January 2010 Vol.12 No.4

SoftwareTech
www.softwaretechnews.com NEWS

Model-Driven Development

Better systems through advanced automation

The Data & Analysis Center for Software

http://iac.dtic.mil/dacs
Unclassified and Unlimited Distribution

TECH VIEWS

by Robert Vienneau, Senior Analyst for the DACS STN

Model Driven Development (MDD) is an emerging practice supported by research and commercially-available tool suites. Artifacts in MDD are used to formally describe user behavior, requirements, architecture, and designs. MDD includes the transformation of models of systems into platform-dependent architectures and designs, the automatic generation of source code, and the automatic generation of test data. MDD is one of a number of technologies raising the level of abstraction in software development and maintenance.

MDD is a tool-intensive practice, and various professional groups have developed standards to support tools throughout the tool chains. Particularly notable are standards maintained by the Object Management Group (OMG), such as Model-Driven Architecture (MDA), the Object Constraint Language (OCL), the Unified Modeling Language (UML), and the XML Metadata Interchange (XMI).

Domain-Specific Modeling (DSM) provides another approach to MDD. Juha-Pekka Tolvanen provides an overview of DSM in the first article in this issue. He provides two examples of DSM and Domain-Specific Languages, reports productivity and quality improvements in assorted domains, and discusses approaches for adopting DSM.

In Model-Based Testing (MBT), test suites, rather than source for the application under test, are automatically derived from the models. Three articles in this issue present aspects of MBT. In the first, Bruno Legeard and Mark Utting provide an overview of MBT. They describe the stages of an MBT process, developer roles in the process, an example, success factors, and benefits of MBT.

In the second article on MBT, Jesse Poore briefly describes his research program at the University of Tennessee. This research merges a view of software testing as an application of statistical experimental designs with a treatment of software applications as specified by mathematical functions, as functions are defined in set theory. The group has developed automated tools based on their approach to modeling users and applications, and some these tools have been employed by the research sponsors.

In the third article on MBT, Peter Feiler and Jorgen Hansson provide a perspective from research at the Software Engineering Institute. They present the use of the Architecture Analysis and Design Language (AADL) to model non-functional quality attributes and provide data on the benefits of this approach from a case study.

Finally, Gordon Morrison, in his article, "Understanding Temporal Logic – The Temporal Engineering of Software", describes, with an example, a methodology for producing a Coherent Object System Architecture (COSA). This methodology evolves a Backus-Naur Form (BNF) specification of a system into a table describing a state machine with a certain structure. Mr. Morrison compares and contrasts qualities of the resulting system with a traditional state-machine implementation. The methodology could someday be supported by tools for MDD.

About the Author

Robert L. Vienneau is a senior analyst at the DACS. He has published articles on formal methods, on financial analysis techniques in software cost modeling, and on a real-time Synthetic Aperture Radar (SAR) demonstration of High Performance Computing (HPC). Mr. Vienneau holds a BS in mathematics from Rensselaer Polytechnic Institute and a MS in Software Development and Management from the Rochester Institute of Technology.

Author Contact Information

Email: Robert Vienneau: rob.vienneau@itt.com

Domain-Specific Modeling for Full Code Generation

THE ADVANTAGE OF USING EXTERNAL LANGUAGES IS THAT THEY CAN BETTER GUARANTEE THAT NORMAL DEVELOPERS FOLLOW THE DOMAIN-SPECIFIC CONSTRUCTS AND RULES.

by Dr. Juha-Pekka Tolvanen

Domain-Specific Languages are becoming hard to avoid. There is a natural reason: they are more expressive and therefore tackle complexity better, making software development easier and more convenient. Most importantly, they raise the level of abstraction and, together with domain-specific generators, can automate the creation of production quality code. In this article we introduce Domain-Specific Languages (DSLs), and Domain-Specific Modeling (DSM) in particular, along with industry experiences. We give guidelines for moving successfully from coding to modeling with full code generation.

Today there are two schools of Domain-Specific Languages: those preferring to embed the new language constructs into an existing host language, and those who prefer to keep them in a separate external language. While both have their place, the advantage of using external languages is that they can better guarantee that normal developers follow the domain-specific constructs and rules. This resembles past developments in textual programming languages; inline assembler in C did not become widespread since most people wanted to keep these abstractions separate.

The other division is in representation. Should specifications be expressed in text, diagrams, matrices, tables, or some other form? The right answer obviously depends on the application domain. We should use the representation that feels most natural to solve the problem. For many cases, I favor graphical models because they have several advantages over textual specifications, which are inherently linear and often written to satisfy the compiler rather than support problem solving. Numerous scientific studies have shown that graphical models are easier to read and understand since they can express things like conditions, parallelism, and structures better than text. Graphical models are also especially good for humans since we are good at spotting visual patterns. Remember the well-known saying that a picture is worth a thousand words. Let's next look at two examples of DSM.

Examples from Practice

The examples are selected from practice and from different domains. Figure 1 shows a case from the automotive industry,

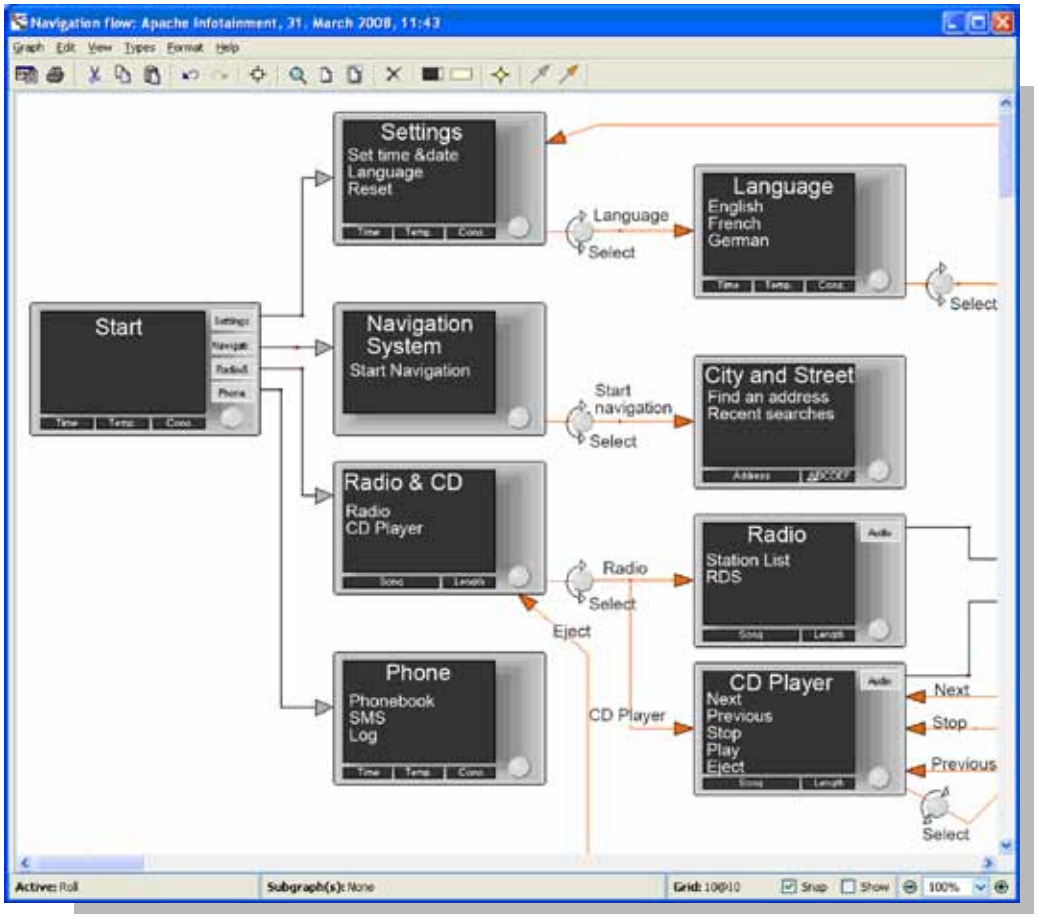


Figure 1: Designing automotive infotainment system with a DSM language

the menu design for an infotainment system. The language used for this model covers the user interaction, navigation and logical structure. Other languages are integrated with this one to cover the detailed display layout and graphics settings for fonts, icons, colors, etc. From these models the full code for controlling the infotainment system can be generated. Here the language is clearly operating at a higher level, above the code, since the developer does not need to know which target code is generated. Several generators can be created, building the same system but for different target platforms. Each generator can take advantage of the features of that particular target platform to best implement the modeled system.

The other example is from a domain where there is no User Interface, embedded device control logic. The model in Figure 2 describes how phone calls are handled by a telephony server. An unanswered call from a certain location is routed to a second number, email or voice mail. Similar telephony services are created with this language by telco service engineers, e.g. for companies or emergency services. The code generator creates the full service specification in the XML format required by the telephony server. The service engineer, however, can focus on call processing and not on the low-level details of the XML. The modeling language's rules for connections between objects prevent the creation of unimplementable or ambiguous specifications.

Industry Experiences

Raising the level of abstraction with DSM as illustrated above always improves productivity. The exact improvement is largely set when the language and generator are defined for a particular case. Panasonic ran a comparison with traditional manual practices by implementing the same features by manually coding them and by using a DSM solution. The productivity with DSM was about 500% higher than manual coding – across the whole lifecycle from design to running production code. Polar Inc. conducted a study in which six engineers implemented features for a heart rate monitoring device using DSM. The engineers estimated the productivity improvement to be 750% compared to manual practices. At Nokia, mobile phone developers experienced a 1000% productivity increase, with development time for a standard feature cut from two weeks to one day. At Lucent, several DSLs were developed and deployed, giving productivity increases between 300% and 1000% depending on the case. Figure 3 shows further examples of productivity increases in various domains.

Productivity is not the only benefit. For safety critical applications, quality matters more. US Air Force studies on military systems have shown that a dedicated specification language and generator resulted in 50% fewer errors than manual coding using components. Both the language and the generator contribute to the improved quality. The language prevents modelers from creating specifications that are illegal, lead to errors, or even cause poor performance. Preventing errors at this early stage is, as always, significantly cheaper than

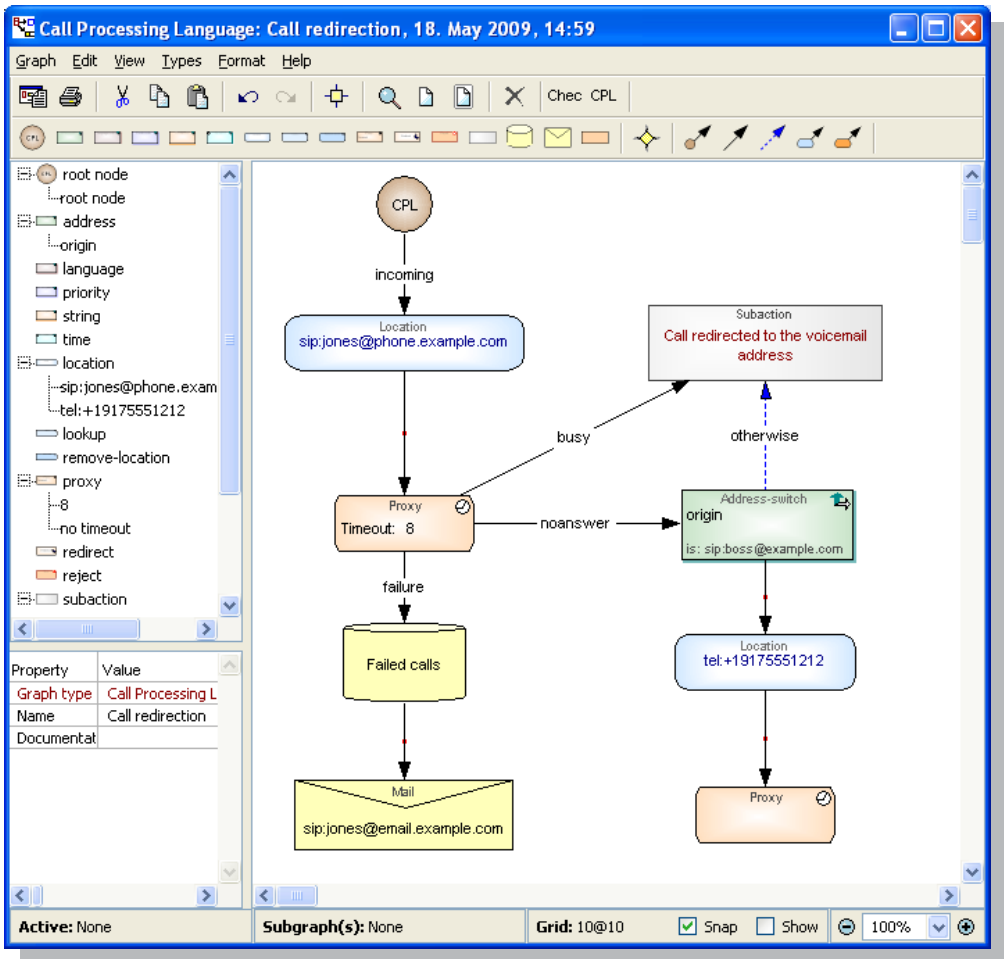


Figure 2: Example design using Call Processing Language

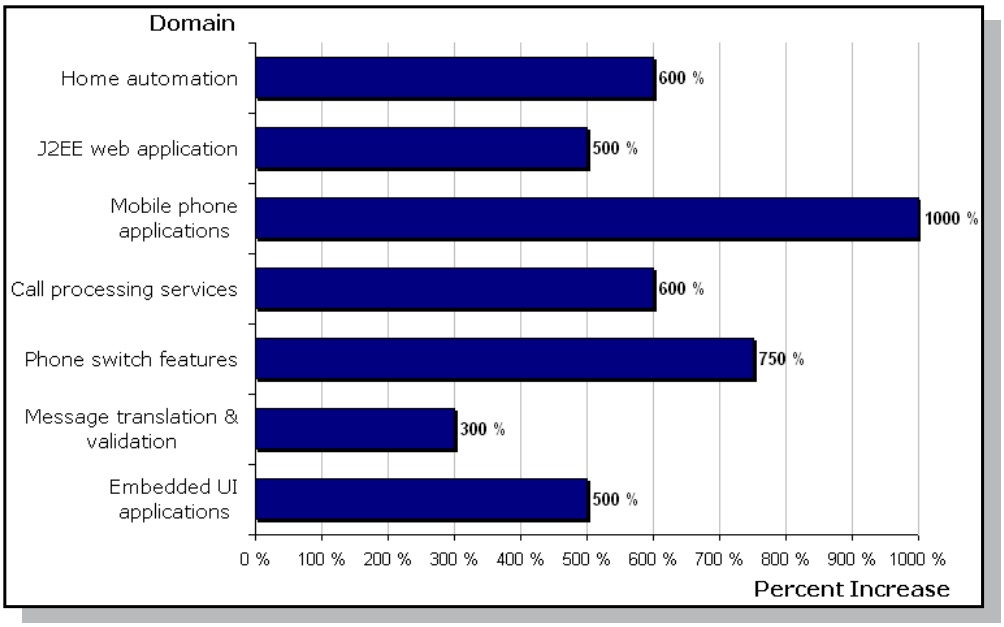


Figure 3: Measured productivity improvements in various domains

catching them later. Automotive software developer Denso found that implementation and testing requirements are both reduced by having the domain rules (AUTOSAR in their case) in the language. With code generators many typical errors — missing references, typos, forgotten initializations, referencing freed memory — simply don't occur anymore. For instance, EADS Secure Networks has detected that there are significantly more errors, and a wider range of errors, with manual coding than with DSM and code generation.

Best practices for moving from coding to modeling and code generation

These improvements in productivity and quality are possible because experienced developers, knowing the domain and how to write good code for it, have defined the DSM solution. It is therefore worth emphasizing that only one or two developers need to master language and code generator development. The other developers simply use the modeling languages these experts have created. This division of labor enables the best developers to package their experience, empowering other members of the development team.

The process of creating a DSM language and generators for a given domain often starts with a feasibility study, which creates a partial implementation over a few days. After this proof-of-concept phase, a pilot project will be launched. This phase usually lasts a few weeks. The language is fleshed out, and then incrementally tested and improved by building a real system with it. Concrete outcomes of the pilot include the full version of the modeling language and its tool support, plus code and documentation generators. Figure 4 illustrates the resource usage for developing modeling languages and generators in some industry examples from our customers; results with other tools vary.

Defining a DSM language is a new experience for most people, and they may thus see it as an obstacle. In reality, all developers are already using the concepts and terms that are specific for the domain and company they are working for. These are the terms found in requirements documents and initial whiteboard sketches, that developers have had to learn how to translate into UML or code. DSLs thus do not contain newly invented concepts, but rather offer the possibility to build systems directly using terms that are already known,

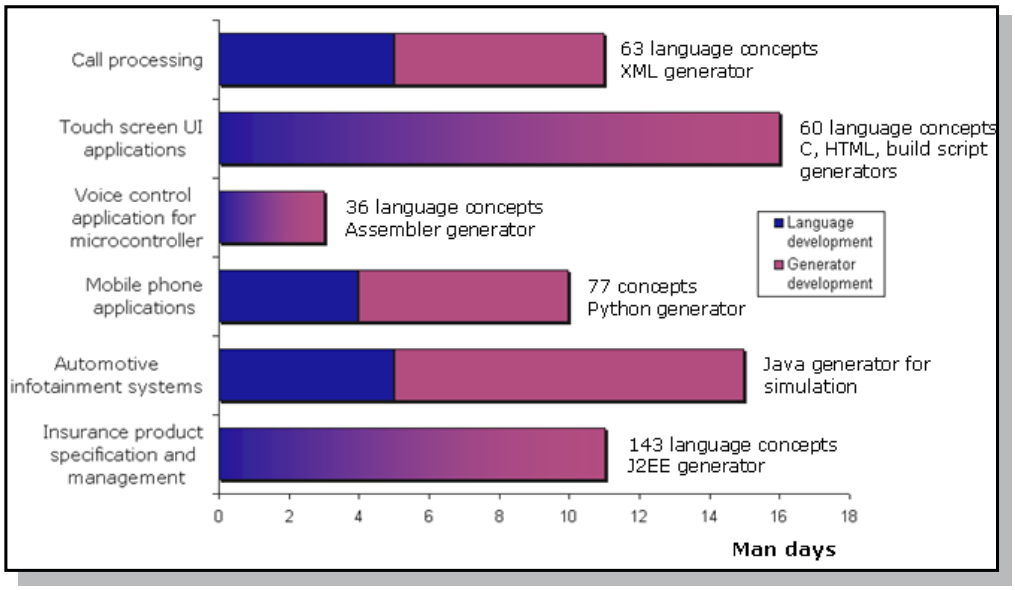


Figure 4: Use of resources in DSM creation

tried and tested in that domain.

