Abstract. The evolution of a new technology depends upon a good theoretical basis for developing the technology, as well as upon its experimental validation. In order to provide for this experimentation, we have investigated the creation of a software testbed and the feasibility of using the same testbed for experimenting with a broad set of technologies. The testbed is a set of programs, data, and supporting documentation that allows researchers to test their new technology on a standard software platform. An important component of this testbed is the Unified Model of Dependability (UMD), which was used to elicit dependability requirements for the testbed software. With a collection of seeded faults and known issues of the target system, we are able to determine if a new technology is adept at uncovering defects or providing other aids proposed by its developers. In this paper, we present the Tactical Separation Assisted Flight Environment (TSAFE) testbed environment for which we modeled and evaluated dependability requirements and defined faults to be seeded for experimentation. We describe two experiments that we conducted on the testbed. The first experiment studies a technology that identifies architectural violations and evaluates its ability to detect the violations. The second experiment studies model checking as part of design for verification. We conclude by describing some ongoing experimental work studying testing, using the same testbed. Our conclusion is that even though these three experiments are very different in terms of the studied technology, using and re-using the same testbed is beneficial and cost effective.

1. Introduction

The evolution of a new technology depends upon a good theoretical basis for developing the technology as well as upon its experimental validation. In most sciences, there is an established mechanism for performing experimentation with a set of tools well adapted for that purpose (e.g., particle accelerators in physics; optical and radio telescopes for astronomy; microscopes, and test tubes in chemistry). Software engineering has no such established protocol for experimental validation. In order to rectify this limitation, we investigated the creation of a software testbed that is useful for such experimentation with software technologies (i.e. development and quality assurance tools, techniques and methods). The testbed is a set of programs, data, and supporting documentation that will allow researchers to test their new technologies on a standard software platform. With a collection of seeded defects and known issues of the target system, we are able to determine if a new technology is adept at uncovering defects or providing other aids proposed by its developers. Creating standardized testbeds would allow more formal validations of many of the technologies now proposed for improvement of software development practices.

This paper, which is an extension of a workshop paper [1], discusses a particular testbed. In order to test its usefulness we had to apply it to a particular software development problem, but from the perspective of different technologies and investigating the feasibility of “reusing” the same testbed for different experiments. We focus in this paper on technologies for achieving software dependability, but the concept is easily extendable to any other attribute one wishes to address.

We investigated software dependability as part of the NASA’s High Dependability Computing Program (HDCP), a cooperative research agreement between NASA and various universities and research centers. The HDCP project investigated achieving high dependability by introducing new technologies developed by the participating research partners. Developing high dependability software requires a) specifying the dependability requirements of the software, b) using technologies to build in high dependability as the software is developed, and c) using technologies to verify that the required level of dependability has been achieved. A standardized testbed would be an appropriate vehicle for this verification. For our testbed, we chose a prototype air traffic control piece of software, Tactical Separation Assisted Flight Environment (TSAFE) [11];[9]. We previously demonstrated that the TSAFE testbed could be used to study technologies that detect similar dependability issues [19]. In this paper, we analyze whether that same testbed could be used in experiments studying very different technologies that detect very different kinds of dependability issues.

The paper is organized as follows. In Section 2, the TSAFE testbed is introduced. In Section 3, the Unified Model of Dependability and its application to the testbed is described. UMD is a requirement engineering approach designed to elicit and model non-functional dependability requirements for a system and was used to specify TSAFE’s dependability requirements. Section 4 presents the development of the testbed from the TSAFE software. As an illustration of using the testbed for evaluation of various dependability technologies’ contribution to building in high dependability, we then present an overview of two completed experiments. Section 5 describes an experiment that studied the Software Architectural Evaluation (SAE) method [25]. The input to SAE is a set of rules that describes desired properties of the software architecture. The output is a diagram that indicates violations of the rules in terms of coupling between components, highlighting extra and missing relations. This experiment was costly to set up since it included the development of the initial testbed. Section 6 describes an experiment that studied the “Design for Verification with Concurrency Controllers (DVCC)” technology. The goal of DVCC is to eliminate synchronization errors in Java programs using model checking technologies in conjunction with design patterns that facilitate automated verification [4]. This experiment turned out to be cost-effective since not much change of the testbed was required, even though model checking is very different from architectural evaluation. Section 7 describes yet another study, which is still ongoing, using the same testbed. It studies testing where test coverage criteria are used to drive the test-case generation process. After summary, discussion, and future work, we provide information for getting access to the testbed.

2. TSAFE Overview

A key strategy for the HDCP initiative was to accelerate adoption of new software engineering technologies by evaluating them on testbeds representative of NASA software. One such testbed is TSAFE, a component of a proposed Automated Air Traffic Control system. TSAFE was defined by Erzberger [11] at NASA Ames Research Center, implemented as a prototype by Dennis at MIT [9], and then instrumented and packaged for experimentation by University of Maryland and Fraunhofer Center Maryland as described in Section 4.

The US Air Traffic Control (ATC) system, consisting of a large network of people and equipment that monitor and direct aircraft, is a critical infrastructure that manages more than 30,000 commercial flights to move 2,000,000 passengers safely each day [12]. The main goal of the system is to keep a safe distance between the aircrafts while achieving efficient air traffic movement in order to minimize delays. The proposed Automated ATC (AATC) software system encompasses all the characteristics that make establishing dependability a challenge, such as distributed computation, concurrency, safety critical functionality, communication protocols, sensitive data and user interaction. Hence, the AATC system raises a large set of implementation issues that propagate to many of its system components.

TSAFE is such a software component of the future AATC designed to aid air traffic controllers in detecting and resolving short-term conflicts between aircrafts. At present, air traffic controllers maintain aircraft separation by surveying radar data for potential conflicts and issuing clearances to pilots to alter their trajectories accordingly. Under the current system, only part of the airspace capability is exploited. Exploiting the full airspace capacity requires the new Automated Airspace Concept to be implemented, which means that automated mechanisms play a primary role in maintaining aircraft separation. The role of TSAFE is to act as a reliable independent safety net from inevitable imperfections in this proposed system. Its aim is to detect conflicts between 2 and 7 minutes in the future and, in its full implementation, to issue avoidance maneuvers accordingly.

TSAFE seemed like a perfect candidate for our testbed application since it contained many of the implementation and behavioral issues present in many large systems, without being overwhelmingly large in terms of lines of code, and its application in the air traffic domain was relevant to NASA, the HDCP sponsor. The implementation of TSAFE used for our testbed provides the air traffic controller with a graphical representation of the conditions (position, planned route, forecasted synthesized route) and of the status (*conformance* or *not conformance* with the planned route) of selected flights within a defined geographical area (Figure 1).

