Predictable Assembly

The crystal ball to software!

Magnus Larsson, Anders Wall, Kurt Wallnau

Anybody who works with computers knows that software sometimes doesn't behave as expected. On an office PC, some way can usually be found of working around this – the bug is more a nuisance than a real obstacle. In industrial applications, however, it is more serious. If a machine malfunctions, this not only leads to lost production, but safety and quality issues must be addressed. The sheer complexity of software means that errors often slip unnoticed through test phases – traditional testing methods are no longer in a position to evaluate all situations that can occur. Mathematically based approaches to development and verification greatly reduce risks and contribute to quality management. Carnegie-Mellon University and ABB are jointly developing techniques for designing high-quality software.

ABB relies on software to deliver innovative solutions to its customers. This software often executes in environments that have strict timing, safety, reliability, and security requirements. Software malfunctions in these applications are expensive, and possibly catastrophic. Even so, the steep cost of developing high-reliability software poses a challenge for the entire software industry. The scale of today's systems, not to mention those of tomorrow, exposes the fundamental inadequacies of relying on testing to achieve high assurance. ABB and Carnegie Mellon's Software Engineering Institute have developed an approach to ensure that the critical runtime behavior of systems is predictable by construction. This will reduce testing costs and speed the introduction of new high-assurance software into the market.

It has often been said that the three fundamental principles of programming are modularity, modularity, and modularity. Soon it will also be said that the three fundamental principles of software engineering are constrain, constrain, and constrain.

Constraints lie at the heart of all engineering disciplines. An engineering problem may present unique challenges, but the skilled engineer knows how to coerce it into a form that can be solved with proven and well-defined techniques. These techniques impose constraints on both the problem being solved and on the form that solutions can take. The loss of freedom implied by these constraints is more than compensated by making it possible to predictably and routinely solve entire classes of problems.

Software engineering is concerned less with programs per se than with large scale networks of interacting programs. At this scale engineering challenges emerge that go well beyond functional correctness (the purview of programming), and encompass equally crucial non-functional qualities (sometimes called "quality attributes") such as security, performance, availability, fault tolerance, and so forth. A pivotal challenge for software engineering research is to provide techniques to routinely construct systems that have predictable non-functional quality. It follows that these techniques will impose constraints on how future software systems will be constructed.

Mathematically based approaches to software development and verification greatly reduce risks and contribute to quality management.

This article shows how "smart constraints" can be introduced into software development practice so that software systems routinely exhibit predictable quality. Smart constraints can be embedded in software infrastructure so that systems are predictable by construction; the days of testing quality into software may finally be numbered. Moreover, predictability by construction can be used to impose objective and measurable quality standards on third-party software; such standards have strong predictive utility.

Predictable by construction

The approach presented in this article is governed by two premises:

1) Smart constraints lead to systems with predictable runtime qualities.
2) Component technology packages constraints to make software predictable by construction.

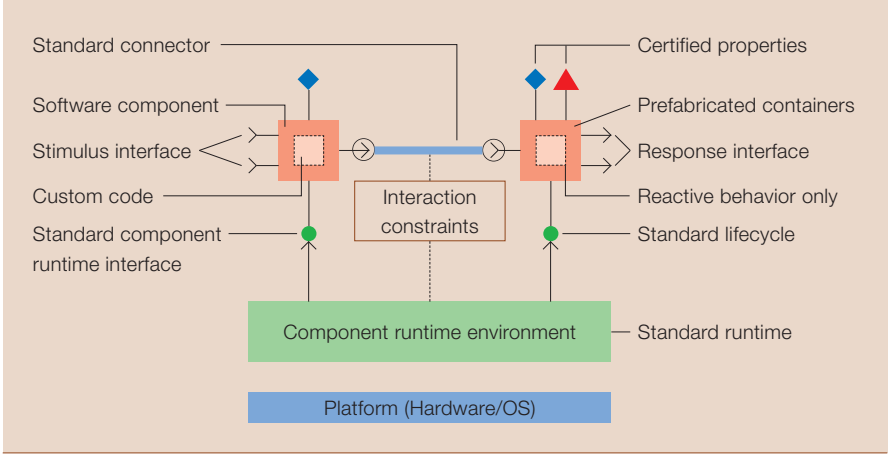
To make sense of these premises, some ideas are introduced: a runtime quality must be defined in terms of observations that can be made on execution traces. A runtime quality is *predictable* if and only if there is a theory (rule) predicting future observations. The crucial point here is that quality is defined relative to a predictive theory, and that this theory must yield confidence about its predictions.

