Engineering High Confidence Medical Device Software

Arnab Ray

Fraunhofer Center for Experimental Software Engineering

ARay@fc-md.umd.edu

Raoul Jetley Paul Jones

US Food and Drug Administration, Center for Devices and Radiological Health

{raoul.jetley, paull.jones}@fda.hhs.gov

Abstract

The increasing complexity of medical device software has created new challenges in ensuring that a medical device operates correctly. This paper discusses how two technologies — model-based development and static analysis — may be used to facilitate the successful engineering of medical software and some possible regulatory side benefits.

Keywords model-based development, formal verification, static analysis, instrumentation based verification

1. Introduction

The amount of software present in medical devices has dramatically increased over the last decade. Many infusion pumps today contain tens of thousands of lines of code. This number can run into the millions for proton beam therapy devices. Software is considered by many to be easier to configure, change and re-use than hardware. It is a technology that enables robust device designs. The need for high-integrity software in the health-care industry has become more important than ever as remote surgery, intelligent operating rooms, autonomous assisted living environments, and bio-feedback based prosthetics become the norm in the not-so-distant future.

The increasing complexity of device software presents considerable engineering challenges. In 1998, close to 8% of device failures could be traced to software errors [5]. Currently, the number of device recalls due to software problems is believed by some to be about 18%. It is likely that device failures and subsequent recalls will continue to increase until software is better engineered.

Figure 1 depicts a generalized software development workflow process typically followed by device manufacturers. The quality of the code in this workflow process is generally ensured through verification activities such as manual inspections, code walkthroughs and testing. Integrated system testing typically takes place at the end of the development lifecycle. Such verification activities, in the context of a quality system, have historically been considered sufficient for developing quality software. However, history has shown that common practices within this workflow process are insufficient for developing highly dependable software [10]. The reason for this is that these largely human resource intensive activities simply cannot fathom, unaided, the interdependencies of complex requirements and code.

Some of the limitations associated with traditional software development techniques can be summarized as follows:

- No formal, mathematics-based verifiable relationship is established between the design and the code
- Without a formal relationship established, it is difficult to demonstrate that the design and the code conform to each other structurally as well as behaviorally
- Without a formal methods foundation, rigorous verification and validation results are difficult to demonstrate throughout the life-cycle process
- Without the use of statistically based testing methods code coverage is difficult to characterize objectively
- No formal relationship is established between system property requirements (e.g., safety, security, privacy, etc), code, and the test suite used to verify the software. As a result, there is no reliable way to ensure that the software addresses these property specific requirements.

The risks associated with current software development practices will likely increase as medical device cyber-physical systems[1] such as non-homogeneous interoperable medical devices begin to enter the health care system. Configurations of such devices will be highly variable and reconfigurable in order to provide support in operating rooms, in hospital rooms, and in home care environments. For example, during a surgical operation, a number of "off-the-shelf" medical devices may be networked together to monitor and safely react to a patient's changing physiology.

HCMDSS 2009 April 16 2009, San Francisco.

[1] The term cyber-physical systems refers to the tight conjoining of and coordination between computational and physical resources