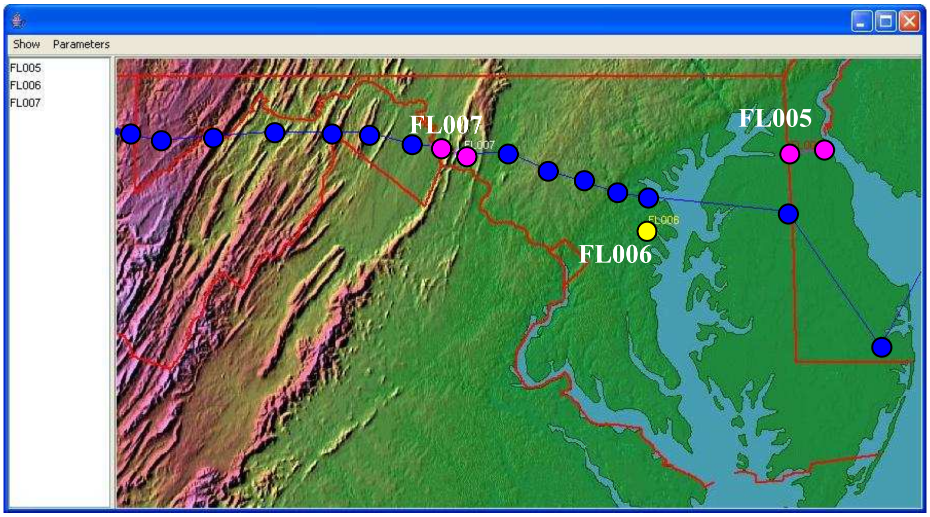


Figure 1. The TSAFE display showing three planes flying over Maryland

3. Modeling TSAFE Dependability

The International Federation for Information Processing [15] defines dependability as *the trustworthiness of a computing system that allows reliance to be justifiably placed on the services it delivers*. "Reliance" is contextually subjective and depends on the particular stakeholders' needs. Different stakeholders will focus on different systems attributes, e.g., availability, ability to avoid catastrophic failures, and prevention of deliberate intrusions, as well as on different levels of adherence to such attributes. In addition, the same attribute can mean different things to different people, and it is common to find multiple definitions for the same attribute [5][14][16];[23];[24]. Dependability assumes a precise meaning only when applied to a specific context: of system and particular stakeholders' goals.

From this perspective, we have adopted a requirements engineering approach specially devised to model dependability in context, i.e., the Unified Model of Dependability (UMD) [2];[10]. UMD is both stakeholder-oriented and issue-centered. UMD is a framework that defines a set of dimensions of interest to any stakeholder (i.e., Figure 2), into which the stakeholder defines issues of concern (e.g., Table 1). It permits stakeholders to express their dependability requirements by specifying what they consider the actual dependability *issues* (i.e., a failure of the system or hazard to users of the system) and their *scope* (the affected system or specific service). For each issue, stakeholders may also specify the tolerable manifestations (*measure*) and the desired *reaction* (something to mitigate if the event occurs). In addition, whenever necessary, stakeholders could also specify the external *event* which could trigger the issue. A supporting tool (Figure 4) helps stakeholders in this process.

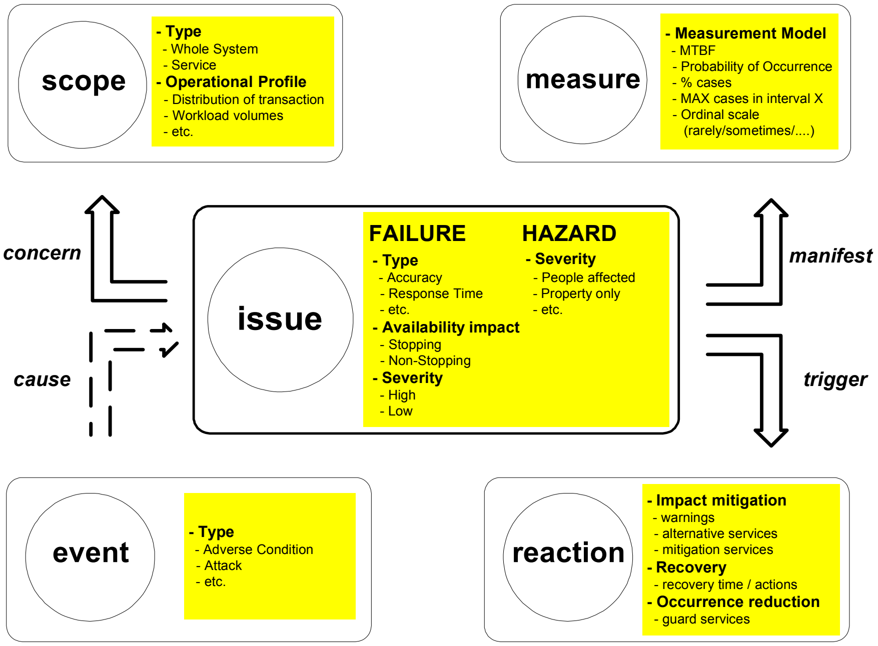


Figure 2. The dependability modeling framework

For example, to help stakeholders identify failures that should not affect the system or a service, UMD may suggest the different types of failures that could occur (e.g., response time failures, accuracy failures). To allow for the specification of more precise requirements, UMD may introduce different levels of severity and impact on availability (e.g., stopping and non-stopping failures).

Table 1. UMD's Failure characterization applied to TSAFE

Failure characterization:

Failure Types:

- *Functional correctness*: System or service does not implement the functional requirements.
- *Throughput*: Average or peak number of items (aircraft, routes, etc.) per unit of time dealt with by the system or service is less than expected.
- *Response time*: Response time of the system or the service greater than expected.
- *Peak load*: Max number of items handled by the system or the service is less than expected.
- *Accuracy*: The accuracy (Lateral, Longitudinal, Vertical) of the computed aircraft position or projected trajectory is less then expected.
- *Data freshness*: The frequency of data updating is less than expected.

Failure Impact:

- *Stopping*: Failure makes the system or service unfit for use.
- *Non-Stopping*: Failure does not make the system or service unfit for use

Failure Severity:

- *High severity*: Failure has a major impact on the utility of the system for the operator.
- *Low severity*: Failure has a minor impact on the utility of the system for the operator.

Hazard characterization:

- *Catastrophic*: Risk of total aircraft destruction.
- *Severe*: Risk of serious damage to the aircraft, serious emergency situation, and loss of human lives possible.
- *Major*: Risk of emergency situation, high stress on cockpit crew.

Event characterization:

- *Adverse Condition*: Any unintentional event that could have some effect on the system.
- *Attack*: Any intentional action carried out against the system.

Measure characterization:

Measurement Models:

- *Mean Time Between Failures* (MTBF)
- *Percentage of cases*

Reaction characterization:

Services Types:

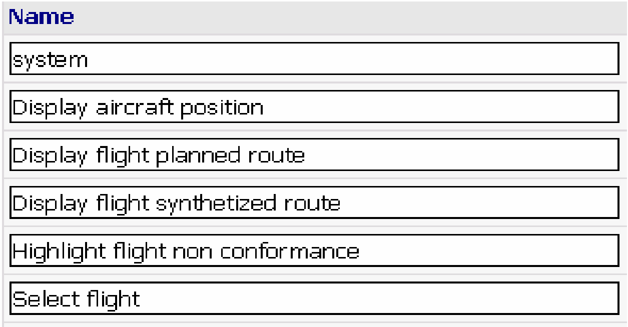
- *Warning Services*: Warn user about the situation.
- *Alternative Services*: Provide alternative ways to perform same tasks.
- *Mitigation Services*: Reduce issue impact on the user.
- *Guard Services*: Reduce probability of occurrence of the issue.

Recovery Behavior:

- *Mean Time To Recover* (MTTR)
- *Max Time To Recover* (MaxTTR)

Similarly, stakeholders can select the measurement model most suitable to express the tolerable manifestation of an issue (e.g., Mean Time Between Failure), and use different reaction types. For example, stakeholders could express the reaction of the system to a specific issue in terms of warning services (to make the user aware of the situation), mitigation services (to reduce the impact of the failure on the user), alternative services (to provide alternative means to perform the same activity), or desired recovery behavior (the time and the actions necessary to recover from the failure). Finally, different event types can be suggested to facilitate stakeholders in recognizing external situations that could harm the system (e.g., attacks and adverse conditions).

UMD has been implemented as a web-based tool [2][10] organized around two tables: the Table "Scope" (Figure 3), which allows stakeholders to describe all the services of the system for which dependability could be of concern; and the Table "Issue" (Figure 4), which allows users to specify their dependability needs by defining, for the whole system or a specific service (selected from the "Scope" table), the potential issues (failures or hazards), their tolerable manifestations, the possible triggering external events, and the desired reactions.



