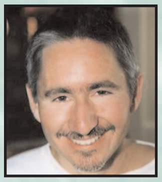
ISR **Institute for Software Research**

University of California, Irvine

Proceedings of the 2002 Workshop on the State of the Art in Automated Software Engineering



David F. Redmiles, Editor
University of California, Irvine
redmiles@ics.uci.edu

July 2002

ISR Technical Report # UCI-ISR-02-1

Institute for Software Research
ICS2 210
University of California, Irvine
Irvine, CA 92697-3425
www.isr.uci.edu

www.isr.uci.edu/tech-reports.html

Proceedings of the 2002 Workshop on the State of the Art in Automated Software Engineering

David F. Redmiles, Editor
Institute for Software Research
University of California, Irvine

redmiles@ics.uci.edu

ISR Technical Report # UCI-ISR-02-1

July 2002

Abstract: The 2002 Workshop on the State of the Art in Automated Software Engineering brought together leading researchers in the field to present their most recent or best work exemplifying automation in software engineering. The workshop focused on identifying emerging trends and challenges, such as: evolving requirements; software adaptability; validation of requirements and systems; complexity of software engineering tasks and artifacts; diversity of models and notations; and the need for efficient tool support and tool infrastructure. Position papers which were presented at the workshop comprise the Proceedings. A summary of the workshop was included in the *Proceedings of the17th IEEE International Conference on Automated Software Engineering (ASE 2002)* and is available at the workshop web site:

http://www.isr.uci.edu/events/ASE-Workshop-2002/

This Proceedings may also be referenced as Technical Report UCI-ICS-02-17.

Recent Experiences with Code Generation and Task Automation Agents in Software Tools

John Grundy[1,2] and John Hosking[2]

Department of Electrical and Electronic Engineering[1] and Department Computer Science[2]
University of Auckland, Private Bag 92019, Auckland, New Zealand
{john-g, john}@cs.auckland.ac.nz

1. Introduction

As software grows in complexity, software processes become more flexible yet complex, and more developers must co-operate and co-ordinate their work, software tools providing developers editing, reviewing and management facilities are not in themselves sufficient to ensure optimal project productivity. The number of tasks developers must manually perform with their tools, no matter how effective and efficient the tools are, continues to increase. Eventually this either overwhelms developers or leads to them not performing (often critical) tasks e.g. they avoid or reduce appropriate project management metrics capture, detailed design analysis and rigorous software testing.

The solution is provision of various forms of automation in the software tools developers use - the tools carry out perhaps a wide range of activities for the developer at appropriate times and inform the developer of results of actions in appropriate ways [2, 4, 5]. Many automation facilities have been used in tools, and in recent years more and more have tended to be added. Examples of automated tool support include information analysis (i.e. checking of software artefacts for consistency); autonomous agents (that perform tasks for users, including notification, information update and change propagation, and task co-ordination); code generation (generating user interface, data management and/or information process code from specifications); and

We have focused in recent years on two areas of automation in software tools: (1) generating code from high-level software specifications; and (2) utilisation of high-level software information by agents to support collaborative work, change management and component testing. From our experiences developing a number of software tools using these automation approaches, we have learned a number of lessons for further research in these areas. These include:

- the need to support software tool meta-model extension
- the need for on-the-fly enhancement of tool notations, event processing and code generation facilities
- support for software artefact change propagation and annotation
- the need to have reflective, high-level information to running software system components
- the continuing challenges of enhancing COTS tools with these kinds of automation facilities, including the need for sharable, extensible software information models for software tools and open tool infrastructure

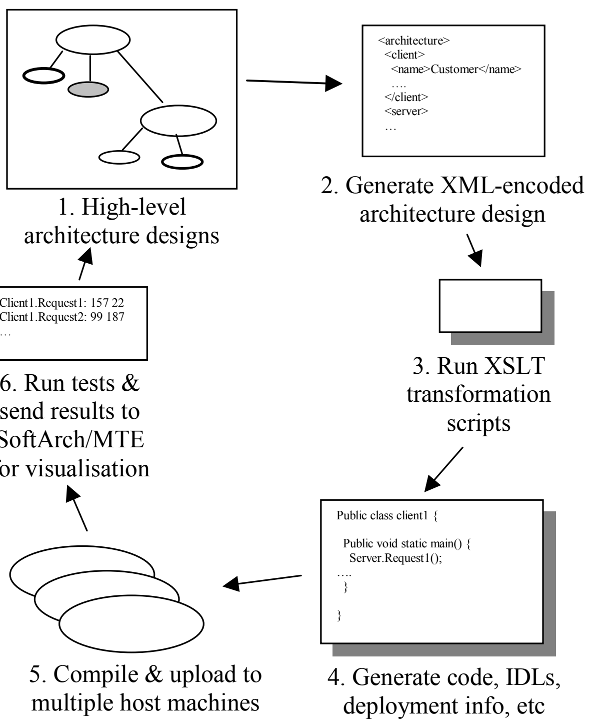
In the following two sections we briefly review some of our recent automated software tools. We give three examples of tools generating code from high level software descriptions, including a performance test-bed generator, a data mapping tool and an adaptive user interface generation tool. We describe three tools utilising event-driven software agents, including collaborative work components, requirements management and component testing tools. We then review the key lessons we have learned from this work and summarise future directions for our research on automated software tools.

2. Code Generation Examples

The three tools described in this section all generate large amounts of complex code from high-level descriptions of different aspects of software. Their ability to do so is dependent on the software information model they generate code from and the developer's ability to construct instances of this model via appropriate user interfaces and design metaphors.

2.1. SoftArch/MTE

Determining if software architecture designs will meet required performance benchmarks is very challenging [3, 14]. SoftArch/MTE is a distributed system performance test-bed generator [6]. It takes high-level descriptions of software architectures and generates client and server code that is automatically deployed and run to inform developers of likely architecture performance. As real code is generated and is deployed and run on real machines, quite accurate performance measures can be obtained very quickly by developers. Figure 1 outlines how Softarch/MTE works. A tool (SoftArch) is used to model software architectures at a high level of abstraction. This generates an XML-encoding of the architecture design including clients, servers, client requests, server operations, database operations and tables, and middleware and host characteristics. XSLT transformation scripts convert the XML into code and deployment scripts, which are uploaded and run on distributed client and server machines by deployment agents. Performance results are sent back to SoftArch/MTE and visualised with MS Excel™.


1. High-level architecture designs
2. Generate XML-encoded architecture design
3. Run XSLT transformation scripts
4. Generate code, IDLs, deployment info, etc
5. Compile & upload to multiple host machines
6. Run tests & send results to SoftArch/MTE for visualisation

```
<architecture>
  <client>
    <name>Customer</name>
    ….
  </client>
  <server>
  …
```

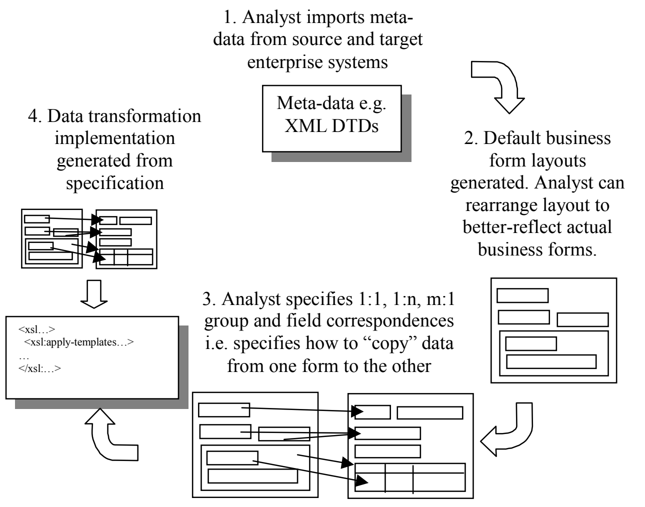
```
Public class client1 {

  Public void static main() {
    Server.Request1();
  ….
  }

}
```

```
Client1.Request1: 157 22
Client1.Request2: 99 187
…
```

Figure 1. SoftArch/MTE performance test-bed.

2.2. Form-based Data Mapping Specification

Implementing mappings between complex data structures is needed for various domains, including business-to-business e-commerce, but is time-consuming and hard to maintain with convention languages and tools [7]. We have developed a form-based data mapping tool that provides an environment in which non-programmer end-users (business analysts) specify correspondences between complex data models [12]. These data models are rendered as "business forms", and analysts specify form element correspondences using a drag-and-drop, form-copying metaphor. A transformation implementation is then generated from this high-level correspondence specification that when run transforms data in the source form format into target form data. Figure 2 illustrates this form-based mapping specification approach. Meta-data is imported from schema files or design tools. Business form representations are generated, and then analysts specify correspondences between form elements, effectively programming-by-demonstration of mappings. XSLT transformation scripts are generated by the mapping tool that implement the data transformations specified.

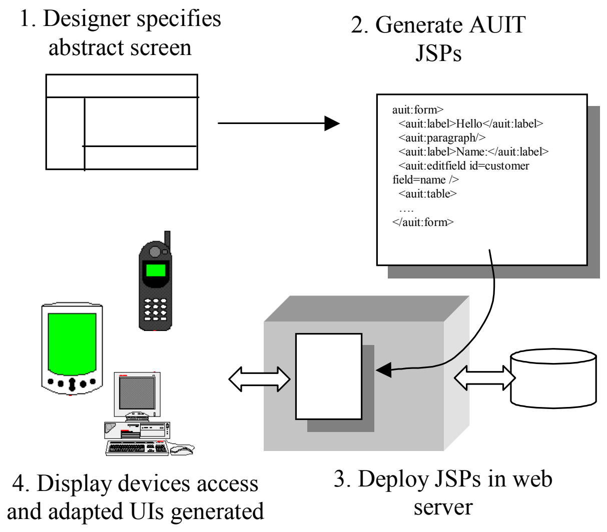

1. Analyst imports meta-data from source and target enterprise systems
Meta-data e.g. XML DTDs
2. Default business form layouts generated. Analyst can rearrange layout to better-reflect actual business forms.
3. Analyst specifies 1:1, 1:n, m:1 group and field correspondences i.e. specifies how to "copy" data from one form to the other
4. Data transformation implementation generated from specification

```
<xsl…>
  <xsl:apply-templates…>
…
</xsl:…>
```

Figure 2. Form-based data mapper.

2.3. Adaptive User Interface Technology

Many systems require thin-client interfaces that will run on multiple display devices and will suit different kinds of users and user tasks [13]. For example a customer accessing an on-line store via a wireless PDA will have quite a different interface for the same functions as a staff member accessing the system from a desktop PC web browser. Building such interfaces with conventional techniques results in large numbers of very similar server-side web page implementations. We have developed a GUI design tool and adaptive interface mark-up generator to make design and implementation of such adaptive interfaces easier [8]. Figure 3 illustrates this Adaptive User Interface Technology (AUIT) system. A designer uses an abstract representation of an interface to specify generic screen components, layout and interaction. This tool generates Java Server Pages with a set of custom tags describing the adaptable interface. When deployed in a web server and accessed by a user, the tags generate a user interface tailord to the accessing user's display device, user characteristics and current task.


1. Designer specifies abstract screen
2. Generate AUIT JSPs
4. Display devices access and adapted UIs generated
3. Deploy JSPs in web server

```
auit:form>
  <auit:label>Hello</auit:label>
  <auit:paragraph/>
  <auit:label>Name:</auit:label>
  <auit:editfield id=customer
field=name />
  <auit:table>
  ….
</auit:form>
```

Figure 3. Adaptive User Interface Technology.

3. Task Automation Examples

The following examples are of software tools we have developed that incorporate software agents to assist developers by automating various tasks. The agents are driven by event subscription or user request. The agents access and manipulate software artefact information for the user in various ways.

3.1. Collaborative Work Agents

Most software engineering tools require some degree of collaborative work support, though most hard-code this and are thus inflexible and require extensive engineering to build [1, 5]. We have developed a set of plug-in software agents that interact with tool client and server components to add collaborative work support to tools [9]. Figure 4 illustrates the basic structure of our approach. Collaboration agents support collaborative editing, group awareness and version control. Communication agents support messaging, annotation and dialogue between developers. Co-ordination agents provide change notification actioning, locking, to-do list task scheduling and even workflow co-ordination. The agents can be plugged into or removed from tools at run-time. In order to add these agents to tools, they need to determine various user interface, distribution and persistency support of the tool components. This is done by having the tool components publicise "aspects" which describe this information and can be introspected by our collaboration agents.

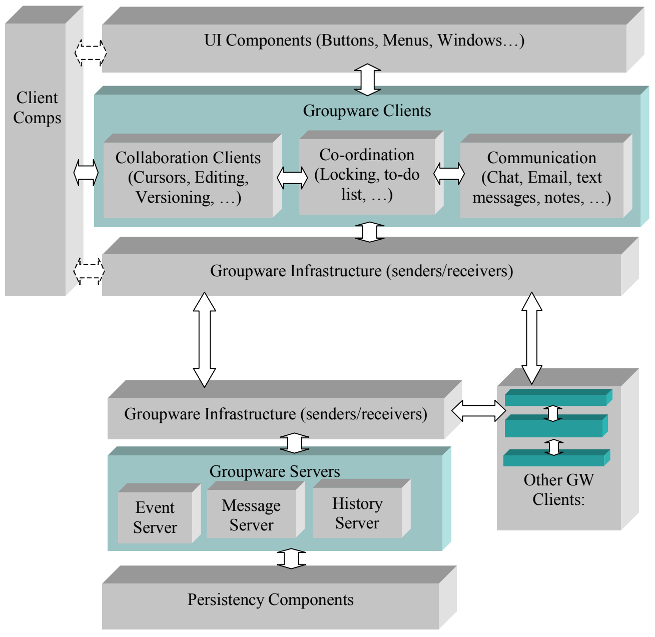

Client Comps
UI Components (Buttons, Menus, Windows…)
Groupware Clients
Collaboration Clients (Cursors, Editing, Versioning, …)
Co-ordination (Locking, to-do list, …)
Communication (Chat, Email, text messages, notes, …)
Groupware Infrastructure (senders/receivers)
Groupware Infrastructure (senders/receivers)
Other GW Clients:
Groupware Servers
Event Server
Message Server
History Server
Persistency Components

Figure 4. Collaborative work components.

3.2. Requirements Management Agents

Based on an empirical study of software engineering practitioners use of abstract information models [17], we have built a prototype tool for managing relationships between functional and non-functional requirements, use case models, and black-box test plans.

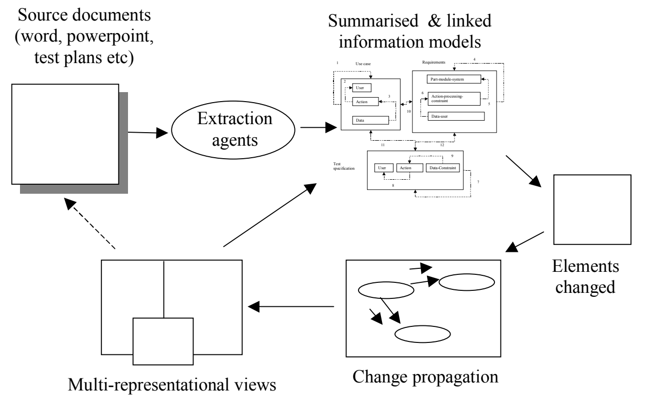

Source documents (word, powerpoint, test plans etc)
Summarised & linked information models
Extraction agents
Elements changed
Multi-representational views
Change propagation

Figure 5. Requirements management.

This environment contains software agents that extract information about these three different abstract software representations, summarising the key parts of each information model. Other agents create implicit links between elements in different representational models or allow developers to create explicit links and modify artefact information. When elements in one representation change, descriptions of these changes are captured and sent to other models. Developers can view the impacts of these changes, trace sources of changes, and view information from different representations in multi-representational views. We hope to provide other agents that can update source, 3rd party software artefact documents in the future. Figure 5 outlines the main facilities of this prototype tool.

3.3. Aspect-oriented Component Validation Agents

Validating that deployed software components meet their required functional and non-functional constraints is very difficult [11, 15]. We have developed software agents that inspect the constraints on deployed software components and perform validation tests on these components. The components are designed with the aspect-oriented component engineering method [10]. Their implementations have information characterising system aspects, such as persistency, distribution, security and transaction processing characteristics, associated with them as XML documents. Our validation agents inspect these component aspects and formulate tests to ensure the component's aspect-encoded constraints (functional and non-functional) are met in their current deployment situation. Some agents deploy 3rd party testing tools, like SoftArch/MTE, to carry out complex tests and analyse the results produced.

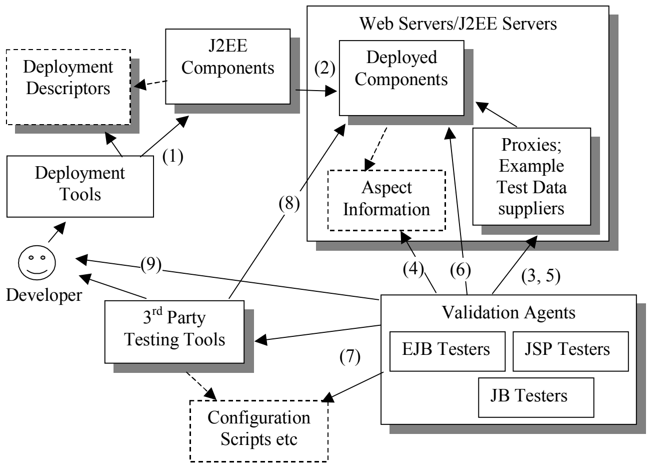


Figure 6. Agent-based component testing.

4. Key Issues and Future Research

We have identified several key issues when building the tools described in the previous two sections. These are summarised below, along with some of the research directions we are investigating to make the development of such automated software tools easier and more feasible.

4.1. Software Information Model Extension

Many of our tools require extensible meta-models in order that their capabilities can be enhanced by developers as required. For example, we have added new kinds of architectural characteristic support to SoftArch/MTE as we have extended the tool to support a wider range of target test bed generation (e.g. message-based systems and web-based interfaces). Similarly, the information models the requirements management agents use needs to be extensible as different users have different degrees of detail in each model they are interested in capturing.

Our experience with these tools has indicated that ideally many automated software tools will have software information models that can be extended as required. Versioning these information models and ensuring compatibility between old and new models often may need to be supported. We are developing a new software meta-tool with a fully extensible meta-model.

4.2. Tool Notation and Behavioural Extension

Many of our tools need to allow developers to add additional notational representations (in order to make use of meta-model extensions or support new kinds of artefact views), and similar require behaviour extensions (such as new code generation extensions or constraints on models built). Examples include extending the modelling notations of SoftArch/MTE, AUIT and our requirements modeller, and adding new target code generation for SoftArch/MTE, our form-based mapper, AUIT and component validation agents.