The best approach for building DSLs is incremental. Build a little of the language, model a little, make some changes to the language, model some more, etc. This iterative process minimizes risks, allows early testing, and smoothes the path of organizational change: moving from coding to modeling. Iteration does not stop with initial language deployment. Once in use, the language will be evolved and extended later as the domain evolves and gaps are filled. Good tools can greatly reduce the time and effort required to build languages, keeping the language definition process agile by automatically updating earlier models as the language changes.

The other recipe for success is focusing only on a narrow area of interest — as in the cases above. The narrower the domain, the easier it becomes to build a good, high-level language and make generators produce first class code. This is the reason that most successful DSM solutions are defined inside a single company: you only have to solve your own problems, not the whole world's. Keeping language development in house gives full control of the automation to the company creating the language, rather than to the tool vendor or other forces outside the company.

The bottom line for DSM languages is how much they raise the level of abstraction up from coding to the problem domain. Languages that apply programming concepts as modeling constructs often end up visualizing source code, and thus fail to provide any real improvements in abstraction — or productivity. Basing the language on problem domain concepts raises the level of abstraction more reliably, as the cases above illustrate.

Concluding remarks

The consistency with which DSM has improved productivity and quality has led companies to increasingly invest in creating DSLs. These languages will be production quality, but not productized. Their benefit is in their narrowness, unlike general purpose modeling tools that must appeal to the widest possible market. The task of language creation will thus become increasingly common, as did library and framework creation before it.

Increasing system complexity and the move from hardware to software in new domains also drive our industry towards languages on a higher level of abstraction. Rising popularity comes from improved tooling. DSM pioneers had to create the modeling tools for their languages from scratch, making DSM practical only for large projects. Today's mature language workbenches make language creation cost effective, providing quality tools for modelers with the minimum effort.

References

EADS Case Study, MetaCase, 2007; www.metacase.com/papers/MetaEdit_in_EADS.pdf.

Kelly, S., Tolvanen, J-P., Domain-Specific Modeling: Enabling Full Code Generation, Wiley-IEEE Society Press, 2008

Nokia Mobile Phones Case Study, MetaCase, 2007; www.metacase.com/papers/MetaEdit_in_Nokia.pdf.

Safa, L., The Making of User-Interface Designer, A Proprietary DSM Tool, Proc. 7th OOPSLA Workshop Domain-Specific Modeling, Montreal, 2007; www.dsmforum.org/events/DSM07/papers/safa.pdf.

About the Author

Dr. Juha-Pekka Tolvanen is CEO at MetaCase and has over 15 years' experience on the creation of modeling languages and code generators. He has recently co-authored a book on Domain-Specific Modeling (Wiley, 2008).

Author Contact Information

Email: Dr. Juha-Pekka Tolvanen: jpt@metacase.com

Model-Based Testing – Next Generation Functional Software Testing

MBT RENEWS THE WHOLE PROCESS OF FUNCTIONAL SOFTWARE TESTING: FROM BUSINESS REQUIREMENTS TO THE TEST REPOSITORY, WITH MANUAL OR AUTOMATED TEST EXECUTION.

by Bruno Legeard and Mark Utting

Model-Based Testing (MBT) is an increasingly widely-used technique for automating the generation and execution of tests. There are several reasons for the growing interest in MBT:

- The MBT approach and the associated commercial and open source tools are now mature enough to be applied in many application areas, and empirical evidence is showing that it can give a good Return On Investment (ROI);
- The complexity of software applications continues to increase, and the user's aversion to software defects is greater than ever, so our functional testing has to become more and more effective at detecting bugs;
- The cost and time of testing is already a major portion of many projects (sometimes exceeding the costs of development), so there is a strong push to investigate methods like MBT that can decrease the overall cost of test by designing tests automatically, as well as executing them automatically.

MBT renews the whole process of functional software testing: from business requirements to the test repository, with manual or automated test execution. It supports the phases of designing and generating tests, documenting the test repository, producing and maintaining the bi-directional traceability matrix between tests and requirements, and accelerating test automation.

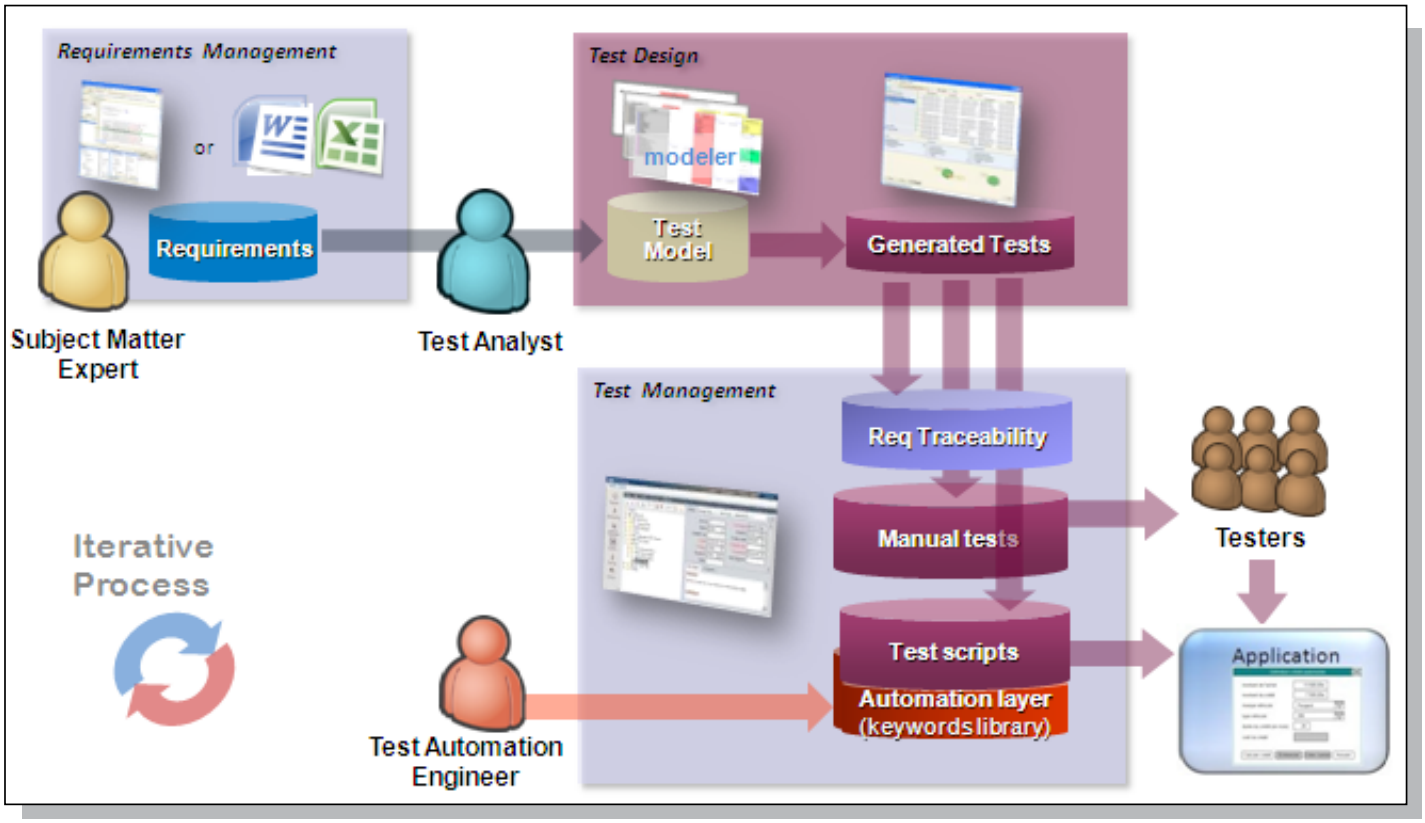


Figure 1: The MBT Process.

This paper addresses these points by giving a realistic overview of model-based testing and its expected benefits. We discuss **what** model-based testing is, **how** you have to organize your process and your team to use MBT, and give some examples of MBT approaches and tools.

What is MBT?

Model-based testing refers to the processes and techniques for the automatic derivation of abstract test cases from abstract formal models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases.

Therefore, the key points of model-based testing are the modeling principles for test generation, the test generation strategies and techniques, and the concretization of abstract tests into executable tests. A typical deployment of MBT in industry goes through four stages (Figure 1).

1. **Design a Test Model.** The model, generally called the test model, represents the expected behavior of the System Under Test (SUT). Standard modeling languages, such as Unified Modeling Language (UML) or Simulink, are used to formalize the control points and observation points of the system, the expected dynamic behavior of the system, the business entities associated with the test, and some data for the initial test configuration. Model elements such as transitions or decisions are linked to the requirements, to ensure bi-directional traceability between the requirements and the model, and later to the generated test cases. Models must be precise and complete enough to allow automated derivation of tests from them.

2. **Select some Test Generation Criteria.** Usually an infinite number of possible tests could be generated from a model. The test analyst chooses Test Generation Criteria to select the highest priority tests or to ensure good coverage of the system behaviors. One common kind of test generation criteria is based on structural model coverage, using well known test design strategies such as equivalence partitioning, cause-effect testing, pair-wise testing, process cycle coverage, or boundary value analysis[1]. Another useful kind of test generation criteria ensures that the generated test cases cover all the requirements, perhaps with more tests for requirements that have a higher level of risk. In this way, model-based testing can be used to implement a requirement and risk-based testing approach. For example, for a non-critical application, the test analyst may choose to generate just one test for each of the nominal behaviors in the model and each of the main error cases; but for one of the more critical requirements, they could apply more demanding coverage criteria such as all loop-free paths, to ensure that the businesses processes associated with that part of the test model are thoroughly tested.

3. **Generate the tests.** This is an automated process that generates the required number of high-level (abstract) test cases from the test model. Each generated abstract test case is typically a sequence of high-level SUT actions, with input parameters and expected output values for each action. These generated test sequences are similar to the high-level test sequences that are designed manually in keyword-based or action-word testing [2]. They are easily understood by humans, but too high-level to be directly executed on the SUT. A further concretization phase automatically translates each abstract test case into a concrete (executable) test [3], using user-defined mappings from abstract data values to concrete SUT values, and mappings from abstract operations into SUT Graphical User Interface (GUI) actions or Application Programming Interface (API) calls. For example, if the test execution is via the GUI of the SUT, then the action words are linked to the graphical object map, using a test robot, such as HP QuickTest Pro, IBM Rational Functional Tester or Selenium. If the test execution of the SUT is API-based, then each action word must be implemented on this API via some glue code. The expected results of each abstract test case are translated into oracle code that will check the SUT outputs and decide on a test pass/fail verdict. Tests generated from the test model may be structured into multiple test suites and published into standard test management tools, such as HP Test Director, IBM Rational Quality Manager, or the open-source TestLink tool. Maintenance of the test repository is done by updating the test model, then automatically regenerating the test suites;

4. **Execute the Tests.** Generated concrete tests are typically executed within a standard automated test execution environment, such as HP QuickTest Pro or IBM Rational Functional Tester. Alternatively, it is possible to execute tests manually – i.e. a tester runs each generated test on the SUT, records the test execution results, and compares them against the generated expected outputs. Either way, when the tests are executed on the SUT, we find that some tests pass and some tests fail. The failing tests indicate a discrepancy between the SUT and model, which needs to be investigated to decide whether the failure is caused by a bug in the SUT, or by an error in the model or the requirements. Empirically, MBT finds SUT errors, but is also highly effective at exposing requirements errors [4].

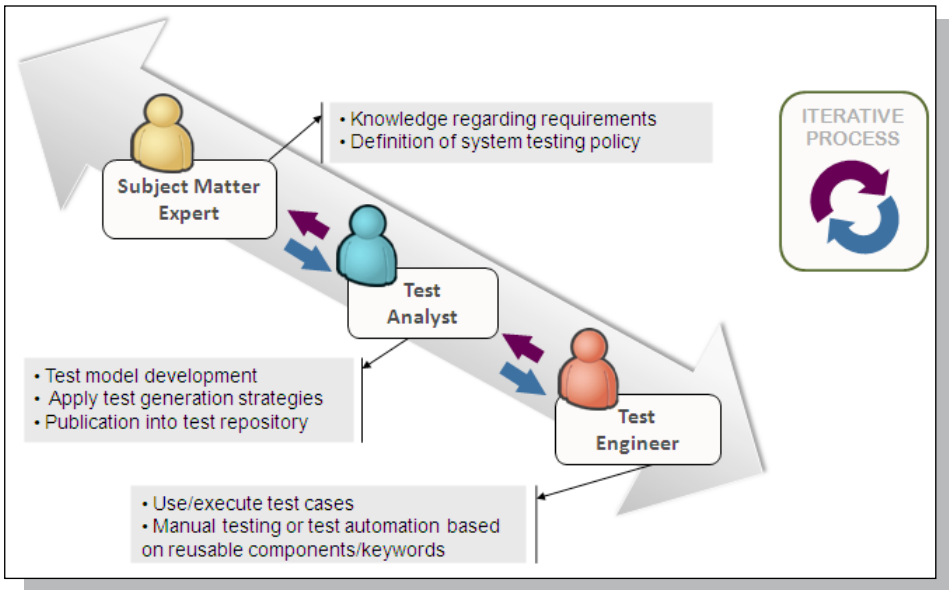


Figure 2: Main Roles in the MBT.

Roles in the MBT process

The MBT process involves three main kinds of people (Figure 2 on the following page):

1. The **Subject Matter Expert** is the reference person for the SUT requirements and business needs. He/she dialogues with the test analyst to clarify the specifications and testing needs.
2. The **Test Analyst** interacts with the customers and subject matter experts regarding the requirements to be covered, and then develops the test model. He/she then uses the automated test generation tool to generate tests and produce a repository of test suites that will satisfy the project test objectives.
3. The **Test Engineer** is responsible for connecting the generated tests to the system under test so that the tests can be executed automatically. The input for the test engineer is the test repository generated automatically by the Test Analyst from the test model.

The test analyst is responsible for the quality of the test repository in terms of coverage of the requirements and fault detection capability, so the quality of his/her interaction with the subject matter expert is crucial. In the other direction, the test analyst interacts with the test automation engineer to facilitate test automation (implementation of key-words). This interaction process is highly iterative.

The Scope of Model-Based Testing

MBT is mainly used for functional black-box testing. This is a kind of back-to-back testing approach, where the SUT is tested against the test model. Differences in behavior are reported as test failures. The model formalizes the functional requirements, representing the expected behavior at a given level of abstraction.

Models can also be used for encoding non-functional requirements such as performance or ergonomics, but this is currently a subject of research in the MBT area rather than common practice. Security requirements can often be tested using standard MBT techniques for functional behavior.

Regarding the testing level, the current mainstream focus of MBT practice is system testing and acceptance testing, rather than unit or module testing. Integration testing is considered at the level of integration of subsystems. In the case of a large system of systems, MBT may address test generation of detailed test suites for each sub-system, and manage end-to-end testing for the whole system.

An Example of MBT

In this section, we present a MBT approach based on a representative commercial tool: ***Microsoft Spec Explorer.*** We illustrate the main characteristics of the approach by giving an example of modeling an application and generating tests from the model.

Microsoft Spec Explorer [5] is a new Microsoft tool that is planned to be released with Visual Studio 2010. It has been used internally within Microsoft for several years, by several hundred testers. It is now becoming a product for external use. To illustrate how it works, we model a simple container object that can have strings added or removed from it. For extreme simplicity, we focus on testing the container with just a single string.

The first modeling step is to decide which actions of the SUT (the container API) we want to model. In Spec Explorer, this is documented in a configuration file in a C-like notation called Cord (Figure 3 on page 12). We model just two actions: *Add1* corresponds to adding the string to the container, and *Remove1* corresponds to removing the string from the container.

To model the behavior of these two actions, we write a

```
config Main
{
// The two implementation actions that will be modeled and tested
action abstract static void StringSetModel.Add1();
action abstract static bool StringSetModel.Remove1();
...
}
```

Figure 3: Configuration file (Config.cord) that defines the actions we want to model and test.

small C# class that defines each of the actions as a method (Figure 4). The class also has a private Boolean flag called *str1*, to remember whether the string is currently in the container or not. The actions simply set or clear this flag. The *Remove1* action also returns a Boolean result that tells the caller whether the string was actually removed from the collection, or whether it was already missing from the collection. Since this model is written in a familiar language (C#), it is not difficult for someone with a programming background to develop quite sophisticated test models.

```
// This C# program models a set that contains a single string.
public static class StringSetModel
{
  private static bool str1;

  [Action("Add1")]
  static void Add1()
  {
    str1 = true; // str1 is now in the set.
  }

  [Action("Remove1/result")]
  static bool Remove1()
  {
    bool wasPresent = str1;
    str1 = false; // str1 is now out of the set.
    return wasPresent;
  }
}
```

Figure 4: Model.cs, which defines a simple model of adding and removing one string.

After designing this test model, which describes the expected behavior of the collection SUT, we can explore the model visually, to see if we've got the model right. Spec Explorer includes an 'Exploration Manager' tool that can explore all the possible actions and internal states of a model, and when we run this on our model we get the diagram shown in Figure 5.

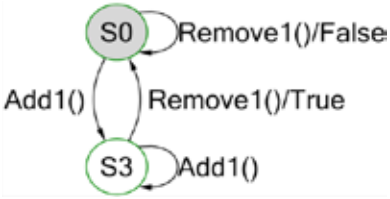


Figure 5: A finite state diagram of all the states and transitions of the model.

This looks good - the grey S0 state is the initial state where the string is not in the collection, which is why the *Remove1* action returns *False*. Spec Explorer has figured out that removing the string from the S0 state has no effect (leaves the state unchanged), and that adding the string twice is the same as adding it once (we stay in state S3), etc. For larger models, the Cord language has several ways of limiting exploration to a subset of the model, so that it is possible to visualize one aspect of its behavior at a time.

The most interesting and important part of using Spec Explorer is generating tests from the model. There are a variety of test generation strategies possible, such as generating one long test that will test all the states and transitions of the model, generating multiple shorter tests, or defining specific scenarios that focus the testing on just one aspect of the model.

Figure 6 (on page 14) shows a *'machine'* declaration in the Config.cord file that says we want to generate several short tests. The left side of Figure 6 gives a graphical view of the tests generated from that machine. For example, the longer test sequence checks that *Remove1* returns *False* on the empty container, then adds the string and checks that removing the string returns true, but removing it a second time returns false (because it has already been removed). As well as this graphical view of the generated tests, Spec Explorer also generates a test suite that implements the same tests as C# code. This can be connected to a real implementation of the SUT, and then executed by Visual Studio to test that SUT and report any failures.

So with Spec Explorer, the human tester designs a high-level C# class that models the expected behavior of the SUT, and also writes several Cord commands to specify how that model should be explored and what kinds of tests should be generated. Then Spec Explorer automates the detailed test case design, generates executable C# tests, and displays those tests visually.