Name
system
Display aircraft position
Display flight planned route
Display flight synthetized route
Highlight flight non conformance
Select flight

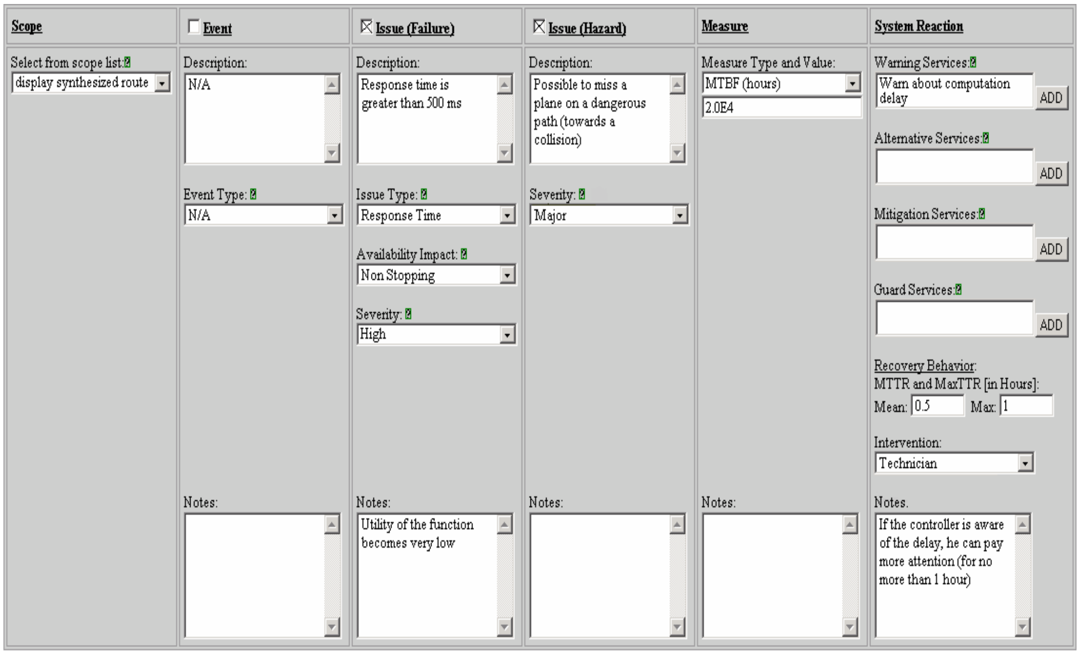
Figure 3. "Scope" of the UMD model for TSAFE, which is used to identify relevant services for the software under consideration.

3.1. Applying UMD to identify the dependability requirements for TSAFE

We used UMD to define the dependability requirements for TSAFE. A small group of computer science researchers and students acted as stakeholders (specifically as air traffic controllers), after being given a short introduction to TSAFE and its purposes, while one person acted as an analyst [10].

UMD was applied in two main steps, scope definition and requirements elicitation and modeling.

Scope definition: All stakeholders, working together and supported by the analyst, selected from the functional requirements available for TSAFE the services for which they believed dependability could be relevant. The identified services are described in the scope table (Figure 3).



Scope	□ Event	☒ Issue (Failure)	☒ Issue (Hazard)	Measure	System Reaction
Select from scope list: display synthesized route	Description: N/A	Description: Response time is greater than 500 ms	Description: Possible to miss a plane on a dangerous path (towards a collision)	Measure Type and Value: MTBF (hours) 2.0E4	Warning Services: Warn about computation delay
	Event Type: N/A	Issue Type: Response Time Availability Impact: Non Stopping Severity: High	Severity: Major		Alternative Services: Mitigation Services: Guard Services: Recovery Behavior: MTTR and MaxTTR [in Hours]: Mean: 0.5 Max: 1 Intervention: Technician
	Notes:	Notes: Utility of the function becomes very low	Notes:	Notes:	Notes. If the controller is aware of the delay, he can pay more attention (for no more than 1 hour)

Figure 4. UMD Tool – TSAFE issues not related to an external event

Requirements elicitation and modeling: Each stakeholder, supported by the analyst and guided by the structure provided by the tool, filled as many tables as necessary to define her/his dependability needs (Figure 4). The characterizations of the UMD concepts of scope, issue, event, measure, and reaction provided useful guidance to stakeholders. Each stakeholder used the characterizations already available, or, whenever necessary, extended it with his/her own definitions. The characterizations used for TSAFE are described in Table 1. As example of the requirements collected with UMD, we describe one of the tables filled by the stakeholders. Figure 4 illustrates an example of an issue not related to an external event. The stakeholder signals a potential failure for the service “display flight synthesized route,” when the “Response time is greater than 500 ms.” This is a Response Time, Non-Stopping, High Severity failure, given the high impact on the service’s utility for the operator. For the stakeholder, this failure is also a “Major Hazard,” given that he thinks he could miss spotting a plane on a dangerous path. This could lead to an emergency situation and possibly cause high stress on the cockpit crew, required to perform sudden escape maneuvers by the very short-term conflict avoidance systems. The stakeholder identifies this failure as a highly critical one, leading the analyst to suggest MTBF of $2*10^4$ (between the values suggested for very high and mission critical availability in Table 1). In order to be more confident in the system, the stakeholder introduces a warning service that will advise in case computational time becomes greater than 500 ms, alerting him when attention is needed. Finally, the stakeholder defines the recovery to be performed within one hour. If this failure condition lasts more than one hour, he would be unable to perform his duties due to the need to maintain a higher than usual level of attention.

4. Building the testbed environment

This section discusses why and how the testbed was built.

4.1. Goals for the testbed environment

In order to implement the dependability requirements for TSAFE that were elicited with support from UMD, there are several major categories of actions, such as detecting and eliminating the faults that can lead to failures, or, if failures cannot be prevented, then assuring proper recovery within desired time. The appropriate selection of technologies that ensures or increases the dependability of the software for a specific project raises questions such as: How effective are these technologies for TSAFE? How costly is it to apply them? What is the most appropriate technology strategy for this context? The remainder of this paper proposes and illustrates the use of a testbed to help answer such questions.

Previous research typically has provided the means to answer these questions for the specific environment in which each technology is being evaluated (which however might be different when technology is applied to other contexts), or to produce a broad cost prediction based on a predetermined set of variables. However, methods for providing truly general decision support, of the kind that can predict the likely effects of using a particular technology in a wide swathe of development environments, have still not been produced. This is partly due to some of the following factors:

- *Empirical studies are expensive, resulting in relatively few studies being carried out.* Carrying out empirical work is complex and time consuming; this is especially true for software engineering. Unlike manufacturing, we do not build the same product, over and over, to meet a particular set of specifications. Software is developed and each product is different from the last. So, software artifacts do not provide a large set of data points permitting sufficient statistical power for confirming or rejecting hypotheses. Moreover, human factors tend to increase the cost of experimentation (the most relevant subjects for most studies come from the population of professional software developers, whose time is almost always over-booked and highly expensive) making it more difficult to achieve statistical significance.
- *Empirical results are difficult to replicate.* Empirical researchers have argued for some time that empirical studies cannot be “one shot deals,” i.e. knowledge must be built up through families of closely-related experiments that allow factors to be carefully studied. However, unless experimental results can be validated somehow this strategy runs the risk of including suspect or clearly invalid results in the overall data set, and thus abstracting faulty conclusions. In other fields, this risk is mitigated by researchers performing “strict replications” of another’s work; that is, replications that conform as closely as possible to the original study to allow the results to be

checked. This is not always feasible in software engineering where the most relevant studies (i.e. those in industrial contexts) are not only human dependent but often are extremely expensive to repeat and may rely on proprietary artifacts.