Most of our tools have very limited notational support, and limited run-time behavioural modification. This results in frustrating turn-around time when enhancing tools and requires developers to have in-depth knowledge of tool internal structures to make any enhancements. Our new meta-tool architecture supports flexible view notation definition as well as a wide range of run-time behaviour enhancement by allowing developers to incorporate new code into the tools at run-time. This code includes constraint checking, event/action rules and XSLT transformation scripts which we have found very useful for implementing code generation.

4.3. Change Propagation and Artefact Annotation

Many of our tools need to track changes made to software artefacts. These include our requirements modeller, collaborative work supporting agents and component validation agents. SoftArch/MTE and our requirements modeller require support for specifying links between model elements and for annotating elements with additional, semi-structured information such as design rationale and change explanation.

While many software tools have moved to adopting publish-subscribe event-based architectures the use of these architectural facilities is still relatively limited. We have found using this architecture important in driving many task automation agents, particularly those supporting collaborative work. The ability to link, refine and annotate software artefacts in many of our tools is important and hence should be supported within a tool infrastructure.

4.4. Reflection Information

Some software tool automation facilities need access to detailed information about running tool components. Examples include the plug-in collaborative work agents, the data mapping tool and the aspect-based component validation agents. The collaborative work agents need to adapt tool component interfaces to integrate new facilities and make use of publicised component event mechanisms. The data mapper needs to obtain meta-data information from source and target structures. The validation agents need to determine what the requirements on deployed components are in order to perform appropriate tests to validate these are met.

In our recent work we have developed a mechanism to annotate software components with information about the "aspects" of a system they provide or require services [10, 9]. This is used by our collaborative work and validation agents. Interestingly, tool automation is required in order to generate this information from annotated component design models. We are investigating adding these aspects as annotations to SoftArch/MTE architecture designs to better-organise the many properties of some of its architecture abstractions.

4.5. Tool Integration

Software tool integration has been a long-standing problem for tool developers and those developing automated support for tools [16, 18]. Some of our tools utilise 3rd party systems in limited ways e.g. SoftArch/MTE uses MS Excel™ to visualise performance data and our requirements modeller extracts summarised data from save files. Many of the automation support described in our tools above could however be very useful if integrated into 3rd party, commercial software development tools. For example, SoftArch/MTE test beds could be generated from (greatly) annotated Rational Rose™ deployment diagrams; mapper transformations from cross-linked MS Access™ screen designs; collaborative work agents potentially added to a very large range of tools; and inter-representation requirements change management added to integrate several different tools.

Three key problems preventing such enhancements of existing tools we have identified are lack of agreed, high-level tool information models that can be shared between tools, lack of adequate tool event and operation APIs, and sufficiently open technologies implementing these APIs and run-time inspection facilities allowing other tools to discover them. We are investigating "componentising" some of the tool automation facilities outlined in the previous section in order to add them to COTS software tools and to allow easier use of these tools by our own.

5. Summary

We have been developed a range of software tools with automation features, in particular ones that generate code from various high-level software information models and ones that leverage "agents" to perform various task automation facilities for developers. Some of the key issues in building such tools we have encountered include the need to support extensible information models, notations and event handling behaviour, change propagation and information element annotation capabilities, detailed reflective information encoded with software components, and tool integration. We are developing a meta-tool with these capabilities to enable us to better support the construction of various automated software tools, and developing various integration components to support the integration of our new tools and enhancement of existing COTS software tools with automation.

References

1. Bandinelli, S., DiNitto, E., and Fuggetta, A. Supporting cooperation in the SPADE-1 environment. *IEEE Transactions on Software Engineering,* vol. 22, no. 3, December 1996, 841-865
2. Fischer, G, Domain-oriented design environments, In *Proceedings of the Seventh Knowledge-Based Software Engineering Conference*, McLean, Virginia, 1992, pp. 204-213.
3. Gorton, I. And Liu, A. Evaluating Enterprise Java Bean Technology, In *Proceedings of Software - Methods and Tools*, Wollongong, Australia, Nov 6-9 2000, IEEE CS Press.
4. Green, C., Luckham, D., Balzer, R., Cheatham, T., Rich, C. Report on a Knowledge-Based Software Assistant, Technical Report RADC-TR-83-195, Rome Air Development Center, August 1983, Reprinted in: C.H. Rich, R. Waters (eds.): *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann Publishers, Los Altos, CA, 1986, pp. 377-428,.
5. Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. An architecture for decentralised process modelling and enactment, *IEEE Internet Computing*, vol. 2, no. 5, September/October 1998, IEEE CS Press.
6. Grundy, J.C., Cai, Y. and Liu, A. Generation of Distributed System Test-beds from High-level Software Architecture Descriptions, In *Proceedings of the 2001 IEEE Automated*

Software Engineering Conference, San Diego, 26-29 Nov 2001, IEEE CS Press, pp. 193-2000.

7. Grundy, J.C., Mugridge, W.B., Hosking, J.G. and Kendall, P. Generating EDI Message Translations from Visual Specifications, In Proceedings of the 2001 IEEE Automated Software Engineering Conference, San Diego, CA, 26-28 Nov 2001, IEEE CS Press.
8. Grundy, J.C. and Zou, W. An architecture for building multi-device thin-client web user interfaces, In *Proceedings of the 14th Conference on Advanced Information Systems Engineering*, Toronto, Canada, May 29-31 2002, Lecture Notes in Computer Science.
9. Grundy, J.C. and Hosking, J.G. Engineering plug-in software components to support collaborative work, to appear in *Software – Practice and Experience*.
10. Grundy, J.C. Multi-perspective specification, design and implementation of software components using aspects, *International Journal of Software Engineering and Knowledge Engineering*, Vol. 10, No. 6, December 2000, pp. 713-734.
11. Haddox, J.M., Kapfhammer, G.M. An approach for understanding and testing third party software components, In Proceedings of 2002 Annual Reliability and Maintainability Symposium, Seattle, WA, 28-31 Jan. 2002, IEEE CS Press.
12. Li, Y., Grundy, J.C., Amor, R. and Hosking, J.G. A data mapping specification environment using a concrete business form-based metaphor, In *Proceedings of the 2002 International Conference on Human-Centric Computing*, IEEE CS Press.
13. Marsic, I. Adaptive Collaboration for Wired and Wireless Platforms, IEEE Internet Computing (July/August 2001), 26-35
14. McCann, J.A., Manning, K.J. Tool to evaluate performance in distributed heterogeneous processing. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing*, IEEE, 1998, pp.180-185.
15. McGregor, J.D. Parallel Architecture for Component Testing. Journal of Object-Oriented Programming, vol. 10, no. 2 (May 1997), SIGS Publications, pp.10-14..
16. Meyers, S. Difficulties in Integrating Multiview Editing Environments, IEEE Software, **8** (1), 1991, pp. 49-57.
17. Olsson, T., Runeson, P., Software document use: A qualitative survey, Technical report, Dept. of Communication systems, Lund University.
18. Reiss SP. The Desert environment. *ACM Transactions on Software Engineering & Methodology*, **8** (4), Oct. 1999, pp.297-342.

Open Modeling in Multi-stakeholder Distributed Systems: Model-based Requirements Engineering for the 21st Century[1,2]

Robert J. Hall
AT&T Labs Research
180 Park Ave, Bldg 103
Florham Park, NJ 07932
bob-3OpenModel-@channels.research.att.com

Abstract

Multi-stakeholder distributed systems (MSDSs), wherein the constituent nodes are designed or operated by distinct stakeholders having limited knowledge and possibly conflicting goals, challenge our traditional conception of requirements engineering. MSDSs, such as the Internet email system, networks of web services, and the Internet as a whole, have globally inconsistent high-level requirements and, therefore, have behavior which is impossible to validate according to the usual meaning of the term. We can sidestep this issue by changing the problem from "does the system do the right thing" to "will the system do the right thing for me (now)?" But to solve that simpler problem, we need a way to predict behavior of the system on inputs of interest to us. OPENMODEL *proposes to solve this by establishing open standards for behavioral modeling: each node will provide via http (or through a central registry) a behavioral model expressed in terms of shared domain-specific function/object theories. A tool will support validation by assembling these models and simulating, animating, or formally analyzing the assembled model, helping the user to detect unfavorable behaviors or feature interactions in advance. This paper presents the* OPENMODEL *proposal and discusses its potential advantages, challenges, and limitations.*

1. Multi-stakeholder Distributed Systems

Definitions and Examples. Requirements engineering has traditionally assumed that the *system* to be designed is under the control of a single stakeholder who (at least in principle) determines a consistent set of requirements. Modern distributed systems, however, do not fit this mold, so requirements engineering must adapt to handle them.

A *multi-stakeholder distributed system (MSDS)* is a distributed system in which subsets of the nodes are designed, owned, or operated by distinct stakeholders. The nodes of the system may, therefore, be designed or operated

- in ignorance of one another, or
- with different, possibly conflicting goals.

The Internet electronic mail (email) system is an MSDS. Different entities (companies, universities, internet service providers (ISPs), and individuals) operate servers of the email system. Individual users (private and commercial) act as clients, sending messages and receiving them via the servers. Each of these entities operates its node(s) according to its own goals and priorities, using software packages designed at diverse times by different groups of developers each having more or less limited knowledge of each other and of the governing standards documents (Internet RFCs).

The emerging field of Web Services provides more examples of MSDSs. A Web Service is simply a service on a network which performs some function through a published remote procedure call interface, using the world-wide web's HTTP protocol as its "transport layer", typically using a distributed object protocol, such as SOAP, on top of that. The intended benefits of this architecture include the ability to dynamically and easily compose these services into useful business functionalities, for example, sending orders and payments down through supplier trees and invoices and services back up. Each web service is built, owned, and operated by a distinct entity having its own capabilities, knowledge, goals, and priorities.

Other examples of MSDS include the Internet as a whole, where hosts designed and governed by literally millions of different stakeholders interoperate at the extremely low level of the Internet Protocol, and today's telephone network, where many companies of widely varying service

[1]A preliminary version of the ideas in this paper were discussed in a talk given at the February 2002 meeting of IFIP Working Group 2.9.

[2]© Copyright 2002, AT&T Corp.

scope and geographical extent must interoperate their nodes at the signaling and voice transport level, yet are governed by their own business and national priorities. Clearly, in a highly interconnected world, MSDSs will be ubiquitous.

MSDS: No Such Thing as Requirements? From a requirements perspective, the interesting thing about an MSDS is that *it typically has inconsistent high level requirements.* Different stakeholders have conflicting goals which, in turn, place inconsistent requirements on system behavior. Of course, to operate at all there must be some level of consistency so that the nodes can communicate information.

For example, the Internet email system's consistent requirements include the SMTP, POP3, and IMAP mail protocols, as well as message format definitions (such as defined by RFC 822). However, it has many examples of inconsistency as well. Spammers (senders of unwanted email) want their messages to get to as many people as possible, yet innocent users want to avoid receiving spam messages. Users want the content and (often) recipient identities of their messages to remain private, yet various jurisdictions (such as the U.S. government) feel it is their right to snoop ISP traffic to watch for criminal activities or intent. And users who send Word (or other executable files) as attachments enjoy the convenience, yet users whose files are exposed or destroyed or whose service is denied by email viruses would like to prohibit such attachments. In each of these three examples, one set of stakeholders wants a capability while another wants to deny it.

In the Web Services domain, consistent requirements include the HTTP, SOAP[7], UDDI[8], WSDL[10], and WSCL[9] protocols. At a higher level, however, different stakeholders place conflicting requirements. For example, an end user of a web service may be forced to supply personal information. This end user typically intends that this be used only as minimally necessary for order fulfillment, billing, and customer support. Some web service providers, however, may store this information in databases and reuse it in ways that would not be agreed to by the end user. Another area of inconsistency lies in the terms used to define the specification of the service (often expressed in WSDL, but possibly just in natural language on the opening web page of the service). Unspecified measurement units, or ambiguous evaluative terms like "reliable", "accurate", "best in class", etc, can be used inconsistently between client and service provider. Such terminological inconsistencies will inevitably arise until standard ontologies are developed and required.

Validation Without Requirements? If an MSDS has inconsistent requirements, how could we possibly hope to validate its design or operation? More precisely, *no* system satisfies an inconsistent requirement. Leaving aside the inconsistency problem for the moment, however, no single individual even *knows* the requirements of all the stakeholders of a typical MSDS. In fact, for the large scale MSDSs discussed above, no one even knows how all the nodes (components, features) behave in detail. And all these factors change rapidly and uncontrollably in an MSDS.

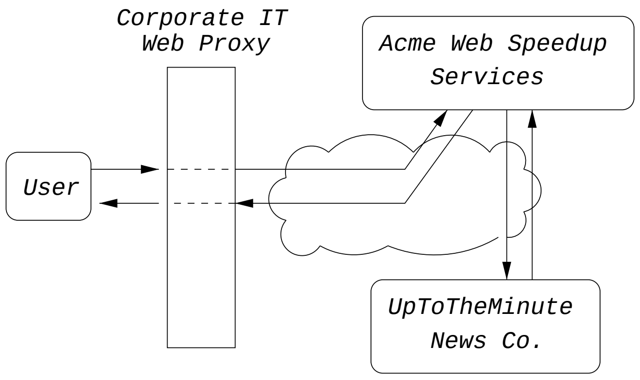


Figure 1. A problematic web service configuration.

Consider the simple web services scenario depicted in Figure 1. Here we assume that all of the end user's web accesses are sent by the corporate IT department through the Acme Web Speedup Service, a caching proxy service. There are four stakeholders here, one for each box in the diagram. The user wishes timely access to the UpToTheMinuteNews web service for the very latest news updates. Corporate IT, on the other hand, wishes to speed up "the average web access" for all users. UpToTheMinuteNews provides (and charges money for) the latest news updates. Acme Web Speedup knows it is appropriate for speeding up access to relatively static pages. The end result is that the user fails to get up to the minute news updates, even though he has paid for them. The reason is that no individual in the system knows the behaviors and requirements of all the nodes and so there is no one to diagnose the problem: a cached page is not timely.

Rather than tackle what appears to be an intractable problem, I propose we change the problem to match the way in which MSDSs are designed and used today.

> **Key Idea:** Change the validation problem from "does the system do the right thing?" to "does the system do the right thing *for me (now)*?"

By doing this, we get rid of the inconsistencies, because there is now only one stakeholder who matters and that stakeholder can (in principle) define a consistent set of requirements. We are still left with significant difficulties, of course. First, the (now single) stakeholder must have a way to find out what the various parts of the system do in order to validate that the system will behave desirably. Existing descriptions of node functionalities are often ambiguous, informal, incomplete, lacking in detail, or even purposely incorrect (due to hidden agendas). For example, web services are often described in ways including natural language passages, which are subject to the well known ambiguities and informalities of NL. Often, all that is known about a remote email service is that it is available on TCP port 25. While that usually implies it will accept email messages using the SMTP protocol, it is no help in discerning what will happen to a message once it is accepted by it.

The next section proposes an approach to solving this *ignorance problem.* Using it, we can effectively reduce the MSDS validation problem to more familiar model-based validation problems, which can be attacked by known techniques. We are still, of course, subject to the "usual" software engineering validation problems of state and theory explosion, and feature/component interaction; however, we have large bodies of research and a developing base of tools to attack these more traditional problems.

2. OPENMODEL: Going Beyond Modularity

The traditional way in which components publish their capabilities so that others (developers, users) can find out their behavior has been through interface descriptions in languages such as CORBA's IDL or WSDL[10] and WSCL[9] for web services. However, interfaces, or even allowed interaction sequences as definable within WSCL, do not provide enough information to validate the behavior of a complex node within an MSDS. For example, almost all email servers satisfy the well known SMTP protocol server behavior as defined in RFC 821, and yet consider the wide range of behaviors possible once a message is accepted: relaying, spam filtering, forwarding, decryption, and even anonymous remailing. Clearly, we must go well beyond simple component interface descriptions in order to support validation of requirements within MSDSs.

OPENMODEL solves the ignorance problem of MSDSs through *open modeling*. The key ideas of OPENMODEL are

- Each MSDS node has an *executable specification model*;
- Each node serves this online in a standardized XML-based format, either directly via HTTP, or through a central registry;
- A tool can retrieve models of the relevant nodes of the MSDS and assist the user in validating single behaviors or classes of behaviors of the composed system.

A node is depicted schematically in Figure 2. The actual node component is abstracted by the executable spec model through the abstraction function **A**. Every concrete input

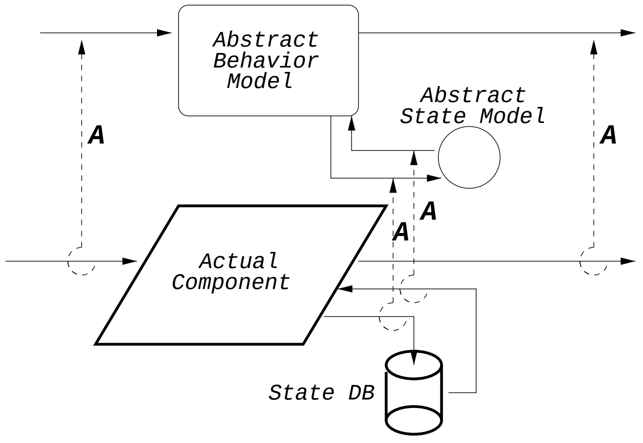


Figure 2. An OPENMODEL MSDS node.

sequence can be abstracted and simulated by the model, and outputs (and state read/update actions) can also be abstracted and compared with simulation outputs.

Potential Benefits. There are several potential benefits of OPENMODEL. First, once the models are retrieved and a composite model assembled, the user can validate that the system behaves desirably. Most basically, the user can create, simulate, and animate concrete behavior scenarios. Scenario coverage measurement tools and other (heuristic) testing methods can help the user gain confidence in sets of behaviors. But more systematic approaches, such as theorem proving and model checking can also be applied to the precise, executable models, finding bugs or increasing confidence in infinite sets of behaviors. Note that this can be done during design of a node (possibly before its implementation even exists) or during use by end users.

Another benefit of OPENMODEL is "contract enforcement". If a node claims to obey its model, an observer can sometimes actually verify that the open model correctly abstracts the actual input/output sequences. There are many ways in which this may be possible, for example when one upgrades a component to a new version (or one by a different vendor). One can run the old component in parallel with the new and compare the abstract outputs (using the abstraction function **A** to map actual inputs and outputs) with each other to see if the new one matches the old one.

Another benefit of OPENMODEL is its support for reuse. A component model provides checkable formal documentation of behavior. It also allows checking proposed uses of the component for feature interactions.