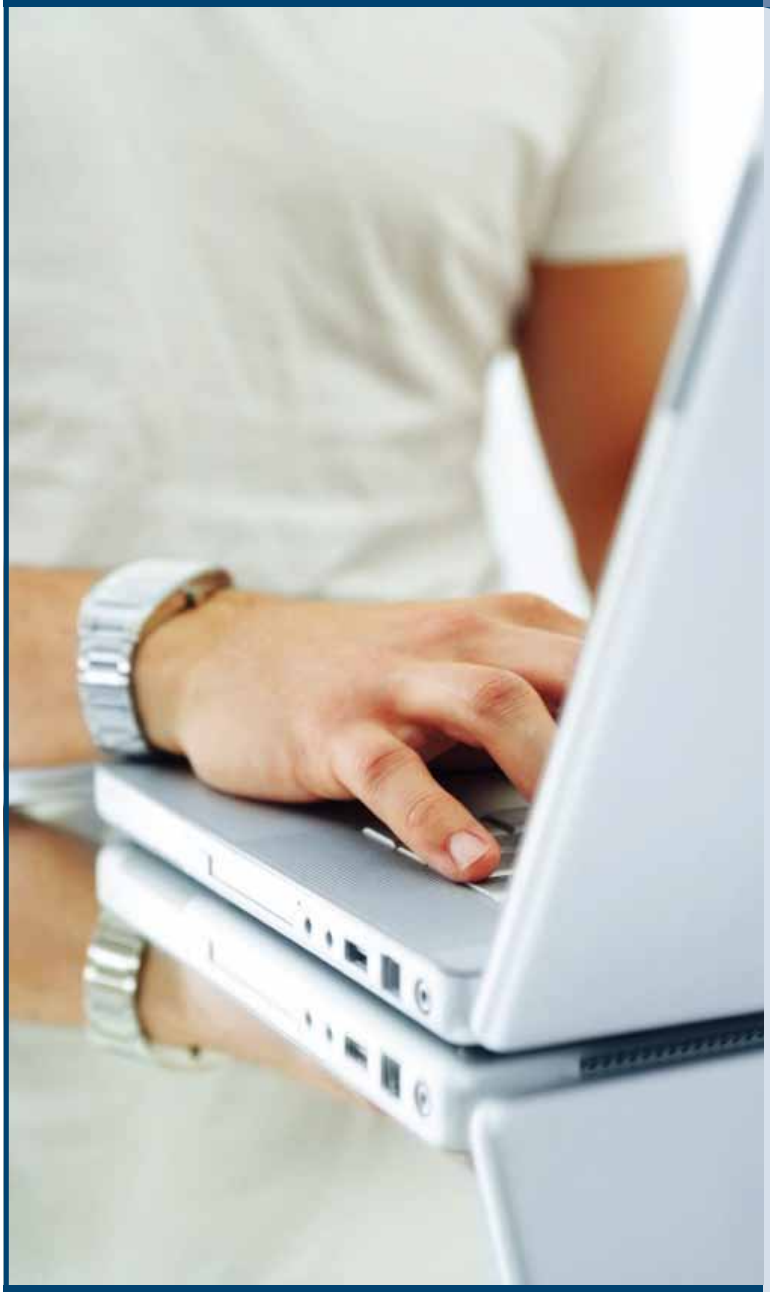
Continued on pg 14

The Data & Analysis Center for Software

Online Learning Center

Technical Training On Demand

Cost effective way for organizations to provide continuous learning opportunities for their employees



Accessible

Classes available 24/7/52
From:

- Home
- Office
- Travel

Accreditation Certification

- CEUs granted
- IACET accreditation supported

Affordable

- Pay one annual subscription fee ($475)
- Take as many classes as you want

Comprehensive

- 450+ classes
- 12,000 topics
- Programming & Web Development
- Course Catalog

Latest Technologies

- Java XML
- Oracle
- Server technologies
- .NET Framework

Flexible

- Fit course work into your schedule
- Self Paced
- Refresh Knowledge
- Study during lunch or after work

To view the catalog visit:
www.thedacs.com/training
For details call: 1.800.214.7921

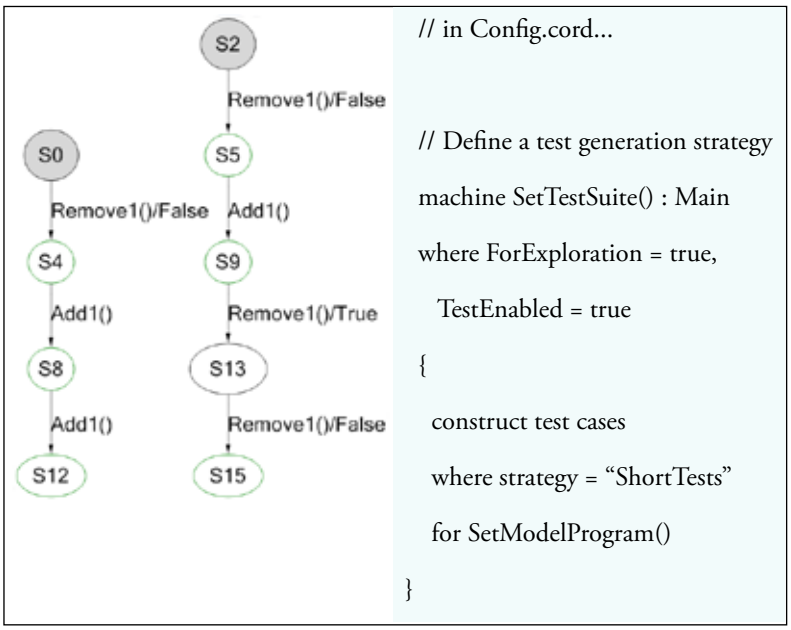


Figure 6: The two tests generated by the 'ShortTests' strategy.

IT applications with Smartesting Test Designer

Test Designer, from Smartesting, is a commercially available model-based testing tool dedicated to IT applications, secure electronic transactions and packaged applications such as SAP or Oracle E-Business Suite. Test cases are generated from a behavior model of the SUT, using requirement coverage and custom scenarios as test selection criteria. Test Designer models are written in a subset of standard UML (class and object diagrams as well as state machine diagrams [6], with OCL annotations [7]). Test Designer supports both a transition-based modeling style (i.e. UML State Machines) and a Pre/Post style (i.e. OCL).

Model elements such as transitions and OCL decisions can be linked to the informal requirements that they cover. Test coverage can then be based on requirements coverage. A range of structural model coverage criteria are also supported, such as transition, decision and effect coverage. The test engineer may also define business scenarios as custom test case specifications that use the UML operations. Test Designer supports both manual and automated test execution, using an offline approach. The generated test cases can be output to test management systems like HP Quality Center or IBM Rational Quality Manager, with bidirectional traceability and full change management for evolving requirements.

Now, we illustrate how Test Designer can be used to model and test the actiTIME application.

actiTIME Overview

actiTIME is a time management program developed by Actimind. Details about its features, and free downloads, can be found on the website **www.actitime.com.** Ordinary users have access to their time track for input, review and corrections. They can also manage projects and tasks, do some reporting and of course they can manage their account (see (1) in Figure 7).

In our sample model we focus on the user time-tracking features of actiTIME version 1.5; after logging into the system the user can specify how much time he spent on a specific task.

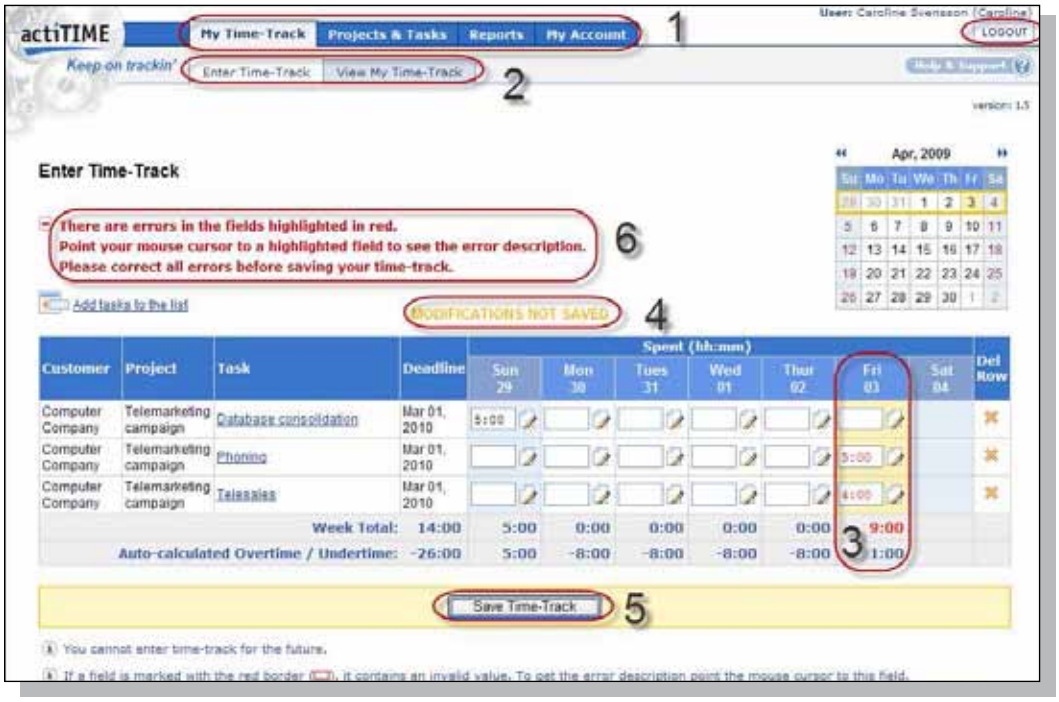


Figure 7: actiTIME User Interface.

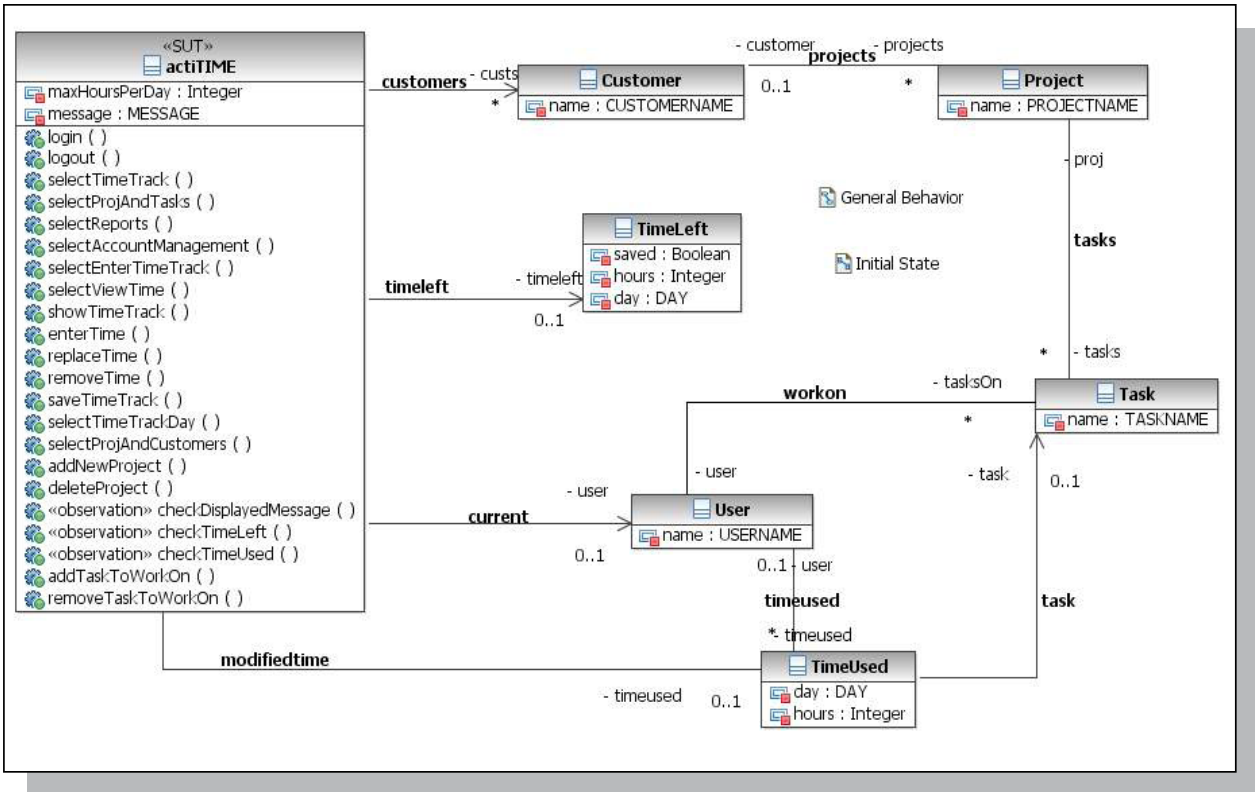


Figure 8: Class Diagram for actiTIME Test Model.

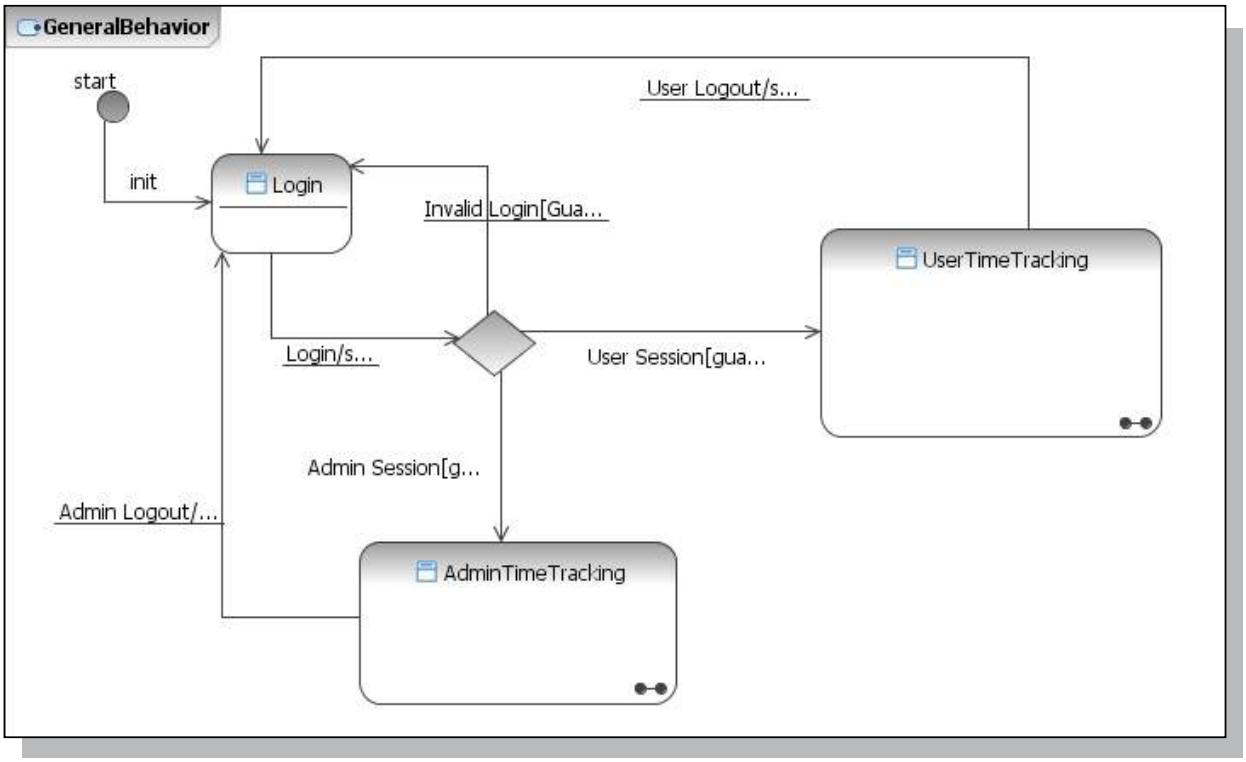


Figure 9: High Level State Machine for actiTIME (partial).

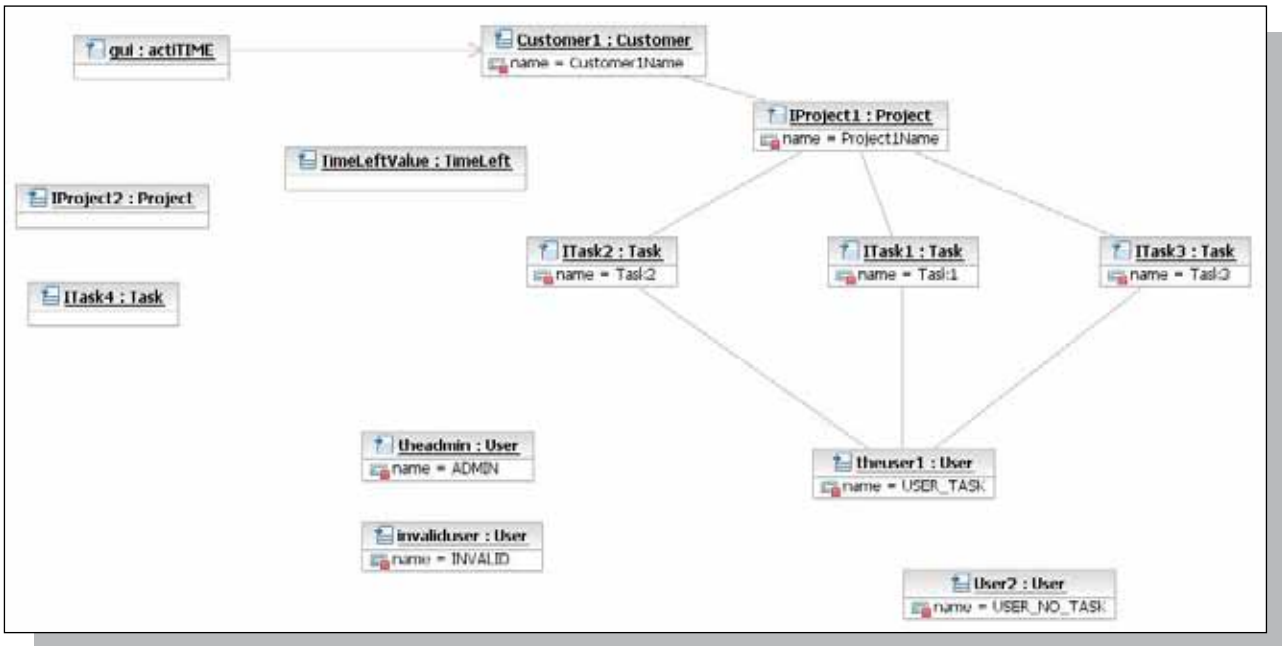


Figure 10: Object Diagram for actiTIME.

actiTIME Test Model

The test model represents the expected behavior of the application, covering the requirements of Table 1. It is based on three UML diagrams (see Figure 8, Figure 9 and Figure 10):

- the class diagram represents the business entities and the user actions to be tested;
- the layered state machine represents the dynamic expected behavior;
- the instance diagram gives some test data and initial configuration of the application.