We have seen that developing reusable software testbeds have mitigated the above difficulties, at least when those testbeds are representative of real software development projects and have appropriate documentation. We have seen that having such testbeds available can address the difficulties discussed above by:

- *Reducing costs of experimentation***.** The cost of an experiment is greatly increased if the preparation of multiple artifacts is necessary. Creating artifacts which are representative of those used in real development projects is difficult and time consuming. Reusing testbed artifacts representative of ones encountered in practice can thus reduce the time and cost needed for experimentation.
- *Facilitating comparison of technologies***.** Once data concerning the use of the testbeds artifacts are available, they provide researchers with a "benchmark" against which future studies can be compared. For example, researchers can obtain subjects with similar experience to those who did the original development, then vary the development practices that were applied to see if they result in fewer defects or more reliability in the final product.
- *Facilitating replication***.** The results of empirical studies involving testbeds need never go without validation since the associated artifacts and development metrics are publicly available.

4.2. Turning TSAFE into a testbed

The TSAFE prototype, upon which we based the testbed, was initially developed at MIT [9] and is a 20 KLOC Java program that performs two primary functions: conformance monitoring and trajectory synthesis.

As part of our effort to turn TSAFE into an experimental testbed, we added a number of specific features, e.g. synthesized faults that can be seeded into the source code as well as a feature that allows the experimenter to enable or disable a specific seeded fault. Another feature was added that allows the experimenter to monitor seeded faults as they get executed by generating system output and traces that can be captured and used to determine the status of the TSAFE system under execution. We also added features to facilitate experimentation with various artifacts of the testbed, synthesized faults that were seeded into artifacts other than the source code (e.g. architectural documentation), and added documentation and other artifacts in order to facilitate understanding the runtime behavior of TSAFE. Some initial studies were conducted and documented to serve as examples for other experimenters interested in using the testbed. The experience from these technology experiments as well as feedback and lessons learned have been collected and will be provided together with the other artifacts as part of the testbed in order to maximize the usefulness as well as to minimize the cost and effort of experimentation.

Currently, the following artifacts are available: a requirements specification, architecture documentation, source code, an installation guide, and some recorded flight data that serve as test input. A tool to create artificial test data is also available.

Fault seeding. We determined that, for experimentation, a set of faults needed to be synthesized for injection in the source code. We identified three kinds of faults that represent developer errors and thus constitute plausible fault classes: Technology-driven, Dependability-driven, and History-driven.

- *Technology-driven* faults are faults that a certain technology claims it will detect. These faults may or may not cause run-time failures in the system under study and may or may not represent historical faults for that system. As the name indicates, technology-driven faults are strongly related to the technology under study. This fault class addresses the fact that different technologies detect different kind of faults. Technologies that address the same kind of problem and therefore will find the same kind of faults belong to the same technology family. Instead of trying to build a testbed that serves all technologies and covers the complete fault space (which is an almost impossible task) we made the decision to start with faults related to the claims of the selected family of technology; technologies that detect violations of the implementation of an architecture as compared to the planned architecture. Thus, we defined a set of faults that are related to the architecture of the testbed relative to this type of technology. For each of these faults, we identified the impact on the system's behavior and how the resulting system failure would be detected. For each of these faults, we also documented what architectural rules they violated. Subsequently, we, we added synthesized faults related to the technologies that were studied, for example, we added concurrency problems that model checking might or might not find (described below).
- *Dependability-driven* faults are faults that will cause run-time failures, if triggered. These faults can be derived from dependability models or requirements of the system. The faults may or may not be detectable by a particular technology, and may or may not represent historical faults for a particular system. In order to synthesize dependability-driven faults, we used UMD described in Section 3 and identified a set of potential failures derived from the dependability model. For each of the possible failures, we identified a set of plausible causing faults. Thus the link between failures and faults was made explicit and it is well known how the system changes behavior when these faults are triggered.

- *History-driven* faults are faults that have been detected in the past during testing or usage of the testbed or similar systems. These faults may be collected from problem reports, inspection meetings, testing reports, quality assessments etc. These faults may or may not cause run-time failures, and they may or may not be detectable by a particular technology. We did not have access to the original fault-history from the implementation of TSAFE at MIT, but used other related fault-histories to determine the plausibility of various suggested faults. We are collecting faults uncovered during TSAFE-related experiments in order to create a fault history for future experimentation.

Focusing on technology-driven faults that were, to as large extent as possible, also dependability-driven and history-driven (as described above), we created several different fault sets. The faults were seeded into the source code of the testbed. There are several reasons for dividing the faults into different sets. First, we deemed it unrealistic, based on our experience from previous analyses, that a source code of the size of the TSAFE source code (~20,000 lines of Java) would contain all faults at once. Second, it was impractical to seed all faults into one version, as the risk of faults overlapping each other was imminent and it would be hard to analyze the results for such faults. Third, we wanted to create several versions of the testbed in order to run several "replicated" experiments on it. The results are different versions of the testbed seeded with different sets of faults. The first version of the testbed without seeded faults is considered a baseline to which we compare faulty versions of the testbed.

4.3. TSAFE instrumentation

The testbed developers instrumented TSAFE as a testbed for designing and executing the following dependability-related experimental activities:

- Define what dependability means for TSAFE, by applying the UMD model.
- According to this definition of dependability, identify potential failures and corresponding faults in the code that could cause these failures.
- Identify test cases that would trigger these faults and cause failures.
- Seed the code with the identified faults.

For each experiment, the technology developers (or someone skilled in using the technology) formulated their hypotheses in terms of faults that can be detected by their technology (and possibly the impact on dependability) and estimated the costs associated with applying the technology. They then:

- Applied the technology on the code containing seeded faults and recorded the detected faults as well as the associated detection cost.
- Exercised test cases on the code containing the seeded faults and recorded the occurring

failures (as well as their frequency of occurrence).

The testbed developers received the results from the technology developers and:

- Analyzed these faults and failures and validated or refuted the technology hypotheses.
- Estimated the effect on dependability and the cost of applying the technology to the TSAFE testbed.

In order to avoid experimental bias, the testbed developers and the technology developers never discussed seeded faults and failures during the experiment. After the experiment had been conducted, faults, failures, and results were discussed as part of final reporting.

The outcomes of these activities are:

- A method for defining dependability and for designing and performing experiments for estimating the effect of a technology with respect to dependability and cost, for a given system.
- An example for the application of this method.
- An instrumented testbed reusable for future experiments.

An elaborated description of this process is provided in [19], using the experiment presented in section 5 as an example.

5. Software Architecture Evaluation Method

Many of the technologies studied in HDCP deal with the architecture of a software system, which is one of the major reasons we decided to start experimenting with this class of technologies. Maintainability is one of the dependability attributes [16] and architecture evaluation assumes that if the implementation conforms to the planned architecture, then the software is easier to maintain.

In short, software architecture models the structure and interactions of a software system. The basic building blocks of the structure of software architecture are components and the interrelationships among them. In addition to structure, behavior is part of software architecture. Constraints and rules describe how the architectural components communicate with one another.

When viewed at the highest levels, a system's architecture is referred to as the macro-architecture of the software system. At lower levels of abstraction, it is referred to as micro-architecture. Architectural styles and design patterns are similar to what Bhansali [3] describes as generic forms of software architecture. Often architectural styles guide the structure and interactions of the system when describing the software architecture of a system at the macro-architectural level. When describing the structure and/or interactions of a system at a micro-architectural level, design patterns can be used.