Finally, we should note that the model published by a component can be the same as (or formally related to, such as by abstraction) a formal model used for validating the node's behavior in isolation. That is, we get a kind of "2 for the price of 1 deal" by reusing the validation model as the open model for sharing.

OPENMODEL **in the Email Domain.** My previous work on feature interaction discovery in the Email domain[4] gives a flavor of how OPENMODEL could be used. In that work, I analyzed ten common email features, discovering 26 unfavorable feature interactions. The approach was based upon modeling each feature as a component of a distributed system, assembling feature instance models into a "typical" configuration, and then simulating scenarios discovered using a systematic heuristic scenario selection strategy. I created the models by abstracting the behavior of well known real email features, such as address books, filters, forwarders, and vacation programs. I created and assembled them by hand, but this is where the OPENMODEL approach would exploit shared models and tool support. Once the models are gathered and composed, validation (using the ISAT tool set) can proceed as usual.

For illustration, suppose an email user wishes to configure and start using new, feature-rich email client software. An OPENMODEL scenario would proceed as follows.

- The user installs and configures the software.
- The OPENMODEL tool queries it for its shared model.
- The OPENMODEL tool retrieves models of the user's ISP's mail servers, as well as those of a representative set of the user's correspondents.
- The OPENMODEL tool (possibly using ISAT tool suite capabilities like simulation and theorem proving) supports the user in checking whether unfavorable behaviors are possible in the way he has configured the new software. If any are found, the user can then reconfigure the software or contact its vendor.
- The OPENMODEL tool remains available as needed as a question answering tool for when the user has questions about his email system.

Note that the email case study cited assumed *static* model compositions. However, in general, executing a scenario in a composed model will result in the discovery of missing models (because, e.g., a message is sent to a node whose model has yet to be retrieved). An OPENMODEL tool will likely support dynamic model retrieval and integration into the current simulation or validation. In the web services scenario of Figure 1, the user's OPENMODEL tool first retrieves a model of the browser, which leads to retrieving that of the corporate IT web proxy. Initial validation reveals that requests are forwarded to Acme, so Acme's model is retrieved. This then leads to retrieving UpToTheMinuteNews's model. At that point, the OPENMODEL tool has the information necessary to anticipate (or diagnose) the problem.

3. Requirements for a Modeling Language

This section discusses some of the critical requirements a standardized OPENMODEL language should obey.

Tool support. It must support *execution (simulation)* of both single nodes and hierarchical networked compositions of nodes. This will enable user validation of behavior as well as animation. Beyond this, it should support systematic validation methods, such as spec coverage measurement and property verification through theorem proving or model checking.

Support for Shared Ontologies. Users and developers want/need to think in terms of domain-meaningful objects and operations. We must standardize the terminology used to describe the objects that pass between nodes of the MSDS so that models from different stakeholders can be shared and interoperate in the OPENMODEL tool. For example, email models must agree on what an "email message" is and how to represent and access its fields. There is a large body of research into ontologies[5], but OPENMODEL ontologies will need to contain automated reasoning support (axioms, rules) in addition to entity-relationship information. These shared ontologies guide the model developer in picking an appropriate "level of abstraction" for the model, by defining the representation and granularity of the input and output objects.

Some Other Requirements. The models (and language) should support single-node validation as well, so that we can gain the 2 for 1 advantage mentioned earlier. The abstract state model and accesses to it must be first class elements in order to enable behavior sampling (e.g. for contract enforcement). And finally, a computable abstraction map **A** must be a required element as well, again to support comparing observed behaviors to model predictions.

Evaluation of a few candidates. There are many plausible candidates for the OPENMODEL language; I will briefly review a selection of them here.

Executables (`.exe`, `a.out`) can support simulation, but they cannot support validation techniques such as coverage measurement or property verification tools. We can't even guarantee they won't crash the system, which is a serious concern when we will be retrieving models from other stakeholders having different (and unknown) goals.

Java Language Source Code is safe and can support simulation, composition, and even coverage measurement. However, the state-model separation is not adequate and it is still too hard to verify properties of Java code.

Z is very expressive and safe, but not executable. Automated reasoning in it is problematic as well.

Low-level model checking languages (e.g. Promela) support execution and model checking, but not arbitrary domain specific theories/ontologies. And model checkers' difficulties with the state explosion problem are well known.

Unified Modeling Language (UML) has a notoriously ill-defined semantics, but a disciplined subset might be useful.

Infinite state executable specification languages (e.g. ISAT's P-EBF, SALSA[1], Action Language[2]) are the best candidates and seem to satisfy most of the requirements. They can be used for modeling single nodes of the MSDS. They can be combined with a module interconnect language, such as EFCs[4] or WSFL[11], to support composition. They have already been shown to safely support simulation, inclusion of domain-specific ontologies/theories, and infinite state property verification.

4. Limitations

OpenModel is not, of course, a panacea and has many limitations. It is inevitable that some nodes will fail to provide models at all, and some will have inaccurate models. However, as long as we are not expecting OpenModel to provide guarantees (it is more appropriate for heuristic bug pre-detection), this should not deter us. Further, even if all models are present and accurate, all known validation tools are subject to the usual complexities of the validation problem: state explosion, theory explosion, and feature interaction. Therefore, models must be abstract in order for validation to be tractable, so modeling may omit behavioral details important to the detection of undesirable behaviors. A balance must be reached so that a useful class of problems can be discovered even after abstraction.

Configurations. A less obvious limitation lies in the difference between a model and its configuration. For example, it may be well known that a given MSDS node runs a particular off-the-shelf component, such as `sendmail`. And `sendmail`'s model should be common knowledge. However, the real issue for validation is how that node's `sendmail` instance is configured. Stakeholders will be much less willing to expose configuration information to the public. However, some configuration information is more sensitive than others, and it may not all be necessary in order to answer questions of interest to another stakeholder. For example, the actual encryption keys used in encrypting email between users are probably not needed, but information about peer relationships (which hosts are relays and which implement which features) presumably is. Moreover, it may be that node owners can be motivated to reveal some of this information; e.g., "convince me you have configured `sendmail` securely and then I will use your service".

Hidden Agendas. Another critical limitation is the realization that some nodes may *purposely* hide or obfuscate certain of their activities or attributes that are unfavorable to other stakeholders. For example, they may collect personal information for one purpose and secretly use it for other, less desirable, purposes. Or, they may advertise a high level of service, but provide a lower level of service to save money (e.g., claiming a fully recoverable backup system but not really backing up the user's information at all).

To deal with this problem, *the validator must remain aware of the underlying game theory (costs and payoffs) of the MSDS*. OpenModel is simply not reliable in scenarios where other stakeholders are motivated to cheat. However, I believe there are many domains in which the game theory is favorable to OpenModel. For example, in the Internet email system of clients and servers, server providers tend to be motivated by the best interests of similar large groups of end users, and "good email service" is best served by cooperation among servers. It is the individual users (clients) who have other motives, such as spamming. An OpenModel user will only retrieve models from server components or from clients run by people with whom the user has a cooperative relationship. Messages from adversaries must be treated as part of the environment, since their models would not be trustworthy even if they existed.

OpenModel should also be useful and reliable in enterprise application integration (EAI) scenarios, where the applications to be integrated are controlled by a single (logical) stakeholder. A good example of the latter case is when a company acquires assets in a merger and must integrate them into its own asset base. All models should be accurate, because they were built for internal use (system evolution and maintenance) within the respective companies.

OpenModel is also useful in situations where behavior sampling (for contract enforcement, see Section 2) is possible, and where legislation can enforce model fidelity.

Even in the face of these limitations, the email feature interaction case study[4] gives us hope that there is still significant heuristic value in open modeling.

5. Related Work

Fickas et al[3] describe a system, Emu, for monitoring the execution of a system as it carries out a plan for achieving what they term "ephemeral" requirements: those highly dependent on context and not likely to persist for long periods of time. The similarity with this work is that they, too, have made the conceptual leap from global system requirements to single-stakeholder requirements that may not be true forever. OpenModel could complement their system nicely by allowing them to discover behaviors of relevant interacting nodes of the MSDS as needed during monitoring of particular requirements.

Modularity and "black box reuse" are not enough. Distributed object systems (CORBA/IDL, J2EE/EJB, .NET/DCOM) and black box reuse technologies do not provide enough information for a user to discover whether an MSDS will operate desirably. Moreover, there is empirical evidence that modular composition of components in an MSDS is unlikely to "just work": I found that roughly

17% of 156 scenarios examined in the email domain resulted in undesirable behavior. OPENMODEL complements these technologies by supplying missing information.

UDDI is starting in the right direction. The Universal Description Discovery and Integration (UDDI) protocol suite is essentially a "yellow pages for the web". That is, it is intended to provide a way to discover and learn about web services. UDDI defines *tModels*, which are various types of declarations of behavioral properties of web services. Web Services Description Language (WSDL)[10] allows declaration of interface signature information, similar to CORBA's IDL. Web Services Conversation Language (WSCL)[9] goes beyond interfaces, defining allowed *conversations*, which are sequences of queries and responses. Web Services Flow Language (WSFL)[11] provides a way to declare static compositions of web service components, essentially a module interconnect language for web services. WSDL and WSFL address aspects of the ignorance problem that OPENMODEL is intended to solve, but do not go far enough. WSCL, while useful in its own right, seems to overlap with the information provided by a full behavioral model. OPENMODEL models should fit into the UDDI framework as a particularly rich form of tModel.

A lesson from P3P. The Platform for Privacy Preferences (P3P)[6] is a world wide web consortium initiative intended to help users protect their privacy while web browsing. The idea of it is that each web site declares a P3P description of how it handles sensitive information. The user declares preferences about how he wants his information handled, and the P3P enabled browser decides at each web site whether the site conforms to the user's wishes. This is analogous to the OPENMODEL approach, in that each node declares a model of its privacy behavior. Thus, P3P serves as a simple example of OPENMODEL in the narrow domain of privacy behavior. However, it suffers from *unfavorable game theory* (mentioned above). It only works if the web site operators are nice guys who don't game the system. P3P has no way to enforce declared policies, nor even to detect when a site's behavior is inconsistent with its policy.

6. Summary and Future Work

Multi-stakeholder distributed systems present major new challenges to requirements engineering: globally inconsistent requirements placed by stakeholders with conflicting goals, and the ignorance problem induced by the limited and unreliable communications among designers and operators of the nodes. We can avoid the global inconsistency problem by focusing on the question of whether a system meets the needs of a single stakeholder. Once that shift is made, the OPENMODEL proposal addresses the ignorance problem. OPENMODEL-based tools will first gather the open, shared models declared by MSDS nodes relevant to the user. It will then apply a range of validation tools to see whether the system will meet the user's needs. The assembled system model can then be incrementally maintained over time so it is available to the user to answer future questions, or to support the user in designing a new node capability. A pilot study in the email domain provides evidence that OPENMODEL can be useful in validating MSDSs.

Of course, there is a great deal of future work. First, we must settle on a modeling language and define the appropriate XML DTDs for representing models. Next, in a given domain, we must establish appropriate shared (de facto standard) ontologies to guide stakeholders in the model building efforts. Email and web services are two promising domains to pursue. Also, we need to engineer a set of highly usable OPENMODEL tools, based on existing modeling, simulation, and validation tools. There may also be somewhat less ambitious courses that could be followed as well. For example, if we relax our concern over formal verification, we could just use Java as modeling language and support simulation and coverage measurement of models. Techniques for retrieving and loading classes into running JVMs are well known from applet design.

References

[1] R. Bharadwaj and S. Sims. SALSA: combining constraint solvers with bdds for automatic invariant checking. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, 2000.

[2] T. Bultan, R. Gerber, and C. League. Verifying systems with integer constraints and boolean predicates: a composite approach. In *Proc. 1998 Intl. Symp. Software Testing and Analysis, SEN 23(2)*, pages 113 – 123. ACM SIGSOFT, 1998.

[3] S. Fickas, T. Beauchamp, and N. Mamy. Monitoring ephemeral requirements. Technical report, University of Oregon Computer Science Dept., May 2002.

[4] R. J. Hall. Feature interactions in electronic mail. In *Proc. Sixth Intl. Workshop on Feature Interactions in Telecommunications and Software Systems*. IOS Press, 2000.

[5] Ontology.org: enabling virtual business (web site). `http://www.ontology.org/`.

[6] Platform for privacy preferences project (web site). `http://www.w3.org/P3P/`.

[7] Simple object access protocol (soap) 1.1 (web site). `http://www.w3.org/TR/SOAP/`.

[8] Universal description, discovery and integration of business for the web (web site). `http://www.uddi.org`.

[9] Web services conversation language (wscl) 1.0 (web site). `http://www.w3.org/TR/wscl10`.

[10] Web services description language (wsdl) 1.1 (web site). `http://www.w3.org/TR/wsdl`.

[11] Web services flow language (wsfl) 1.0. `www-4.ibm.com/software/solutions/ webservices/pdf/WSFL.pdf`.

Using the Semantic Web to Construct an Ontology-Based Repository for Software Patterns

Scott Henninger
Department of Computer Science & Engineering
University of Nebraska-Lincoln
scotth@cse.unl.edu

Abstract

Patterns, particularly design and usability patterns, have become a popular way to disseminate the current state of knowledge in certain software development issues. Many books have been written and people are using the pattern approach to encode knowledge ranging from management practices to risk assessment patterns.

The continued explosion of patterns collections have caused a couple of clear problems. The fir is the issue of quality and how one knows whether a pattern provides sound advice. The second is finding the right pattern for a particular problem. In this abstract, I propose the semantic web as a medium to start representing the relationships between patterns and track which are used most often or rated highly by peers. This approach not only supports the process of finding patterns, but also allows for the construction of agents that let developers know when a given pattern is applicable.

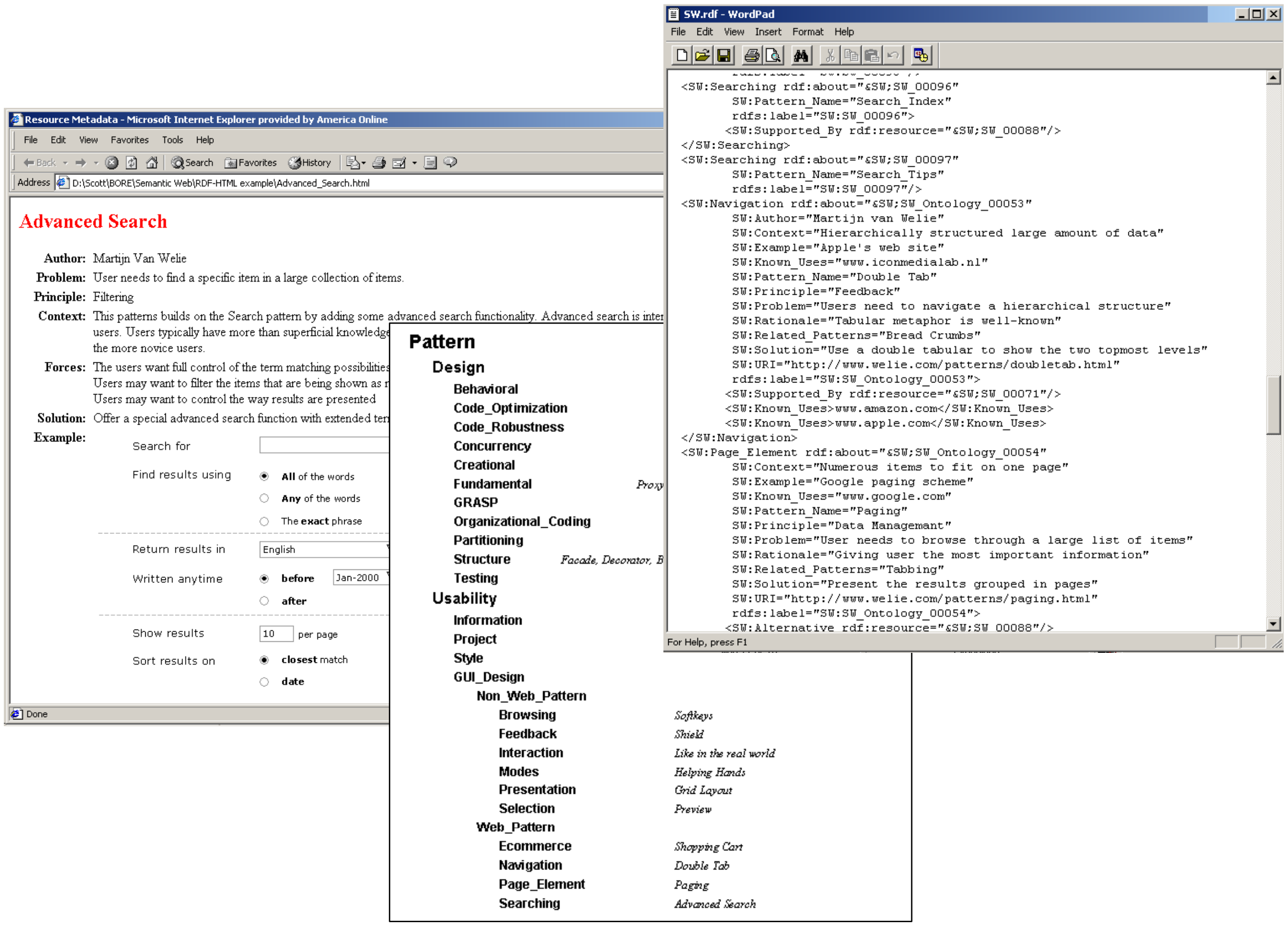
1. Design and Usability Patterns

Beginning with the seminal Gang-of Four design pattern book [Gamma et al. 1995], the software development community has embraced the pattern concept first sued by Alexander on architecture [Alexander 1979]. The general idea is to describe a commonly occurring problem, one or more solutions that have been shown to be effective, along with other contextual information such as why the problem occurs (forces) and the context in which the solution is effective. Perhaps the largest pattern communities in the software development area are the design and usability patterns communities [Borchers 2000], but other types of patterns, such process and management patterns have recently emerged, underscoring the effectiveness of the pattern format to describe problem-solution pairs.

While many researchers have focused on creating and validating patterns, few are thinking about how the patterns should be delivered to or otherwise made usable for software developers. A specific problem is that while Alexander made strong statements about the relationships between patterns to create a "language" capable of creating the whole solutions, at best the relationships between patterns in a collection are weak, and relationships between patterns in different collections is nonexistent. There is also no formal mechanism to assess and refine usability patterns on the community level.