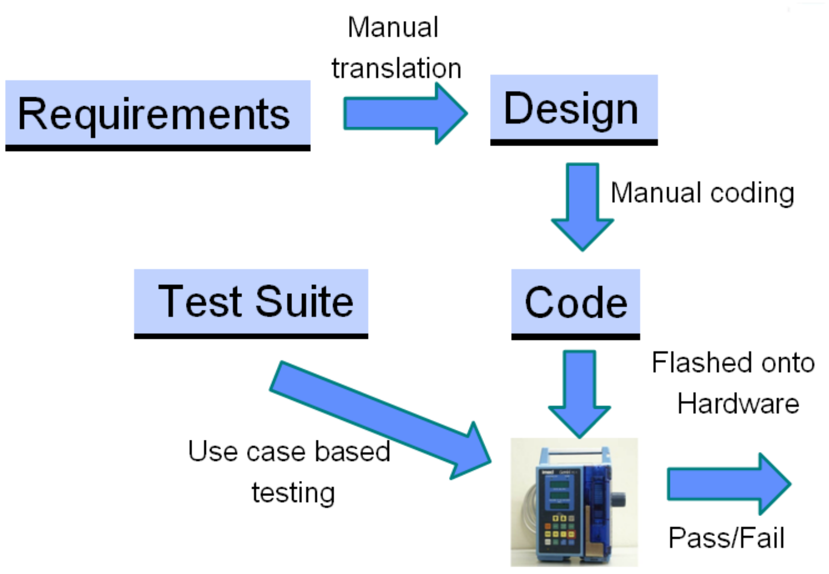

Figure 1. Traditional Software Development Workflow

Emerging medical device cyber-physical systems, such as interoperable medical device systems and prosthetics, bring new engineering challenges to the medical device development space in terms of scale, security, privacy, timing, human factors, composition, sensing, coordination, control, and certifiable evidence based verification and validation. Ultimately, research and development is needed to establish cyber-physical device composition and integration technologies and certifiable tool chains that address issues of logical and physical interoperability [8].

The remainder of this paper discusses some mathematically well-founded technologies for design, development and verification of medical device software. These techniques have been used with considerable success in other safety-critical industries such as aerospace and automotive engineering. In particular, we discuss model-based development and static analysis, and discuss how these technologies might be leveraged in a regulatory environment.

2. Model-based Development

A model can be thought of as a formal representation of a design specification. A model can also be used to capture the essential structure and behavior of a component or system. Of particular interest to developing high-confidence medical device software is the notion of ”executable modeling notations.” Executable modeling notations can be distinguished from conventional design notations by the fact that they are based upon a mathematically precise notion of what it means for a model to perform a behavioral action. This means that designs rendered in an executable modeling notation can be simulated and debugged just like normal code that has been written in a traditional programming language like C or C++.

The principal advantage of using executable models, hereafter referred to as simply as models, over conventional programming languages is that they free the developer from implementation details like pointer management and memory allocation. This is analogous to the way programming languages abstract away low-level details of processor instruction sets, facilitating the separation of software and hardware concerns. Modeling makes it easier for the developer to focus resources on various aspects and properties of a particular design.

To verify models on a particular hardware platform they must be converted to code through a process called “automatic code generation.” Modeling tools are used to convert modeling notations into code. Compilers then transform this code into machine language that can then be executed on the hardware.

This kind of software development, where the model serves as the primary artifact, is often referred to as model-based development. Over the years, model-based development techniques have become standard practice in the production of high-integrity embedded software in the aerospace and automotive industries.

A major advantage of a model-based development workflow is that it facilitates catching and correcting errors early in the development lifecycle. Since models can be constructed much faster than code, designers can rapidly create prototypes of their system and study various design alternatives before committing to a final implementation. Also, owing to the executable nature and formal semantics of these models, various analytical verification and validation (V&V) methods like model-checking [4] or instrumentation-based verification (IBV) [1] can be used to formally prove

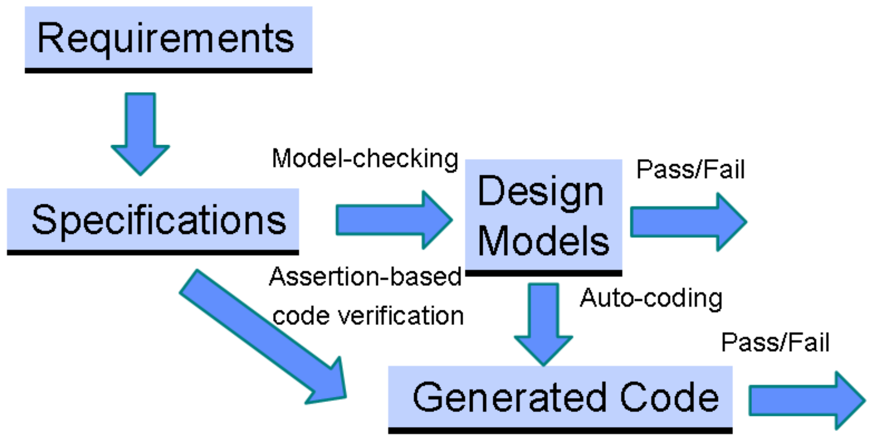

Figure 2. Model-checking based Verification Workflow

that the software design satisfies functional and specific property requirements (e.g. safety, security, etc).

When using model-based V&V techniques, natural language requirements are first converted to formal specifications, which may be expressed as either temporal logical formulae [2] or monitor models (*monitors* for short) [1]. Temporal logic formulae are typically employed for model-checking. Monitors may be thought of as encodings of idealized system behavior that are executed concurrently with the models to guarantee consistent results during IBV.