Software architectural evaluations are investigations into a software's structure and behavior with the purpose of suggesting areas for improvement or understanding various aspects of a system (e.g., maintainability, reliability, or security). In many cases, a software architectural evaluation is performed before a system has been designed or implemented. Often, this type of architectural evaluation is performed to compare alternatives or to determine whether or not the architecture is complete or appropriate for the application and its requirements. In other cases, a software architectural evaluation is performed after the system has been implemented. Such post-implementation architectural evaluations are typically performed to ensure that the actual implementation of a system matches the planned architectural design [25]. Some of the technologies in HDCP evaluate architectures before implementation, some after. The technology discussed in this paper is of the latter kind and is called *Implementation-oriented software architectural evaluation* [25]. Since this type of software architectural evaluation is performed after a version of the software system exists, it can utilize data measured from the actual source code and associated documentation. Implementation-oriented software architectural evaluations can be used for similar goals to pre-implementation software architectural evaluations. For example, the source code and associated documentation can be used to reconstruct the actual software architecture in order to compare it to the planned or conceptual software architecture. Recovering the actual architecture of an implemented system is used for risk assessment and maintenance cost prediction as well. The analysis of the actual software architecture can be used to evaluate whether the implemented software architecture fulfills the planned software architecture and associated goals, rules and guidelines.

Violations of architectural guidelines affect the maintainability of the software, as resulted from the analysis of Mozilla [13]. Software decay is not a new problem. In 1969, Lehman found that "the main problem of large systems is unintentional interaction between components, which require changes to the components for their elimination" [17]. The observations were formed into a collection of "laws" stating, for example, that: "As a program is evolved, its complexity increases unless work is done to maintain or reduce it" [18]. Brooks drew the conclusion that all systems will eventually require a complete redesign as a consequence of such degeneration [7]. A technology able to detect architectural violations increases maintainability of the system and thus increases its dependability. For initial experimentation with architectural technologies, we decided to use the *Software Architectural Evaluation (SAE) method* [25]. SAE requires a set of rules that describe properties the architecture should have. The output of the tool that supports SAE is a diagram that shows relations among components in the code. The diagram indicates violations of the rules of communication in terms of coupling between components, highlighting extra and missing relations as compared to the planned architecture.

Setup of testbed. The testbed developers first prepared a set of rules describing properties the architecture of TSAFE should possess. Altogether, 42 rules were identified; of those, 29 were considered to be relevant to the SAE method. The other 13 rules were related to TSAFE-specific use of design patterns and were considered to be outside of the immediate scope of SAE.

Next, a version of the TSAFE code was prepared that included 13 rule violations. That is, 13 instances of the code where the architecture as implemented by the code did not conform to the planned architecture as defined by the testbed developers. Of these, 7 were considered relevant to SAE. The other 6 rules violated design patterns. These 7 *expected* rule violations were considered to map to 4 distinct faults in the code.

An example of an architectural rule is EC1, which states that the *engine* component must not access any classes in any of the other components (*client*, *feed*, or *main*) but the *data* component. Rule EC1 is violated, for example, by the seeded faults 7.1.1 and 11.1.1. Both faults are structural faults that introduce couplings between components that are not specified in the architecture thus violating rule EC1.

Fault 7.1.1 will not cause run-time failures and is seeded by introducing a Java import statement of a component that is not used. Seeding this fault only required an addition of one line of code in one class.

Fault 11.1.1 will cause run-time failures under certain circumstances, when executed. More specifically, the run-time failure is that the TSAFE display map will not be updated when the threshold parameters are changed. The TSAFE map is updated at a certain frequency during the software's execution and changes to the thresholds should influence the conformance status of a flight, but this fault will interfere with this update.

Originally, the class *ParametersDialog* contained the *ActionListener* that calls the *EngineParameters* class to inform it about changed parameters. The fault was seeded by altering the source code in the following way. The attribute holding the reference to the *EngineParameters* object is moved from the *TsafeClient* class to the *TsafeEngine* class, which resides in another component. The *TsafeClient* now uses the attribute of the *TsafeEngine* class to access the methods of *ParametersDialog*. When the *ParametersDialog* is created, it gets a new created object of the *EngineParameters* and not the one that is used from the other classes. Thus, it informs the wrong object about changes, which will cause a run-time failure. In order to seed this fault, nine changes (deletions, additions, and changes of nine lines of code) to four different classes were conducted. The static view of the change that represents fault 11.1.1 is illustrated in Figure 5 and was constructed using the SAVE tool [25]. SAE is expected to detect the new inter-component connections between the *TsafeEngine* and the *ParametersDialog* and the *TSAFEClient*, but not the deleted intra-component connection between the *TsafeClient* and the *ParametersDialog*.

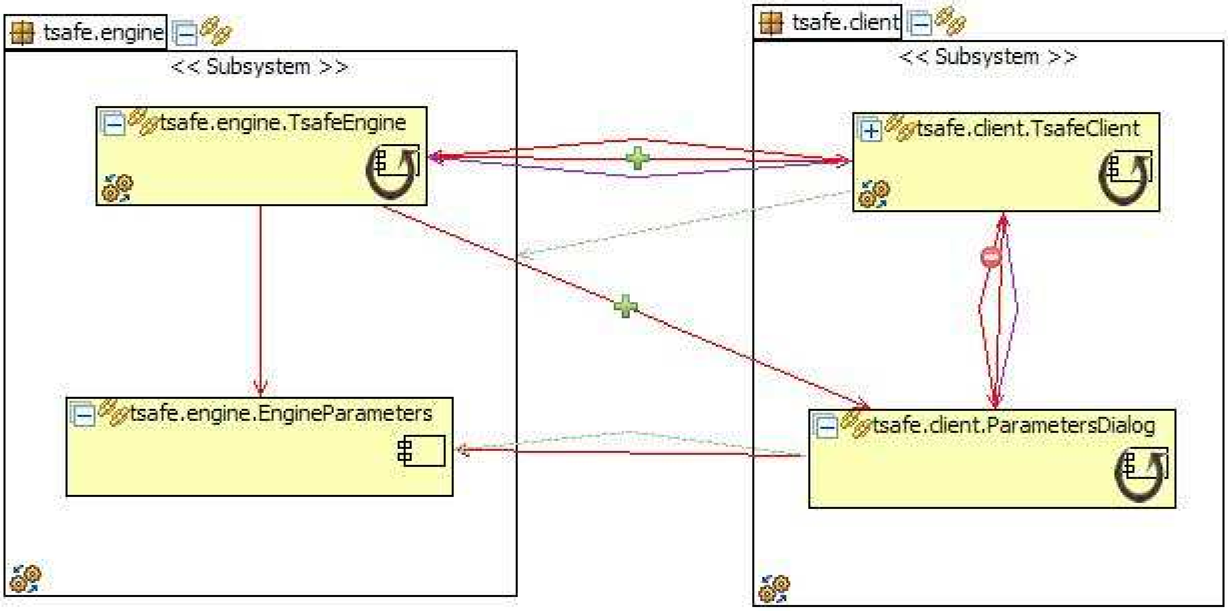


Figure 5. Zooming in on the *engine* and *client* components highlighting added (+) and deleted (-) connections.

Measures. To assess the effectiveness of the SAE technology, this study measured the percentage of the seeded rule violations and the percentage of the seeded faults that were found by the subject (a senior person skilled in using the technology) conducting the SAE task. This required a two-step process:

(1) The issues reported by the subject were checked against the list of seeded rule violations. This required some subjectivity since the subject was not expected to independently formulate the same wording as was used to describe the seeded violations; the testbed developers had to assess whether the underlying issue described was the same as the item on the seeded list. However, since every item on both the subject list and the seeded list had to refer to a specific area of the architecture, the amount of subjectivity was reduced.