This is not a new idea in science or in software. The timing behavior of a software system may be predictable using generalized rate monotonic scheduling theory or real-time queuing theory. Both theories (generally all theories) make assumptions about the systems that are their subjects, and any system that satisfies these assumptions is predictable in these theories. Smart constraints ensure that these assumptions are satisfied, ie, they are smart because they are informed by predictive theories.

It is one thing to define a smart constraint, but another to guarantee the constraint is satisfied. One recurring component-technology idiom that is particularly effective in packaging smart constraints is depicted in 1.

In this idiom, custom software is deployed into prefabricated containers [1]. A container restricts the visibility of custom code to its external environment, and restricts visibility to the custom code from the environment. Different types of containers can play different roles in a global (architecture-defined) coordination scheme. A software component in this idiom is a container combined with custom code. Components are strictly reactive: they react only to stimuli received through the container interface, and respond only through the container interface. A component runtime environment provides coordination mechanisms (or "connectors") and implements other policies for managing resources shared by components.

1 The container idiom.



The point here is that the user should not get fixed on a particular component technology: many implementations of the idiom are possible and simple implementations can often be realized. What matters is that container types, connector types, runtime environment, and an ability to place constraints on allowable patterns of component interaction can all be used to encode, or package, smart constraints. Moreover, the small number and uniformity of the abstractions in this idiom considerably simplify the task of automating substantial portions of the construction and prediction process – to yield predictability by construction.

What matters is that container types, connector types, runtime environment, and an ability to place constraints on allowable patterns of component interaction can all be used to encode, or package, smart constraints.

The idea is simple, and best understood by analogy. A Java or C# compiler checks that programs are well-formed. One check of well-formedness is that a program satisfies the type theory of the programming language. If this constraint is satisfied, then the compiler guarantees certain properties of program execution (technically, safety properties). Here the same idea is applied, but at the level of assemblies of components instead of at program level. In place of type theories we have behavior theories for non-functional runtime qualities.

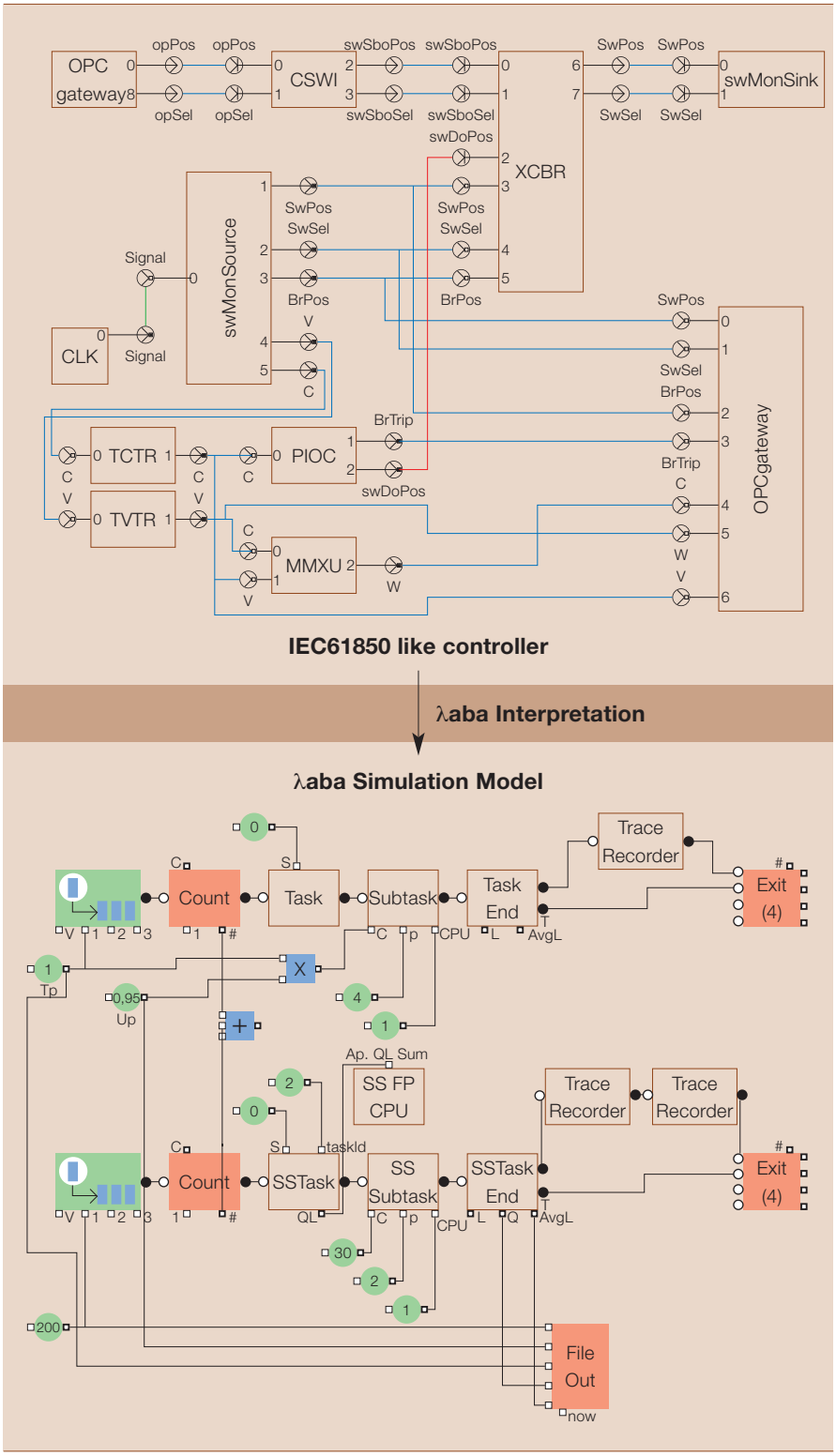
In place of specifications in a programming language, specifications in an architecture description language (CCL [4]) are used. In effect, CCL formalizes the container idiom, and makes possible automated prediction and code generation. The result is predictability by construction. If specifications in CCL are well formed according to the container idiom, and satisfy additional reasoning-framework specific constraints, the systems they specify will be predictable by construction. The ultimate expression of predictability by construction is to build only systems whose behaviors can be predicted.