If model checking is used (as shown in Figure 2), then the logical specifications (or temporal logic formulae) are checked against finite-state representations of the design model using either sophisticated graph traversal or equation solving techniques. A model is said to be verified against a set of specifications, if for all possible model executions, it is not possible for any of the specifications to be violated. On the other hand, if a specification is violated by an execution trace, the model is deemed to be erroneous. (This execution trace is often generated by the model-checker as proof of the specification violation.)

If IBV is used as the V&V method of choice, as shown in Figure 3, the design model is first instrumented with monitor models. A test-generation engine is then used to check the composite of the design model and the monitor against a series of automatically generated tests. The aim here is to determine whether the actual behavior of the design model and the idealized behavior of the monitor instrumentation diverge from each other. In other words, the test-generation engine takes the role of a pessimistic observer and generates tests so as to “break” the design. If it is successful in observing a divergence between the design model and the monitor, it outputs the relevant test case as the rationale for why the specification is not satisfied.

The metric that specifies how extensively the model’s behavior is covered by the tests is known as a coverage criterion. Various coverage criteria can be used to verify the model based on how rigorous the test cases need to be. For example, line coverage stipulates that each model element needs to be executed at least once for the test suite to be complete. Decision coverage, on the other hand, enforces that boolean expressions tested in control structures (such as the if-statement and while-statement) must evaluate to both true and false. The coverage criterion typically used by IBV is known as MC/DC (modified condition decision coverage). MC/DC stipulates that tests should be generated until each boolean sub-expression in a conditional expression has been shown to independently affect the outcome of the expression. MC/DC is considered by the Federal Aviation Agency (FAA) to be the most exhaustive coverage criterion and is used for testing the most critical type of aerospace code.

Once the model has been verified, using either model checking or IBV, automatic code generation is used to derive the core source code for the device. The generated code typically needs to be instrumented by hand in the same way outputs of compilers need to be optimized for certain applications[2].

There are two principal ways to perform code verification in the model-based development process. The first is applied using model checking techniques. In this method, the logical specifications are first converted to assertions, the generated code is instrumented with these assertions, and code verification tools [6] run on the modified code. In contrast to this rather direct method of re-verifying the requirements on the code, one may adopt an alternative strategy, where the code and the design are shown to be behaviorally equivalent to each other [12]. Since the design has already been verified, we may conclude that the code also satisfies the requirements. This alternative strategy makes use of IBV work, wherein the test suite generated as part of the model

[2] With advances in code-generators we expect to see production-level highly-optimized code being produced directly from models in the future.

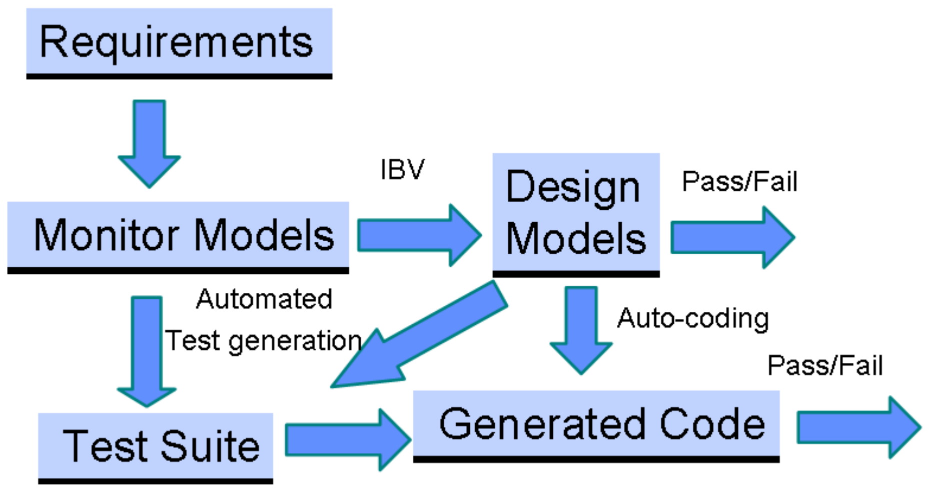

Figure 3. Instrumentation-Based Verification Workflow

verification is re-used for code. The code is verified to be correct if outputs of the model and the code are equivalent. If they are not, one may suspect that behavior has been introduced in the code that may lead to the violation of a requirement.