(2) The rule violations were mapped to specific faults, by which we mean specific instances within the code that would affect system flexibility or maintainability that were directly caused by the rule violations. This required a higher degree of subjectivity since it required reasoning about likely problems on future development increments of the system.

Along with the list of discrepancies found, the subject was required to report for each discrepancy the following information:

1. Which rule was being violated?
2. What part of the architecture was being examined when the discrepancy was found?

This information was used to understand the process followed by the subject as well as to help improve process conformance (relative to the process provided to the subject as part of the experiment), since it was reasoned that making up plausible answers to those questions would require more effort from the subject than simply applying the SAE documented process and reporting the results according to the provided form and instructions.

Results. The results of the study were useful for the evolution of the SAE method:

- SAE found 6 of the 7 *expected (*relative to the claims of the technology) seeded rule violations.
- The 6 seeded rule violations found by SAE were considered to map to 3 out of the 4 seeded faults. Fault 11.1.1 was among the detected seeded faults.
- The one seeded rule violation that was missed helped identify a bug in the tool that has since been fixed. Thus, the bug in the tool caused one (fault 7.1.1) of the four seeded faults to be missed.
- An additional 3 out of the 6 *unexpected* seeded rule violations were found, which indicates that SAE can be useful even outside the set of narrowly targeted problems it was designed to catch.

There were several instances where misstated rules caused some confusion as to what exactly should be checked in the code. As a result, the language describing the rules was improved.

Since the SAE method addresses faults related to maintainability and since a failure means not being able to maintain the software in a specific amount of time, exercising the test cases on the code containing the seeded faults would mean performing maintenance tasks on the code and measure the time it takes to complete them. For this experiment, we simply assumed that failure of the implementation to comply with the planned architecture would cause such failures. To address the question in more depth, we designed a separate experiment, now ongoing, to test the hypothesis that such architectural deviations do indeed create maintenance problems. The maintainability experiment is carried out as part of a Software Engineering class at the University of Maryland in which students, working in teams, are adding features to two versions of TSAFE. One version adheres to all of the guidelines required by the SAE method, but the other version does not. Our hypothesis is that the one that adheres to these guidelines will be easier to enhance with extensions and be more dependable than the other. We will report on this in the future, but that does not discount the results from our current experiment, which show that we can easily and in a cost effective manner apply the architectural evaluation method to a fairly complex piece of software.

The conclusion from applying SAE to TSAFE is that SAE detects most of the faults that it claims to detect. These faults are related to the maintainability of the software and thus when these faults are detected and removed, the assumption is that dependability increases.

Cost. It took 331 hours to prepare the experiment, which was dominated by the time it took to develop the testbed. The cost involved in applying SAE was 4 hours. Technologies such as SAE can therefore be recommended as a cost effective way to address these kinds of dependability issues. Although this experiment does not prove the reduction in cost due to the use of testbeds, this shows clearly from a follow-up experiment studying a technology similar to SAE that we previously reported on [19]. That experiment did not require any new faults to be seeded or any other alterations of the testbed or to the experimental design. Applying that technology also took 4 hours and produced similar results in terms of detected defects and the cost of developing the testbed was eliminated. The object of our next experiment was model checking, a technology from a different family than architectural evaluation, and was discussed in this paper because we were interested in analyzing the feasibility of using the same testbed for an experiment studying a very different technology.

6. Model Checking

In this experiment, we evaluated the "Design for Verification with Concurrency Controllers (DVCC)" technology using the TSAFE testbed. The goal of the DVCC technology is to eliminate synchronization errors in Java programs using model checking techniques in conjunction with design patterns that facilitate automated verification [4]. Model checking techniques exhaustively explore all the states of a system looking for violations of its properties. If a system does not satisfy the given property, the model checker generates a counter-example behavior demonstrating the fault. Model checking has been applied to a variety of problems related to software development such as verification of formal requirements, concurrent programs, and system code. Model checking has also been applied to security in analyzing formal models of security protocols and in verification of security related properties in software applications. One of the biggest challenges in model checking is scalability due to state space explosion. In order to apply model checking to software, one needs to generate manageable models of the software artifacts that can be analyzed by a model checker and to specify an environment characterizing possible inputs. The model and environment generation problems typically require reverse engineering of the code to discover models and constraints that are often known by the developers at design time.

The DVCC technology promotes the use of verifiable design patterns that facilitate model construction and environment generation to enable scalable verification. The basic idea is to specify critical behaviors in controller classes, and to separate a controller's behavior from its environment using stateful interfaces that specify the interactions between a controller and its environment. Using this approach, the model extraction problem reduces to construction of compact models for controller behaviors, and environment generation problem is resolved by using controller interfaces as the characterizations of their environments.

In the concurrency controller design pattern, concurrency controller classes are used to control the accesses to data objects that are shared among multiple threads. For example, a concurrency controller class implementing a reader-writer lock can be used to control the accesses to a shared data object. The methods of a concurrency controller class are written as guarded commands that specify the behavior of the controller. The controller interface specifies the order the methods of the controller and the shared data object should be called. The controller interface is specified as a finite state machine. The concurrency controller design pattern provides helper classes which support specification of guarded commands and finite state machines.

The DVCC technology supports a modular verification approach, which separates the verification of the concurrency controller behavior (behavior verification) from the verification of the threads that use them (interface verification). For behavior verification, we use a symbolic and infinite-state model-checking tool called Action Language Verifier (ALV) [8], which enables verification of controllers with parameterized constants, unbounded variables and arbitrary number of client threads. For interface verification we use an explicit state model checking tool called Java Path Finder (JPF) [6] which enables verification of arbitrary thread implementations without any restrictions.

This experimental study evaluates the DVCC technology, addressing questions regarding

1. The applicability of the DVCC technology to safety critical air traffic control software (i.e., is it possible to reengineer the TSAFE software using the DVCC approach where the synchronization statements are only used in the concurrency controller classes, and in the rest of the code calls to the methods of the concurrency controller classes are used to control the access to the shared data) and
2. The effectiveness of the DVCC technology in finding concurrency errors in safety critical air traffic control software (i.e., can the verification tools that are used for the DVCC approach find behavior errors in the concurrency controller classes and interface errors in the code that uses the concurrency controller classes?).

Setup of testbed. Two teams conducted this experimental study: 1) The University of California at Santa Barbara (UCSB) team, which consists of the developers of the DVCC technology and 2) the Fraunhofer Center for Experimental Engineering, Maryland (FC-MD) team, which consists of the developers of the TSAFE testbed.

Normally, in the DVCC approach, software developers use concurrency controllers during software design and development. However, in order to apply the DVCC approach to the existing TSAFE code, in this experimental study, we introduced the concurrency controllers to the code as a reengineering activity.

The technology developers reengineered the TSAFE software as follows:

1. They identified all synchronization statement in the code and the shared objects they protect.
2. They replaced the synchronization statements in the TSAFE code with calls to the appropriate concurrency controller classes (also provided by the technology developers). In the reengineered TSAFE code *all* synchronization statements are in the controller classes.

The technology developers then used the reengineered TSAFE code to create modified versions using seeded faults. Each modified version may contain no faults, one behavior fault, or one interface fault. These faults are technology-driven faults because they are based on the claims of the technology. The faults are related to concurrency issues and will cause run-time problems of the TSAFE software. The technology developers seeded behavior faults by modifying the guarded commands in the controller classes. Interface faults were seeded by changing the order of the calls to the controller classes or by removing the calls to the controller classes.