2. Formalizing a Community of Practice for Patterns

A major focus of both the BORE (Building an Organizational Repository of Experiences) [Henninger 1997] and GUIDE (Guidelines for Usability through Interface Development Experiences) [Henninger et al. 1997] efforts is to improve the process through the experiences


Resource Metadata - Microsoft Internet Explorer provided by America Online
File Edit View Favorites Tools Help
Back Search Favorites History
Address D:\Scott\BORE\Semantic Web\RDF-HTML example\Advanced_Search.html
Done

Advanced Search

Author: Martijn Van Welie
Problem: User needs to find a specific item in a large collection of items.
Principle: Filtering
Context: This patterns builds on the Search pattern by adding some advanced search functionality. Advanced search is inter
users. Users typically have more than superficial knowledge
the more novice users.
Forces: The users want full control of the term matching possibilities
Users may want to filter the items that are being shown as r
Users may want to control the way results are presented
Solution: Offer a special advanced search function with extended ter
Example:

Search for	
Find results using	[✓ **All** of the words]
	Any of the words
	The **exact** phrase
Return results in	English
Written anytime	[✓ **before**] Jan-2000
	after
Show results	10 per page
Sort results on	[✓ **closest** match]
	date

Pattern

Design	
Behavioral	
Code_Optimization	
Code_Robustness	
Concurrency	
Creational	
Fundamental	*Proxy*
GRASP	
Organizational_Coding	
Partitioning	
Structure	*Facade, Decorator, B*
Testing	
Usability	
Information	
Project	
Style	
GUI_Design	
Non_Web_Pattern	
Browsing	*Softkeys*
Feedback	*Shield*
Interaction	*Like in the real world*
Modes	*Helping Hands*
Presentation	*Grid Layout*
Selection	*Preview*
Web_Pattern	
Ecommerce	*Shopping Cart*
Navigation	*Double Tab*
Page_Element	*Paging*
Searching	*Advanced Search*

```
<SW:Searching rdf:about="&SW;SW_00096"
       SW:Pattern_Name="Search_Index"
       rdfs:label="SW:SW_00096">
      <SW:Supported_By rdf:resource="&SW;SW_00088"/>
</SW:Searching>
<SW:Searching rdf:about="&SW;SW_00097"
       SW:Pattern_Name="Search_Tips"
       rdfs:label="SW:SW_00097"/>
<SW:Navigation rdf:about="&SW;SW_Ontology_00053"
       SW:Author="Martijn van Welie"
       SW:Context="Hierarchically structured large amount of data"
       SW:Example="Apple's web site"
       SW:Known_Uses="www.iconmedialab.nl"
       SW:Pattern_Name="Double Tab"
       SW:Principle="Feedback"
       SW:Problem="Users need to navigate a hierarchical structure"
       SW:Rationale="Tabular metaphor is well-known"
       SW:Related_Patterns="Bread Crumbs"
       SW:Solution="Use a double tabular to show the two topmost levels"
       SW:URI="http://www.welie.com/patterns/doubletab.html"
       rdfs:label="SW:SW_Ontology_00053">
      <SW:Supported_By rdf:resource="&SW;SW_00071"/>
      <SW:Known_Uses>www.amazon.com</SW:Known_Uses>
      <SW:Known_Uses>www.apple.com</SW:Known_Uses>
</SW:Navigation>
<SW:Page_Element rdf:about="&SW;SW_Ontology_00054"
       SW:Context="Numerous items to fit on one page"
       SW:Example="Google paging scheme"
       SW:Known_Uses="www.google.com"
       SW:Pattern_Name="Paging"
       SW:Principle="Data Managemant"
       SW:Problem="User needs to browse through a large list of items"
       SW:Rationale="Giving user the most important information"
       SW:Related_Patterns="Tabbing"
       SW:Solution="Present the results grouped in pages"
       SW:URI="http://www.welie.com/patterns/paging.html"
       rdfs:label="SW:SW_Ontology_00054">
      <SW:Alternative rdf:resource="&SW;SW_00088"/>
```

Figure 1: The left window of this figure shows a pattern rendered through an RDF representation using an XSL file to render the image. The middle window shows a sample pattern, showing the GOF and usability patterns in particular. Note that the names to the right in italics are instances of the patterns. The Right-hand window shows an example of RDF code. Note that it is compatible with XML.

gained in software development efforts, hence the terms "Experience" in both of the system's acronyms. Perhaps more important is the focus on using the feedback to refine both the patterns and the context in which the patterns are most applicable. Both models have a meta-process in which development teams review the adequacy of the information supplied by the rule-based system and submit deviation requests when the recommendations are inappropriate or inadequate. These are forwarded to process and/or usability specialists in charge of the repository. Changes based on these deviation requests can then be made, facilitating a continuous learning process involving both creating new patterns and refining contextual information about the patterns [Henninger 2001]. The end result is the continuous creation of new and refined patterns that fit the needs of a given development organization, or the patterns community as a whole. Thus, we are aiming to create a community of practice [Brown, Duguid 1991] in which people exchange "stories" or patterns about what has worked in the past and when it may be applicable.

3. Using the Semantic Web to Deliver Software Development Knowledge

The Semantic Web [Berners-Lee 1998] is a Web-based technology that extends XML by providing the means to define ontologies, the definition of objects and relationships between them. This allows machines to make intelligent inferences about objects across the Web. This can allow intelligent agents [Hendler 2001] embodying knowledge about certain aspects of software development (much of it may be organization-specific) to make intelligent inferences that can be used as the basis for improved decision-making on software development processes, usability issues, etc.

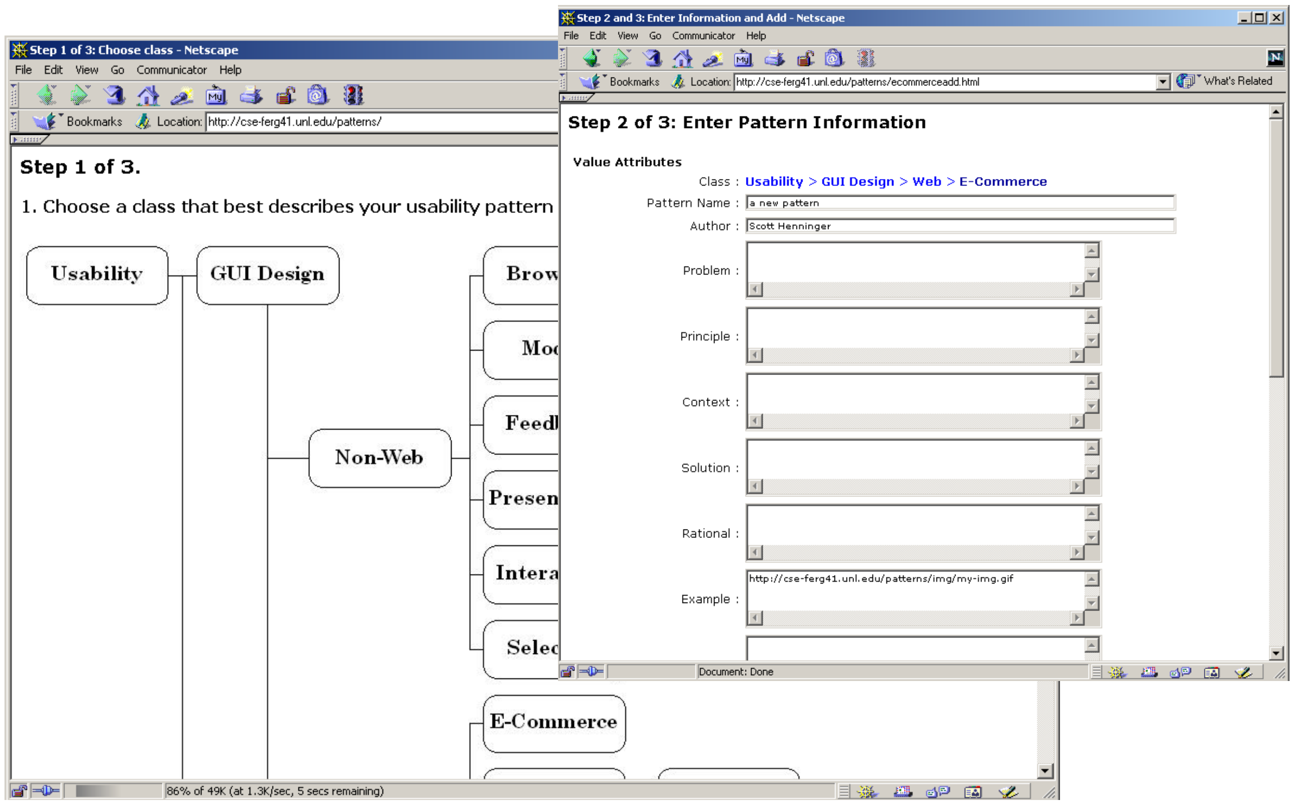
Currently, the technology is in flux, but we have been experimenting with the Resource Description Framework (RDF) [Klein 2001] and the newer and less stable Defense Agency Markup Language (DAML) [Burke 2002], which also includes an Ontology Inference Layer (OIL). Both are knowledge representation languages with roots in semantic networks, but built to work on the Web. Our plan is to begin collecting patterns and relationships that are represented in RDF and/or DAML files. Figure 1 shows a sample pattern, an ontology focusing on usability patterns, and part of the RDF representation for the usability pattern domain.

3.1 Using the Semantic Web as a Communication Medium for Communities of Practice

Our overall goal is to set up a repository where pattern designers can post their proposed patterns and have them evaluated by peers and experts. Not only would this help consolidate some of the knowledge in the area, it will provide the means to collect the patterns in a common area (a virtual common area, as RDF/DAML files can be stored on servers worldwide) so people can argue and come to a consensus on both the validity of patterns and where the patterns belong. A key to the success of such a program is agreement on structure, only part of which can be alleviated by the object-oriented and flexible structured that can be created by these knowledge representation languages. We are experimenting with just this part of the problem by looking specifically at usability patterns.

Here are the following steps we are following to turn isolated collections of patterns into a world-wide repository structure designed for computation and r intelligent agents:

1) *Choose a Web-Based Knowledge Representation Language.* This choice has been made easier by Protégé [Noy et al. 2001] and other ontology tools that will translate an ontology into a number of knowledge representation languages. Our initial steps have concentrated on the more stable RDF, but we will soon migrate to the more powerful DAML.
2) *Create an Initial Domain-Specific Ontology*. An initial ontology is being constructed that describes the domain of usability patterns, as shown in Figure 1. An ontology is a formal, explicit specification of a shared conceptualization, meaning that defined terms and relationships between them are specified in machine-readable manner, and generally agreed upon by humans.
3) *Create a site for Collecting Patterns*. As shown in Figure 2, we are constructing a Web site that allows people to add both instances of pattern classes to the repository, including relationships between patterns. New patterns instances and classes will go through and evaluation process and posted in the RDF/DAML file(s) for use by other organizations.


Step 1 of 3: Choose class - Netscape
File Edit View Go Communicator Help
Bookmarks
Location:
Usability
GUI Design
Non-Web
Brow
Moc
Feedl
Presen
Intera
Selec
E-Commerce
Step 2 and 3: Enter Information and Add - Netscape
What's Related
Document: Done

Location: http://cse-ferg41.unl.edu/patterns/

Step 1 of 3.

1. Choose a class that best describes your usability pattern

86% of 49K (at 1.3K/sec, 5 secs remaining)

Location: http://cse-ferg41.unl.edu/patterns/ecommerceadd.html

Step 2 of 3: Enter Pattern Information

Value Attributes

Class :	Usability > GUI Design > Web > E-Commerce
Pattern Name :	a new pattern
Author :	Scott Henninger
Problem :	
Principle :	
Context :	
Solution :	
Rational :	
Example :	http://cse-ferg41.unl.edu/patterns/img/my-img.gif

Figure 2: The left-hand window shows the current class structure of the Usability Pattern ontology. The right-top window shows a partially filed in window to add a new instance to the E-Commerce class.

4) *Other Organizations Place Instances on the Web.* Provided that people agree on the overall ontology, RDF/DAML files containing instances can be placed anywhere on the Web. For example, someone creating a collection on e-Commerce usability patterns can place their instances in an RDF/DAML file for others to use. If, however, this person or group wanted to create, for example a subclass of e-Commerce, such as "Web Storefront", then they would need to get agreement amongst the patterns community to change the schema (class structure), otherwise, other agents would not be able to utilize their class structures.

5) *Instances are Collected for Processing.* The instances are collected in a database or some representational medium. This can be accomplished by either Web crawlers or brokerage services. The advantage of the first is that organizations can work independently, and the advantage of the second is enhanced trust – the broker can be used to verify the source, keep statistics on customer satisfaction, etc. We are using he second strategy.

6) *Agents are Created to Make Intelligent Decisions.* Given the ontological structures, agents can be created that make inferences about the information supplied. For example, an e-Commerce pattern for "Server-Side Information Collection" cold have a relationship with a "Database Server" class stating that one of the instance of the Database Server patterns (which could include e-Commerce solutions using Oracle, Sybase, etc.) must also be chosen. Other constraints could also be imposed by the agent that makes an automatic selection of the database given the project attributes, possibly all represented in patterns. Other kinds of inferences are also possible [Fensel et al. 2001].

The end result should be the creation of agents that can reason about usability pattern and deliver them to software developers to improve the quality of user interfaces. In our case, the agent is BORE (which has been combined with GUIDE) [Henninger 2001], but others may create their own agents with different assumptions and computational inferences.

4. Current Status and Future Work

The current Web site shown in Figure 2 is being developed and should be ready for deployment at the end of the summer. We will draw on the emerging Usability Patterns community [Borchers 2000] to place their existing patterns in the repository and create new ones to fill in details not found in the repository. We will initially seed the repository with some patterns available on the Web, with author permission, such as van Welie's Amsterdam Collection [van Welie 2002]and Tidwell's Common Ground [Tidwell 1999] collection.

We will initially perform the validation process, but will gradually turn this over to subject area experts [Ackerman, Malone 1990]. In parallel, we will continue to evolve the BORE system to utilize these patterns in the software development process.

5. References

[Ackerman, Malone 1990] Ackerman, M.S. and Malone, T.W., Answer Garden: A tool for growing organizational memory. *Proceedings of the Conference on Office Information Systems*, (1990), ACM, New York, 31-39.

[Alexander 1979] Alexander, C. *The Timeless Way of Building*. Oxford Univ. Press, New York, 1979.

[Berners-Lee 1998] Berners-Lee, T. Semantic Web Roadmap, *W3C Semantic Web Vision Statement*, 1998, http://www.w3.org/DesignIssues/Semantic.html, Last accessed on 2/20/2002.

[Borchers 2000] Borchers, J. CHI Meets PLoP: An Interaction Patterns Workshop. *SIGCHI Bulletin*, *32* (1). 9-12.

[Brown, Duguid 1991] Brown, J.S. and Duguid, P. Organizational Learning and Communities-of-Practice: Toward a Unified View of Working, Learning, and Innovation. *Organization Science*, *2* (1). 40-57.

[Burke 2002] Burke, M., The DARPA Agent Markup Language Homepage. http://www.daml.org/, 2002.

[Fensel et al. 2001] Fensel, D., van Harmelen, F., Horrocks, I., McGuinness, D. and Patel-Schneider, P. OIL: An Ontology Infrastructure for the Semantic Web. *IEEE Intelligent Systems*, *16* (2). 38-45.

[Gamma et al. 1995] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Hendler 2001] Hendler, J. Agents and the Semantic Web. *IEEE Intelligent Systems*, *16* (2). 30-37.

[Henninger 1997] Henninger, S., Tools Supporting the Creation and Evolution of Software Development Knowledge. *Proceedings of the Automated Software Engineering Conference*, (Lake Tahoe, NV, 1997), 46-53.

[Henninger 2001] Henninger, S., An Organizational Learning Method for Applying Usability Guidelines and Patterns. *8th IFIP Working Conference on Engineering for Human-Computer Interaction (EHCI'01)*, (Toronto, 2001).

[Henninger et al. 1997] Henninger, S., Lu, C. and Faith, C., Using Organizational Learning Techniques to Develop Context-Specific Usability Guidelines. *Proc. Designing Interactive Systems (DIS '97)*, (Amsterdam, 1997), 129-136.

[Klein 2001] Klein, M. XML, RDF, and Relatives. *IEEE Intelligent Systems*, *15* (2). 26-28.

[Noy et al. 2001] Noy, N., Sintek, M., Decker, S., Crubezy, M., Fergerson, R. and Musen, M. Creating Semantic Web Contents with Protege-2000. *IEEE Intelligent Systems*, *16* (2). 60-71.

[Tidwell 1999] Tidwell, J., COMMON GROUND: A Pattern Language for Human-Computer Interface Design. http://www.mit.edu/~jtidwell/common_ground.html, 1999.

[van Welie 2002] van Welie, M., Guidance/Feedback Patterns. http://www.welie.com/patterns/index.html, January, 2002.

Managing Software Projects in Spatial Hypertext: Experiences in Dogfooding

Frank Shipman
Center for the Study of Digital Libraries and Department of Computer Science
Texas A&M University
College Station, TX 77843-3112, USA
shipman@cs.tamu.edu

Abstract

Managing long-term, research-oriented software projects requires more flexibility and open-endedness than most production-oriented software processes provide. We have been exploring the use of spatial hypertext to manage such projects. Spatial hypertext allows users to place information objects in visual spaces and use visual cues and spatial relations to represent inter-object relations. Over time, users develop a visual language to express characteristics of their task. The Visual Knowledge Builder (VKB), our particular spatial hypertext system, uses heuristics to recognize structure in user-generated layouts and includes navigable history for returning to earlier states of the spatial hypertext. This paper reflects on our experiences in dogfooding – using our own research prototype – for two projects for more than two and a half years and what these experiences might mean for using spatial hypertext in other software development contexts.

Keywords

spatial hypertext, software development

1. Introduction

Software development takes place in a wide variety of contexts, yielding an equally wide variety of software engineering processes. Some contexts, such as life-critical applications, require stability and correctness and are willing to expend resources (including delaying delivery time) in order to achieve these goals. Other contexts, including much software developed for home use, are time-to-market driven, looking to gain market share with the potential of providing bug fixes later and enhancements through a series of versions of the software. Research software is even more extreme – the software is being developed to explore what is possible within a design space. In the research context, reliability need only support the project's mode of evaluation, e.g. proof-of-concept demonstrations, laboratory experiments, and limited real-task usage being common.

Different project management software is appropriate within these different contexts. We are exploring the use of spatial hypertext, with its emphasis on free-form expression, in the project management of research software. Spatial hypertext users place information objects in visual spaces and use visual cues and spatial relations to represent inter-object relations [4]. For example, they may categorize objects by creating lists or using color. Over time, users develop a visual language to express characteristics of their task. Our system, the Visual Knowledge Builder (VKB), includes a spatial parser for recognizing structures in the layout and supporting users in the manipulation of these structures [7]. Additionally, VKB records the evolution of the workspace as a navigable history with multiple access mechanisms.