It should be noted that both these techniques for code verification are driven by requirements. In the case of assertion-based verification, the code is checked against assertions that are derived directly from the requirements. In the testing equivalence method, the test set that is used to prove behavior conformance between design and code is generated primarily by referencing the requirements. In both approaches, a direct traceable connection between the requirements and code verification activities is established. This traceability, base in mathematics, can help establish a convincing argument that the software has been checked with respect to its requirements at each stage of the development life-cycle.

The use of model-based V&V techniques reduces the dependence on testing as the principal means for verification, while at the same time providing a means for detecting design errors early in the development life-cycle. Clearly, the earlier errors are detected and corrected, the greater are the benefits in terms of time and cost; a fact expressed succinctly by the great architect Frank Lloyd Wright — "You can use an eraser on the drafting table or a sledge hammer on the construction site".

The nature of these design formalisms is such that they could be used in a regulatory context to challenge manufactured products for specific properties, such as safety, security, etc., acting as pseudo reference standards. In the FDA/CDRH/OSEL[3] software laboratory we were able to establish an infusion pump safety model using these methods. From this model, we were able to establish a set of alarm safety assertions and insert them in code from a real infusion pump implementation. The Verisoft[4] tool was used to perform systematic state space exploration of the code and check if any of these assertions were ever violated without triggering the appropriate alarm. Several alarms were not triggered that should have been [9]. An advantage of using Verisoft was that the assertions could be checked on all possible paths of the program and not just a specific execution path, as with runtime checking.

Clearly, it is impractical for regulators to develop such reference models for all medical devices. However, it is eminently practical for device manufacturers to carry out their own property-specific verification activities, and get "regulatory credit" for the work. One way of presenting this work is in the form of an assurance (or dependability) case [10]. For example, the *claim* might be that the device is safe. The *evidence* might be a test result report showing that all safety properties are met. And, the *argument* might be a safety model and an explanation of how it relates to the test results.

3. Static Analysis

Static analysis can be defined as an analysis of software that is performed without executing code, i.e., by analyzing some static artifact like source code or object files. Using static analysis facilitates detecting errors while the code is under development, thus reducing development and maintenance costs and the risk of expensive device recalls. In the context of high-confidence medical software, static analysis may be carried out for two principal purposes: a) checking the

[3] Food and Drug Administration/Center for Devices and Radiological Health/Office of Science and Engineering Laboratories

[4] Verisoft is a freely available state-space exploration tool for C programs. The use of Verisoft for the research study does not imply FDA endorsement of the tool.

source code to ensure that architectural constraints are not violated, and b) discovering errors in the source code.

3.1 Checking Architectural Constraints through Static Analysis

In the previous section, we described how assertion-based code verification and testing equivalence aims to establish the identical behavior of design models and code with respect to satisfying the requirements. However, these techniques do not check whether structural constraints defined in the design architecture are actually implemented in code. Static analysis can be used to make such checks.

The structural constraints that designers impose on code stem from considerations of extensibility and maintenance. For example, in a layered protocol, a layer is only allowed to use functions provided by its immediate subordinate so that a layer implementation may be replaced easily with another. However, such constraints formulated at the design phase are often not followed in the implementation. This often leads to spaghetti (highly-coupled) code that while perhaps still functionally correct, is extremely difficult to maintain and modify. In order to prevent this architectural degeneration, the code needs to be checked for off-specification dependencies. This can be done by using static analysis techniques to extract the implemented architecture from code [11]. The extracted architecture can then be compared to the required architectural specifications. This comparison can help identify dependencies that are present in the code but should not be and dependencies that should be present in the code but are not.

As an example, consider the architecture diagrams shown in Figure 4. Figure 4(a) shows a design architecture where component A is expected to communicate with B and B with C. However, after performing static analysis, we find that even though a dependency exists between A and B as planned, the expected dependency between B and C is missing and an extra dependency between A and C, that was not supposed to exist, is now present.

3.2 Detecting Runtime Errors using Static Analysis

While dependency analysis on extracted architectures may help guard against design errors, it does not afford any kind of protection against low-level coding errors. Coding errors usually manifest themselves as run-time bugs, such as null pointer dereferences, buffer overruns, arithmetic errors and memory leaks. Until recently, the only way to detect these errors was by means of rigorous code reviews and dynamic testing. However, with advances in lightweight formal methods techniques, a number of these defects can now be detected using static analysis.