The technology developers received each version of TSAFE without knowing which types of faults were in which version (or if there was any fault in a version) and applied DVCC to it. They used the ALV tool to detect behavior errors in the controllers. The guarded commands from the controller classes were automatically translated to the input language of the ALV by the DVCC support tools developed by the technology developers. Behavior verification was achieved by checking each controller with respect to a set of invariant properties that should be satisfied by the controller (these properties are not automatically generated, they have to be written during the reengineering of TSAFE and they correspond to the class invariants of the controller classes). If a property is violated, the ALV tool generates a counter-example behavior demonstrating an execution sequence for the controller, which results in the violation of the invariant.

The technology developers used the Java Path Finder (JPF) [6] tool to detect interface errors. JPF is a tool that searches for assertion violations in Java code exhaustively by investigating all possible input valuations and all possible thread interleavings. However, JPF can only handle pure Java code, therefore, in order to use JPF all part of the TSAFE software that use native code (such as user interface calls, RMI calls, network communication) must be replaced with drivers and stubs. The technology developers investigated the TSAFE software for native code and replaced the calls to methods that involve native code with drivers and stubs. Then, the technology developers used these drivers and stubs and the JPF tool to detect interface errors. Note that the same set of drivers and stubs are used for each version of TSAFE.

Measures. For assessing the effectiveness of DVCC, we collected the number of faults found by the behavior and interface verification and the number of faults missed by the behavior and interface verification. For assessing the cost of applying DVCC, we collected the effort involved in the experiment:

the time and memory usage by the ALV tool were recorded as well as the time and memory usage by the JPF tool.

Results. There were a total of 14 controller faults and 26 interface faults in versions v1–40. Verifying the controllers in versions v1–40 with ALV identified 12 faults. The two faults that were not found by ALV were the faults in versions v5 and v13 which were spurious faults, i.e., they are modifications in the controller classes which do not cause any failures in the controller behavior. Among the 26 interface faults, interface verification using JPF identified 21 of them. Among the five faults that were not caught by JPF, two were spurious faults (v22 and v33). The faults in versions v18, v19, and v20 were real faults which can cause failures but were not found by JPF. These faults demonstrate that there is a limit to the depth of the faults that can be identified using explicit state verification techniques without running out of memory. One of the results of this study is that DVCC was able to distinguish spurious faults from the real faults. That is, the verification process did not report any violations (both interface violation and concurrency controller property violation) when the seeded fault was spurious. The experimental study also resulted in a fault classification specific to DVCC and helped identify new directions for improving the DVCC approach.

Cost. The time to apply the DVCC each time was minimal since it was a matter of initially setting up and repeatedly executing a set of scripts and collecting the output. The experiment involved executing the scripts 40 times (one for each version of TSAFE) requiring a total effort of 8 hours. It took about 8 hours to change the testbed architecture in order to introduce some concurrency characteristics by turning TSAFE into a client-server application, and it took another 8 hours to introduce the concurrency controllers to the code. This 16 hours effort is small compared to the effort it would have taken to develop a testbed from scratch.

In Table 2, the results from experiment one and two are summarized.

Table 2. Results from experiment 1 and 2

	No. of seeded defects	No. of detected seeded defects	Cost
SAE	Total of 13 seeded rule violations. Of these, 7 were architectural violations and 6 were violations of design patterns	SAE detected 6 of the 7 seeded architectural violations, thus one architectural violation remained undetected. SAE detected 3 out of the 6 seeded design pattern violations. Since SAE was not expected to detect any of these violations, this is a positive finding.	A total of 331 hours including developing the initial testbed A total of 4 hours to apply the technology in one session.

Model Checking	Total of 40 faults. Of these, 14 were controller faults and 26 were interface faults	ALV identified 12 of the 14 seeded controller faults. The 2 undetected controller faults were spurious faults and were not expected to be detected by ALV. JPF identified 21 of the 26 seeded the interface faults. Two of the faults that were not caught by JPF were spurious faults and were not expected to be detected by JPF. Three of the undetected faults were real ones.	A total of 16 hours to reengineer the testbed A total of 8 hour to set up and apply the technology 40 times.

7. Software Testing (Ongoing Study)

We are expanding the testbed to allow for experiments using standardized execution-based technologies, i.e., software testing. During software testing, the software's behavior is studied by executing test cases, which represent controlled input. The software's execution is checked against an expected behavior and the software's specifications and test coverage criteria are used to drive the test-case generation process.

We have multiple goals for experimenting with software testing on the testbed. First, we want to define a process for future execution-based verification and validation technologies. Since software testing is well-understood, we can focus on the process-definition issues rather than the intricacies of the technology. In the future, we will apply the same process to other, less understood execution-based technologies. Second, since software testing has been widely studied, the results of experiments on testing will give us a baseline technology, which we can use to compare other technologies. Finally, with our results, we can contribute to software testing research and its applicability to dependability.

A senior person skilled in testing is leading the development of a "test pool" [21] for the TSAFE software. The test pool will consist of a large number of test cases which can be used to generate many types of test suites, satisfying many coverage criteria. Specifications of TSAFE are used to generate *black-box* test cases as well as code coverage to develop *white-box* test cases. In order to design "fair" experiments, it is important that the test pool be constructed in such a way so as not to make any one testing technique seem superior. For example, consider an experiment that compares the fault-detection effectiveness of branch-coverage and statement-coverage adequate suites. If the test pool contained exactly one test case *t* that covers a particular branch *b* in the code and detects a fault *f*, any branch-coverage adequate suite generated from the test pool will surely contain *t*. Consequently, the fault *f* will always be detected by all branch-coverage adequate test suites, giving an unfair advantage to branch-coverage. However, it may be possible to create additional test cases that cover *b* but do not detect *f*. To circumvent the abovementioned problem such test cases must be added to the test pool. In general, the test pool must be designed very carefully. Consequently, the test pool (which currently contains 121 test cases) has the following characteristics:

1. Each statement in the TSAFE application is covered by at least 30 test cases.
2. Each branch is covered by at least 30 test cases.
3. If the test case was developed using some requirements specification, we provide traceability to the particular requirement that was being tested.

In addition, we describe the expected output [20]. This is the TSAFE output that we expected to see as a result of the execution of the test case. For example, for a test case that should result in a blundering flight, the expected output was an encoded form of "the flight XYZ should blunder."

As we use the test pool to conduct new experiments (e.g., compare test suites created using equivalence-class partitioning and boundary-value analysis), we will need to augment the test pool with additional test cases. We have executed all the test cases on the original TSAFE application as well as its fault-seeded versions. Since the original TSAFE produces a graphical output, we augmented it so that each graphics primitive was also written in text-form to a file. Our test-case verifier examined this text file to determine the output of a test case. For each test case execution, we have collected the following information:

1. Whether the TSAFE application passed/failed for the test case compared to the expected result.
2. The path (in terms of statements and their ordering) that was executed by the test case.
3. Whether the fault-seeded versions passed/failed for the test case. If they failed, then we record the difference between the original and fault-seeded output.

It is common practice to run all test cases in a test pool first. The pool then contains the actual output observed [22]. We have started to use the test pool to conduct experiments on TSAFE. Since we executed all the test cases, we need not re-run them during these experiments. We can simply simulate the process of test execution. That way, if we want to create 200 branch-coverage test suites, each consisting of a number of test cases from the pool, we will immediately know the fault-detection ability of the 200 suites, since we know what each test case actually does. In addition, as part of this process we enhanced TSAFE with a harness to automatically executing the test cases. The results from this experiment will help us to compare testing's impact on dependability relative to other technologies.

The time to apply testing will be minimal since each run will be a matter of executing a set of scripts that will run a test case. The result (pass or fail) will be automatically recorded. There was no need to change the testbed architecture for this experiment since testing is such a general technique. However, the creation of the test suite required much effort, amounting to several weeks of work.