The next section provides a brief overview of VKB, its spatial parser, and its history mechanism. Following this is a description of our use of VKB for managing software-oriented research projects. The paper concludes with a discussion of how our experiences might inform the use of spatial hypertext in other software development contexts and the development of project management software.

2. The Visual Knowledge Builder (VKB)

The main Visual Knowledge Builder interface, shown in Figure 1, is a two-dimensional workspace with controls at the top and message bars below. Users collect and author

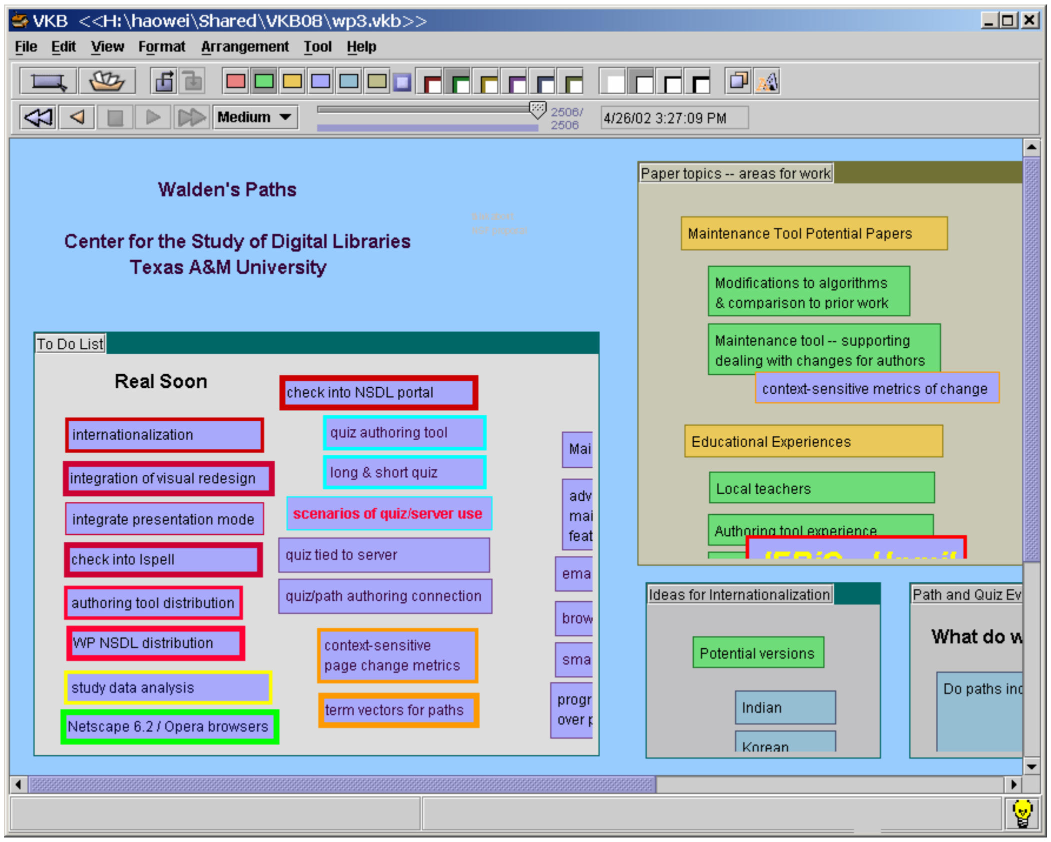

VKB <<H:\haowei\Shared\VKB08\wp3.vkb>>
File Edit View Format Arrangement Tool Help
Medium
2506/ 2506
4/26/02 3:27:09 PM
Walden's Paths
Center for the Study of Digital Libraries
Texas A&M University
To Do List
Real Soon
check into NSDL portal
internationalization
quiz authoring tool
integration of visual redesign
long & short quiz
integrate presentation mode
scenarios of quiz/server use
check into Ispell
quiz tied to server
authoring tool distribution
quiz/path authoring connection
WP NSDL distribution
context-sensitive page change metrics
study data analysis
term vectors for paths
Netscape 6.2 / Opera browsers
Paper topics -- areas for work
Maintenance Tool Potential Papers
Modifications to algorithms & comparison to prior work
Maintenance tool -- supporting dealing with changes for authors
context-sensitive metrics of change
Educational Experiences
Local teachers
Authoring tool experience
Ideas for Internationalization
Potential versions
Indian
Korean
Path and Quiz Ev
What do w
Do paths ind

Figure 1. Project workspace with three collections containing software development tasks, paper-writing tasks, and brainstorming for a particular software extension.

information in the form of rectangular objects containing text, attributes and values, links to files or URLs, and images. User's express interpretations, e.g. categorizations and relations, through the placement of objects and the use of visual attributes, such as border and background color, border width, and font type, size, and color. The workspace also includes "collections", two-dimensional spaces embedded in the top-level workspace or another collection. Users may navigate into a collection to see more of its contents. Figure 2 shows the results of navigating into the "To Do List" collection in Figure 1. For more information on general VKB functionality see [7,8,10].

Figures 1 and 2 show a workspace used for managing a research project with five or more participants at any one time. This space has been in use for over two years in weekly project meetings. Writing tasks are placed in the "Paper topics -- areas for work" collection and system and design tasks are found in the "To Do List" collection. Over the period of use, dozens of tasks have been identified, given a priority, and placed in the "Done" collection on the right side of Figure 2.

The second toolbar in Figures 1 and 2 provides access to the history of the workspace. VKB records all the editing events and allows users to play this record forwards or backwards, navigate to specific types of events, or locate the state of the workspace on a particular date [6]. This history mechanism is similar to Reeves' embedded history [5] and Hayashi's temporally-threaded workspaces [1].

2.1 Structure Recognition

To aid the manipulation and later formalization of visually-represented information, VKB attempts to recognize spatial structures as they are created in the workspace via a spatial parser. The spatial parser was developed to recognize structures found to be common in a variety of virtual and physical layouts [9], such as the lists, stacks, and composites in figures 1 and 2. Users can access different scopes of structure through hierarchic-click selection and the recognized structures are used to generate suggestions for placement, formal attributes, and relations of information objects [10].

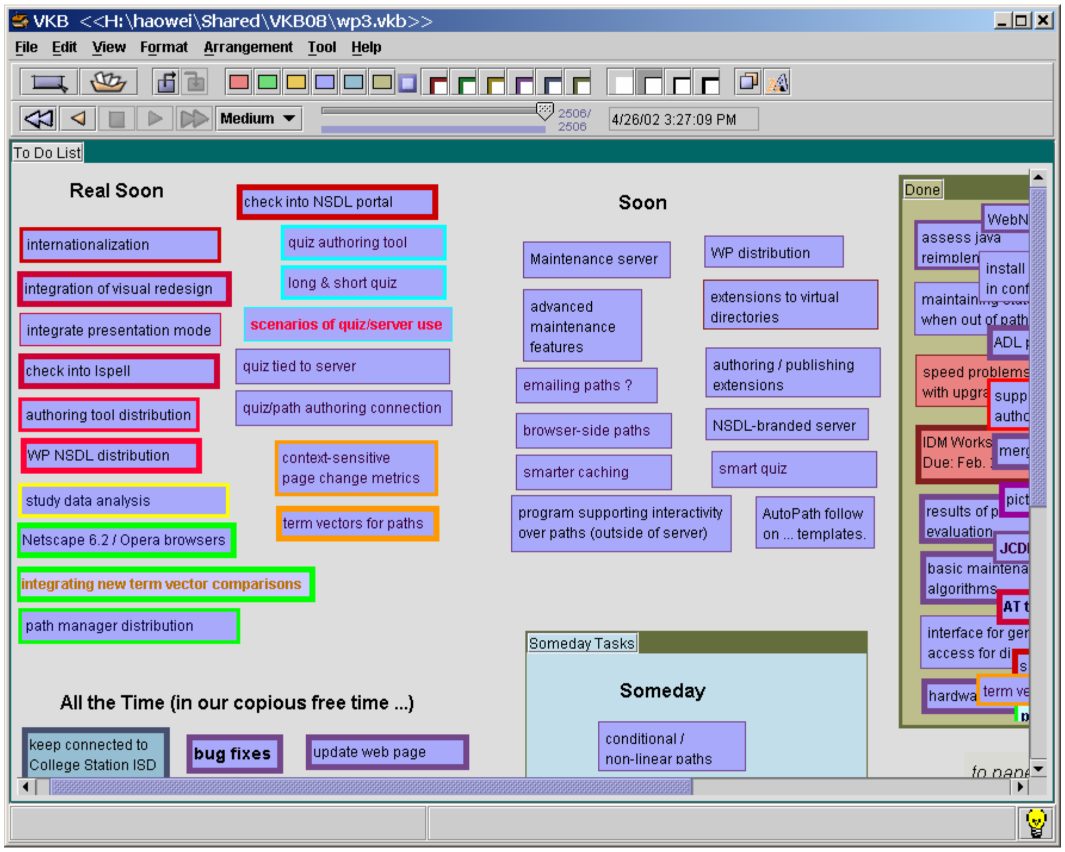

VKB <<H:\haowei\Shared\VKB08\wp3.vkb>>
File Edit View Format Arrangement Tool Help
Medium
2506/
2506
4/26/02 3:27:09 PM
To Do List
Real Soon
check into NSDL portal
internationalization
quiz authoring tool
long & short quiz
integration of visual redesign
integrate presentation mode
scenarios of quiz/server use
check into Ispell
quiz tied to server
authoring tool distribution
quiz/path authoring connection
WP NSDL distribution
context-sensitive page change metrics
study data analysis
Netscape 6.2 / Opera browsers
term vectors for paths
integrating new term vector comparisons
path manager distribution
All the Time (in our copious free time ...)
keep connected to College Station ISD
bug fixes
update web page
Soon
Maintenance server
WP distribution
advanced maintenance features
extensions to virtual directories
emailing paths ?
authoring / publishing extensions
browser-side paths
NSDL-branded server
smarter caching
smart quiz
program supporting interactivity over paths (outside of server)
AutoPath follow on ... templates.
Someday Tasks
Someday
conditional / non-linear paths
Done
to page

Figure 2. Navigating into "To Do List" collection in Figure 1 shows lists indicating the priority of tasks, border colors indicating person responsible for task, and border width indication progress on task.

Figure 3 shows a limitation in the spatial parser. Much like the space seen in figures 1 and 2, this is a project workspace used in weekly meetings. Unlike the arrangement in the prior space, this VKB space uses horizontal position to indicate a continuum of priority. The spatial parser recognizes (some) horizontal lists in this structure but does not include the notion of an ordered list, much less the use of space to represent continuous values of an attribute such as priority. Techniques from Hsieh's VITE [2,3], which supports continuous and discrete mappings between structured data and a visual representation, could aid in this example.

2.2 Navigable History

One feature of VKB that has been particularly useful within the context of project management is the navigable history. The initial to-do lists had relatively few tasks but have grown as dozens of tasks being generated (and fewer being completed) during each year. During this time, the

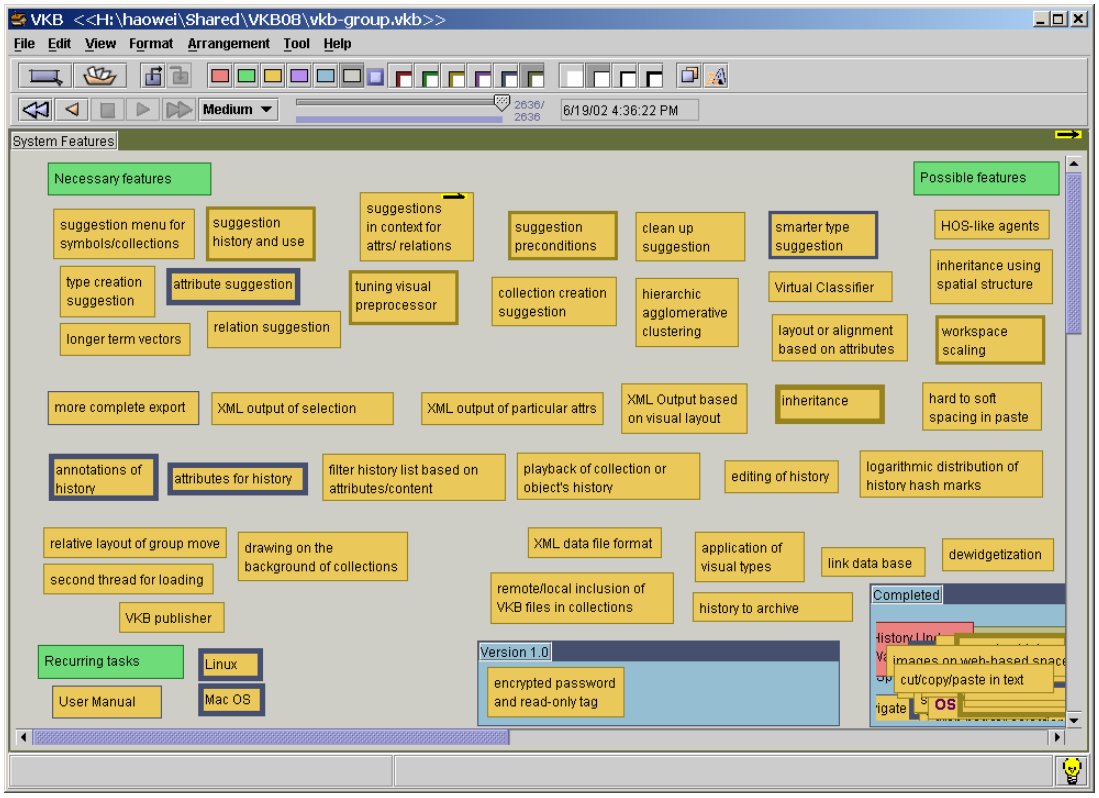

VKB <<H:\haowei\Shared\VKB08\vkb-group.vkb>>
File Edit View Format Arrangement Tool Help
Medium
2636/ 2636
6/19/02 4:36:22 PM
System Features
Necessary features
suggestion menu for symbols/collections
suggestion history and use
suggestions in context for attrs/ relations
suggestion preconditions
clean up suggestion
smarter type suggestion
Possible features
HOS-like agents
type creation suggestion
attribute suggestion
tuning visual preprocessor
collection creation suggestion
hierarchic agglomerative clustering
Virtual Classifier
inheritance using spatial structure
longer term vectors
relation suggestion
layout or alignment based on attributes
workspace scaling
more complete export
XML output of selection
XML output of particular attrs
XML Output based on visual layout
inheritance
hard to soft spacing in paste
annotations of history
attributes for history
filter history list based on attributes/content
playback of collection or object's history
editing of history
logarithmic distribution of history hash marks
relative layout of group move
drawing on the background of collections
XML data file format
application of visual types
link data base
dewidgetization
second thread for loading
remote/local inclusion of VKB files in collections
history to archive
Completed
VKB publisher
Recurring tasks
Linux
Version 1.0
images on web-based space
cut/copy/paste in text
User Manual
Mac OS
encrypted password and read-only tag
OS

Figure 3. System features placed in priority from left to right and classified among different themes from top to bottom.

visual representation – the semantics of different colors and border widths – has evolved to cope with more tasks and to represent characteristics of tasks considered important later in the project's development. As such, the visual attributes of older tasks (particularly those in the "Done" or "Completed" collections) cannot be interpreted based on the current meaning of those attributes. Navigable history allows returning to earlier states of the workspace to determine the meaning of the visual representation. Also, by going back to prior times in the workspace, the creation and modification of tasks can be viewed. Figure 4 shows the workspace from Figure 3 approximately 1 year earlier.

3. Experiences from Dogfooding

Dogfooding is the use of one's own systems for debugging and iterative design. It is not a substitute for getting users involved in design and formative or summative evaluation. We have been using versions of VKB for note taking, preparing papers and presentations, project management, and conference organization. VKB has also been in use by members of the hypertext research community (outside of Texas A&M) for over two years.

Our experiences using VKB to help manage the research projects shown in the above figures began in November, 1999. The VKB spaces are projected and edited during weekly meetings. Because the project team members (faculty and students) are in the room during most editing of the space, the implicit nature of the representation remains comprehensible. The face-to-face setting promotes the use of conversation to repair breakdowns when visual changes are not understood by all participants.