Other ABB projects have already exploited the affinity of component technology with predictable non-functional behavior, for example PECOS [13]. However, no previous work has generalized these ideas to multiple non-functional attributes, or emphasized the role of validation and certification to the extent done in the work reported here.

Certifiable quality

Analytic theories reveal which properties of the software must be known if its effects are to be predictable. A component technology imposes a standard packaging of software; this

2 Interpretation



includes how components are specified and what details about a component implementation must be exposed by component suppliers. Taken together, these provide a practical basis for establishing objective quality standards for third-party software.

To illustrate, the case is considered where the timing behavior of component assemblies using Lehoczky's real-time queuing theory [2] is predicted. Among other things, this theory assumes:

- A scheduling discipline such as "earliest deadline first" (EDF).
- Identification of schedulable entities, for example threads.
- Second moment of expected service time of each schedulable entity.

The first property is satisfied by the component runtime environment, the second by containers. The third, however, must be satisfied by the component supplier. A few points are worth noting from this illustration. First, what must be satisfied is an accurate measurement of the second moment of service time; this corresponds to the "certified properties" in 1. While it might be desirable to impose requirements on the values these measurements may take, this is considered a separate issue. Second, the performance theory is required to give a precise definition of the measure; it may also provide strong guidance on the measurement process itself.

Of course, the quest for trusted software does not end with certifiable quality. For this wider goal, a more comprehensive approach is required – one that requires improvements in software technology and process – and in social processes as well [10] [12]. Still, the approach presented in this article is a first step towards establishing quality standards that are both objective and predictive. In particular, it is worth observing the described approach deals directly with software artifacts rather than indirectly, for example through measures of the process maturity of developer organizations.

Predictable assembly is not a universal solution to software quality. It must fit within an overall quality strategy based on mature software development processes, well-trained and motivated software developers, and well-documented architectural standards.