8. Summary, Discussion, and Future work

Testbeds have proven to be an effective vehicle for testing new technologies. We previously demonstrated that the same testbed could be used to study technologies that detect similar dependability issues [19]. In this paper, we studied whether one testbed could be used in experiments studying different technologies that detect different kinds of dependability issues. Our conclusion is that even though these three experiments are very different in terms of the technology that was the subject for experimentation, it was feasible to apply them to the same testbed. However, there are some limitations to the use of testbeds. For example, the technologies studied in the experiments described in this paper all assume that the Java programming language is used. Thus, it would probably not be cost-efficient to reuse/rewrite the testbed for experiments on technologies that analyze languages that are conceptually different from Java. Another limitation to the use of testbeds, their development and maintenance, is the rapid evolution of technology in general, which means that the testbed has to be constantly updated to serve the latest emerging technologies. Service-Oriented Architectures is an example of a relatively new technology that has not been extensively empirically studied yet and would benefit from using a testbed. However, such a testbed would probably be very different from the testbed we have discussed in this paper. In addition, we do not suggest a broad development of testbeds by each organization that whishes to conduct technology experiments. Our vision is instead that a few independent organizations are funded to develop and maintain testbeds as a service to technology developers allowing technologies to be studied outside of the inventor's laboratory before applying them to real situations. This will keep the cost of experimentation down and will allow for coordination and experience sharing between all stakeholders in an efficient manner.

Developing and maintaining several versions of the testbed is indeed a challenge. In order to manage the asset that the TSAFE testbed constitutes, we are applying product line modeling describing all available versions of artifacts and how they can be combined. The source code versions are managed using CVS and the documentation is managed using a document management system (Hyperwave). This management is, however, time-consuming and there is always a risk that it will deteriorate since funding for such activities are always difficult to obtain.

Our future work is to enhance the testbed by synthesizing and seeding more faults that both make the testbed more interesting for other families of technologies. We will also run several additional experiments, analyze, and interpret the results in order to build a selection of examples and baselines that can be reused by technology developers who want to design and run their own experiments on the testbed. If necessary, we will then evolve the experiment, the testbed and the technology under investigation. Based on these results and the explicit links from these faults to failures, we will be able to reason about the impact of these technologies on dependability.

9. Acknowledgements

The authors acknowledge support from the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298 as well as from the NSF Science of Design program. We thank our HDCP team members at the University of Southern California, especially Dr. Barry Boehm for fruitful collaboration. We also thank the reviewers for insightful comments on the paper and Jen Dix for proof-reading.

10. Access to the testbed

Please contact Mikael Lindvall at mlindvall@fc-md.umd.edu for further information on how to obtain access to the testbed and related artifacts. Several versions of the testbed exist and determining exactly which one is the most appropriate depends on the purpose of the experiment.

11. References

[1] Asgari S., Basili, V., Costa, P., Donzelli, P., Hochstein, L., Lindvall, M., Rus, I., Shull, I., Tvedt, R., and Zelkowitz, M. 2004. Empirical-based Estimation of the Effect on Software Dependability of a Technique for Architecture Conformance Verification, ICSE/DSN 2004 Twin Workshop on Architecting Dependable Systems (WADS 2004).

[2] Basili, V., Donzelli, P., Asgari, S., 2004. A Unified Model of Dependability: Capturing Dependability in Context, IEEE Software. 21 (6): 19--25.

[3] Bhansali S. and Nii, H. P. 1992. Software Design by Reusing Architectures. Proceedings of the Seventh Knowledge-Based Software Engineering Conference. McLean, Virginia. 100--109.

[4] Betin-Can, A. and Bultan, T. 2004. Verifiable Concurrent Programming Using Concurrency Controllers. The 19th IEEE International Conference on Automated Software Engineering (ASE 2004). 248--257.

[5] Boehm, B., Huang, L., Jain, A., Madachy, R. 2003. The Nature of Information System Dependability – A Stakeholder/Value Approach. USC Technical Report.

[6] Brat, G., Havelund, K., Park, S., and Visser, W. 2000. Java PathFinder - A second generation of a Java model checker. Proceedings of the Workshop on Advances in Verification.

[7] Brooks, F. P. 1995. The Mythical Man-Month. Addison Wesley.

[8] Bultan, T. and Yavuz-Kahveci, T. 2001. Action Language Verifier. Proc. 16th IEEE International Conference on Automated Software Engineering. 382--386.

[9] Dennis G. 2003. TSAFE: Building a Trusted Computing Base for Air Traffic Control Software. Masters Thesis.

[10] Donzelli, P., Basili, V. 2006. A Practical Framework for Eliciting and Modeling System Dependability Requirements: Experience from the NASA High Dependability Computing Project. Journal of Systems and Software 79(1): 107--119.

[11] Erzberger, H. 2001. The automated airspace concept. 4th USA/Europe Air Traffic Management R&D Seminar.

[12] Erzberger, H. 2004. Transforming the NAS: The Next Generation Air Traffic Control System. 24th International Congress of the Aeronautical Sciences.

[13] Godfrey, M. W. and Lee, E. H. S. 2000. Secrets from the Monster: Extracting Mozilla's Software Architecture. Proc 2nd Symp. Constructing Software Engineering Tools (CoSET00).

[14] Huynh, D., Zelkowitz, M., Basili, V., and Rus, I. 2003. Modelling dependability for a diverse set of stakeholders. Distributed Systems and Networks 2003.

[15] International Federation for Information Processing (IFIP WG-10.4), www.dependability.org

[16] Laprie J-C. 1992. Dependability: Basic Concepts and Terminology, Dependable Computing and Fault Tolerance. Vienna, Austria: Springer-Verlag.

[17] Lehman, M. M. and Belady, L. A. 1985. Program Evolution: Processes of Software Change, London: Harcourt Brace Jovanovich.

[18] Lehman, M. M. 1996. Laws of Software Evolution Revisited. European Workshop Software Process Technology.

[19] Lindvall, M., Rus, I., Shull, F., Zelkowitz, M. V., Donzelli, P., Memon, A., Basili, V. R., Costa, P., Tvedt, R. T., Hochstein, L., Asgari, S., Ackermann, C., and Pech, D. 2005. An Evolutionary Testbed for Software Technology Evaluation. NASA Journal of Innovations in Systems and Software Engineering. 1: 3--11.

[20] Memon, A., Banerjee, I., and Nagarajan, A. 2003. What Test Oracle Should I use for Effective GUI Testing? IEEE International Conference on Automated Software Engineering (ASE'03). Montreal, Canada. 164--173.

[21] Memon, A., Nagarajan, A., and Xie, Q. 2005. Automating regression testing for evolving GUI software. Journal of Software Maintenance and Evolution: Research and Practice. 17: 27--64.

[22] Memon, A., Soffa, M., and Pollack, M. E. 2001. Coverage Criteria for GUI Testing. 8th European Software Engineering Confere nce (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9). Vienna University of Technology, Austria.

[23] Randel, B. 1998. Dependability, a unifying concept. Proceedings of Computer Security, Dependability and Assurance: from needs to solutions.

[24] Rus, I., Basili, V., Zelkowitz, M., and Boehm, B. 2002. Empirical evaluation techniques and methods used for achieving and assessing high dependability. Workshop on dependability benchmarking, Int. Conf. on Dependable Systems. Washington.

[25] Tvedt, R. T., R., Costa, P., and Lindvall, M. 2002. Does the Code Match the Design? A Process for Architecture Evaluation. Proceedings of the International Conference on Software Maintenance. 393--401.