The visual languages have also become the focus of humor. For example, border color in Figures 1 and 2 indicates who has primary responsibility for each task. When new tasks are created there can be a variety of suggestions as to what color it should be assigned. As border thickness indicates progress, there is competition surrounding the changing of task borders and movement of tasks to "Done". Finally, for some of the most dreaded tasks – like writing journal papers – the font gets bigger each week until some progress is achieved. Clearly, group personality plays a large role in such activity. While we

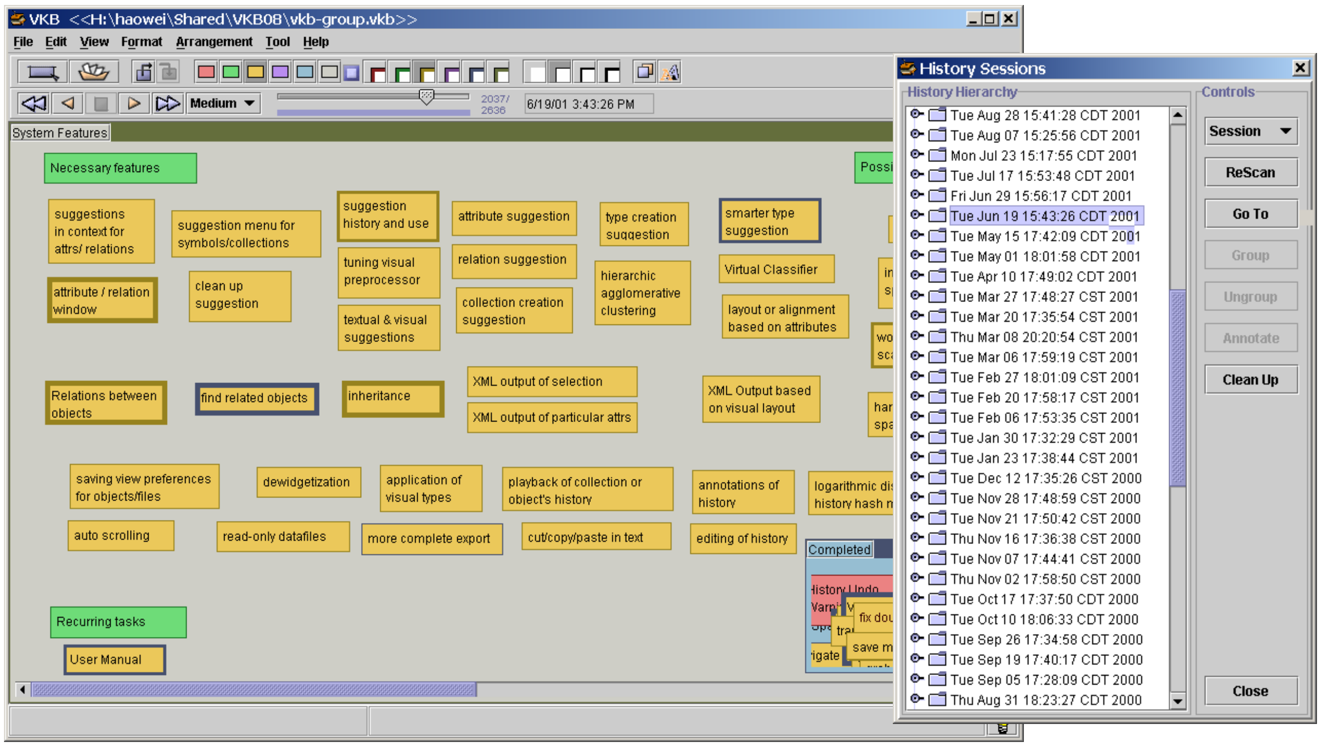

VKB <<H:\haowei\Shared\VKB08\vkb-group.vkb>>
File Edit View Format Arrangement Tool Help
Medium
2037/ 2636
6/19/01 3:43:26 PM
System Features
Necessary features
suggestions in context for attrs/ relations
suggestion menu for symbols/collections
suggestion history and use
attribute suggestion
type creation suggestion
smarter type suggestion
attribute / relation window
clean up suggestion
tuning visual preprocessor
relation suggestion
hierarchic agglomerative clustering
Virtual Classifier
textual & visual suggestions
collection creation suggestion
layout or alignment based on attributes
Relations between objects
find related objects
inheritance
XML output of selection
XML output of particular attrs
XML Output based on visual layout
saving view preferences for objects/files
dewidgetization
application of visual types
playback of collection or object's history
annotations of history
logarithmic dis history hash n
auto scrolling
read-only datafiles
more complete export
cut/copy/paste in text
editing of history
Completed
Recurring tasks
User Manual
History Sessions
History Hierarchy
Tue Aug 28 15:41:28 CDT 2001
Tue Aug 07 15:25:56 CDT 2001
Mon Jul 23 15:17:55 CDT 2001
Tue Jul 17 15:53:48 CDT 2001
Fri Jun 29 15:56:17 CDT 2001
Tue Jun 19 15:43:26 CDT 2001
Tue May 15 17:42:09 CDT 2001
Tue May 01 18:01:58 CDT 2001
Tue Apr 10 17:49:02 CDT 2001
Tue Mar 27 17:48:27 CST 2001
Tue Mar 20 17:35:54 CST 2001
Thu Mar 08 20:20:54 CST 2001
Tue Mar 06 17:59:19 CST 2001
Tue Feb 27 18:01:09 CST 2001
Tue Feb 20 17:58:17 CST 2001
Tue Feb 06 17:53:35 CST 2001
Tue Jan 30 17:32:29 CST 2001
Tue Jan 23 17:38:44 CST 2001
Tue Dec 12 17:35:26 CST 2000
Tue Nov 28 17:48:59 CST 2000
Tue Nov 21 17:50:42 CST 2000
Thu Nov 16 17:36:38 CST 2000
Tue Nov 07 17:44:41 CST 2000
Thu Nov 02 17:58:50 CST 2000
Tue Oct 17 17:37:50 CDT 2000
Tue Oct 10 18:06:33 CDT 2000
Tue Sep 26 17:34:58 CDT 2000
Tue Sep 19 17:40:17 CDT 2000
Tue Sep 05 17:28:09 CDT 2000
Thu Aug 31 18:23:27 CDT 2000
Controls
Session
ReScan
Go To
Group
Ungroup
Annotate
Clean Up
Close

Figure 4. System features placed in priority from left to right and classified among different themes from top to bottom.

have not performed any comparison, it seems less likely that such good-natured banter and competition would occur using a traditional project management tool to track progress. The implicit nature of the visual representation makes these judgments less threatening.

Besides promoting a common understanding of tasks and progress, the workspaces act as a community memory of project activity. When an annual report is due, going back through the history helps identify activities to report. Seeing the edits replayed triggers memories of not just what was recorded but more general meeting discussions that occurred. Another use of the history has been to determine when particular ideas or tasks were introduced into the project. Thus the workspace and its history act as a record of the intellectual development of the projects.

4. Discussion

To determine how our experiences with the use of spatial hypertext for project management can inform its potential for use in other software development contexts, we must first understand the similarities and differences in these contexts. Large research software projects are characterized by their continual exploration of a design space. This exploration is driven by the interests of the members of the project, the availability of funding, the research methods, and the venues for reporting results.

A successful project is one that generates publishable results, which often includes developing software prototypes that can be studied in use. A really successful project uncovers new issues that cause the project to continue indefinitely.

While this may seem very different from commercial software development, there are a number of similarities. The continual integration of design and development can be found in many iterative development methods. Also, most software developed that is successful continues to evolve for years after the initial release. No one believes the current version of MS Word will be the last. Thus, potential for indefinite future development is also of value for commercial software. Many of the differences between commercial and research software seem to be in the required level of stability, features, support, and scale.

The primary advantage to using spatial hypertext is its permissive nature. The visual representation allows expression of characteristics of the task as desired. Schemas do not have to be changed when new it is determined that new attributes of tasks need to be considered. Also, the informal nature of the representation removes the need to have separate tools for recording design discussions and managing the software development.

The informal nature means that this would not work in environments where there is the need to automatically generate reports on the status of projects. Also, while there can be an indefinite number of objects in a spatial hypertext, limitations of displays pose practical problems for working with more than a few hundred tasks in one space. Thus, spatial hypertext seems well suited to relatively small software development activities that require, or would benefit from, flexibility.

Looking to the future, spatial hypertext could better support our own development in a couple ways. The most obvious direction is connecting the visual workspace with structured or semi-structured data used by other tools. Integrating VITE's ability for visual changes to objects in the workspace to be reflected as semantic changes to objects in an underlying database would allow integration with other software development and project management tools.

5. Acknowledgements

This work was supported in part by the National Science Foundation under Grant Number IIS-9734167.

6. References

[1] Hayashi, K., Nomura, T., Hazama, T., Takeoka, M., Hashimoto, S., and Gudmundson, S. Temporally-threaded Workspace: A Model for Providing Activity-based Perspectives on Document Spaces, *Proceedings of ACM Hypertext '98 Conference*, 1998, pp. 87-96.

[2] Hsieh, H. and Shipman, F. VITE: A Visual Interface Supporting the Direct Manipulation of Structured Data Using Two-Way Mappings, *Proceedings of ACM Conference on Intelligent User Interfaces 2000*, 2000, pp. 141-148.

[3] Hsieh, H. and Shipman, F. Manipulating Structured Information in a Visual Workspace, to appear in *Proceedings of ACM Conference on User Interface Software and Technology 2002*, 2002.

[4] Marshall C.C., and Shipman, F.M. 1995. Spatial Hypertext: Designing for Change. *Communications of the ACM*, 38, 8 (August 1995), 88-97.

[5] Reeves, B. 1993. Supporting Collaborative Design by Embedded Communication and History in Design Artifacts. Ph.D. Dissertation, Department of Computer Science, University of Colorado.

[6] Shipman F. and Hsieh, H. "Navigable History: A Reader's View of Writer's Time", *New Review of Hypermedia and Multimedia*, Vol. 6 (2000), pp. 147-167.

[7] Shipman, F., Hsieh, H., Airhart, R., Maloor, P., Moore, J.M., Shah, D. Emergent Structure in Analytic Workspaces: Design and Use of the Visual Knowledge Builder. *Human-Computer Interaction: INTERACT 2001*, 2001, pp. 132-139.

[8] Shipman, F., Hsieh, H., Airhart, R., Maloor, P., and Moore, J.M. The Visual Knowledge Builder: A Second Generation Spatial Hypertext, *Proceedings of the ACM Conference on Hypertext*, 2001, pp. 113-122.

[9] Shipman, F.M., Marshall, C.C., and Moran, T.P. 1995. Finding and Using Implicit Structure in Human-Organized Spatial Layouts of Information. In *Proceeding of the ACM Conference on Human Factors in Computing Systems (CHI '95)*, ACM, New York, 346-353.

[10] Shipman, F., Moore, J.M., Maloor, P., Hsieh, H., and Akkapeddi, R. Semantics Happen: Knowledge Building in Spatial Hypertext, *Proceedings of the ACM Conference on Hypertext*, 2002, pp. 25-34.

The TOBIAS test generator and its adaptation to some ASE challenges
Position paper for the ASE Irvine Workshop

Y. Ledru
Laboratoire Logiciels Systèmes Réseaux/IMAG
BP 72, F-38402 Saint-Martin-d'Hères CEDEX, FRANCE
Yves.Ledru@imag.fr

Abstract

In the past decade, a scientific community has emerged around the notion of "Automated Software Engineering". This community has made several advances in two kinds of challenges: the complexity of processing software engineering information, and the difficulty to capture knowledge about software. This position paper first recalls these challenges. It then describes how these challenges influenced the design of the TOBIAS test generation tool.

1 Challenges of ASE research

Automated Software Engineering tries to develop software tools that help in the software development activities. Such tools start from digital information and try to produce other digital information for the software engineer. Most of the time, this digital information is a structured or a formal document. Structured or formal documents include: source code, tests, formal specifications, but also semi-formal specifications (e.g. UML diagrams) or structured documents (e.g. XML documents). At longer term, research work on natural language recognition (both spoken and written) may increase the spectrum of potential input documents.

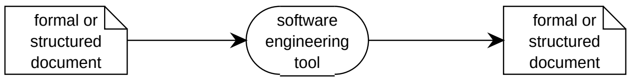

formal or structured document
software engineering tool
formal or structured document

Figure 1. Structure of an ASE tool

The first challenge faced by ASE tool designers is to design efficient and powerful tools. This is not a trivial task, and it is confronted to fundamental problems.

- The complexity of the input documents: real-life applications usually involve thousands or millions of artifacts (lines of code, diagrams elements, ...). This complexity is inherited by the tools that process these artifacts, and it requires optimisations even in the case of linear algorithms.
- Many algorithms in this field have a non-linear complexity. They must face the challenge of combinatorial explosion (e.g. in model-checking techniques).
- In many other cases, the input documents use languages whose expressiveness makes processing activities undecidable (e.g. theorem proving on first order predicate logic).

These three fundamental problems are at the heart of ASE research, and significant advances have been made in each of these fields. One of the main results of the ASE community was the identification of domain specific knowledge which helps face these complexity and decidability problems. By accumulating knowledge about a class of problems, it is possible to design domain specific tools, or tools which use a domain knowledge base. Such tools exploit the domain information to narrow their search for solutions and converge more rapidly.

This introduces a second challenge: how do you capture domain knowledge and other relevant information. Although much effort has been done in order to educate software engineers to the virtues of specification and documentation, many software development activities still correspond to CMM level 1. Approaches to software engineering like Extreme Programming even try to reduce the production of documents to a minimum, taking for granted that software engineers only like to write code! Also the widespread use of computer technology has turned millions of people into amateur software engineers, with poor education in software development techniques.

Two kinds of answers have been proposed to this second challenge.

- The first answer is to provide useful and efficient tools. It is important that tools bring some benefits and are applicable to real-size software. Benefits can be of two kinds: either an improvement in productivity, or a speed-up of the process, or improvement in quality. If any of these factors (productivity, time, quality) is a critical factor for a company, it will motivate efforts to adopt the new technology. The capabilities of software engineers to adapt themselves and their process to new technology should not be underestimated. Software engineers are confronted to a constant evolution of target technologies (programming languages, hardware and software platforms). They have the ability to learn new specification languages if they perceive the induced benefits.

- The second answer is to design formalisms or tools that are easy to use for the software engineer. Much work has been done in trying to design graphical formalisms, supported by GUI tools. This approach was successful in several projects. For example, the AMPHION project uses a graphical relational language as input that was used with success by experts in astrophysics to formalize their problems.
- The third answer is the integration of the tool in the development process. Many novel approaches are based on new activities and new notations. They require a revolution in the way software is developed. In most cases, a company cannot afford such a revolution because it not only requires to educate its engineers, but also to deeply reorganise the company itself or its processes (and hence loose some maturity during a transition period).

Obviously, the cost of integrating new techniques will be compared to the expected benefits. From there, several approaches can be adopted by ASE tool designers: some will try to maximize the benefits of their tools, whatever be the cost, others will develop more modest tools which bring less benefits at a lower cost.

These challenges are not new and they have already been reported in various studies related to ASE, software engineering or formal methods. The rest of this paper will present the TOBIAS tool[1] , aimed at the generation of large testsuites and will discuss how these challenges are taken into account by the tool.

2 TOBIAS

TOBIAS is a tool for the automatic generation of test cases from a given test pattern. Writing test cases is a very tedious and repetitive task, especially when we need a large set of test cases. This is where TOBIAS helps produce a large set of similar test cases. We have experimented that many test cases feature the same sequence of operations but with different parameter values [1]. Other sequences may also differ by exchanging an operation with a similar one.

[1]TOBIAS has been developed within the COTE project of the french national network in software technology (RNTL).

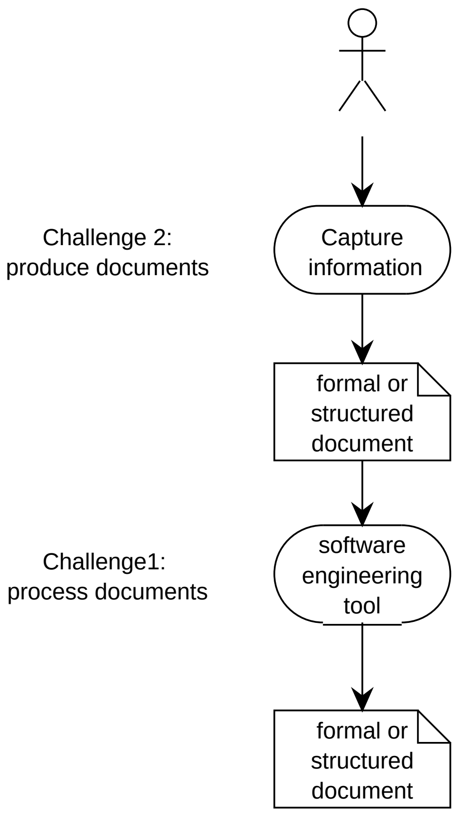

Challenge 2: produce documents
Capture information
formal or structured document
Challenge1: process documents
software engineering tool
formal or structured document

Figure 2. Challenges of ASE research

TOBIAS allows the user to define a set of relevant values for each operation parameter or to identify sets of similar operations (named "groups" in TOBIAS). These form the basis for the definition of test patterns (named "test schemas" in TOBIAS). A test schema is a bounded regular expression over operations and groups. Test schemas are then unfolded by TOBIAS into a large set of test cases.

We expect that TOBIAS will help test engineers generate more tests cases and in a more systematic way for about the same effort than "manually" written test cases. Generating more tests may increase the confidence in the testsuite. Generating these more systematically will help cover more behaviours of the system, including situations that could be overlooked or forgotten by the test engineer. So we may reasonably expect that TOBIAS increases the chance of detecting errors.

In a recent experiment [3], we generated a large test suite (4320 test cases) and compared it to a manually produced test suite (45 test cases). Our experiment showed that

- The testsuite generated by TOBIAS discovers more errors than the manual testsuite. It also exercised some known errors in several different ways, making the testsuite more robust towards evolutions of the specification.

- Writing TOBIAS test schemas requires a similar effort than writing the small manual testsuite.

2.1 Principles of TOBIAS

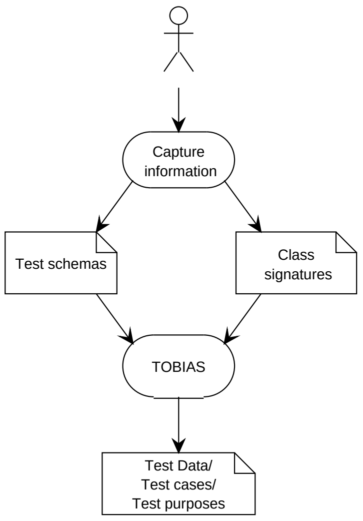


Figure 3. Basic view of TOBIAS

TOBIAS takes two inputs (Fig. 3):

- the signatures of the classes of the application under test

- a test schema

and produces sequences of method calls which can be used as test data, test cases (if an oracle is available) or test purposes such as the ones required as input of the TGV tool [2].

For example, let us consider a simple class "IntegerSet" with two methods: "add(v:int)" and "remove(v:int)". Starting from this signature and the following test schema:

```
add(x)ˆ1..3;remove(y)ˆ0..2
where
x : {0,1,2,3}
y : {0,1,2}
```

TOBIAS will generate all sequences which feature one to three calls to "add" with values 0 to 3, followed by zero to two calls to "remove" with values 0 to 2. In total, this schema generates 1092 different sequences ((4 + 4*4 + 4*4*4) * (1 + 3 + 3*3)). Fig. 4 shows some of the generated sequences.

```
1: add(0)
2: add(1)
3: add(2)
4: add(3)
5: add(0); add(0)
6: add(0); add(1); add(2); add(3)
7: add(3); add(2); add(1); add(0)
8: add(0); remove(0)
9: add(0); remove(0); remove(0)
10: add(1); remove(0)
11: add(0); add(1); add(2); add(3);
    remove(0); remove(1); remove(2)
```

Figure 4. Some test cases generated by TOBIAS

They correspond to classical test cases like adding or removing several elements, but also adding or removing twice the same element, or trying to remove an absent element. Actually, in order to turn the output of TOBIAS into test cases, you need an oracle which will evaluate the effects of method calls and deliver a verdict.

The test schema (4 lines) specifies this large set of test cases, and TOBIAS helps the software tester to construct this test suite at a lower cost than manual production of the test cases. This simple example shows how test schemas generate test cases by iterating over parameter values and the number of successive calls to the same method. TOBIAS also allows to iterate over a set of instances of the class or over a set of methods.

TOBIAS is a tool that amplifies the work of the test designer. The tool is based on a simple idea: to exploit similarities between test cases in order to specify these by a generative pattern. In the next sections, we will see how it addresses the challenges of ASE.

3 Tobias and complexity

Unfolding a TOBIAS test schema is not intrinsically difficult, but the output is subject to combinatorial explosion: the tool generates a large number of test cases, and it is precisely the purpose of the tool. Having a large number of systematically generated test cases helps finding more errors because it exercises the combinatorial complexity of the application. Still, the testsuite may not be arbitrarily large because its execution may require untractable resources for minor additional benefits. Therefore, the challenge of TOBIAS users is to produce an optimal number of test cases.

