions is that their sensitivity to feedback irregularities highly depend on the *state* of the overall system. In particular, the performance deterioration due to feedback irregularities is prominent in a *transient* phase compared to in the *steady* state. For example, Fig. 7 shows the output behavior of the DC motor speed control with a higher sampling rate at the transient phases (resulting from external disturbances) and a much lower sampling rate at the steady state. Therefore, by utilizing such *multi-rate* sampling schemes it is possible to improve the bandwidth utilization of buses and ECUs without compromizing either the performance or the stability of the system. Some of the recent research in this area utilizes similar ideas for control/scheduler co-design [8, 32, 19].

6. CONCLUDING REMARKS

A large number of control functions run on modern automotive architectures consisting of several ECUs and one or more communication buses. The typical development stages involved are (i) specifying high-level functional requirements using the Simulink/Stateflow framework, (ii) automatically generating code from Simulink/Stateflow, (iii) validating the software models so that timing requirements at the high level are met (iv) testing and verification of the software models implemented on a platform (v) performance analysis to trigger suitable modifications through a control/platform co-design. In this paper, we outlined the state-of-the-art in this domain and discussed some of the challenges are yet to be addressed.

7. REFERENCES

[1] Absint. aiT worst-case execution time analyzers. http://www.absint.com/ait/, 2010.

[2] AUTOSAR Specification of FlexRay interface, version 3.0.3, www.autosar.org.

[3] Applied Dynamics International. www.adi.com.

[4] E. Bini and A. Cervin. Delay-aware period assignment in control systems. In *IEEE RTSS*, 2008.

[5] G. Bosman. A survey of co-design ideas and methodologies. Master's thesis, Vrije Universiteit, 2003.

[6] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java (Volume 1: Introduction to Ptolemy II). *EECS Department, University of California, Berkeley*, 2007.

[7] K. Butts. Presentation at the 2011 Workshop on open problems and challenges in automotive control, UC Berkeley. http://www.mpc.berkeley.edu/2011-workshop/presentations, 2011.

[8] R. Castane, P. Martí, M. Velasco, and A. Cervin. Resource management for control tasks based on the transient dynamics of closed-loop systems. In *ECRTS*, 2006.

[9] A. Cervin and P. Alriksson. Optimal on-line scheduling of multiple control tasks: A case study. In *ECRTS*, 2006.

[10] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and Karl-Erik Årzén. How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. *IEEE Control Systems Magazine*, 23(3), 2003.

[11] Chrona GmBH. Chrona creation suite. http://www.chrona.com/en/products/creation-suite/, 2011.

[12] Chrona GmBH. Chrona validation suite. http://www.chrona.com/en/products/validation-suite/, 2011.

[13] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, pages 366–390, 1999.

[14] Esterel Technologies. Understanding how SCADE KCG generates safe C code (white paper). http://dnlcenter.esterel-technologies.com, 2011.

[15] C. Ferdinand, R. Heckmann, F. Martin, T. L. Sergent, D. Lopes, and X. Fornari. Combining a high-level design tool for safety-critical systems with a tool for WCET analysis of executables. In *The 4th European Congress on Embedded Real Time Software*, 2008.

[16] A. A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik, and K. C. Shashidhar. Automotgen: Automatic model oriented test generator for embedded control systems. In *CAV*, 2008.

[17] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[18] D. Goswami, R. Schneider, and S. Chakraborty. Co-design of Cyber-Physical Systems via controllers with flexible delay constraints. In *ASP-DAC*, 2011.

[19] D. Goswami, R. Schneider, and S. Chakraborty. Re-engineering Cyber-Physical control applications for hybrid communication protocols. In *DATE*, 2011.

[20] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[21] N. He, P. Rummer, and D. Kroening. Test-case generation for embedded Simulink via Formal Concept Analysis. 2011.

[22] T. A. Henzinger, C. M. Kirsch, M. A.A. Sanvido, and W.Pree. From control models to real-time code using Giotto. *Control Systems Magazine, IEEE*, 23(1):50–64, 2002.

[23] S. Hong, X. S. Hu, and M. D. Lemmon. Reducing delay jitter of real-time control tasks through adaptive deadline adjustments. In *ECRTS*, 2010.

[24] H. Huebert. A survey of HW/SW co-simulation techniques and tools. Master's thesis, Royal Institute of Technology, Stockholm, Sweden, 1998.

[25] INCHRON. The CHRONSIM simulator. http://www.inchron.com/chronsim.html, 2010.

[26] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 2010.