Test Generation with Test Designer

Figure 11 shows the GUI of Test Designer for the project actiTIME. A list of the generated test cases (structured by test Suites) is displayed on the left, and the details of one test case are displayed on the right. The details of the requirements and test aims that are covered by a particular test step are shown in the right-hand bottom corner.

After test generation, the generated tests are published into a test repository. These tests are ready for manual test execution. Each test is fully documented in the Design Steps Panel.

For test automation, complete script code is generated and

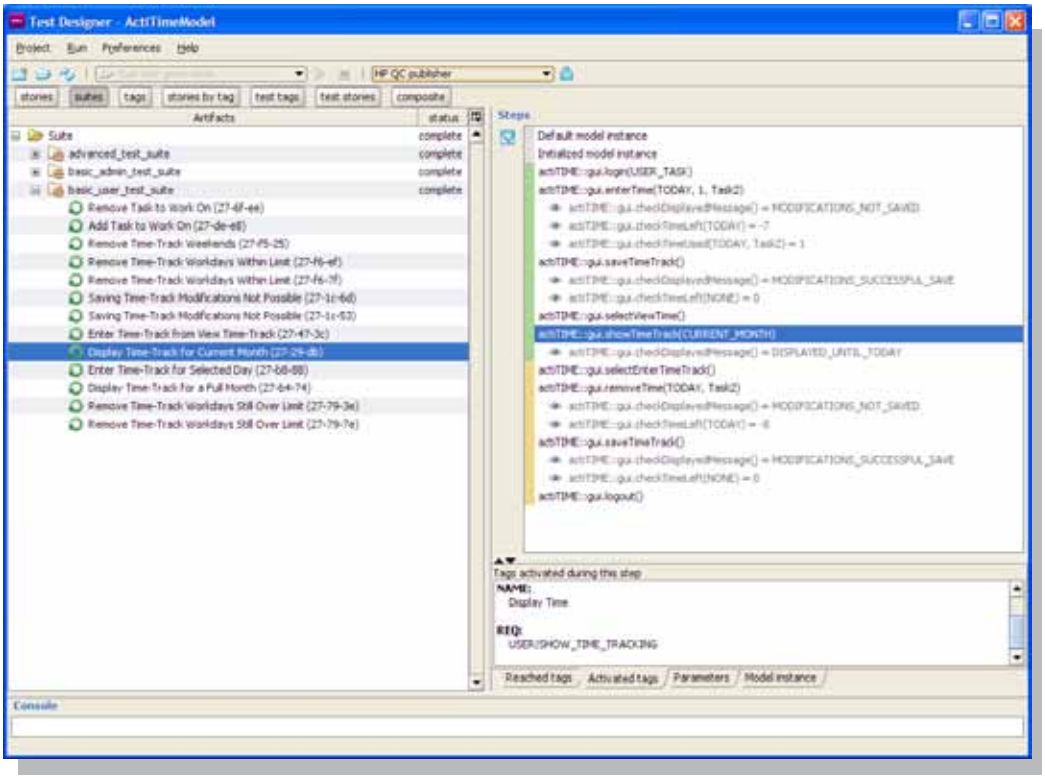


Figure 11: Smartesting Test Designer user interface. Project actiTIME.

maintained for each test case. The remaining (optional) task for the test automation engineer is to implement each key-word used in UML test model so that it is defined as a sequence of lower-level SUT actions. If this is done, the generated test scripts can be executed automatically on the SUT. An alternative approach is to leave the key-words undefined, in which case a human tester must execute the scripts manually.

To sum-up, Test Designer is a typical MBT solution for IT applications, using a subset of UML as input language (class diagrams, state diagrams, instance diagrams, and OCL specification language), providing test generation and publication features both for manual and automated testing.

Key factors for success when deploying MBT

The key factors for effective use of MBT are the choice of MBT methods, the team organization, the qualification of the people involved, and a mature tool-chain.

1. **MBT Methods, requirements and risks:** MBT should be implemented on top of current best practices in functional software testing. The SUT requirements must be clearly defined, so that the test model can be designed from those requirements. The product risks should be well understood, so that they can be used to drive the MBT test generation.
2. **Organization of the test team:** MBT can be a vector for process and productivity improvements. Roles (for example, the test analyst who designs the test model and the test automation engineer who implements the adaptation layer) are reinforced.
3. **Team Qualification - test team professionalism:** The qualification of the test team is an important pre-requisite. Test analysts, test automation engineers, and testers should be professional and have appropriate training in MBT techniques, processes, and tools.
4. **The MBT tool chain:** To maximize the effectiveness of MBT, it is important use an integrated tool chain, including a MBT test generator that integrates with your test management environment and test automation tools.

Expected benefits of MBT

Model-based Testing is an innovative and high-value approach compared to more conventional functional testing approaches. The main expected benefits of MBT may be summarized as follows:

- Contribution to the quality of functional requirements:
  - Modeling for test generation is a powerful means for the detection of "holes" in the specification (undefined or ambiguous behavior).
- Contribution to test generation and testing coverage:
  - Automated generation of test cases;
  - Systematic coverage of functional behavior;
  - Automated generation and maintenance of the requirement coverage matrix;
  - Continuity of methodology (from requirements analysis to test generation).
- Contribution to test automation:
  - Definition of action words (UML model operations) used in different scripts;
  - Test script generation;
  - Generation of skeleton code for a library of automation functions;
  - Independence from the test execution robot.

Conclusion

The idea of model-based testing is to use an explicit abstract model of a SUT and its environment to automatically derive tests for the SUT: the behavior of the model of the SUT is interpreted as the intended behavior of the SUT. The technology of automated model-based test case generation has matured to the point where large-scale deployments of this technology are becoming commonplace. The prerequisites for success, such as qualification of the test team, integrated tool chain availability and methods, are now identified, and a wide range of commercial and open-source tools are available.

Although MBT will not solve all testing problems, it is an important and useful technique, which brings significant progress over the state of the practice for functional software testing effectiveness, and can increase productivity and improve functional coverage.

References

[1] M. Utting, B. Legeard, *Practical Model-Based Testing: A Tools Approach*, Morgan Kauffman, 2007.

[2] Hung Q. Nguyen, Michael Hackett, and Brent K. Whitlock, *Global Software Test Automation: A Discussion of Software Testing for Executives*, Happy About, 2006.

[3] Elfriede Dustin, Thom Garrett, and Bernie Gauf, *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*, Addison-Wesley Professional, 2009).

[4] Keith Stobie, Model Based Testing in Practice at Microsoft, *Electronic Notes in Theoretical Computer Science, Volume 111,* 1 January 2005, pages 5-12, *Proceedings of the Workshop on Model Based Testing (MBT 2004).*

[4] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson, *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, in *Formal Methods and Testing*, Springer Verlag, 2008.

[5] Martin Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)*, Addison-Wesley Professional, 2003.

[6] *Jos Warmer* and *Anneke Kleppe*, *The Object Constraint Language: Getting Your Models Ready for MDA (2nd Edition)*, Addison-Wesley Professional, 2003.

About the Authors

In 2007, Mark Utting and Bruno Legeard authored the first industry-oriented book on model-based testing, "Practical Model-Based Testing: A Tools Approach".

Bruno Legeard is Chief Technology Officer of Smartesting, a company dedicated to model-based testing technologies and Professor of Software Engineering at the University of Franche-Comté (France). He started working on model-based testing in the mid 1990's and has extensive experience in applying model-based testing to large information systems, e-transaction applications and embedded software.

Mark Utting works for Netvalue.net.nz, using agile techniques to develop Next Generation Genomics Software, and is also an Associate Professor in Computer Science at The University of Waikato in New Zealand. He is the author of more than 50 publications on model-based testing, formal methods for object-oriented and real-time software, and language design for parallelism.

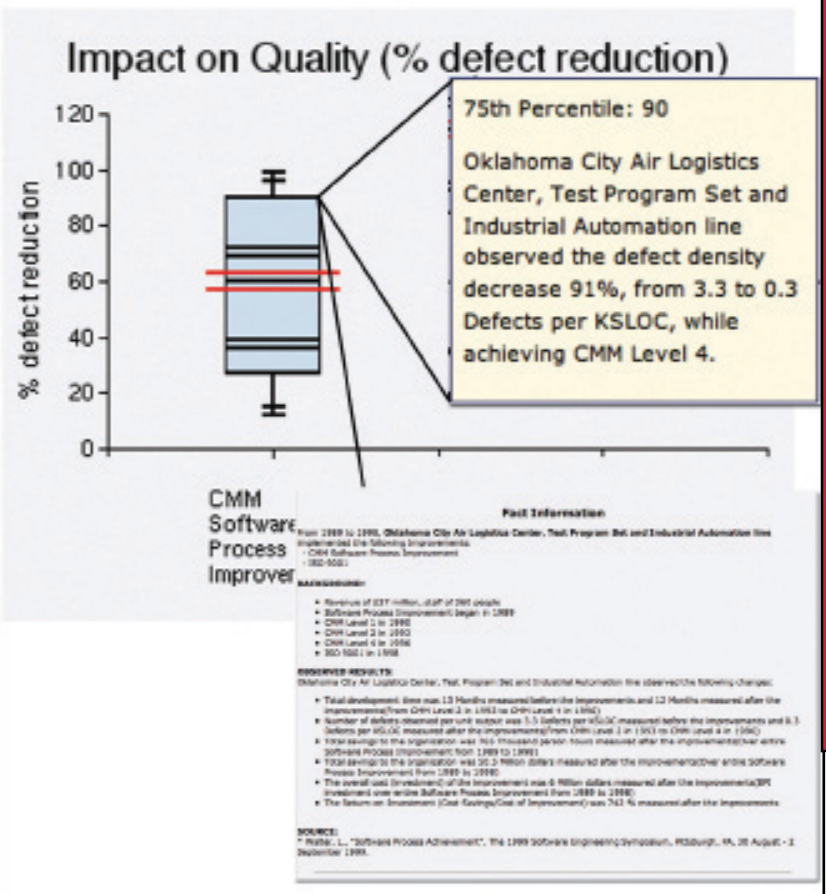
Author Contact Information

Email: Bruno Legeard: legeard@smartesting.com

Email: Mark Utting: marku@cs.waikato.ac.nz

ARE YOU GETTING THE MAX FROM YOUR SOFTWARE INVESTMENT?

The DACS ROI Dashboard



Technologies Covered:

- SEI/CMM/CMMI
- SEI Team Software Process (TSP)
- SEI Personal Software Process (PSP)
- Inspections
- Reuse
- Cleanroom

And Many More!

Graphs Showing Impact of Software Technologies on:

- ROI
- Productivity
- Quality

Summarizes Facts from Open Literature

Access the DACS ROI Dashboard!

http://www.thedacs.com/databases/roi/

The Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400
http://www.thedacs.com

Theory—Practice—Tools for Automated Statistical Testing

WE WERE SOMEWHAT AHEAD OF THE TIMES, BUT RECENT INTEREST IN PROVIDING EVIDENCE FOR "DEPENDABILITY ASSURANCE CASES" AND A GROWING NEED TO DEMONSTRATE COMPLIANCE WITH INDUSTRIAL STANDARDS ARE CATCHING UP WITH THE IDEA OF STATISTICAL TESTING. .

by Jesse H. Poore

The Software Quality Research Laboratory (SQRL) at the University of Tennessee has a long-standing program of research and development,* flowing from the foundations of Cleanroom software engineering [1; 2]. Most of our efforts have focused on two areas: (i) treating testing as a problem to be addressed by statistical science, and (ii) treating software development from a function theory point of view. Originally, these represented two separate threads of development but soon merged as the methods for deriving both specifications and statistical models from requirements merged. At all times the program has focused on building accurate theoretical foundations, followed by field experience with industrial partners in order to develop engineering practices. These practices are then embodied in tools that enforce the workflow (see Figure 1) and assure adherence to theory. The engineering practices define the workflow and separate the syntactical and mathematical details, which are made available by the tools, from the crucial information from human domain engineers and software developers. The following is a brief summary of the theory, practice, and tools now available with reference to the key literature. See also the glossary of related terms displayed in the accompanying sidebar to this article.

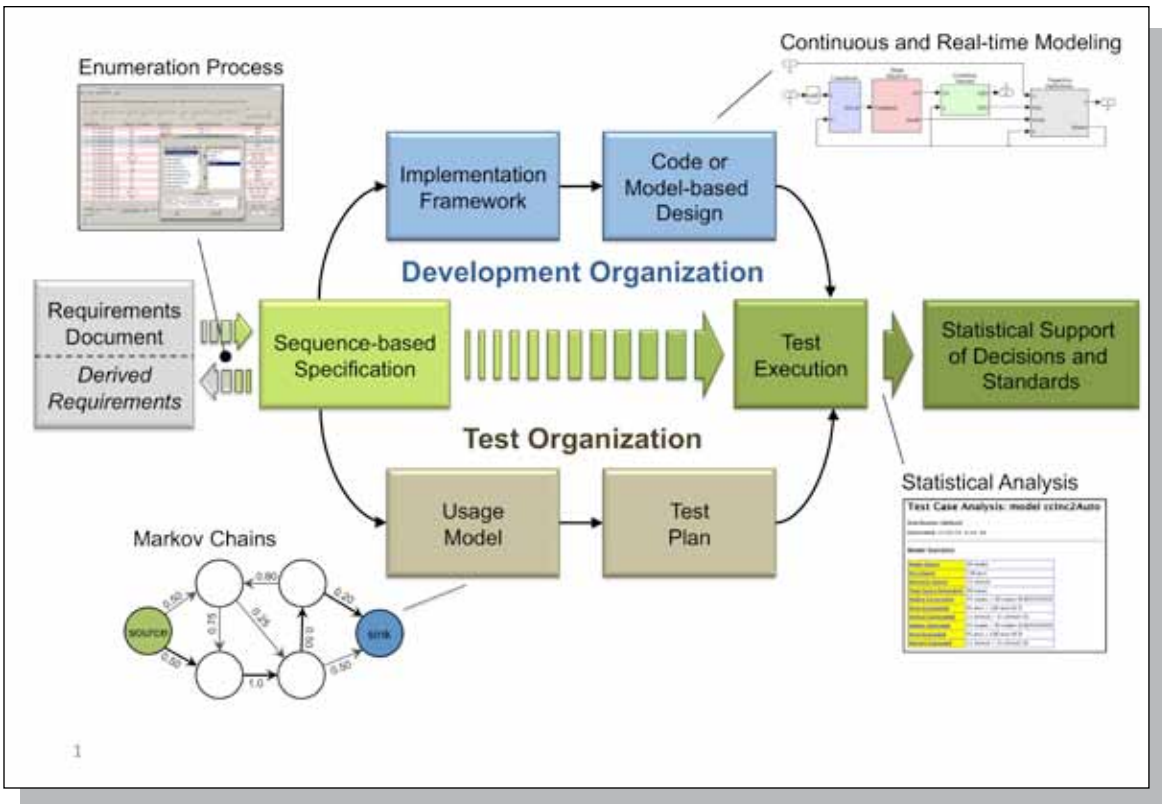


Figure 1: Work Flow in a Statistical Testing Environment

* This research program has been funded entirely by contracts and gifts from industry, including at various times BNR, Ericsson, CTI, IBM, Nortel, Oak Ridge National Lab, Raytheon, and Verum Consultants. We acknowledge not only the funding but as importantly the willingness to try our theories, help to establish engineering practices, and provide valuable feedback on our work.

Automated Statistical Testing

Our earliest efforts were in statistical testing. We were somewhat ahead of the times, but recent interest in providing evidence for "dependability assurance cases" and a growing need to demonstrate compliance with industrial standards are catching up with the idea of statistical testing. Today, a complete and mature library of command-line tools to support automated statistical testing, the Java Usage Model Builder Library (JUMBL), is freely available from SQRL. JUMBL has been downloaded by a few thousand organizations and is embedded in several proprietary and commercial testing tools and services.

Our methods use stochastic processes with Markov chain usage models, sampling theory, a Bayesian reliability model [3], and related stopping criteria. The general process is described in [4]. A usage model encoded as a directed graph and Markov chain represents a population of use cases (at the chosen level of abstraction). We use statistics from the Markov chain to validate the model and plan our testing process. The test cases are generated from the usage model by walking the graph, using graph algorithms, hand tracing test cases, or sampling. We also pay attention to edges of the graph because they carry scripts that are instructions to automated test runners. Pass-fail data determine a model of test experience, which is the basis for reliability estimates, coverage analysis and the stopping decisions. Every detail of the process from model definition to testing record becomes a part of the documentation.

One of the first lessons we learned from practice was that if testing were to adequately satisfy development engineers, product managers, and industrial standards, it would have to be automated. Testing protocols usually begin with model coverage based on graph algorithms, followed by testing of specific cases to address contractual and policy obligations and ending with random sampling to satisfy standards of interest. The result is hundreds of thousands of test steps. To achieve this magnitude, it is necessary to automatically generate, execute, evaluate, and record each test case and result.

The JUMBL command-line library, which reads and writes several open file formats, is designed to interface with commercial automated test environments. While we aren't familiar with all the sites that are actively using or exploring these methods and tools, we do collaborate with users in IBM, Verum Consultants, Fraunhofer IESE, and Oak Ridge National Laboratory, among others. See [5] for a recent report of these methods and tools applied to Philips Healthcare medical imaging systems.

GLOSSARY

Assurance Case: A structured set of arguments and a corresponding body of evidence demonstrating that a system satisfies specific claims with respect to its security, safety, or reliability properties.

Bayesianism: A philosophical interpretation, due to Thomas Bayes, Bruno de Finetti, and others, of probabilities as self-consistent subjective odds. Bayesians emphasize the application of Bayes' theorem to use prior information and update assessments of probability in the light of observed data.

Cleanroom software engineering: A methodology for software development combining incremental development under statistical quality control, box structure top-down design, functional verification of code to specifications and statistical testing.

Dependability: The property of software that ensures the software always operates as intended.

Discrete System: A system which can take only a countable number of inputs and states, and only transitions between states at discrete increments of time. (contrast continuous and hybrid System)

DOORS: A requirements management tool available from IBM. http://www-01.ibm.com/software/awdtools/doors/

Experimental Design: A branch of statistics in which the values of independent variables are chosen to test hypotheses about relationships of causality or dependence.

Functional Requirements: The statement of what functions a system must perform, typically in terms of inputs, outputs, and the transformation of one into the other. Non-functional requirements are often specified as quality attributes, such as availability, safety, security, maintainability, performance, etc,

Lexicographic Order: Alphabetic order, for example, of text strings of a specified length.

Markov Chain: A stochastic process for a discrete system in which the probability distribution over the states after the next time increment depends only on the current state and not on the transition history that resulted in that state.