To this end, it is necessary to select a subset of the generated testsuite. Two kinds of test cases should be eliminated from the testsuite:

- redundant test cases:
- non-conform test cases.

For example, in Fig. 4, test cases 1,2,3 and 4 are redundant, because they correspond to add a single element and the result of the test should not be influenced by the actual value of the parameter. Similarly, test cases 6 and 7 are also redundant. Test cases 9 and 10 could correspond to non-conform test cases if the pre-condition of remove requires that the element that is removed is an element of the set.

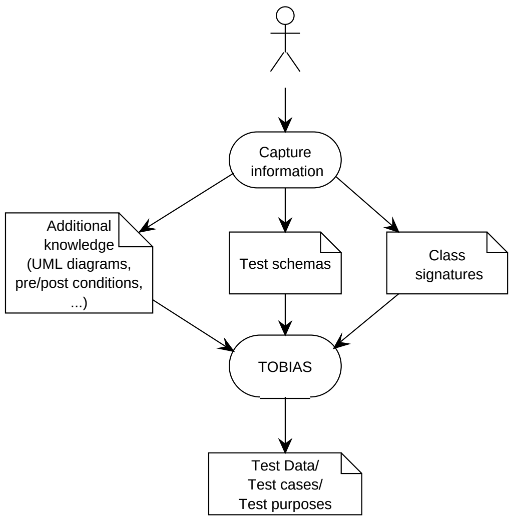

Capture information
Additional knowledge (UML diagrams, pre/post conditions, ...)
Test schemas
Class signatures
TOBIAS
Test Data/ Test cases/ Test purposes

Figure 5. Providing more information to TOBIAS

In order to detect redundancy and non-conformance, additional information must be provided to the tool (Fig. 5). This information can take two forms.

- Several specification documents (UML diagrams, state machines, pre-/post-conditions,…) can be exploited in order to detect and eliminate non-conform test cases. This process can become difficult if these specifications are complex or written at a very high level of abstraction. They may require the use of verification techniques with their usual cost. Still, if the tester does not need to detect every non-conform test case, he may use simpler information or less expensive algorithms which will detect some kinds of non-conformances. For example, TOBIAS will soon take into account the relations of the UML class diagram in order to detect test cases that feature non-conform communication between instances [2]. This kind of verification is quite elementary but may lead to the elimination of a significant number of non-conform test cases. We are also working on the integration of TOBIAS with the CASTING tool [4] which takes into account pre-/post-conditions and a state transition diagram to detect non-conformant test cases. Here, the tool involves a more complex computation, using constraint logic programming techniques.
- The language for expressing test schemas can be extended to allow the test engineer to provide a finer description of the test schema and to express test hypotheses. For example, in the IntegerSet class, values of integers are not significant. This equivalence can be provided as a test hypothesis and used by TOBIAS to avoid redundant test cases. Currently, we are experimenting an extension of the language that allows the test engineer to specify constraints on the values of the parameters used in a test case. For example, the user can express that the x_i parameters are pairwise distinct, which would eliminate test case 5.

To summarize this point, TOBIAS is exposed to the combinatorial complexity of the generated testsuite. This complexity is intrinsic to the tool because we wanted a tool that would systematically test combinations of values and method calls. Still, it has to be controlled because too large testsuites may require untractable resources without providing additional benefits. Therefore several techniques are used to generate more pertinent test cases, but these techniques require additional domain information about the tests or about the application. As for many ASE tools, mastering the complexity requires additional domain knowledge. In the next section, we will see how this knowledge is captured in the context of TOBIAS.

4 Capturing knowledge for TOBIAS

Fig. 3 shows that TOBIAS requires two kinds of inputs: a test schema and signatures of the classes. In many applications, these signatures can be extracted either from the code of the application under test, or from its UML specification. The current version of TOBIAS is based on the UML class diagram because the tool was developed in a project where the availability of such documents is taken for granted. Very soon, we plan to allow this extraction from Java source code also (Fig. 6).

If signatures are extracted from existing UML specifications or from Java source code, TOBIAS can be used by

[2]Actually, a test schema allows to specify not only the method invoked but also the instance which will activate the method call, and the instance which will process the call.

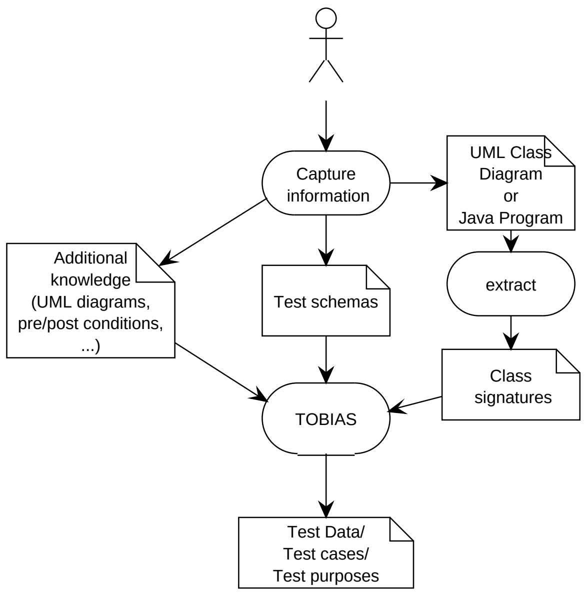


Figure 6. TOBIAS with additional knowledge

simply providing a test schema. The example of Sect. 2.1 shows that test schemas are expressed in a few lines. The current version of TOBIAS also provides a graphical user interface to help the user define a test schema. For example, the interface prompts the user for values of the parameters, or for names of the instances which will execute these methods. It must be noted that the test schema encourages the user to provide specific information about the application under test. For example, the choice of values for the method parameters forces the user to analyse which values are of interest; the quality of the information provided by the user has a direct effect on the quality of the test suite.

TOBIAS has been designed to allow a new user to start using the tool at low cost. Starting from an existing class diagram, the user only has to provide a first test schema. He will immediately get a first large sequence of method calls. We expect that the user will then try to refine this sequence either by using the extensions of the schema language (e.g. constraints), or by providing more information to the tool about conformance issues. This conformance information is actually a specification of the application under test. The planned extensions of the tool will try to exploit existing UML diagrams of the application under test (class diagram, state transition diagram, OCL pre/post-conditions). If such specifications don't exist, we hope that the benefits of producing more pertinent and conformant test suites will encourage the software engineer to specify parts of the application. Again, we plan that the tool will be able to bring small but useful results from simple elements of the specification (e.g. the relations in the class diagram), and more precise and complete results from detailed specifications.

Another way to encourage the user to provide such specifications is to integrate TOBIAS with other test generation tools which are based on the same kinds of diagrams. In the COTE project, TOBIAS will be combined with UMLAUT/TGV and CASTING, which both exploit elements of UML specifications. Specifications, preferably executable ones, can also be used as a basis for the test oracle, in order to turn sequences of method calls into real test cases.

In summary, our approach is to provide the first benefits at the single cost of expressing test schemas. It exploits the availability of several documents (source code, UML specifications), and hence should be easier to integrate in the company process. Then, it encourages the evolution of the processes by delivering new benefits for additional elements of specification.

5 Conclusion

This paper has shown how the TOBIAS test generator tries to face two of the major challenges of each ASE tool: complexity and information acquisition. Our approach is to start with simple solutions that fit into standard software development processes. Then we intend to gradually incorporate more refined processing based on more precise information.

Acknowledgments

TOBIAS is the result of joint efforts with Lydie du Bousquet, Pierre Bontron, Olivier Maury and several students of UJF. I'd also like to thank our colleagues of the COTE project, in particular our colleagues from Gemplus who provided motivation for developing the tool.

References

[1] L. du Bousquet, H. Martin, and J.-M.Jézéquel. Conformance testing from UML specifications - experience report. In *UML2001 Workshop on Practical UML-Based Rigorous Development Methods*, Toronto, 2001.

[2] T. Jéron and P. Morel. Test Generation Derived from Model-checking. In *Computer Aided Verification (CAV'99)*. LNCS 1633, Springer, 1999.

[3] O. Maury, Y. Ledru, P. Bontron, and L. du Bousquet. Using test hypotheses to build a UML model of object-oriented smart card applications. In *Int. Conf. on Software and Systems Engineering and their Applications (ICSSEA)*, Paris, 1999.

[4] L. Van Aertryck, M. Benveniste, and D. Le Métayer. Casting: A formally based software test generation method. In *The 1st Int. Conf. on Formal Engineering Methods, IEEE, ICFEM'97*, Hiroshima, 1997.

Sleeping at Night: Perpetual Monitoring of Environmental Assumptions

Stephen Fickas[1], Max Skorodinsky[1], Martin Feather[2]

[1]Computer Science Department, University of Oregon
[2]Jet Propulsion Lab, Pasadena

1. Introduction

We are interested in failure. In particular, we are interested in failures that are discovered at analysis time, but are left to simmer. Figure 1 places our work in context.

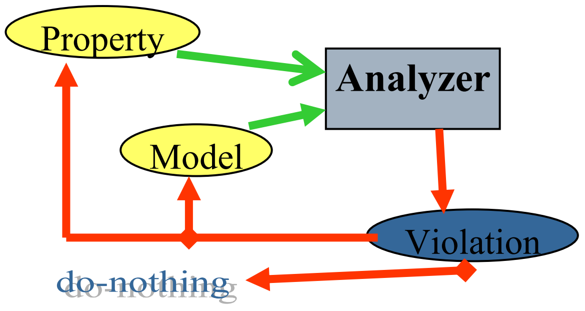


Figure 1: Reactions to failure

Given a violation, the traditional approach is to change the model to remove the cause of the violation (the arrow leading from violation to model). In essence, "design out" the problem [Garcez et al, 2001]. A less traditional, but still interesting approach is to change the property (the arrow leading from violation to property) so that the violation is eliminated [van Lamsweerde&Letier, 2000; Durney, 1993]. We are interested in a third approach that is quite untraditional in the formal analysis literature but one that seems quite common. The "approach" is to acknowledge that a violation is possible but is simply not going to be handled by changes to the model or property. There are various reasons that might be given for taking this approach, but they all lead to a cost/benefit argument: the cost of trying to design them out or lower our goals is not worth the benefit of getting rid of them.

This paper takes up one aspect of the do-nothing approach: the assumptions underlying cost/benefit arguments. In particular, we are interested in arguments based on the likelihood of the environment acting in certain ways. (The Model of figure 1 is actually a representation of the artifact under design as well as its environment. In our project, we use closed models that include both.) Our experience is that domain experts, participating in the analysis process of figure 1, often make statements about the expected behavior of the deployment environment. While domain experts are domain experts, it can be difficult to predict either the initial environment where the artifact will run, or a changing environment looking out over time. We will argue in the remainder of the paper that we need to carry analysis arguments to runtime, leading to what we call "runtime requirements engineering". We will introduce an example and then discuss future work.

2. Example: The Fault Protection Engine

Our group took on the task of analyzing basic liveness and safety properties for a one component of the operating system for a spacecraft. The component is called the Fault Protection Engine or FPE for short. The FPE component is interesting in its own right: it attempts to diagnose and treat runtime faults that occur during the mission. In some ways, it is a runtime instantiation of figure 1! However, it is not the details of the FPE that we will focus on, but our attempts to analyze its behavior to discover problems.

Our group was given a state chart representation of the FPE component. This representation was developed by domain experts at JPL. We were asked to use model checking to verify that the FPE component, as represented by the state charts, met simple distributed system properties such as non-termination, non-starvation, deadlock free, etc. A large part of our effort was getting to a Promela model (we chose to use Spin as our model checker) that actually gave useful results. We have documented this effort in a separate paper [Feather et al, 2001]. Where we take up here is at the final version of our model, one capable of finding violations in reasonable time. To provide some context, we will use a piece of the original state chart diagram as illustration (see figure 2).

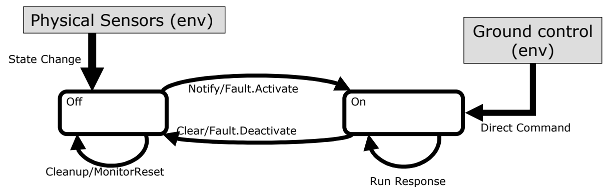
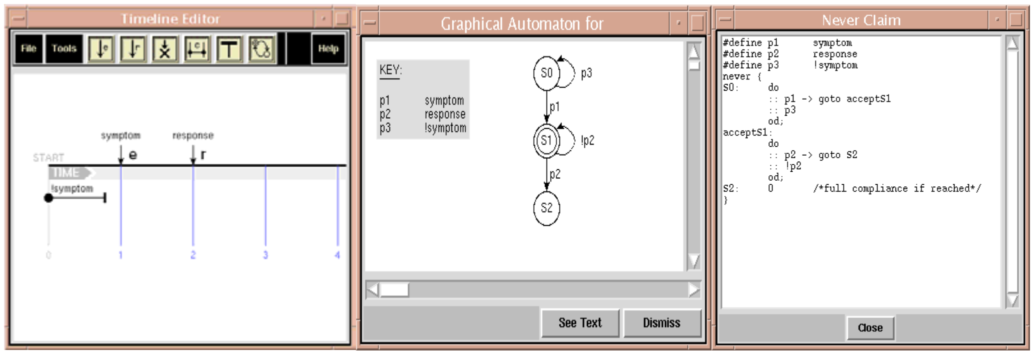


Figure 2: the basic detect/repair cycle

There are two environmental pieces in the figure, the physical sensors on the spacecraft and the human staff (ground control). Both provide external events to the FPE component itself. The general operation is for faults to be detected and then queued up for processing. Once a fault has had a repair routine run, it is cleared. Ground control can ask that a repair routine be run irrespective of any actual faults detected on the spacecraft.

Our modeling approach is to start with a wildly under constrained environment and gradually refine it as called for. This same approach proved useful in our earlier work in implementing highly non-deterministic specifications [Fickas, 1985; London&Feather 1986]. Using this approach, both physical sensors and ground control were allowed to “run open”: on every cycle, they had the opportunity to produce an event for the FPE component. Before describing violations, we need to describe a property we were interested in proving. In English, we wished to show that for every fault detected, a response was eventually run. We are going to use a tool that we have found valuable in stating properties such as this, the Timeline editor tool [Smith et al, 2001]. Figure 3a shows the GUI for the tool with the specification of our property. Figure 3b shows the Buchi automata that the tool produces. Figure 3c shows the Promela/Spin never-claim that is actually inserted into our model.



```
#define p1      symptom
#define p2      response
#define p3      !symptom
never {
S0:     do
        :: p1 -> goto acceptS1
        :: p3
        od;
acceptS1:
        do
        :: p2 -> goto S2
        :: !p2
        od;
S2:     0       /*full compliance if reached*/
}
```

Figure 3a: GUI spec 3b: Buchi automata 3c: never-claim

When we inserted the never-claim into our FPE model with the unconstrained environment, we turned up the following violation: the input buffer to the repair component could be overrun with repair requests. In other words, the environment could flood the FPE with enough faults so that some were lost because of finite buffer size. Figure 4 is a representation of the states leading towards a full buffer, and hence failure: once a buffer is full, new messages are lost.

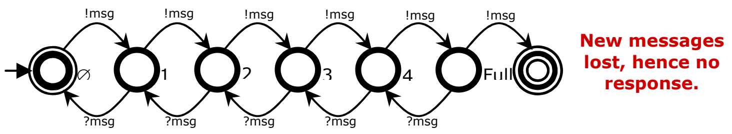


Figure 4: progression towards buffer overflow

When we presented this violation to domain experts, they had the following comments:

1) There are safeguards built-in to the hardware sensors that prevent them from flipping back and forth too quickly spurious signals).
2) If there actually are enough *real* faults to fill the buffer, there are bigger problems than buffer overflow.
3) Our experience on past missions says that this is not a problem worrying about.

Given these comments, we modified the environment model. In essence, we defined environmental assumptions that said the environment was not allowed to fill the buffer. With these assumptions in place, one violation was eliminated. At this point, a new violation arose: ground control could flood the input buffer with requests. We dutifully presented this new violation to domain experts. The response was as follows:

1) Ground control staff are well-trained and would not flood the spacecraft with requests.
2) Experience says that ground control rarely are required to send requests.

However, after further discussion, the possibility of a “flight rule” was put forth. Flight rules are constraints on the actions humans can take when interacting with the spacecraft. In our biased view, they fall out of composite system design [Fickas&Helm, 1992]. All components in a composite system must control their behavior to reach a larger goal. Flight rules are a manifestation of control of the human agents in the system. It is important to note that flight rules are currently not checked by machine at runtime: ground staff is expected to follow them in the

form of an operations manual. A paraphrase of the flight rule proposed was “do not, through combination of onboard requests and ground control requests, exceed the request-buffer size”.
To summarize this first example, several violations were found of the requirement that all requests be acted upon. The two we have presented were both explained away, i.e., the do-nothing approach was chosen. The exception was the possibility of adding a flight rule to the operations manual of ground control. The question is whether we can sleep at night after we are finished with the analysis phase. Will our environment assumptions all hold once the spacecraft is launched and on mission?

3. What makes us sleepless

There are two concerns we have after completing the FP engine case study. First, did we turn up all the environmental assumptions that are being made? In essence, did we uncover all the ways the system can fail at the hands of the environment? We are feeling skeptical about this given the lack of elicitation methodology for building environment models in Spin. As previously noted, we did follow an informal approach documented in [London&Feather, 1982], one that starts with a completely unconstrained environment and gradually adds constraints as needed. However, we used mostly ad hoc methods to decide what to model and what to not model as we interacted with the domain experts. Second, some of the do-nothing failures that we did turn up during modeling seemed to warrant some further validation at runtime. A brief scan of the failure assessment literature shows a non-trivial number of system failures due to erroneous engineering assumptions about the runtime environment. We do not believe you can remove the need for assumptions – no engineered artifact would ever get built if it had to handle all worst case scenarios thrown at it by its environment. However, it seems that once these assumptions are explicated (by a good elicitation method!), we can do more than do-nothing.

4. Looking to Cryptography for Assumption Elicitation

In our FP engine case study, we used a seat-of-the-pants approach to working with domain experts. In particular, we wrestled with mundane and uninteresting modeling issues at the same time we pursued the main goal of accurately representing the composite system. One might argue that the tool we chose, Spin, was the problem. However, our experience in our year-long modeling seminar is that all modeling tools have their quirks and none can said to be easy to use when starting from scratch. The question is do we have to start from scratch? We look with interest to the more general software engineering area where notions of patterns and frameworks are proposed as building blocks. We conjecture that the same ideas, if not the same content, can find a home in formal modeling efforts. In this section, we will provide what we believe are starting points for modeling frameworks, and in particular, frameworks that focus on the environment model of a composite system.