[27] A. Kanade, R. Alur, F. Ivancic, S. Ramesh, S. Sankaranarayanan, and K. C. Shashidhar. Generating and analyzing symbolic traces of Simulink/Stateflow Models. In *CAV*, 2009.

[28] A. Kanade, R. Alur, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow Models. In *EMSOFT*, 2008.

[29] J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli. Software timing analysis using HW/SW co-simulation and instruction set simulator. In *CODES/CASHE*, 1998.

[30] J. Liu and E. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine*, 23:65–75, 2002.

[31] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, pages 416–426, 2007.

[32] A. Masrur, D. Goswami, R. Schneider, H. Voit, A. Annaswamy, and S. Chakraborty. Schedulability analysis of distributed Cyber-Physical applications on mixed time-/event-triggered bus architectures with retransmissions. In *IEEE SIES*, 2011.

[33] Mentor Graphics Corporation. Seamless co-verification environment user's and reference manual, version 2.2. Wilsonville, Oregon, 1996-1998.

[34] Object Management Group. Model driven architecture, 2010.

[35] J. Offutt and J.M. Voas. Subsumption of condition overage techniques by mutation testing. 1996.

[36] W. Pree and J. Templ. Modeling with the Timing Definition Language (TDL). In *ASWSD*, 2006.

[37] Reactive Systems. www.reactive-systems.com.

[38] S. Resmerita, P. Derler, A. Naderlinger, and W. Pree. Migration of legacy software towards correct-by-construction timing behavior. In *MontereyWorkshop on Modelling, Development and Verification of Adaptive Computer Systems*, 2010.

[39] S. Resmerita, P. Derler, W. Pree, and A. Naderlinger. Modeling and simulation of TDL aplications. *Lecture Notes in Computer Science*, 2010.

[40] J. A. Rowson. Hardware/software co-simulation. In *DAC*, 1994.

[41] TNI-Software. www.tni-software.com.

[42] I. Saha, K. Chakraborty, S. Roy, B. V. Reddy, V. Kurapati, and V. Sharma. An approach to reverse engineering of C programs to Simulink models with conformance testing. In *ISEC*, 2009.

[43] Alberto Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, 2002.

[44] M. Satpathy, A. Yeolekar, P. Prakash, and S. Ramesh. Efficient coverage of parallel and hierarchical Stateflow models for test case generation. *Journal on Software Testing, Verification and Reliability*, 2011.

[45] M. Satpathy, A. Yeolekar, and S. Ramesh. Randomized directed testing (redirect) for Simulink/Stateflow Models. In *EMSOFT*, 2008.

[46] R. Schneider, D. Goswami, S. Zafar, M. Lukasiewycz, and S. Chakraborty. Constraint-driven synthesis and tool-support for FlexRay-Based automotive control systems. In *CODES+ISSS*, 2011.

[47] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, 2005.

[48] S. A. Seshia and J. Kotker. Gametime: A toolkit for timing analysis of software. In *TACAS/ETAPS*, 2011.

[49] J. Sifakis, S. Tripakis, and S. Yovine. Building models of real-time systems from application software. In *Proceedings of the IEEE Special issue on modeling and design of embedded*, pages 100–111, 2003.

[50] S. Soheil, P. Eles, Z. Peng, P. Tabuada, and A. Cervin. Dynamic scheduling and control-quality optimization of self-triggered control applications. In *IEEE RTSS*, 2010.

[51] sal.csl.sri.com.

[52] T-VEC technologies. www.t-vec.com.

[53] S. A. Vilkomir and J. P. Bowen. From MC/DC to RC/DC: Formalization and analysis of control-flow testing criteria. *Formal Methods and Testing*, pages 240–270, 2008.

[54] H. Voit, R. Schneider, D. Goswami, A. Annaswamy, and Chakraborty S. Optimizing hierarchical schedules for improved control performance. In *IEEE SIES*, 2010.

[55] Yin W., Stéphane L., Terence K., Manjunath K., and Scott M. The theory of deadlock avoidance via discrete control. In *POPL*, 2009.

[56] G. C. Walsh, H. Ye, and L. G. Bushnell. Stability analysis of networked control systemss. *IEEE Trans. on Control System Technology*, 10(3):438–446, 2002.

[57] R. Wilhelm, J. Engblom, A. Emerdahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C.n Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), 2008.

[58] F. Zhang, K. Szwaykowska, W. Wolf, and V. J. Mooney. Task scheduling for control oriented requirements for Cyber-Physical Systems. In *IEEE RTSS*, 2008.