Real-Time System: A system (discrete, continuous or hybrid) in which constraints specify the deadlines by which the system must respond to certain events. The time to respond is often measured along a continuum.

CONT.

GLOSSARY cont.

Reliability Model: A model, often based on failure data obtained during functional testing with tests selected from a usage model, for estimating the probability that a system will execute in a given environment for a given time without failure. (Software reliability models may estimate the probability that an input sequence randomly chosen from all possible sequences will execute correctly.)

Sampling Theory: The study of the "selection and implementation of statistical observations in order to estimate properties of an underlying population." http://mathworld.wolfram.com

Simulink®: A tool, available from Mathworks, for graphical modeling of time-varying systems. http://www.mathworks.com/products/simulink/

Sequence-Based Specification: A collection of techniques for software specification in which "software behavior is specified in terms of the appropriate next externally observable response to each sequence of external inputs."

Statistical Testing: Methods for deciding about the acceptance or rejection of a hypothesis based on a sample of the outcomes of a random experiment. (Applied to software, it is the use of statistical science to demonstrate that a system satisfies various criteria and to estimate reliability.)

Stochastic Process: An indexed set of random variables. A stochastic process can be used to specify how the probability distribution over the states of a system evolves over time. (A Markov chain is a stochastic process.)

Test Coverage: A measure of how much of a system is executed by the test suites. Test coverage metrics in terms of code include statement coverage, branch coverage, and definition-use coverage. (In the case of software usage models, coverage of inputs, states, transitions, and paths may be measured.)

Test Oracle: "Any (human or mechanical) agent that decides whether the program behaved correctly on a given test."

Usage Model: A model ,at some level of abstraction, in the form of a directed graph of the possible uses of a system and (when known) the relative likelihood of taking the next step from each node of the graph.

Requirements Analysis and Specification Development

Functional verification as prescribed by Mills requires a precise functional specification [6]; acquiring the specification is more difficult than the subsequent verification. Although Mills used set-theoretical rules to define functions, we began to explore an alternative: explicit roster listing of ordered pairs (<argument, value> or <input, output>) for the function definitions. To construct this list of definitions consistently and completely requires systematic enumeration that characterizes the infinite set of all finite input sequences. At first thought, this seems impractical because of the combinatorial growth, but experience has proven it both efficient and effective.

The enumeration process [7] considers every sequence of inputs in length-lexicographic order, beginning with length one, then length two, etc., until the process terminates, as it surely must. Through short sequences, the early phase brings to light all the startup considerations before a process reaches steady state; then enumeration proceeds to identify all the control states, finally concluding with the shut-down or continuous-cycle phase. We have found that enumeration requires frequent interaction between the owners of requirements (product managers, for example) and the specification developers, if they are to construct a complete, consistent, and (traceably) correct specification. Every decision in constructing the specification must be tagged to a statement of the functional requirements as justification for decisions made. At the conclusion of the enumeration process, you can generate the state machine, the code framework, and much of the code itself (see [8] for a case study).

To enumerate, we use a prototype tool that enforces strict adherence to the process and prompts the user to answer the questions required to construct and document a complete specification. (We know of two organizations that have developed enumeration tools using Excel macros and a third one implemented in DOORS.) The prototype collects, constructs, and maintains documentation information as the process evolves, which can be generated in standard form at any time.

We start with ordinary functional requirements, which might take the form of text documents, predecessor systems, competitor systems, or ideas in the heads of product developers. Through a prescribed series of steps, the enumeration process attempts to construct a precise specification; if at any point a step cannot be justified by the requirements, then the requirements must be enhanced. Thus, errors and omissions in the requirements are exposed and corrected (at the working

level of abstraction) as the process evolves to its conclusion. The specification also can be used to generate a directed graph to serve as the basis for a statistical testing usage model. See [9] for a case study with a tool chain from specification development to the conclusion of statistical testing.

This specification process results in many changes to requirements, at least in resolving errors, omissions, and inconsistencies. Moreover, changes coming from product managers and others outside the enumeration process must be accommodated as well. Analysis shows that changes in functional requirements induce twelve distinct types of changes to specifications (for example, adding an input element, changing a response, judging a sequence to be physically impossible, changing the equivalence of a sequence, etc. [10]). Each of the twelve has a unique effect on the existing specification. The prototype tool applies algorithms to automatically make all the syntactical or mathematical changes possible and highlights all the cases where a human must consider the change effect and provide additional information.

All our experience to date employs a discrete enumeration process, in which we must introduce abstractions to handle timers and continuous events. For example, an abstract element "start timer" or "timer expired" would be introduced into the input alphabet to take such details into account. This is efficient and effective when one knows or intuits the correct abstraction; however, some trial and error usually precedes its identification.

Real-time, Continuity, Non-determinism

Although the discrete process has been used for hybrid and switching real-time systems, special insight was required to abstractly represent timing, non-determinism, and continuity. Recent SQRL research extended the enumeration process explicitly to treat these details [11]. Our process for deriving precise hybrid specifications has been designed to retain the strengths of the process for discrete systems, to build on a sound mathematical foundation for hybrid systems, to focus on generating Simulink® models (or similar model-based design products), and to connect with existing automated statistical testing methodology and tools.

Model-based design tools have emerged as an effective way to prototype complete systems without incurring the costs of constructing physical components. These system models incorporate discrete and continuous behavior operating in unison. Designers can use the tools to refine entire model blueprints until the software and physical device model work together to meet a client's requirements.

Tools such as Simulink® make it "easy" to produce complex systems. They do not, however, guarantee that the intended behavior is implemented correctly in the model design. Requirements must drive the modeling process and, therefore, provide the basis for testing complete system function.

Following the enumeration process, we systematically produce a hybrid, sequence-based specification that satisfies our functional requirements; as with the discrete process, this step-by-step process improves the requirements statements. We then generate Simulink ®models, test-case generators, and test oracles from this specification; produce test cases targeting the system at various levels of abstraction; and statistically evaluate the results to quantify system behavior.

The key discoveries to be made during the hybrid system specification process are:

1. The critical internal actions—the interactions between the discrete subsystem and the continuous subsystem.
2. The discrete states of the system, or operation modes.
3. The atomic functions describing the behavior of each continuous variable as a function of time.
4. The combinations of atomic functions that define essential system trajectories.

In a discrete system, machine state is time-independent. When an action produces a state change, the system's state is fixed until another action occurs. In a hybrid automaton, an action may cause a mode change and a trajectory change. In other words, the operation modes of the system and the characteristics of their trajectories are determined by this action.

Future Work

We believe this enumeration approach to software and system specification forces developers to ask the detailed, appropriate questions, in the right order, necessary to achieve a full understanding of their design task and ultimately to get the system model right. The specifications support generation of code, testing models, and test oracles.

In further work, we will continue to:

1. Work with industry to make the methods and tools as effective as possible.
2. Extend the discrete, sequence-based specification

methods to incorporate hybrid behavior for various patterns.

3. Extend and refine the enumeration tools to support hybrid enumeration.
4. Explore tools that automatically generate model-based designs—including trajectory definitions—from specifications.
5. Extend and refine statistical testing, contributing to dependability assurance cases.

In 1987 Mills observed "a surprising synergy between mathematical verification and statistical testing of software." [1] Sequence-based enumeration as the means of functional specification has improved the synergy between implementation and testing models flowing from complete, consistent, and (traceably) correct specifications.

References

[1] Mills, H. D., M. Dyer, and R. Linger, "Cleanroom Software Engineering," *IEEE Software*, pp. 19-24, September 1987.

[2] Prowell, S.J., C.J. Trammell, R.C. Linger, and J.H. Poore, *Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1999.

[3] Prowell, S.J. and J.H. Poore, "Computing system reliability using Markov chain usage models," The *Journal of Systems and Software*, 73:219-225, 2004.

[4] Poore, J.H. and C.J. Trammell, "Engineering Practices for Statistical Testing," *Crosstalk: The Journal of Defense Software Engineering*, April 1998.

[5] Bouwmeester, L., G. Broadfoot, and P. Hopcroft, "Compliance Test Framework," *Proceedings of 2nd Workshop on Model-based Testing in Practice* (MoTiP 2009), Ed.: T. Bauer, H. Eichler, A. Rennoch, Fraunhofer IRB Verlag, Stuttgart, 2009.

[6] Linger, R., H. Mills, and B. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, 1979.

[7] Prowell, S.J. and J.H. Poore, "Foundations of Sequence-based Software Specifications," *IEEE Trans¬actions on Software Engineering*, 29(5), May 2003.

[8] Broadfoot, G.H. and P.J. Broadfoot, "Academia and Industry Meet: Some experiences of formal methods in practice," *Proc. of the Tenth Asia-Pacific Software Engineering Conference*, Chiang Mai, Thailand, pp. 49-59, 2003.

[9] T. Bauer, T. Beletski, F. Bohr, R. Eschbach, D. Landmann, and J.H. Poore, "From requirements to statistical testing of embedded systems," *Fourth International ICSE Workshop on Software Engineering for Automotive Systems*, May 2007.

[10] Lin, L., S.J. Prowell and J.H. Poore. "The impact of requirements changes on specifications and state machines," *Software--Practice and Experience*, 39:573-610, 2009.

[11] Carter, J., *Sequence-Based Specification of Embedded Systems*, PhD thesis, The University of Tennessee, Knoxville, Tennessee. 2009. (Web link.)

About the Author

Jesse H. Poore is a professor of computer science at the University of Tennessee and holds the Ericsson-Harlan D. Mills Chair in Software Engineering. He is the director of the University of Tennessee-Oak Ridge National Laboratory Science Alliance. His interests are the economical production of high-quality software and federal science policy. He received a PhD in information and computer science from the Georgia Institute of Technology. He is a member of the IEEE, the IEEE Computer Society, and the ACM and is a fellow of the AAAS. He received the 2002 IEEE Software Engineering Award.

Author Contact Information

Email: Jesse H. Poore: jpoore@tennessee.edu

The DACS Gold Practice Initiative:

- Promotes effective selection/use of software acquisition & development practices
- Defines essential activities/benefits of each practice
- Considers the environment in which each practice is used
- Addresses the timeliness of practice benefits
- Recognizes interrelationships between practices that influence success or failure
- Contains quantitative and qualitative information
- A continually evolving resource for the DoD, Government, Industry and Academia
- Free to use/free to join

Learn More About the DACS Gold Practice Initiative:
http://www.goldpractices.com

Current Gold Practices:

- Acquisition Process Improvement
- Architecture-First Approach
- Assess Reuse Risks and Costs
- Binary Quality Gates at the Inch-Pebble Level
- Commercial Specifications and Standards/Open Systems
- Develop and Maintain a Life Cycle Business Case
- Ensure Interoperability
- Formal Inspections
- Formal Risk Management
- Goal-Question-Metric Approach
- Integrated product and Process Development
- Metrics-Based Scheduling
- Model-Based Testing
- Plan for Technology Insertion
- Requirements Management
- Requirements Trade-Off/Negotiations
- Statistical Process Control
- Track Earned Value

The Data & Analysis Center for Software

P.O. Box 1400
Rome, NY 13442-1400
http://www.thedacs.com

Toward Model-based Embedded System Validation through Virtual Integration

IF WE CAN DISCOVER A REASONABLE PERCENTAGE OF THESE LATE SYSTEM-LEVEL FAULTS EARLIER IN THE DEVELOPMENT PROCESS WE CAN EXPECT CONSIDERABLE COST SAVINGS.

by Peter H. Feiler and Jörgen Hansson

Embedded systems are safety-critical and mission-critical systems that have become increasingly software-intensive - with millions of lines of code executing on a distributed networked set of processors. Developing such systems has indeed shown to be increasingly challenging as embedded software subsystems compete for computer system resources and face unpredictable interaction behavior due to the time-sensitive nature of the application. Several studies [1,2,3] have shown that 70% of faults are introduced early in the life cycle, while 80% of them are not caught until integration test or later with a repair cost of 16x or higher. The diagram illustrated in Figure 1 shows percentages for fault introduction, discovery, and cost factors. The cost of developing the software for an aircraft has become unaffordable – reaching $10B or more. If we can discover a reasonable percentage of these late system-level faults earlier in the development process we can expect considerable cost savings.

There are many cases of systems failing due to a software error despite the fact that the best design techniques and fault-tolerance techniques are being used. [7] examines some of them and the root causes due to mismatched assumptions. A well-publicized example of a system failure due to software is the explosion of the Ariane 5 rocket during her maiden flight. The destruction was triggered by the overflow of a 16-bit signed integer variable due to Ariane 5's greater acceleration in a reused Ariane 4 software component to perform a function that was

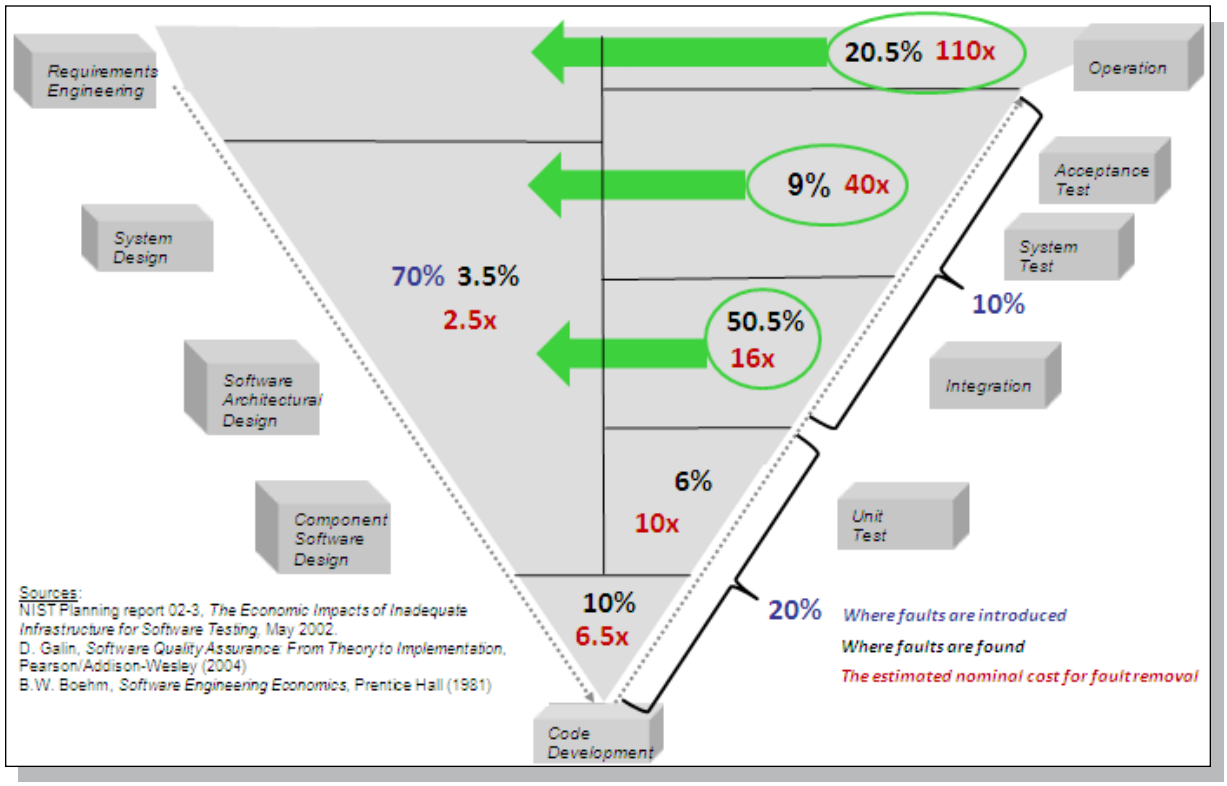


Figure 1: Fault Introduction, Discovery and Cost Factors

"not required for Ariane 5". This fault was not caught because the handler was disabled due to efficiency considerations and cascaded into a total system failure [8]. Even simple systems such as the ITunes music program failed when it was migrated to a dual-core processor system. Two tasks were involved in ripping CDs. When executing on a single processor first one task and then the other task was performed. Once on a dual-core processor, the two tasks executed at the same time and got in each other's way updating the music catalog [9]. Finally, with regard to the Mars Pathfinder after a successful landing, not long after it started gathering meteorological data the spacecraft began experiencing system resets. The press reported these failures in terms such as "software glitches" and "the computer was trying to do too many things at once". An analysis on the ground identified the problem as priority inversion due to a low priority task blocking a high priority task on a lock to a shared data area. Once recognized the engineers could remotely enable the use of a priority ceiling protocol that overcomes the inversion problem and the mission completed successfully [10].

Quantitative architectural modeling, verification and validation of software and system behaviors have shown significant promise in addressing defects related to requirements and design shortcomings for embedded software.

The Promise of Model-based Engineering

Model-based engineering (MBE) involves the creation of models of the system with focus on a certain aspect of the system such as functionality, performance, or reliability. Modeling, analysis and simulation has been practiced by engineers for a number of years. Traditionally, this has led to the creation of different models of the same system to represent the interests of different stakeholders. For example, computer hardware models have been created in Very High Speed Integrated Circuits Hardware Description Language (VHDL) [15] and validated through model checking [16]. Control engineers have used modeling languages such as Simulink for years to represent the physical characteristics of the system to be controlled and the behavior of the control system. Characteristics of physical system components, such as thermal properties, fluid dynamics, and mechanics, have been modeled and simulated. Dependability engineering resulted in the creation of fault trees for fault analysis and Markov models for reliability analysis. Resource utilization analysis is based on resource demands and resource capacities. Scheduling analysis is based on a timing model of the application tasks. Security analysis involves the creation of a model in terms of security levels and domains applied to subjects and objects.

Despite these modeling efforts system-level problems remain late into the development life cycle. These different models are created at different times by different stake holder teams interpreting architecture & design documents to gain their understanding of the system. These independently created models may not be consistent with each other and with the actual system as it evolves. Changes to the system are not always communicated to the various modeling teams and the various models become out of date. The result is a multiple truth problem that industry has experienced with model-based engineering. The analysis results have limited value as they

A number of recent studies confirm that a paradigm shift towards analysis and formal validation at the architecture level must occur to meet the challenges of these time-sensitive software-reliant systems with high safety and reliability demands:

- GAO Space-based Software Study [11] highlighting the reality of more testing than planned (exhausting vs. exhaustive testing) due to the increasingly complex interactions between system components;
- NASA Software Complexity Study [12] on flight software growth & complexity, and the need for integration of fault prevention, detection, and containment with nominal system operation;
- Leveson System Safety Engineering Study [13] on accident model based systems theory focusing on accident factors (understanding of root causes);
- National Research Council Study [14] by the committee for certifiably dependable software systems addressing the issue of sufficient evidence for software for dependable systems through analysis and formal validation.

reflect inconsistent and out of date models.