We suggest that we can take a general modeling tool, Spin, and develop the engineering practice around it that will give us what we want. What do we want? We want a method or framework that focuses on the behavior of the environment in a composite system. For the sake of discussion, we will use an existing modeling methodology from the cryptographic-protocol world, strand spaces. We propose strand spaces for at least three reasons: (1) It is environment-centric. It puts the environmental component of a composite system first and foremost. In fact, its whole raison d'être is to explore means that the environment, acting badly, can cause a system to fail. (2) It is tool independent. It really is a methodology or way of thinking about a problem. (3) There is some evidence that it can be coupled with Spin (discussed in next section). We will next introduce the strand space concept and discuss its applicability to the larger problem of environmental monitoring. At the end of the section, we will discuss similar environment-centric frameworks and how they might fit into a modeler's toolbox.

Strand Space as a Framework

The strand space is a framework for modeling security protocols [Fabrega et al, 1999]. The strand space modeling technique provides a means for succinctly specifying the actions of legitimate protocol participants. Additionally, the framework provides an explicit model of a protocol *penetrator* that is independent of a specific protocol. In other words, the strand space captures a means to reason about, as well as model, a malicious entity which exists/acts in the same system as the legitimate entities. Furthermore, the technique establishes a bound on the capabilities of the malicious entity, which allow for rigorous proofs of security properties of a specific protocol based on the bounded capabilities of the penetrator.

Although methods for proofs of protocol properties such as authentication and secrecy are clearly presented by the framework's designers, a major limitation of this technique is that the models as well as the proofs must be produced by hand. There is, however, at least one documented study that reports success with using the Spin automated model checker to verifying a security protocol, where the protocol has been represented as a model built with techniques nearly identical to those of the strand space mode [Maggi&Sisto, 2002]. The following introduction to the strand space modeling and verification technique will combine an explanation of concepts central to strand space with a demonstration of how these concepts can be translated into a model built in Promela. A famous security protocol, the Needham-Schroeder-Lowe Public-Key Authentication Protocol, will be used to illustrate the main ideas/concepts.

A central concept to the Strand Space model is a *strand*, which is a set of nodes, where each node captures an action (specified by a protocol) valid for the entity to which the strand corresponds. A node is an element of the set N (all nodes in the model), such that every node $n \in N$ belongs to a unique strand and each node in a strand is indexed. A predecessor relation is defined on the nodes in the same strand. This is done, by connecting two nodes, one of which precedes the other, with a special symbol ($\Rightarrow$). If n_1, n_2 are nodes, $n_1 \Rightarrow n_2$ means n_1, n_2 occur on the same strand and index (n_1) = index $(n_2)-1$. In addition to the immediate causal predecessor relation, a causal link relation is defined on nodes occurring in different strands. If n_1, n_2 are nodes, $n_1 \rightarrow n_2$ implies that n_1 sent a message which was received by n_2. Each node contains a *term*, which corresponds to a message either being sent or received. A term is said to *originate* on a node if that term is preceded by the symbol + (plus) which also corresponds to a message being sent whereas a term, preceded by the symbol – (minus), corresponds to a message being received. A term *uniquely originates* on a node if it originates on a unique node $n \in N$. In the domain of cryptographic protocols, a uniquely originating term has the significance of representing a nonce or a session key.

The designers of strand space focus on two security properties in their work: agreement and secrecy. In both cases the authors use the concept of a uniquely originating term and causal relations between nodes to establish proofs of the two properties. For example, the agreement property is defined in terms of participants committing to a run of a protocol using a data item on which they agree. This property is verified by checking that a bundle that contains a strand which receives a data item *x* has a unique strand which sends *x*. The Needham-Schroeder-Lowe protocol in conventional specification illustrates some of the strand space concepts presented thus far:

1. A $\rightarrow$ B: $\{N_a A\}_{K_B}$
2. B $\rightarrow$ A: $\{N_a N_b B\}_{K_A}$
3. A $\rightarrow$ B: $\{N_b\}_{K_B}$

The goal of this protocol is that the legitimate participants A and B gain possession of N_a and N_b and associate these values with each other. No other party should have access to these values. The Needham-Schroeder-Lowe protocol in Strand Space:

$$\begin{tikzcd}
\text{strand } s' & & \text{strand } s'' \\
\textbf{\textit{a node:}} & & \\
\langle s', 1\rangle & & \langle s'', 1\rangle \\
\textbf{\textit{a sent message:}} & & \textbf{\textit{a received message:}} \\
{[\,+\{N_a A\}_{K_B}\,]} \arrow[rr] \arrow[d, Rightarrow] & & {[\,-\{N_a A\}_{K_B}\,]} \arrow[d, Rightarrow] \\
\langle s', 2\rangle & & \langle s'', 2\rangle \\
{[\,-\{N_a N_b B\}_{K_A}\,]} \arrow[d, Rightarrow] & & {[\,+\{N_a N_b B\}_{K_A}\,]} \arrow[ll] \arrow[d, Rightarrow] \\
\langle s', 3\rangle & & \langle s'', 3\rangle \\
{[\,+\{N_b\}_{K_B}\,]} \arrow[rr] & & {[\,-\{N_b\}_{K_B}\,]}
\end{tikzcd}$$

Thus far, only legitimate protocol participants have been illustrated. In terms of modeling these in Promela, the legitimate entities can be expressed as processes and the actions allowed by them by the protocol as atomic actions. This is described in detail in [Maggi&Sisto, 1999].

Adding the Environment

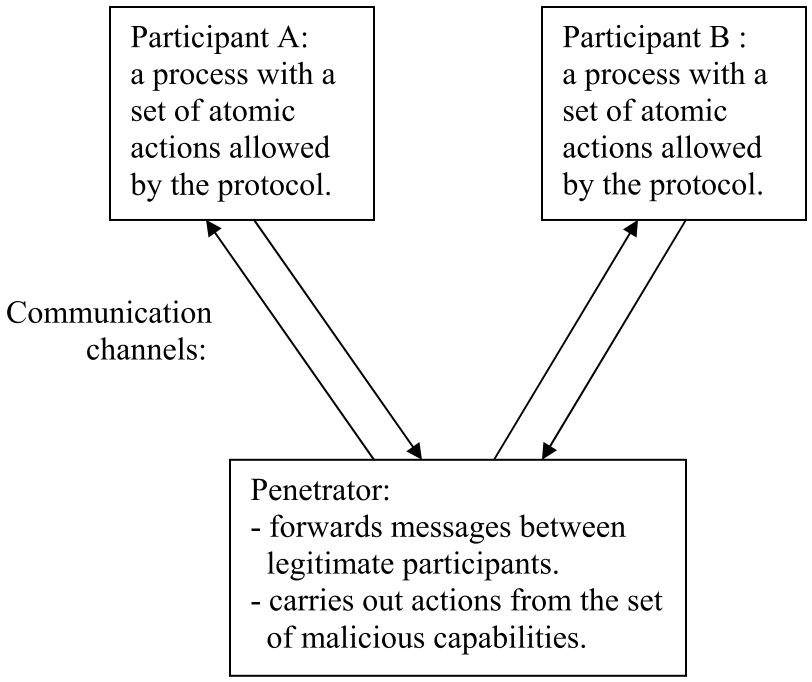
The most powerful concept developed in strand space is that of the model of a penetrator, or malicious entity. Using well-accepted notions of a security protocol bad guy, the authors present a bounded model of a penetrator, which embodies a finite set of capabilities. For example, the following are a few of the capabilities of the penetrator taken from [Fabrega et al, 1999]:

M[a]: send message *a* given that *a* is a term initially known to the penetrator, [+a].
F[a]: receive message *a*, [-a].
T[a]: tee, [-a, +a, +a].
V[a, b]: concatenation, [-a, -b, +ab].
R[ab]: separation into parts, [-ab, +a, +b].

…

The verification of a property is done with respect to these capabilities and any data of which the penetrator is in possession. A notion of an infiltrated strand space is defined, which consists of strands that represent legitimate protocol participants as well as the penetrator strands, where each strand corresponds to one possible penetrator action. According to the authors of strand space, the penetrator model can be easily expanded to include other capabilities without necessitating any other modifications to the framework.

The model of the Needham-Schroeder-Lowe protocol presented in [Maggi&Sisto, 2002] very closely resembles that of the infiltrated strand space. As mentioned previously, the legitimate participants of the protocol are able to take atomic actions prescribed to them by the specification of the protocol. The penetrator is injected into the space by connecting each legitimate participant to the penetrator with a Promela channel. Thus, the penetrator is the first recipient of any protocol dictated action. Furthermore, once the penetrator receives a message destined to a legitimate participant it forwards the message to the intended recipient. Most importantly, the penetrator is able to commit any of the actions in its set of capabilities at this point. Thus, the infiltrated space is an extremely compact way to model the influence of malicious or otherwise interfering entities with the actions of legitimate ones. The following is an illustration of the infiltrated space in a Spin model.


Participant A: a process with a set of atomic actions allowed by the protocol.
Participant B : a process with a set of atomic actions allowed by the protocol.
Communication channels:
Penetrator:
- forwards messages between legitimate participants.
- carries out actions from the set of malicious capabilities.

Is There a Framework Somewhere Here?

We believe the answer is yes, there is a potential framework for eliciting failures and environmental assumptions. Our belief rests on (1) the ability to provide different penetrator actions for different domains, and (2) the ability of the Promela architecture above to contain the state-space explosion. Neither of these beliefs has been satisfactorily evaluated by us: they remain conjectures. (We have recently used the Promela architecture above to model another crypto protocol, but have not attempted to use it on a non-crypto problem.)

In summary, the general research questions around the strand space approach to assumption elicitation are as follows:

a. Can the method be refocused, or at least focused more broadly, on embedded systems. Its general model of actors/agents of the artifact/system working under deterministic rules is a fit with applications like the FP engine. More generally, one does not often want non-determinism in designed artifacts. The strand space focus is on the environment: how can the environment, viewed as an agent, screw up the system. Can this all be translated into a methodology that allows modeler and domain expert to explore environmental assumptions?

b. Assuming that the strand space method is effective for embedded systems, what tool support can we give to it? Given that we propose to use Spin as our base modeling tool, what software engineering tools can we provide for deriving environmental assumptions and dependability metrics from the strand space model? Do patterns or frameworks make sense in Spin in the same way they do in more general programming languages? More directly, can we develop a strand space framework for Spin, i.e., add a software engineering layer on top of the Spin modeling tool?

Beyond Strand Space: A Framework Toolkit

Looking a bit more closely at the strand space concept, it is based on *exchange protocols*. It centers on a non-cooperating environment thrown in with a set of correctly functioning system components. This covers some of the problems we turned up in our FP engine study. However, there are clearly other ways that a composite system can be viewed. For instance, the (human) ground control component is assumed to act correctly. But if this component acts incorrectly it can have disastrous results. A framework that focuses more closely on this problem is the non-repudiation protocols that deal with a trusted component acting badly. Pascoe describes the APPROVE framework for modeling these types of problems [Pascoe, 2001]. Still other problems crop up when dealing with black-box components not under project control. Recent work on Interface Modeling [Alfaro&Henzinger, 2001] centers on the types of assumptions one must make about such components for the system to avoid failure. In our FP engine study, domain experts had a tendency to treat sensor hardware in this way. Assumptions were made about the way it would behave as seen through its interface.

It is tempting to propose an effort that unifies all of these different methods into a single framework. However, we propose to follow a different path: different frameworks for different pieces of a composite system. In essence, we are bowing to the fact that there is not one "environment" that needs to be reckoned with, but a set of relationships between system components and non-system components. It is even greyer than this. For instance, the domain experts do not view ground control as part of the environment. And in some ways they are right: there is a degree of control over it. However, its behavior remains only plausible, not guaranteed. One can view it as the "human user" of the system. Other components of the environment are engineered but come as black-box entities. Yet other components of the environment reflect the physical aspects of deep space missions. Instead of lumping these together, we propose to build a collection of frameworks to match known system-environment relationships. The goal is to supply a modeler with a framework that teases out the environment model, and in particular, the environmental assumptions that exist with different environment components, e.g., with humans in the loop, with black-box components, with white-box trusted components, with the physical environment. In a larger sense, this can be viewed as falling under the domain-specific rubric: power by constraint. Build methods and tools that leverage knowledge of constrained aspects of a system.

5. Sleeping Better: Monitoring Assumptions

We would like to consider a compromise to the do-nothing approach: go ahead and make assumptions at analysis time, but record them and carry them to runtime for monitoring. To give a bit more context, we are interested in knowing more than simply fail/no-fail information about an assumption (e.g., more than what would be given by an assert statement in the deployed code). In particular, we are interested in observing the states leading up to assumption failure, and using that information in various ways:

- At the least, we might be given a warning that failure is certain but some time remains to "man the life boats". The Mocha group calls these doomed states: the system will fail under all future events but has not reached the failure state yet [Alur et al, 1998].
- Better, we might be able to provide some control over the environment. For instance, both our examples had human agents as part of the larger composite system. There is potential to influence their actions (e.g., actual monitoring of flight rules in the FPE example).
- Best, we might be able to change the artifact (the components under our control) to head off failure without relying on help from the environment. In some systems, this means an artifact moving to a fail-safe mode until a danger has passed (e.g., the FPE example).

We have done some experimentation with assumption monitoring, and that work is described in detail in [Fickas et al, 2002]. Here we will briefly outline our approach and describe what we see as future research directions.

Step 1. We capture environmental assumptions in the Timeline editor tool (see figure 3a). The Timeline editor provides a means of explicitly stating assumptions made about the environment of a required system property.
Step 2. We take the output of the editor, a Spin never-claim (see figure 3b,c), and use it as the specification of a runtime monitor.
Step 3. We translate the never-claim to an Emu event tree. Emu is a tool we have developed for monitoring runtime events (www.emu-project.org). Our translator converts the state-machine represented by the never-claim into the equivalent Emu monitor represented by an event tree.
Step 4. The Emu monitor waits for a triggering event/action, and then provides intermediate information about the current state of the tree (state machine).

To date we have been able to build visual monitors of Emu event trees. These have proven useful to system staff monitoring the accuracy of environmental assumptions. We have yet to connect a monitor with an adaptation mechanism.

Open Issues of Monitoring

We find that tools like the Timeline editor are sorely needed in the formal methods area. They provide an abstract view of scenarios that is at just the right level. And furthermore, we use the tool both for analysis time verification and for the start of the path to runtime monitoring. The problem is that tools of this ilk, ones designed for model checking, carry model checking baggage. The output of the Timeline editor is meant to be integrated with a larger Promela program. And this integration is based on the non-deterministic search space generated by Spin. Further, the editor's output (in reality, a Buchi automata) is based on Linear Temporal Logic (LTL), which is founded on infinite time sequences. Both of these attributes of model checking, non-determinism and infinite time, are knotty issues when moving to a deployed system. We are not the first to notice this: other groups have wrestled with the use of finite traces to prove properties stated in terms of infinite time. However, we are aware of no other work in the area of assumption monitoring (as opposed to runtime verification) that has satisfactorily dealt with the twin problems. Have we solved them with our mapping to Emu? No. We have taken a finesse (actually two). First, we change all infinite behavior in a never-claim to be timed. Technically, we turn a liveness issue into a safety issue. Second, we use a deterministic monitor. This will fail to capture alternative paths through the never-claim. While both of these choices, translating to timed and deterministic monitors, has been sufficient for the simple problems we have studied, we would be surprised if they were enough for larger, more complex scenarios.

6. Summary

Our goal was to convince the reader that there is a portion of modeling that is under supported, that of building an environmental model and eliciting environmental assumptions along the way. There appear to be two hard problems: (1) finding a software engineering method that helps with elicitation, and (2) once elicited, finding a means of runtime monitoring. We have discussed two approaches that we believe hold promise.

References

Luca de Alfaro, Thomas A. Henzinger, Interface Theories for Component-based Design, *Proceedings of the First International Workshop on Embedded Software (EMSOFT '01*), Lecture Notes in Computer Science 2211, Springer-Verlag,

Rajeev Alur, Thomas A. Henzinger, F.Y.C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. Mocha: Modularity in model checking. In *Proceedings of the Tenth International Conference on Computer-aided Verification (CAV 1998*), Lecture Notes in Computer Science 1427, Springer-Verlag, 1998, pp. 521-525.

Durney, B., *Requirements Transformations*, PhD Thesis, Computer Science Department, University of Oregon, 1993

F. Javier Thayer Fabrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2/3):191--230, 1999

M.S. Feather, S. Fickas, A. Razermera Mamy, Model-Checking for Validation of a Fault Protection System, *Proceedings 6th IEEE International Symposium on High Assurance Systems Engineering*, Boca Raton, Florida, October 23-24 2001. IEEE Computer Society

Fickas, S., Automating the transformational development of software, In *IEEE Transactions on Software Engineering*, Vol. 11, No. 11 Nov. 1985

Fickas, S., Helm, R., Automating the design of composite systems, *IEEE Transactions on Software Engineering*, June, 1992

Fickas, S., Beauchamp, T., Razermera Mamy, A., Monitoring Ephemeral Requirements, Computer Science Tech Report 10-02, University of Oregon, May 2002 (www.emu-project.org)

A. S. d'Avila Garcez, A. Russo, B. Nuseibeh, and J. Kramer, An Analysis-Revision Cycle to Evolve Requirements Specifications , *Proceedings of 16th IEEE International Conference on Automated Software Engineering (ASE-2001*), pp.354-358, 26-29 November 2001, San Diego, USA.

P.E. London & M.S. Feather, "Implementing specification freedoms", *Readings in Artificial Intelligence and Software Engineering:* 285-305, 1986, Morgan Kaufmann, 1986 [Originally published in Science of Computer Programming (2): 1-131, 1982]

P. Maggi and R. Sisto, Using SPIN to verify security properties of cryptographic protocols*, Spin Workshop 02,* July 2002

J. S. Pascoe, R. J. Loader and V. S. Sunderam, Working Towards The Agreement Problem Protocol Verification Environment, *2001 Communicating Process Architectures (CPA 2001).*

Smith, M., Holzman,G., Etreeami, K*.,* Events and Constraints: A Graphical Editor for Capturing Logical Requirements of Programs*, International Symposium on Requirements Engineering – RE01,* Toronto, August 2001

van Lamsweerde, A., E. Letier , Handling Obstacles in Goal-Oriented Requirements Engineering , *IEEE Transactions on Software Engineering*, *Special Issue on Exception Handling*, Vol. 26 No. 10, October 2000, 978-1005.