While dependency analysis on extracted architectures may help guard against design errors, it does not afford any kind of protection against low-level coding errors. Coding errors usually manifest themselves as run-time bugs, such as null pointer dereferences, buffer overruns, arithmetic errors and memory leaks. Until recently, the only way to detect these errors was by means of rigorous code reviews and dynamic testing. However, with advances in lightweight formal methods techniques, a number of these defects can now be detected using static analysis.

There are many different types of static analysis techniques for detecting run-time bugs, such as symbolic execution [7] and abstract interpretation [3]. These techniques focus on assessing run-time bugs by evaluating intricate interactions within the software. For example, values of variables as they are manipulated down a path through the code, or the relationship between how parameters of functions are treated and the corresponding return values. To analyze code with this level of sophistication, all possible paths in the software are exhaustively analyzed to check for potential software anomalies.

By searching exhaustively through all paths in the program, these static analysis techniques can uncover bugs that may not be caught by testing alone. Since each test case follows only a specific path in the program, a finite number of tests can only check a limited set of possible execution paths. Usually these paths cover only a small fraction of the total possible paths in the software. Static analysis, on the other hand can evaluate all possible execution paths through the program; subject to the constraints of the tool employed.

Despite providing greater code coverage than testing, static analysis does have its limitations. Since the analysis is performed at compile-time, it is impossible to ascertain the actual values of input parameters and program variables used during execution. Static analysis tools therefore have to assume all possible values for these variables. This makes the analysis computationally intensive and causes high false positive[5] rates. Alternatively, the analysis tools may use heuristics to improve performance, yielding false negatives[6] as a result. In the ideal case, static analysis tools should have no false positives, no false negatives, and run in approximately the same amount of time as is required for compilation. However, this is not possible given the current state of technology. Therefore, most effective static analysis tools instead try to find the elusive sweet spot between false positives, false negatives, and performance to make results useful for every day software development.

It must be noted that static analysis is most effective when used in combination with traditional V&V techniques. It must be viewed as a complement to, rather than a replacement for, conventional V&V methodologies. Ideally, of course, static analysis should be integrated with a manufacturers' software development life-cycle process. Using it as code, is developed helps developers identify and repair defects prior to adding the code to a code baseline. Similarly,

[5] A false positive is any result that a static analysis tool reports that is not actually a defect in the source code.

[6] A false negative is any defect in the code that a static analysis tool does not report.

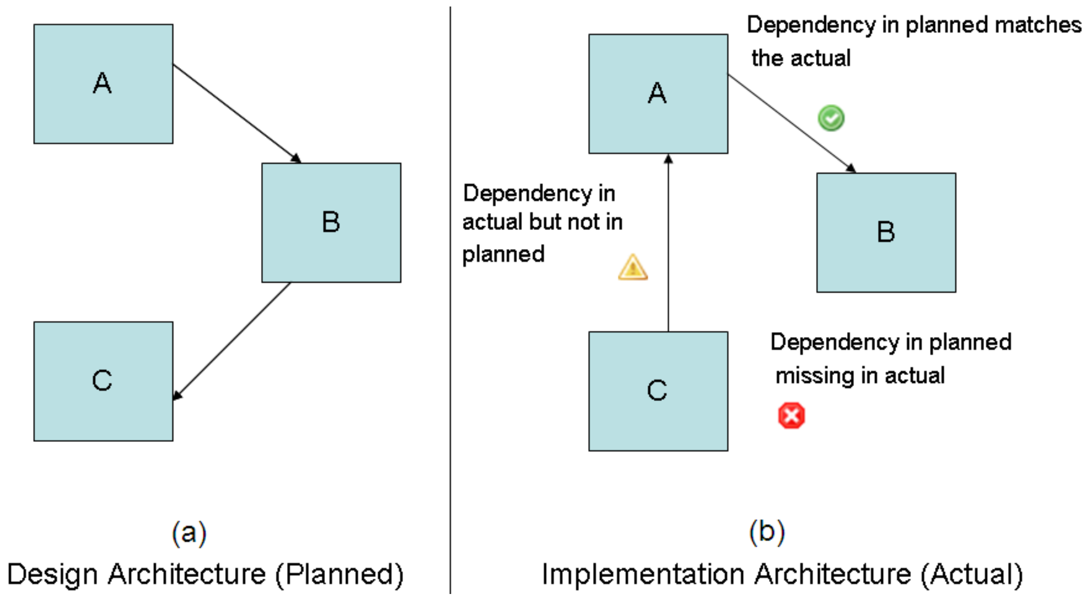
A
B
C
(a)
Design Architecture (Planned)
Dependency in planned matches the actual
A
B
Dependency in actual but not in planned
C
Dependency in planned missing in actual
(b)
Implementation Architecture (Actual)