ABB application

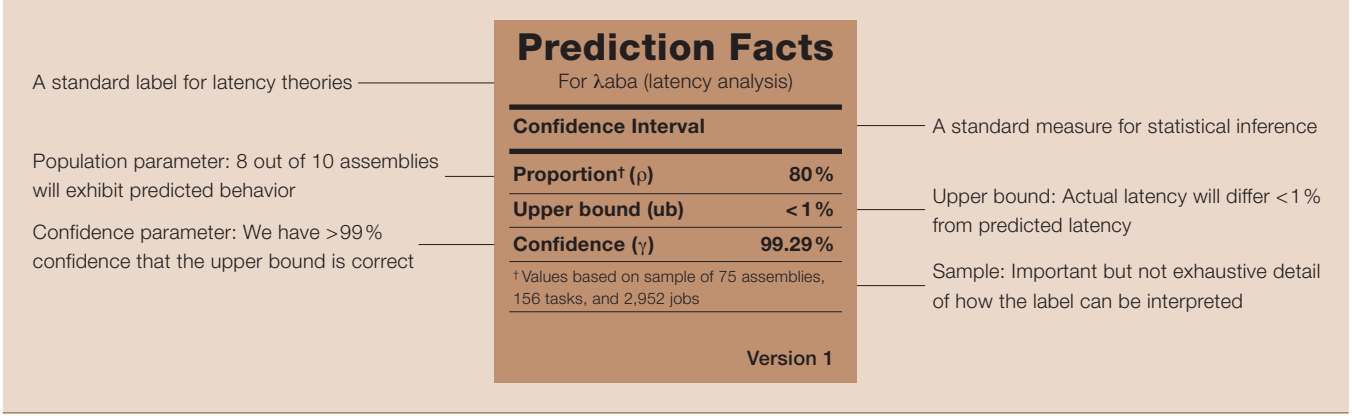
ABB and the Software Engineering Institute (SEI) first tested the feasibility of predictability by construction in the domain of substation automation [3]. This work produced three main results.

1) It was demonstrated how the IEC61850, a standard in the power domain, can be mapped to a software component technology. IEC61850-compliant components and assemblies were specified in CCL. In itself this was a minor result, but it is one that provided a basis for a more direct connection between the standard and software systems that implement it.
2) The λaba reasoning framework was developed ("λ" for latency, "aba" for "average-case, with blocking and asynchronous interactions") for predicting the average-case latency of IEC61850 protection and control assemblies. λaba uses smart constraints to ensure that a number of key assumptions of generalized rate monotonic analysis [5] are satisfied.

   The mechanism of predictability by construction is shown in 2. An IEC61850 assembly is depicted in the graphical notation of CCL (upper half of 2). If the assembly is well-formed to λaba, an analyzable "performance view" is constructed (lower half of 2). Note that CCL is based in abstractions that are familiar to engineers using IEC61131 function blocks or UML. Indeed, CCL can be replaced by any number of design notations.
3) A technique was demonstrated for the rigorous empirical validation of the predictive strength of λaba, or of any other reasoning framework purporting to predict timing behavior [6]. The technique uses constrained random assembly generation to construct representative samples in an application domain, and from these build statistical confidence (or tolerance) intervals that are useful for inferring the accuracy of future predictions. In effect, this statistical label serves to certify the reasoning framework itself. 3 and [7] discuss how statistical labels are interpreted.

ABB and the SEI took what was learned from this initial experiment to the domain of industrial robot control.

3 Certified reasoning framework.



There are two significant results to report from this work.

The first deals with the question: Can a hard real-time periodic control system be safely extended by third party software with stochastic execution behavior, while also guaranteeing best service to the extension? In short, can a hard real-time control system be "open" while still providing firm guarantees on time? To answer this question the λss reasoning framework was developed ("λ" for latency, "ss" for "sporadic server"). λss uses sporadic server containers as its central smart constraint [8]. The sporadic server container protects periodic tasks from stochastic bursts.

The reasoning framework allows all periodic behavior to be effectively "collapsed" into one net periodic effect, and then uses this net effect in a family of queuing equations to establish bounds on average service time of plug-ins 4. The bottom line is that plug-ins are guaranteed to have bounded and predictable invasiveness on periodic timing behavior, are given guaranteed access to the processor, and exhibit predictable latency.

The second result demonstrated how far software model checking (see glossary on Page 54) has progressed in recent years, and its potential to go farther still if it is effectively combined with component-based development. Model checking is particularly effective for verifying the correct behavior of reactive, concurrent software – the kind of software that is prevalent in industrial automation. The IPC (inter-process communication) code of a robot controller was subjected to a commercially-available model checker. The properties checked are typical of IPC code:

- Whenever a message is sent to X, X receives that message (barring timeouts).
- Whenever a message is sent to X, Y never receives that message.
- Whenever a sender receives an answer, it is the answer to the most recently sent message.
- A sender is never blocked while trying to write to a message queue that is not full.
- Messages (or answers) are never written to a slot that has disconnected.