This traditional use of model-based development is understandable, given the lack of a common, precise architecture description language to represent the system architecture. To illustrate the need for an architecture model, consider the complexities of just one quality attribute - security. A system designer faces several challenges when specifying security for distributed computing environments or migrating systems to a new execution platform. Business stakeholders impose constraints due to cost, time-to-market requirements, productivity impact, customer satisfaction concerns, and the like. A system designer needs to understand requirements for the confidentiality and integrity of protected resources (e.g., data) and identify how the application software executing on a distributed computer system and interfacing with physical systems and human operators meets those requirements. In addition they have to predict the effect that security measures will have on other runtime quality attributes such as resource consumption, availability, and real-time performance. However, despite these considerations, security is often studied only in isolation and late in the process. The unanticipated effects of separated design approaches or changes are discovered only late in the life cycle, when they are much more expensive to resolve.

Only in the best case do the independent models created in the traditional approach reflect the same system architecture. Any change to the architecture during its lifetime requires each model to be updated and verified. As challenging as that is, the need to consistently proliferate changes in one analysis to others adds difficulty (e.g., reflect the impact of choosing a different security encryption scheme on intrusion resistance as well as on schedulability and end-to-end latency). Consequently, it has become important to move from the traditional approach toward the integration of the different analysis dimensions into a single, precisely specified architecture model. Creating a single architecture model of the system that is annotated with analysis-specific information can drive model-based development. It allows analysis-specific models to be generated from the annotated model and the regeneration with little effort of those models to reflect changes to the architecture. This new approach also allows the impact of changes across multiple analysis dimensions to be readily analyzed, giving the system designer more insight and the system integrator warning of costly side effects. Even after the development phases have been completed, the model (and its associated analysis models) can be used to evaluate effects of reconfiguration and system revisions.

Modeling the Embedded System Architecture with AADL

To precisely specify architecture, allow non-ambiguous interpretation, support quantitative analysis, and model embedded software-intensive systems, we need a common, standard language with strong semantics that can capture both the structure and dynamics of embedded systems. The requirements of such specification include: support for analyzable system properties to evaluate critical quality attributes; incremental modeling to support all life-cycle phases from early abstraction to final implementation; and evolution compositional representation to support hierarchical and abstract layering, allowing multiple levels of integration with suppliers and providing integrators with component integration extensibility to allow new representations of system behaviors as analysis approaches differ and grow.

As the importance of architecture has been recognized, industrial standards for architecture modeling have emerged: OMG SysML [6] for system engineering and SAE Architecture Analysis and Design Language (AADL) [4-5] for embedded software systems. SysML is a graphical modeling language in the form of an extensible Unified Modeling Language (UML) profile used early in system development to represent the requirements, component structure, discrete state behavior, and parametrics (i.e., physical behavior as equations) of a system. AADL is an international industry standard extensible architecture modeling language for embedded software systems with graphical and textual representations, well-defined execution semantics, and an XML interchange format.

The focus of AADL is to model the software system architecture in terms of an application system bound to an execution platform. Most of the modeling effort is directed to the application-specific software, incorporating sufficient operating system specifics to ensure that the model is adequate (precise, complete, and realistic) to support evaluation of a system architecture and verification. Many operating system characteristics are already embedded in AADL semantics to support architectural modeling (communication mechanisms, thread behavior, etc.). It is designed to support an architectural MBE development life cycle, including system specification, quantitative analysis, system tuning, efficient integration, and upgrade, and the integration of multiple forms of analyses and extensibility for additional analysis approaches.

AADL supports component-based modeling of the architecture. Components have a type and one or more implementations. Software components include data (to represent data types and shared data areas), subprogram, thread (to represent concurrently executing tasks), and process (to

represent protected address spaces). The hardware components include processor, virtual processor (to represent schedulers and partitions), memory, bus, virtual bus (to represent virtual channels and protocols), and device (to represent physical system components). The system component is used to describe hierarchical grouping of components, encapsulating software components, hardware components and lower level system components within their implementations.

Interfaces to components and component interactions are completely defined. The AADL supports data and event flow, synchronous call/return and shared access. In addition it supports end-to-end flow specifications that can be used to trace data or control flow through components.

The core language supports modeling in several architectural views and addresses quality attribute analyses through explicit modeling of the application system and binding to execution platform components. Real-time analyses are supported through well-defined timing semantics for concurrent execution and interaction between the components, which are expressed through a hybrid automata specification in the standard document.

The AADL supports real-time task scheduling using different scheduling protocols. Properties to support General Rate Monotonic Analysis and Earliest Deadline First and other scheduling protocols are provided in the core standard. Execution semantics are defined for each category of component and specified in the standard with a hybrid automata notation.

Modal and configurable systems are supported by the AADL. Modes specify runtime transitions between statically known states and configurations of components, their connections and properties. Modes can be used for fault tolerant system reconfigurations affecting hardware and software as well as software operational modes.

The AADL supports component evolution through inheritance, allowing more specific components to be refined from more abstract components. Large scale development is supported with packages which provide a name space and a library mechanism for components, as well as public and private sections. Packages support independent development and integration across contractors.

AADL language extensibility is supported through a property sublanguage for specifying or modifying AADL properties, which can be used for additional forms of analysis. The AADL also provides an annex extension mechanism that can be used to specify sub-languages that will be processed within an AADL specification. An example is the Error Modeling Annex which allows specification of error models to be associated with core components. The combination of annex extensions and user defined property sets provide the means to integrate new specification capabilities and new analysis approaches to support the analysis tools and methods desired for multiple dimensions of analysis.

Safety-criticality is supported by AADL in a number of ways:

- AADL is strongly typed, e.g., if a processor specification indicates that it requires access to a Peripheral Component Interconnect (PCI) bus and an Ethernet, then only a PCI bus can get connected to the one bus access feature.
- The protected address space enforcement of processes and resource allocation enforcement of virtual processors ensure time and space partitioning.
- A safety level property on system components, initially used for major subsystems and later attached to more detailed components, is used to ensure that components with high safety criticality are not controlled by or receive critical input from low criticality components.
- AADL has a built-in fault-handling model for application threads and extends into an explicit representation of a health monitoring architecture, and fault management by reconfiguration, which is modeled through AADL modes.
- AADL supports fault modeling through the Error Model Annex standard, which allows us to introduce intrinsic faults, error state machines, and fault propagations across components – including stochastic properties such as probability of fault occurrence. These annotations to the architecture support hazard and fault impact analysis as well as reliability and availability analysis.
- The dynamic behavior of the architecture is represented by modes and further refined through the Behavior Annex standard of AADL. This allows us to apply formal methods such as model checking to validate system behavior.

Virtual System Integration and Validation

A key concept of virtual integration is the use of an annotated architecture model as the single source for architecture analysis. Thus, it becomes important to integrate the different analysis dimensions into a single architecture

model. An architecture model that is annotated with analysis-specific information can drive model-based engineering by generating the analysis-specific models from this annotated model. This allows changes to the architecture to be reflected in the various analysis models with little effort by regenerating them from the architecture model. This approach also allows us to evaluate the impact across multiple analysis dimensions. In other words (see Figure 2), analytical models are auto-generated from an architecture model with well-defined semantics and annotated with relevant analysis-specific information (e.g., fault rates or security properties). Any changes to the architecture throughout the life cycle are reflected in all dimensions of analysis (e.g., substitution of a faster processor to accommodate a high workload not only is reflected in schedulability analysis, but also may impact end-to-end response time, and requires revalidation of increased power consumption against capacity as well as possible change in mass).

A second key concept of virtual integration is the ability to interchange architecture models with their annotations through a standardized interchange format, such as the AADL XML standard. This allows system integrators and suppliers to exchange specifications and models of subsystems in a

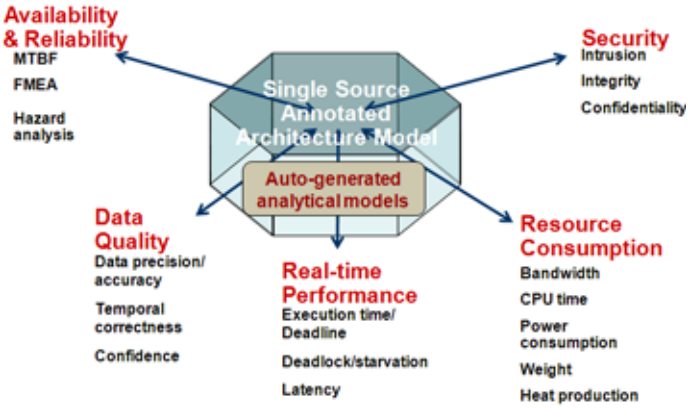


Figure 2: Analytical Models from an Annotated Architecture Model

standardized representation and facilitates the integrator to routinely predict and validate system properties early and throughout the development life cycle.

Virtual integration facilitates quantitative analysis, verification and validation using models. As such, it complements system testing by focusing on system issues due to concurrent execution and non-deterministic interaction of time-sensitive nature. To this end, AADL is being embraced by the V&V (validation and verification) community because it is a vehicle to express high-level system requirements and perform verification across a range of analyses. These analyses are made available through a number of tools such as the Eclipse-based open source OSATE[1], the commercial Stood environment [20], and tool chains put together in industry initiatives such as ASSERT [18], TOPCASED [17], and SPICES [19].

The Aerospace Vehicle Systems Institute (AVSI), a global cooperative of aerospace companies, government organizations, and academic institutions, has launched the System Architecture Virtual Integration (SAVI) program, which is a multiyear and multimillion dollar effort focused on developing a capability for system validation and verification through virtual integration of systems. Major players of the SAVI project include Boeing, Airbus, Lockheed Martin, BAE Systems, Rockwell Collins, GE Aviation, FAA, U. S. Department of Defense (DoD), Carnegie Mellon Software Engineering Institute (SEI), Honeywell, Goodrich, United Technologies, and NASA. The SAVI paradigm necessitates:

- An architecture-centric, multi-aspect model repository as the single source of truth
- A component-based framework to support model-based and proof-based engineering
- A model bus concept for consistent interchange of models between repositories and tools
- An architecture-centric acquisition process throughout the system life cycle that is supported by industrial standards and tool infrastructure

To establish cost-effective management and limit risks of the SAVI program, the Proof-Of-Concept (POC) project [21] has been executed as the first of multiple phases with the following goals:

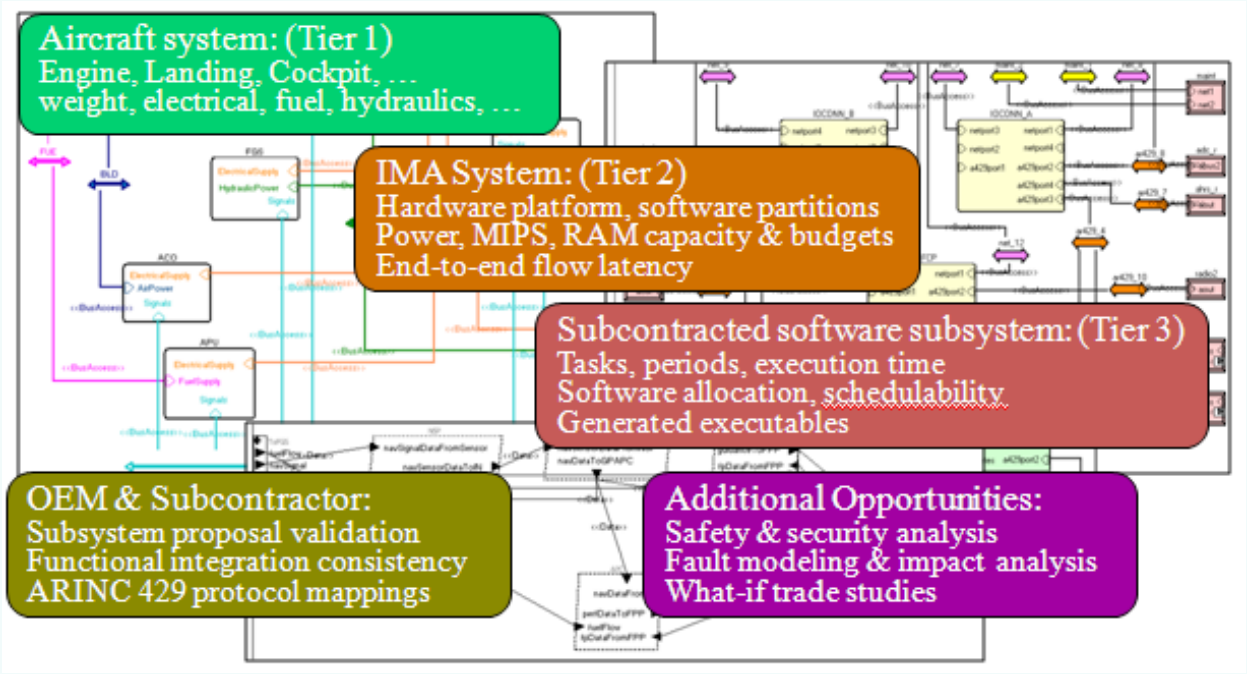
- Document the main differences between a conventional acquisition process and the projected SAVI acquisition process and identify potential benefits of the SAVI acquisition process
- Evaluate the feasibility and scalability of the multi-aspect model repository and model bus concepts central to the SAVI project

[1]OSATE is the open-source AADL tool environment (OSATE) that includes an AADL front-end and architecture analysis capabilities as plug-ins to the open source Eclipse environment.

- Assess the cost, risk, and benefits of the SAVI approach through a return on investment (ROI) study and development of a SAVI development roadmap

Together with the successful proof-of-concept demonstration of SAVI, an analysis to determine the economic gains was conducted, comparing the current acquisition and development practice (referred to "as-is") with a model-driven acquisition and development approach ("to-be"). Conservative estimates show that a cost reduction between 7.6%-26.1% is to be expected, which translates into a cost avoidance of $717M-$2,391M; the range is a function of the various levels of rework cost and anticipated defect removal efficiency.

VIRTUAL SYSTEM INTEGRATION CASE STUDY BY AVSI



For the proof of concept phase of the System Architecture Virtual Integration (SAVI) initiative, a multi-tier aircraft model was created and analyzed along multiple quality dimensions. The demonstration illustrated analyses at the Tier 1 level for system properties such as mass and electrical power consumption, and at the Tier 2 level focusing on the Integrated Modular Avionics (IMA) architecture by revisiting the previous analyses and adding computer resource analyses and end-to-end latency analysis across subsystems. The demonstration continued by showing AADL support to manage subcontracting with suppliers through a model repository. A negotiated subsystem specification could be virtually integrated and inconsistencies between supplier specifications such as inconsistent data representation, mismatched measurement units, and mapping into the ARINC 429 protocol could be detected. Three of the suppliers then developed a task-level specification of their subsystem and performed local validation through analysis. In one case, the subsystem was elaborated into behavioral specifications via UML diagrams, and an implementation in Ada. During this process the suppliers routinely delivered AADL models back to the airframer for repeated revalidation through virtual integration with increasing model fidelity. The demonstration model itself was developed by a team located in Iowa, Pennsylvania, France, and the UK utilizing the model repository located in Texas. At the end of the POC demonstration the SAVI team concluded that "the results of the demonstrations indicate that the SAVI concept is sound and should be implemented with further development".

Summary

In this article we have discussed architectural modeling, verification and validation as a means to realize virtual integration. The approach builds on AADL, which was designed for modeling software-intensive systems with real-time, embedded, fault tolerant, secure, safety-critical behaviors. Using the precise semantics of AADL, developers of subsystems can more readily share and understand architecture specifications. The language is designed to create a machine-analyzable, single architectural model annotated with precise notation. The language supports model creation and analysis throughout a full model-based development life-cycle including system specification, analysis, system tuning, integration, and upgrade. There are several advantages with this approach. Virtual integration activities replace traditional design reviews by:

- Recording subsystem requirements in an initial system model during request for proposals
- Validating supplier model compatibility and initial resource allocations during proposal evaluation
- Validating interfaces and functionality during preliminary design integration
- Validating performance during critical design integration

Conducting early and continuous virtual model integration based on standardized representations has a number of advantages:

- It ensures that errors are detected as early as possible with minimal leakage to later phases
- Models with well-defined semantics facilitate auto-analysis and generation to identify and eliminate inconsistencies
- Automated compatibility analyses at the architecture level scale easily
- Industrial investment in tools is leveraged through well-defined interchange formats

References

[1] The Economic Impacts of Inadequate Infrastructure for Software Testing (NIST Planning report 02-3, May 2002). Available: http://www.nist.gov/director/prog-ofc/report02-3.pdf. [Accessed: May 25, 2009].

[2] D. Galin, Software Quality Assurance: From Theory to Implementation. Pearson/Addison-Wesley, 2004.

[3] J. B. Dabney, "Return on Investment of Independent Verification and Validation Study Preliminary Phase 2B Report," NASA IV&V Facility, 2003.

[4] SAE International. SAE Standards: Architecture Analysis & Design Language (AADL), AS5506, November 2004. Available: http://www.sae.org/technical/standards/AS5506/1. [Accessed May 25, 2009].

[5] P. H. Feiler, D. P. Gluch, and J. J. Hudak. "The Architecture Analysis & Design Language (AADL): An Introduction," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (USA), CMU/SEI-2006-TN-011, 2006.

[6] Systems Modeling Language (SysML). www.sysml.org.

[7] Peter H. Feiler, "Challenges in Validating Safety-Critical Embedded Systems", Proceedings of SAE International AeroTech Congress, Nov 2009.

[8] "Ariane 5 Flight 501.", en.wikipedia.org/wiki/Ariane_5_Flight_501.

[9] ITunes Crashes on Dual-core Processors. discussions.apple.com/thread.jspa?messageID=1235236&.

[10] M. Jones, "What Really Happened on Mars", http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/.

[11] Report to Congressional Committees. "DOD's Goals for Resolving Space Based Infrared System Software Problems Are Ambitious." General Accounting Office. GAO-08-1073. September 2008.

[12] Daniel L. Dvorak. NASA Study on Flight Software Complexity. Technical Report NASA/CR-2005-213912, NASA Office of Chief Engineer Technical Excellence Program, March 2009.

[13] Nancy Leveson. System Safety Engineering: Back To The Future. Technical Report MIT, 2002.

[14] Daniel Jackson Ed. "Software for Dependable Systems: Sufficient Evidence?" Committee on Certifiably Dependable Software Systems, National Research Council. National Academic Press, ISBN: 0-309-10857-8, 2007.

[15] VHSIC (Very High Speed Integrated Circuits) hardware description language en.wikipedia.org/wiki/VHDL.

[16] Model checking hardware en.wikipedia.org/wiki/Model_checking.

[17] Maurice Heitz, Sebastien Gabel, Julien Honoré, Xavier Dumas/CS, Iulan Ober/IRIT and David Lesens/ASTRIUM-ST Supporting a Multi-formalism Model Driven Development Process with Model Transformation, a TOPCASED implementation, Proceedings of 4th International Congress on Embedded Real-Time Systems, ERTS 2008.