Figure 4. Architecture based comparison between design and implementation

using static analysis during code integration can provide an integrated analysis of the entire software system at a holistic level.

Static analysis technology can play a role in a regulatory context as well. In this context regulators can obtain device code and apply this technology to expose errors, without knowing much about the design or code. And, like the modeling technology discussed earlier, manufacturers could get "regulatory credit" for using this technology when presented in an assurance case format. One could further imagine that a verification claim would be strengthened by arguing that both model-based development and static analysis techniques were used in the verification process.

4. Conclusion

In this paper we have presented two complementary software development technologies that can be used to help develop high integrity medical device software: model-based development, which allows the developer to check that the design and implementation adhere to the system (software) requirements and static analysis that helps ensure that the implementation itself is free of errors.

Though these technologies have been used with great success in the aerospace and automotive industries, it should be remembered that the medical device environment has its own idiosyncrasies to consider. This environment is based on the practice of medicine (a rather inexact science) on patients with widely varying physiological conditions and with devices that rely on the notion of "competent human intervention" as a primary means for risk control. In this environment, the consequence of a device malfunction may be death or serious injury.

The technologies discussed provide a glimpse of how the development of high-confidence medical device cyber-physical systems might begin to be realized. An open-systems based research environment seems warranted to facilitate broad involvement in addressing issues underlying the composition and integration of cyber-physical medical device and infrastructure technologies through certifiably dependable tool chains that can represent and resolve cyber-physical properties. At the same time, these tool chains need to explicitly support implementation assurance claims. A broad national research agenda is warranted that brings academics, manufacturers, and regulators together to refine existing technologies; and through innovation, develop new technologies such that future implementations can be established as certifiably dependable.

References

[1] C. Ackermann, A. Ray, R. Cleaveland, J. Heit, C. Shelton, C. Martin. Model-Based Design Verification. A Monitor Based Approach. Society of Automotive Engineers World Congress 2008

[2] M. Ben-Ari, A. Pnueli and Z. Manna. The temporal logic of branching time. Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 1981

[3] P. Cousot and R. Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. POPL 1977

[4] E. Clarke, O.Grumberg, D.Pereld. Model Checking. MIT Press. 2000

[5] General Principles of Software Validation; Final Guidance for Industry and FDA Staff. January 11, 2002

[6] P. Godefroid. Model checking for programming languages using Verisoft. Proceedings of the 24th ACM SIGPLANSIGACT symposium on Principles of programming languages, ACM Press, 1997

[7] H. Hampapuram, Y. Yang, and M. Das. Symbolic path simulation in path-sensitive dataflow analysis. In SIGSOFT Software Engineering Notes, Jan 2006

[8] High-Confidence Medical Devices: Cyber-Physical Systems for 21st Century Health Care A Research and Development Needs Report, Prepared by the High Confidence Software and Systems Coordinating Group of the Networking and Information Technology Research and Development Program, February 2009

[9] R. Jetley and P. L. Jones. Safety Requirements based Analysis of Infusion Pump Software, Proceedings of the IEEE Real Time Systems Symposium, Tuscon, December 2007

[10] D. Jackson, M. Thomas, and L. I. Millet editors. Software for Dependable Systems: Sufficient Evidence? Committee on Certifiably Dependable Software Systems, National Research Council, National Academies Press, 2007

[11] J. Knodel, D. Muthig, M. Naab, M. Lindvall. Static Evaluation of Software Architectures. 10th European Conference on Software Maintenance and Reengineering 2006

[12] A. Ray, R. Cleaveland, S. Jiang, T. Fuhrman. Model-Based Verification and Validation of Distributed Controller Architectures. Society of Automotive Engineers Convergence 2006