The verification uncovered a violation (or "counter-example" in model-checking jargon) of the third property. Product engineers confirmed that the counter-example was indeed a problem, but the problem had been diagnosed and repaired in a subsequent release of the code. What is remarkable, however, is that the problem, though suspected, had remained hidden for several years! In retrospect, this is not surprising, since concurrency errors in software are notoriously difficult to reproduce. This is also reflected in some statistics about this particular model checking exercise.

- The state space of only the relevant parts of the code was $\cong 10^{1932}$ states – after abstraction!
- The error state arose on the 179th interaction between communicating threads.

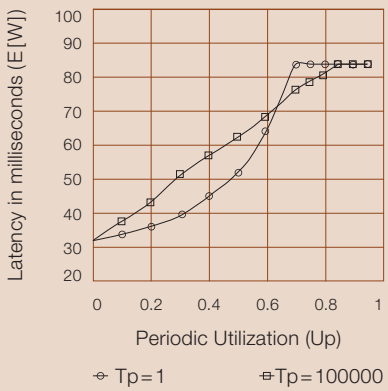
These figures show that it is highly unlikely a conventional testing approach would have revealed the error.

The single most expensive part of the technical infrastructure for predictability by construction is the validated reasoning framework.

Since this work was performed, the ComFoRT model checking reasoning framework has been further developed. ComFoRT exploits software component technology, and implements a variety of complementary state-of-the-art complexity reduction techniques [9] [11].

The biggest challenge faced in the IPC verification task – and faced by anyone using current generation model checkers – is to produce models that are valid abstractions of the system under scrutiny. One key feature of ComFoRT is the fully automatic generation of sound abstract models from CCL design specifications 5. This feature of ComFoRT is one of several that aims to make model checking significantly easier to use for programmers and software engineers.

4 Predictable plug-in latency.

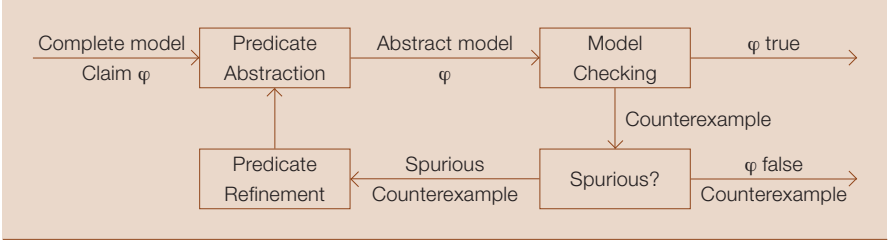


Findings

Predictable assembly is not a universal solution to software quality. It must fit within an overall quality strategy based on mature software development processes, well-trained and motivated software developers, well-documented architectural standards, and a culture of excellence. With this in mind, some conclusions can still be drawn about the viability of this approach to ABB businesses.

Foremost among the findings: the premise of this work – using smart constraints for predictability and packaging smart constraints in component technology – is valid and useful. The premise does not depend on a particular choice of component, specification, or analysis technology. While not all technology choices will be equally effective, this work highlights what is most important in each of these if they are to serve the goal of predictability by construction.

5 ComFoRT's automated model construction using counter-example guided abstraction refinement.



Before the technology of predictability by construction can be adopted, however, two conditions must be satisfied:

1) The organization must have control over software architecture design decisions. There are tradeoffs between predictability and design freedom; an organization that cannot impose or relax constraints on design will be limited in making these essential tradeoff decisions. Legacy systems also pose challenges restricting such tradeoffs.

2) Predictable quality must have real value. This may sound vacuous, but it is not. For example, some management information systems have "loose" design tolerance for behavior such as latency. Highly accurate timing predictions may not be of great value here (although security might be). An embedded controller, however, may have "tight" design tolerance for timing behavior.

The single most expensive part of the technical infrastructure for predictability by construction is the validated reasoning framework. Elements such as component and specification technologies are crucial but not as technically challenging. The lesson here is that reasoning frameworks are more likely to be core ABB assets than those other elements, and may play an important role in establishing "smart" corporate design standards.

Current work

Currently work is concentrating on an approach for a predictable soft protection and control system (Soft P&C), in the substation automation domain. A Soft P&C system is essentially a complete substation automation system that is implemented on a centralized, more or less standard, computer with no proprietary hardware. Technologically, such a system can be built today. However, in order to convince customers that such a solution still fulfills the critical quality requirements such as performance and reliability, evidence must be produced. The concepts presented in this article can provide this.