[18] Eric Conquet "ASSERT: a step towards reliable and scientific system and software engineering" Proceedings of 4th International Congress on Embedded Real-Time Systems, ERTS 2008.

[19] SPICES: Support for Predictable Integration of mission Critical Embedded Systems http://www.spices-itea.org/

[20] STOOD: a real time software toolset supporting a range of methods and languages including UML 2, HRT HOOD, AADL, C, C++, and Ada 95.. www.ellidiss.com.

[21] P. Feiler, J. Hansson, D. de Niz, L. Wrage, "System Architecture Virtual Integration: An Industrial Case Study", SEI Technical Report, CMU/SEI-2009-TR-017 November 2009.

About the Authors

Peter Feiler is a senior member of the technical staff at the Software Engineering Institute (SEI). He is the recipient of Carnegie Mellon's 2009 Information Technology Award for his work on the Society of Automobile Engineers (SAE) standard Architecture Analysis and Design Language (AADL). He joined the SEI in 1985 after a successful career at Siemens Corporation, where he conducted research and led a group in software technology.

Jorgen Hansson is a senior member of the technical staff at the Software Engineering Institute (SEI) and a full professor of software engineering at Chalmers University of Technology, Sweden. His current research interests include embedded real-time systems, architectural design and validation, and dependability in terms of performance, availability, reliability, and security.

Authors Contact Information

Email: Peter Feiler: phf@sei.cmu.edu

Email: Jorgen Hansson: hansson@sei.cmu.edu

Understanding Temporal Logic – The Temporal Engineering of Software

THE GOAL OF TEMPORAL ENGINEERING, USING COHERENT OBJECT SYSTEM ARCHITECTURE (COSA), IS TO ELIMINATE THE WASTE THAT IS OFTEN GENERATED DURING A CHANGE IN THE PROCESS. .

by Gordon Morrison

An extended BNF provides an unambiguous definition for a complete logic specification. The logic of BNF, by definition, flows in step with time, and that leads to a temporal definition for engineered software. Like most demand event-driven applications, logic is the key to creating a reliable system. Temporal logic, as presented in this paper, provides a clear way of creating step-by-step rules to control and respond to events. Temporal logic reduces the complexity and provides a clear derivation from the specification. Tracing logic has always been a bane in engineering. With temporal logic, trace is inherent in the architecture for every step. This paper shows a five-function calculator implementation using temporal logic.

Experienced domain specialists are usually the people that do the application analysis and put together a specification. These people have a track record in the business, but even the best analysts view things differently, and the specification can often change during the process. The goal of Temporal Engineering, using Coherent Object System Architecture (COSA), is to eliminate the waste that is often generated during a change in the process. The COSA engineering process discussed here applies to all logic, including keeping track of information relative to a process or as the result of a process. The COSA process can describe things as diverse as financial systems, calculating dividend investments, chemical process systems, or robot navigation on the surface of Mars. A simple calculator provides an example of the process that fits in this paper and can easily be understood. The process for the calculator example produces an initial specification in Backus-Naur Form (BNF), an expression of that specification in a tree format, and a COSA extended BNF table. The COSA engine can be used to create an execution trace from the COSA extended BNF table.

The Specification using BNF

To explain the COSA process, an example specification is created for a Texas Instruments calculator application. The user enters a number, an operator, a number, and 'equals' to get an expected result. The specification includes the ability to correct entries and clear everything. The specification also

REVIEW OF BNF AND REGULAR EXPRESSIONS

In BNF a vertical pipe allows selections between different possibilities, as in format A or format B. The less than and greater than symbols are used to contain and delineate symbol names that contain more than one word as in the <unary minus>. (Because of space restrictions I will be using <unary ->.) These structures will be read as 'unary minus contained'.

The star '*', plus '+', and question mark '?' characters are taken from regular expressions. The star is used to define zero or more occurrences of a symbol. The plus' is used to define one or more occurrences. The question mark denotes zero or one occurrence, that is, an optional argument. The listing <period> <digit> is the same as <period> & <digit> and means that both are required. The '&' is usually added to clarify meaning.

In a temporal sense, once an assignment has been made, all that is needed is one symbol. There can be no ambiguity. Until now, applications have developed BNF definitions in a spatial domain where the parser doesn't keep track of time; it only kept track of states.

includes the use of the second operand to calculate a percent of the first operand. The framework provided by the platform allows for a Graphical User Interface and real-time interaction. The final logic also provides an interface to the framework and the runtime behavior.

Backus-Naur Form (BNF)[1] is an incredibly powerful strategy used to lay out the application structure. Good examples of BNF usage can be found on the Internet.

BNF can be used to describe and generate state diagrams, tree views, and the complete application. Any change to the tree diagram results in changes to the BNF and state diagrams. Consistency is maintained because the traditional 'code' doesn't exist. Each approach is just a different view of the same piece of logic.

Building Temporal Logic

The example is to build a calculator as a Windows® application. Buttons on the calculator can be clicked with a mouse to create an event that results in an action or behavior. The COSA approach is to start the design with a BNF definition. The operational definition for the calculator is defined as four rules: Oper1, Oper8, Oper2, and then Result. These operational rules are defined with the temporal component running from left to right and top down.

```
Calc = <Oper1> <Oper8> <Oper2> <Result>?;
```

When defining the first operand, consider that a number is defined as a collection of one or more digits ranging from zero through nine. Numbers can exist as positive numbers '+number' and as negative numbers '-number'. Numbers can also have decimal portions containing a period followed by a number. The positive number is the default representation eliminating the need for the plus sign. The acceptable number formats for this example are:

```
Number = <digit>+ | <digit>+ <period> |
        <digit>* <period> <digit>+;
```

The first operand is defined as an optional unary minus followed by the Number definition. This definition of Number satisfies the requirements for entering a number at Oper1 and Oper2.

```
Oper1 = <unary ->? <Number> ;
```

The Operation is defined as four functions:

```
Oper8 = <Add> | <Sub> | <Mul> | <Div>;
```

The second operand is defined to have the same format as the first:

```
Oper2          = <Oper1>;
```

The result is more complex and is defined in terms of the percentage function, the equal sign, and a capability to continue additional operations:

```
Result = <Percent> | <Equals> | <Oper8> <Oper2>;
```

Additional operations include the chaining together of operations such as summing.

The general layout of a simple calculator is shown in Figure 1. The calculator design is implemented by dragging and dropping the various components onto the form. The labeled buttons have an associated On Click event. The two display boxes in the calculator don't have associated click events. All of these events call the calculator object with their respective event data. The 'Trace Logic' tab at the top contains a list box that displays trace or debug information. The 'Calculator' tab

Figure 1 Calculator

[1] ISO/IEC 14977:1996(E) *Information Technology – Syntactic Meta-language* – Extended BNF

contains a smaller list box that displays the entered data, the results of calculations, error messages, and all of the buttons.

Table 1: Initial BNF Specification

```
Calc   = <Oper1> <Oper8> <Oper2> <Result>?;
Oper1 = <unary ->? <Number>;
Oper2 = <Oper1>;
Oper8 = <Add> | <Sub> | <Mul> | <Div>;
Result = <Percent> | <Equals> | <Oper8> <Oper2>;
Number = <digit>+ | <digit>+ <period> |
      <digit>* <period> <digit>+;
```

The definitions of <unary ->, <period>, <digit>, <Add>, <Sub>, <Mul>, <Div>, <Equals>, and <Percent> are assumed.

BNF is similar to diagramming sentences. It is an approach that as processors of language we understand. The task is to keep a BNF specification simple, consistent, and synchronized in its different representations.

At this stage of the specification, time is implicit in the BNF definition. Time flows from left to right and is connected through all of the rules. Without an understanding of time an application must set and test flags to indicate what mode the application is in at any given time. An application that must set and test flags to indicate where it is has lost its primary dimension – time. The result of that lost dimension of time is a complex state chart.

The Logic of an extended BNF: Rules of Engagement

Table 2 shows the initial BNF specification in a vertical tree structure. There are four rules in this calculator. Each rule has steps that perform the necessary actions to complete the rule. The vertical tree will be expanded further to include the steps necessary to complete the rule.

Table 2: BNF Definition in Tree Format

Calc =	
Rule	**State / Step**
Oper1	= <unary ->?
	= <digit>*
	= <period>?
	= <digit>*
	= <CE>?
	= <C>?;
Oper8	= <+>?
	= <->?
	= <*>?
	= </>?;
Oper2	= <any>
	= <unary ->?
	= <digit>*
	= <period>?
	= <digit>*
	= <CE>?
	= <C>?;
Result	= <%>?
	= ⇔ ?
	= <Oper8>?
	= <ierr86>;

The next step in the COSA methodology is to add the True and False actions to each line. Consider rule Oper1 step <unary ->. This step states, "If this state is true, then negate the number and return to the Windows message loop for the next action. If the user entry is not a minus sign, then ignore the entry and move on to the next step in the rule to see if it's a number." In the implementation, Negate is a subroutine invoked when executing the step if the condition is true. This step can be expressed in a table entry.

Table 3: The First Step in the First Rule

Rule	**State / Step**	**True Action**	**False Action**
Oper1	= <unary ->?	Negate	Ignore

Table 4 shows the state transitions with respective actions for all steps in the BNF rules. The internal housekeeping actions that need to be taken may be nonobvious. The user doesn't recognize the internal housekeeping of moving and saving values for further actions. As a part of that housekeeping, the operand values must be kept. The first operand value is saved through an appropriate state and action. The state becomes <Push> and the action PshDsp is created to convert the first operand and save it for the anticipated calculation.

The General COSA Engine

The general COSA Engine consists of a 'while' statement containing at least the local engine state of **'On'** or **'Off'**. The 'while' statement may contain other global switches to control the preemptability of the engine.

An "if" statement is used to compare the dynamic state to the static state contained in the temporal BNF table.

The true section to the 'if' statement binds dynamically to the true behavior, followed by a true trace statement and the next true transitional location in the temporal BNF table.

The false section of the 'if' statement is identical to the true section in structure, and dynamically binds to the false behavior and next false time.

Using a COSA extended-BNF table, the model and the code are one and the same. The COSA extended-BNF table tells the engine what static state is being looked for, what to do in a static state when a match is made, what to do next after performing the true action, what to do if a static state does not match, and what to do next after performing the false action. The last column provides a trace of what the engine has done.

For the sake of discussion, assume the operation divide was clicked and the last true operation was a fractional number. That is, the action associated with the true behavior for rule rOpr1+3 was last executed, and the engine remains in the state associated with rule rOpr1+3. The divide entry doesn't match the static states associated with rOpr1+3, rOpr1+4, rOpr1+5, or rOpr1+6. So the engine executes the false behavior of Ignore for four state transitions and transitions to the state associated with the rule Oper8. The engine Ignores its way through the other three operators until it matches the divide. At this time, the engine transitions to the state associated with the next rule, Oper2. Four lines of code could have been added to turn off the engine - that is, to not process the next user entry - for each of the actions Addition, Subtraction, Multiply, and Division. Instead a single line of logic does the same task: the engine is explicitly turned off with Eng_Off for both the true and false actions.

Table 4: COSA Extended BNF Table

Rule	Static State	True Behavior	Next Rule	False Behavior	Next Rule	Trace
rOpr1,	iNeg44,	Negate,	rOpr1+1,	Ignore,	rOpr1+1,	100
rOpr1+1,	iDigit,	AnyNum,	rOpr1+1,	Ignore,	rOpr1+2,	101
rOpr1+2,	iDot59,	OneDot,	rOpr1+3,	Ignore,	rOpr1+4,	102
rOpr1+3,	iDigit,	AnyNum,	rOpr1+3,	Ignore,	rOpr1+4,	103
rOpr1+4,	iClEnt,	ClrEntry,	rOpr1,	Ignore,	rOpr1+5,	104
rOpr1+5,	iClear,	Clear,	rOpr1,	Ignore,	rOpr1+6,	105
rOpr1+6,	iPush,	PshDsp,	rOpr8,	PshDsp,	rOpr8,	106
// operations						
rOpr8,	iAdd43,	Addition,	rOpr2,	Ignore,	rOpr8+1,	500
rOpr8+1,	iSub44,	Subtract,	rOpr2,	Ignore,	rOpr8+2,	501
rOpr8+2,	iMul42,	Multiply,	rOpr2,	Ignore,	rOpr8+3,	502
rOpr8+3,	iDiv47,	Division,	rOpr2,	Ignore,	rErr,	503
// next number						
rOpr2,	iOff,	Eng_Off,	rOpr2+1,	Eng_Off,	rOpr2+1,	700
rOpr2+1,	iNeg44,	Negate,	rOpr2+2,	Ignore,	rOpr2+2,	701
rOpr2+2,	iDigit,	AnyNum,	rOpr2+2,	Ignore,	rOpr2+3,	702
rOpr2+3,	iDot59,	OneDot,	rOpr2+4,	Ignore,	rOpr2+5,	703
rOpr2+4,	iDigit,	AnyNum,	rOpr2+4,	Ignore,	rOpr2+5,	704
rOpr2+5,	iSave,	SaveDsp,	rOpr2+6,	SavDsp,	rOpr2+6,	705
rOpr2+6,	iClEnt,	ClrEntry,	rOpr2+1,	Ignore,	rOpr2+7,	800
rOpr2+7,	iClear,	Clear,	rOpr1,	Ignore,	rResu,	801
// equals						
rResu,	iPer37,	Percent,	rOpr1,	Ignore,	rResu+1,	901
rResu+1,	iEqual,	Equals,	rOpr1,	Ignore,	rResu+2,	902
rResu+2,	iChain,	Operate,	rOpr8,	Operate,	rOpr8,	903
rErr,	iErr86,	Unknown,	rOpr1,	Error,	rOpr1,	998

Understanding States

This COSA extended BNF Table can easily be displayed in a tree format with the state as a node and red and green lines going to the next respective behaviors. Various levels of expansion provide a complete view of the model. As Albert Einstein said, everything should be made as simple as possible, but not simpler. A definition of cohesion includes "do one thing and do it well". Each row in the COSA extended BNF table is a state. Each row provides for a true and a false behavior, a true and false next step, and a trace. It does this one

Table 5: Engine Code Segment

```
1 while (engLocal) AND (engGlobal) do
2 begin
3  if dynamicState = rRule[iTime].staticState then
4   begin
5     rRule[iTime].pTrueRule;          // True Behavior
6     True_Trace(iTime);// TRUE TRACE
7     iTime := rRule[iTime].iTrueRule;
8   end else
9   begin
10     rRule[iTime].pFalseRule;// False Behavior
11     False_Trace(iTime);// FALSE TRACE
12     iTime := rRule[iTime].iFalseRule;
13   end;
14 end;
```

state very well. Each row only does one state very well. This representation has a great side effect: accurate documentation shown as a spreadsheet, or in tree format either expanded or collapsed. And a narrative can be attached to each row, if desired.

Operational Analysis: Continuing the Model

The COSA extended BNF Table is a part of the model *and* is a part of the application. The COSA extended BNF is consistent with the State Chart. The choice is to view the BNF or the graphics. The graphics are created from the BNF. Changes to the BNF are reflected in the graphics and changes to the graphics are reflected in the BNF. They remain in sync and coherent.

States in the COSA state diagram (Figure 2) transition in chronological order with the possibility to repeat states and the ability to transition to an earlier state forming an iterative loop. All COSA states in this example are binary and are managed by the temporal cursor iTime. So that the rules will fit on a single line portrait orientation, programmer-abbreviated names like 'rOper8' and 'iNeg44' are used. Notice how the BNF in Table 6 has changed style and that it is synchronized with the state diagram of Figure 2.

Figure 2: COSA States

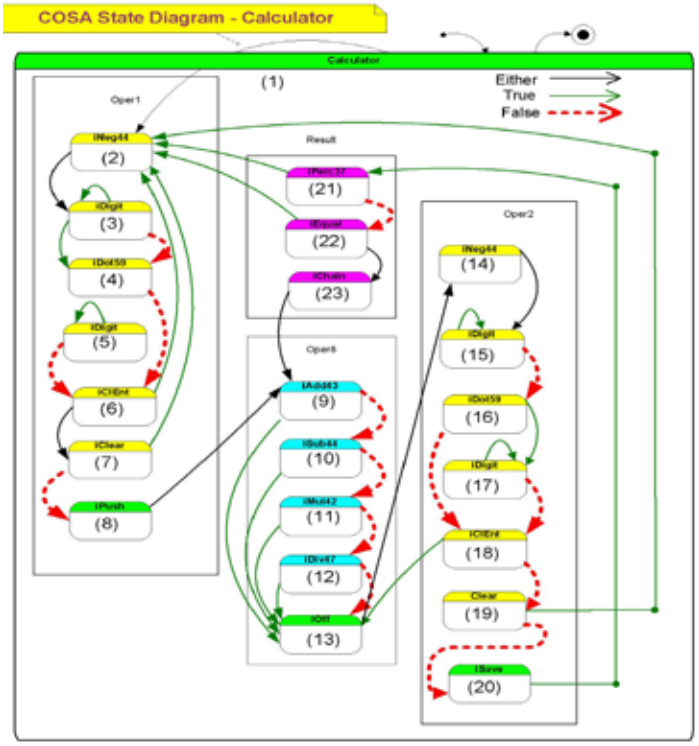


Table 6: Final BNF Specification

```
Engine    = <Calc>
Calc      = <Oper1> <Oper8> <Oper2> {<Result>}
Oper1     = {<iNeg44>} | <iDigit> | {<iDigit>}
            {<iDot58> <iDigit>} | ({<iClrEn>} |
            {<iClear>} ) & <iPush>
iDigit    = (<0> | <1 > | <2> | <3> | <4> | <5> | <6> |
            <7> | <8> | <9>)* & <iOff>
Oper8     = ( <Add> | <Sub> | <Mul> | <Div>) &
            <iOff>
Oper2     = {<iNeg44>} | <iDigit> | { <iDigit> }
            {<iDot58> <iDigit>} | <iSave>
            ({<iClrEn>} | {<iClear>} )
Result    = ((<Perc> | <Equals> ) & <iOff>)
```

The engine definition and a couple of engine controls have been added to the BNF. The explicit 'iOff' control stops the engine and allows control to return to the Windows message loop. In the definition of iDigit any combination of digit can be entered. Every time a digit is entered into the calculator the engine is turned off by the behavior, and control is returned

to the Windows message loop waiting for the next event. This allows the parser to build more complex numbers from a single digit.

The 'iPush' and 'iSave' are the display values being used in the calculation. The iPush state specifically keeps the first value available to be used in the subsequent operation calculation. The iSave state is specific for the potential to chain operations.

The COSA state diagram does not need narrative labels to describe transitions to other states. Because of the high level of cohesion, each state has only true or false transitions. Turning this state diagram over to ten developers will result in one set of logic (it's possible there may be ten different data manipulations, but many will be similar).

The red (dashed) and green (solid) lines represent the false and true logic respectively. Again, the model is the application whether the model is represented as BNF, tree diagram, state diagram, boxes-with-strings, or the code. The consistency is easily kept and nothing is thrown away.

A Step in Time with Trace

A trace with an example subtraction (Table 7) illustrates COSA and contrasts its results with a traditional state space approach. The example subtraction is divided into four parts:

1) Operand one is "–3.14159",
2) The operation is "– ",
3) Operand two is "–2.14159", and
4) Clicking on the "=" produces a result of "–1".

The calculator starts in the 'run' state at the initial time (t1). This simple calculator does not have a change sign button, so clicking on the '-' sign to negate a number can only happen before a number is entered as the first operand. This is the 'iNeg44' state t2. The COSA state diagram or the extended BNF Table shows the state name, e.g. "iNeg44", the true behavior "Negate" and the false behavior "Ignore", a true next state green arrow, and a false next state red arrow. These next state arrows manage time. There is only one temporal target for leaving the negate state, whether the state is true or false and that is the 'iDigit' state t3. Logically, it does not matter whether the first entity is a negative sign or not since the next state transition will likely be the whole portion of the number being built. The temporal component keeps track of its time sequence in the operational loop eliminating the need for any ambiguous group of flags or variables.

Figure 2 shows the 'iDigit' state t3 repeating. This happens until the button clicked on the calculator is not a number. Next the number 3 button is clicked and the next state remains the 'iDigit' state t3. The period button is clicked and the state transitions to the 'iDot59' state t4.

Like the 'iNeg44' state at t1, the 'iDot59' state t4 will transition to the next state 'iDigit' state t5. Since only one period is allowed in a valid number the transition must be to the next state. As the fractional portion of the number is entered, '14159' the state iterates on the 'iDigit' state t5 until the button clicked is not a number.

Table 7: COSA Temporal Calc Trace

	Time Behavior	Value
1	B= Negate;	N= -
2	B= AnyNum;	N= -3
3	B= Ignore;	N=
4	B= Onedot;	N= -3.
5	B= AnyNum;	N= -3.1
6	B= AnyNum;	N= -3.14
7	B= AnyNum;	N= -3.141
8	B= AnyNum;	N= -3.1415
9	**B= AnyNum;**	**N= -3.14159**
10	B= Ignore;	N=
11	B= Ignore;	N=
12	B= Ignore;	N=
13	B= PshDsp;	N=
14	B= Ignore;	N=
15	B= Subtract;	N= -3.14159
16	B= EngOff;	N= -3.14159
17	B= Negate;	N= -
18	B= AnyNum;	N= -2
19	B= Ignore;	N=
20	B= OneDot;	N= -2.
21	B= AnyNum;	N= -2.1
22	B= AnyNum;	N= -2.14
23	B= AnyNum;	N= -2.141
24	B= AnyNum;	N= -2.1415
25	B= AnyNum;	N= -2.14159
26	B= Ignore;	N=
27	B= SavDsp;	N=
28	B= Ignore;	N=
29	B= Ignore;	N=
30	B= Equals;	N= -1

An operation button is clicked, the subtract in this example, and the states transition through 'iClEnt' state t6 as false, through 'iClear' state t7 as false, through 'iPush' push display state t8 as false, through the 'iAdd43' addition state at t9 as false, and finally to the 'iSub44' subtraction state at t10 as true. The true operation rule sends the next temporal state to 'iOff' to turn the engine off at state t13, skipping state times t11 and t12. State 'iOff' engine off sends control back to the Windows message loop to wait for the next button click. Turning the engine off is determined by the BNF grammar definition ending a rule.

Each state in the COSA state diagram has only two possible transitions to the next state: true or false. The predictability of this binary state design greatly simplifies the transition logic, compared to the multiple state transitions allowed in a traditional approach[2], as shown in Figure 3. The traditional state diagram shows three transitions out of the Ready (1) state, four transitions from the Negated1 (3) state, and five transitions into the OpEntered (8) state. The temporal component is missing in the state diagram of Figure 3. The COSA approach produces a much simpler application. This is readily seen when the trace files of the two approaches are compared. In the COSA trace file shown above, the progress for entering "-3.14159" is at step number t9, set in bold. Compare trace number 47 in Table 8, also set in bold.

Table 8: Traditional Spatial Calc Trace

1,	calc,	
2,	calc,	
3,	calc,	
4,	ready,	
5,	ready,	
6,	ready	
7,	ready,	
8,	begin,	
9,	begin,	
10,	begin	
11,	begin,	
12,	begin,	
13,	negated1,	0
14,	begin,	
15,	calc,	
16,	begin,	
17,	ready,	
18,	ready,	
19,	negated1,	0
20,	negated1	
21,	negated1,	-0
22,	negated1,	-0
23,	int1,	-3
24,	negated1,	-3
25,	Oper1,	-3
26,	calc,	
27,	negated1,	-3
28,	Oper1,	-3
29,	Oper1	
30,	int1,	-3
31,	int1	
32,	int1,	-3
33,	int1,	-3
34,	frac1,	-3.
35,	int1,	-3.
36,	int1,	-3.
37,	frac1,	-3.
38,	frac1	
39,	frac1,	-3.
40,	frac1,	-3.
41,	frac1,	-3.1
42,	frac1,	-3.14
43,	frac1,	-3.141
44,	frac1,	-3.1415
45,	frac1,	-3.14159
46,	Oper1,	-3.14159
47,	**frac1,**	**-3.14159**
	.	
	.	
	.	

When the application does not have an inherent temporal component, that application must repeatedly test to determine where in the state sequence the execution is working. In Table 8, the negated1 (3) state is entered six times in the trace between steps 13 and 27. The int1 (6) state is entered six times in the trace sequence between 23 and 36. The Frac1 (7) state is entered ten times in the trace sequence between 34 and 47. This same functionality of entering "-3.14159"

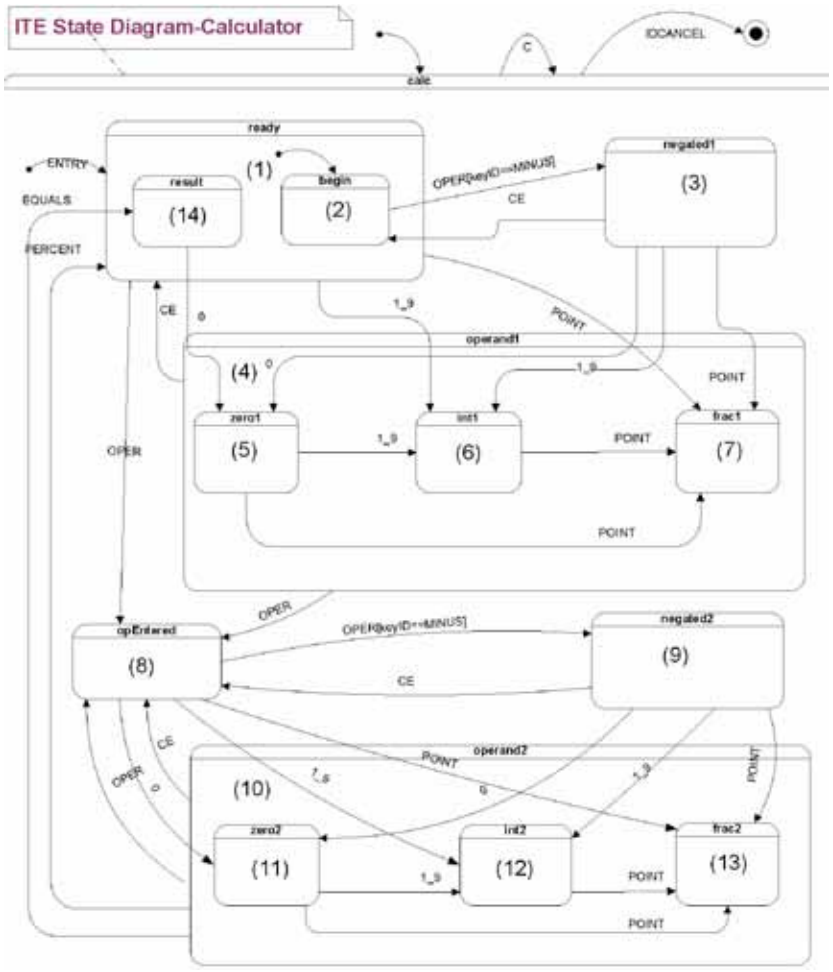
[2] *Practical Statecharts in C/C++*, CPM Books © 2002 Miro Samek, PhD

in COSA is accomplished between trace sequence t2 and t9 because the application knows where it is in time all of the time. The traditional approach never knows what event may occur next. Therefore, the design of an application must consider every possible event to be robust.

In the traditional approach with no temporal component, redundant testing must occur at each entry-point that an event takes place even if it is wrong. When an operation is entered incorrectly the application should report the error, recover, and be ready for a correct operation. Reporting provides learning feedback to the user. If the user is in a critical application, repeating the same wrong sequence without feedback can create disastrous results. Feedback allows the user to recognize the error and take corrective actions.

The user enters seventeen keys for this calculation example. The completed COSA trace shows 30 state transitions, compared to 95 state transitions in the complete spatial trace doing the same calculation. Traditional If-Then-Else (ITE) logic uses more than three times the number of states as COSA to obtain the same results.

Figure 3: ITE States



A Sampling of Efficiency Comparison

There are 1,010 lines of code in the traditional Spatial C++ calculator implementation compared to the 553 lines for the same functionality in the COSA implementation using Borland's v7.0 Delphi Environment. A second COSA calculator implemented in C++ produced 527 lines of code using Microsoft .NET 2003 Developer Studio.

The traditional spatial calculator used in this example was thoroughly designed. Designed applications are smaller than if they had not been designed[3]. This paper indicates that design is important but architecture is critical. A temporal architecture like COSA makes a big difference in the complexity and quality of the final application.

Figure 3, from the non-COSA approach to the calculator example, clarifies why software engineering has so many problems with software correctness and verification. Samek says: "Arriving at this statechart was definitely not trivial, and you shouldn't worry if you don't fully understand it at the first reading (I wouldn't either)." Statechart diagrams should make understanding the solution easier. Statecharts should be highly cohesive, defining one state and doing it very well. Abstract statecharts create complexity. COSA statecharts aren't trivial but are certainly understandable.

Summary

Complex projects are difficult enough; they don't need to be made more complex by the engineering approach.

Starting with an extended BNF provides an unambiguous definition for a complete logic specification. The logic of BNF, by definition, flows in step with time, and that leads to a temporal definition for engineered software. Like most

[3] Carnegie Mellon – Software Engineering Institute, "PSP II-Designing and Verifying State Machines", February 2005, Slide Number 42 & 43.

demand event-driven applications, logic is the key to creating a reliable system. Temporal logic reduces the complexity and provides a clear relationship to the specification. Tracing logic has always been a bane in engineering. With temporal logic, trace is inherent in the architecture for every step.

Consistency is easily kept with the use of a temporal architecture. The model can be represented as a BNF Table, tree diagram, state chart, or boxes with strings. A temporal approach eliminates waste, reduces the complexity, improves the quality, and reduces the costs of a project.

The efficiency of the temporal implementation of this calculator application is clear. In my book[4], I demonstrate the efficiency of the temporal approach in other applications. The lack of efficiency and the cost of spatial implementations have been well-documented and tolerated for nearly sixty years.

It's time to make a change to temporal engineering. There are significant opportunities for anyone that implements the tools to support COSA.

[4] *Gordon Morrison, Breaking the Time Barrier: The Temporal Engineering of Software.* Outskirts Press, Inc. Feb 2009.

About the Author

Gordon Morrison is an inventor, consultant, and author. He has developed real-time weather radar systems, invented the technology known today as multi-core and hyper-threading technology (U.S. Patent 4,847,755) licensed to Intel, IBM, TI, Motorola, Apple, and many others. Gordon has developed extensive database systems, developed high-performance communications systems, and developed micro-code for animation and weather radar. Mr. Morrison's main interest has been improving the quality of software and reducing complexity. To that end, he wrote, *Breaking the Time Barrier.*

Author Contact Information

Email: Gordon Morrison:

gordon.morrison@vsmerlot.com

Two Great Reliability Solutions from the RIAC & DACS.

System Reliability Toolkit

RIAC

The RIAC/DACS System Reliability Toolkit provides technical guidance in all aspects of system reliability, allowing the user to understand and implement techniques to ensure that system and product designs exhibit satisfactory hardware, software and human reliability, and to minimize the inherent risks associated with deficiencies in system reliability.

To purchase, please contact:
The Reliability Information Analysis Center
6000 Flanagan Road
Suite 3
Utica, NY 13502-1348
1-877-363-RIAC
http://theRIAC.org

The DACS Software Reliability Sourcebook

DACS
The Data & Analysis Center for Software

This first edition of the DACS Software Reliability Sourcebook provides a concise resource for information about the practical application of software reliability technology and techniques. The Sourcebook is divided into nine major sections, plus seven supporting appendices.

To purchase, please contact:
The Data & Analysis Center for Software
775 Daedalian Drive
Rome, NY 13441-4909
1-800-214-7921
http://thedacs.com

STN Article Submission Policy

The STN is a theme-based quarterly journal. In the past DACS has typically solicited specific authors to participate in developing each theme, but we recognize that it is not possible for us to know about all the experts, programs, and work being done and we may be missing some important contributions. In 2009 DACS adopted a policy of accepting articles submitted by the software professional community for consideration.

DACS will review articles and assist candidate authors in creating the final draft if the article is selected for publication. Note that DACS does not pay for articles published. Note also that submittal of an article constitutes a transfer of ownership from the author to DACS with DACS holding the copyright.

Although the STN is theme-based, we do not limit the content of the issue strictly to that theme. If you submit an article that DACS deems to be worthy of sharing with the community, DACS will find a way to get it published. However, we cannot guarantee publication within a fixed time frame in that situation. Consult the theme selection page and the Author Guidelines located on the STN web site (see **https://www.softwaretechnews.com/**) for further details.

To submit material (or ask questions) contact **news-editor@thedacs.com**

Recent and upcoming themes include:

- Earned Value
- Software Testing
- Project Management
- Model Driven Development
- Software Quality and Reliability
- Cyber Security

The first 50 people to send in a completed survey will receive a FREE DoD/IT Acronym CD from the DACS.

This valuable CD-ROM contains over 9,000 Department of Defense and Information Technology acronyms. There are hundreds of acronym lists available but none are as well done as this CD AND specifically targeted towards DoD and Information Technology. This unique-shaped CD-ROM plays in your computer's regular, hub-mounted, CD drive. You'll use this great resource over and over again. It's FREE, just for filling out our brief survey on the next page!

▼ Fold Here ▼

Data & Analysis Center for Software (DACS)

http://thedacs.com

▼ Fold Here ▼

SOFTWARE TECH NEWS FEEDBACK

1. Which volume of the Software Tech News did you receive? STN Vol 12, No. 4

2. When did you receive the newsletter? (month/year) _____

3. How satisfied were you with the CONTENT of the newsletter? (Article Quality)
❑ Very Satisfied ❑ Satisfied ❑ Neither Satisfied nor Dissatisfied ❑ Dissatisfied ❑ Very Dissatisfied

4. How satisfied were you with the APPEARANCE of the newsletter?
❑ Very Satisfied ❑ Satisfied ❑ Neither Satisfied nor Dissatisfied ❑ Dissatisfied ❑ Very Dissatisfied

5. How satisfied were you with the OVERALL QUALITY of the newsletter?
❑ Very Satisfied ❑ Satisfied ❑ Neither Satisfied nor Dissatisfied ❑ Dissatisfied ❑ Very Dissatisfied

6. How satisfied were you with the ACCURACY of the address on the newsletter?
❑ Very Satisfied ❑ Satisfied ❑ Neither Satisfied nor Dissatisfied ❑ Dissatisfied ❑ Very Dissatisfied

7. Approximately how much of the newsletter do you read?
❑ The entire issue ❑ Most of the content ❑ About half the content ❑ Briefly Skimmed ❑ Didn't Read

8. Would you read this newsletter in an E-mail newsletter format?
❑ Definitely ❑ Probably ❑ Not Sure ❑ Probably Not ❑ Definitely Not

9. How did you request the product or service?
❑ Phone Call ❑ E-mail ❑ DACS Website ❑ Subscription Form Other_____

10. Would you recommend the Software Tech News to a colleague?
❑ Definitely ❑ Probably ❑ Not Sure ❑ Probably Not ❑ Definitely Not

11. What topics would you like to see this newsletter devoted to? _____

_____

Comments (optional) _____

_____

REGISTER FOR THE FIRST TIME | UPDATE CURRENT SUBSCRIPTION

Name: _____ Position/Title: _____

Organization: _____ Office Symbol: _____

Address: _____ City: _____

State:_____Zip:_____ Country: _____

Telephone: _____ - _____ - _____Fax: _____ - _____ - _____ Email: _____

Functional Role: _____

Organization Type: ❑ Air Force ❑ Army ❑ Navy ❑ Other DoD _____
❑ Commercial ❑ Non-Profit ❑ Non-US ❑ US Government ❑ FFR&D ❑ Other _____

 *Note: You must give us your address to receive the CD.

ABOUT THE SOFTWARE TECH NEWS

STN EDITORIAL BOARD

Distribution Statement:
Unclassified and Unlimited

DACS
P.O. Box 1400
Rome, NY 13442-1400
Phone: 800-214-7921
Fax: 315-838-7130
E-mail: dacs@thedacs.com
URL: http://www.thedacs.com/

ADVERTISEMENTS

The Software Tech News is now accepting advertisements for future newsletters. In addition to being seen by the thousands of people who subscribe to a paper copy, an electronic version is available at the DACS website, exposing your product, organization, or service to hundreds of thousands of additional eyes every month.

For rates and layout information contact: news-editor@thedacs.com

COVER DESIGN

Wendy Butcher
Graphic Designer
ITT Corporation, DACS

ARTICLE REPRODUCTION

Images and information presented in these articles may be reproduced as long as the following message is noted:

"This article was originally published in the Software Tech News, Vol.12, No.4 January 2010."
Requests for copies of the referenced newsletter may be submitted to the following address:

Philip King, Production Editor
Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

Phone: 800-214-7921
Fax: 315-838-7130
E-mail: news-editor@thedacs.com
An archive of past newsletters is available at **www.SoftwareTechNews.com.** In addition to this print message, we ask that you notify DACS regarding any document that references any article appearing in the *Software Tech News.*

ABOUT THIS PUBLICATION

The Software Tech News is published quarterly by the Data & Analysis Center for Software (DACS). The DACS is a DoD sponsored Information Analysis Center (IAC), administratively managed by the Defense Technical Information Center (DTIC). The DACS is technically managed by Air Force Research Laboratory, Rome, NY and operated by ITT, Advanced Engineering and Sciences Division.

Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

Return Service Requested

STN Vol. 12, No. 4 January 2010 Model-Driven Development

IN THIS ISSUE