ABB – CMU collaboration

ABB Corporate Research and the Software Engineering Institute (SEI) began work together on predictable assembly from certifiable components in 2001. This collaboration has been mutually beneficial. CMU/SEI gets an industrial context for developing their technology, which can later be adapted to a larger class of analogous problems. ABB has the benefit of being early in applying new ground breaking technology for predicting quality attributes in complex software systems.

Dr. Magnus Larsson
Dr. Anders Wall
ABB AB, Corporate Research
Västerås, Sweden
magnus.larsson@se.abb.com

Kurt Wallnau
Software Engineering Institute
Carnegie Mellon University
kcw@sei.cmu.edu

Glossary:

Execution trace: The sequence of changes occurring to a system or component when observed over time.

Model checker: A software tool that performs model checking.

Model checking: An approach whereby all possible execution traces of a system (hardware or software) are looked at exhaustively to verify that given properties hold – or when execution traces are not looked at exhaustively, this is because analytical abstraction reveals they yield no new information (for example due to symmetries, relevance or repetition).

State space: A state is a value that the set of variables of a system can assume during execution. For example, a simple switch can assume either of two states: ON or OFF. The set of all states a system can assume is called its state space. The size of this space increases with the complexity of the system, and more especially, exponentially with the number of variables of the system. When a system has a large number of variables, the number of states explodes to such a degree that a model checker is no longer able to analyze them in useful time, or to store them physically. This challenge is called state explosion. Model checkers use powerful abstraction mechanisms to reduce state explosion. Modern model checkers can check very large systems very quickly.

UML: Unified modeling language – a widely used formalism for software description and specification.

References:

[1] **Ward-Dutton, N**., "Containers: A sign components are growing up." Application Development Trends, pp 41–46, Jan 2000.

[2] **Lehoczky, J. P.,** "Real-time queuing theory," in Proceedings of the IEEE Real-Time Systems Symposium, 186–195, IEEE, New York, 1996.

[3] **Hissam et. al.,** *Predictable Assembly of Substation Automation Systems: An Experiment Report, Second Edition,* Technical Report CMU/SEI-2002-TR-031, www.sei.cmu.edu/publications/documents/02.reports/02tr031.html

[4] **Wallnau, K., Ivers, J.,** *Snapshot of CCL: A Language for Predictable Assembly*, Technical Note CMU/SEI-2003-TN-025, www.sei.cmu.edu/publications/documents/03.reports/03tn025.html

[5] **Klein et. al.,** *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems,* Kluwer Academic Publishers, 1993.

[6] **Larsson M.,** Predicting Quality Attributes in Component-based Software Systems, Ph. D thesis Mälardalen University Press, ISBN 91-88834-33-6

[7] **Moreno, G., Hissam, S. Wallnau, K.** "Statistical Models for Empirical Component Properties and Assembly-Level Property Predictions: Toward Standard Labeling," in the 5th ICSE Workshop on Component-Based Software Engineering, May 2002, www.preview.sei.cmu.edu/pacc/CBSE5/Moreno-cbse5-final.pdf

[8] **Hissam et. al.,** *Performance Property Theories for Predictable Assembly from Certifiable Components,* Technical Report CMU/SEI-2004-TR-017, www.sei.cmu.edu/publications/documents/04.reports/04tr017.html

[9] **Ivers, J., Sharygina, N.,** *Overview of ComFoRT: A Model Checking Reasoning Framework,* Technical Note CMU/SEI-2004-TN-018, www.sei.cmu.edu/publications/documents/04.reports/04tn018.html

[10] **Meyer, B.** "The Grand Challenge of Trusted Components," 660–667. Proceedings of the 25th International Conference on Software Engineering (ICSE). Portland, Oregon, May 3-10, 2003. Los Alamitos, CA: IEEE Computer Press, 2003.

[11] **Clarke, E., Kroening, D., Sharygina, N., Yorav, K.,** "Predicate Abstraction of ANSI-C Programs Using SAT," in Formal Methods in System Design, 25, 105–127, 2004, Kluwer Academic Publishers.

[12] **Wallnau, K.,** *Software Component Certification: 10 Useful Distinctions,* Technical Note CMU/SEI-2004-TN-031, http://www.sei.cmu.edu/publications/documents/04.reports/04tn031.html

[13] **Nierstrasz, O., et al,** "A Component Model for Field Devices" Proceedings First International IFIP/ACM Working Conference on Component Deployment, ACM, Berlin, Germany, June 2002. See also http://www.pecos-project.org/