28

SOFTWARE VERIFICATION — A SCALABLE, MODEL-DRIVEN, EMPIRICALLY GROUNDED APPROACH

Lionel C. Briand

Lionel C. Briand
Simula Research Laboratory

PROJECT OVERVIEW

Software Verification and Validation

Research in software verification and validation (V&V) has been conducted for more than three decades. Software systems play an increasingly critical role in society and industry and their complexity tends to grow exponentially over time. As a result, existing V&V technology in many industry sectors is no match for the scale and complexity of software systems, and this makes it difficult to ensure the dependability of software systems in a cost-effective way. For example, safety-critical systems in various industries (e.g., aerospace, automotive, maritime, and energy) increasingly rely on software and need to be certified with respect to their safety. Despite international standards and practical guidelines, no cost-effective, well-established way to ensure software safety at a reasonable level exists.

Software V&V research develops algorithms, strategies, and tools to help develop and automate techniques to detect failures and correct faults in software systems. From a scientific standpoint, this research involves a variety of technologies that cover many technical domains, including software modeling, programming languages, static and dynamic analysis of source code, simulation, and optimization and search algorithms. These domains must be integrated to form a suitable, complete, and practical V&V solution.

Scientific Challenges

The main challenges regarding software V&V relate to devising solutions that scale up to the increasing complexity of software systems. In practice, resources to verify and validate software systems are limited, both in terms of available expertise and time. The challenges at a more detailed level are related to the effective automation of verification techniques and to the evaluation of their cost-effectiveness.

Most of the work on V&V performed at Simula takes a model-driven approach that relies on models of the behavior and properties of the designed software system. We make use of metaheuristic search techniques developed in evolutionary computing to reveal potential problems in the system design and to generate an optimal set of test cases with a high fault-revealing power. The focus of verification spans faults ranging from functional to safety, response time, and concurrency properties. Such an approach is markedly different from mainstream approaches, based for example on static analysis of source code or model checking, which, for different reasons, tend not to scale up to large systems.

Obtained and Expected Results

A number of model-driven techniques have been developed for class testing, component off-the-shelf testing, integration testing, and testing of non-functional properties such as response time, deadlocks, or starvation. Many other aspects of verification must be investigated, with a particular emphasis on safety and robustness. Moreover, providing automated solutions and investigating their cost and effectiveness on large systems requires attention.

We currently are in the process of establishing model-based testing practices at Tandberg in Oslo, and new projects with other companies are beginning. One of these projects, with Det Norske Veritas, that is of particular interest to Norwegian society, is the improvement of software V&V technology in the maritime and energy sectors, where complex software systems play an increasingly important role and where software failures can lead to dramatic consequences in terms of loss of human lives, damage to the environment, or significant financial losses.

SOFTWARE VERIFICATION — A SCALABLE, MODEL-DRIVEN, EMPIRICALLY GROUNDED APPROACH

28.1 Introduction

Software is present in most systems across all industries, including energy, automotive, health care, maritime, aerospace, and banking, to name just a few. Software systems are increasingly taking on safety- and business-critical roles and growing in complexity. One crucial aspect of software development is therefore to ensure the dependability of such systems, that is, their reliability, safety, and robustness. This is achieved by several complementary means of verification, ranging from early analysis of system specifications and designs to systematic testing of the executable software. Such verification activities are, however, difficult and time-consuming. This stems in part from the sheer complexity of most software systems and because they must accommodate changing requirements from many stakeholders.

Software verification potentially has a high impact on the dependability of systems and therefore their economical, human, and environmental impact. Exhaustive verification, however, even on smaller systems, is impossible to achieve and this often leads to the difficult dilemma of how to achieve sufficient confidence with limited resources. This chapter presents a personal assessment of the state of the art in software verification, its achievements and gaps, and a set of research directions that the author believes hold promise for the future.

28.2 Background

Fundamentals

The goal of software verification is to make software-based systems dependable. Software dependability is, however, a multipronged concept. Ideally, one would like software systems to be correct. It is, however, highly difficult to prove the correctness of even small programs [11]. The closest workable concept is that of *reliability*, which is defined as the probability that a system will perform its intended function during a specified period of time under stated conditions [17]. Moreover, there is more to dependability than just reliability. A system also needs to be robust and safe. A system is robust if it acts reasonably under severe, unusual, or illegal conditions [31].

Robustness is often associated with the concept of “graceful degradation”, that is, the capacity of a system to provide partial functionality even under degraded conditions. Safety is related to whether a system can cause damage, for example, threaten human life or cause serious environmental damage. Those three aspects of dependability, namely, reliability, robustness, and safety, are related but distinct. They must all be addressed by verification with dedicated techniques.

There are three ways according to which one can influence dependability. First, through rigorous specification, design, and coding practices, one can limit the introduction of defects, although one cannot entirely avoid them. Second, one can attempt to detect defects as early as possible in the development life cycle, for example, by inspecting, analysing, or testing various artefacts, such as design documents, models, and source code. Such activities are referred to as *verification*. Last, software can be designed to be fault tolerant in order to contain run-time failures and, in the worst case, provide degraded but graceful functionality in the event of failure [38].

Verification can be performed according to different processes, depending on the artefact to be verified and the technology available. As far as non-executable artefacts are concerned, common practices include inspections, walkthroughs, and code reviews [46]. Those are all variants of informal but systematic manual or partially automated analyses of development artefacts, ranging from planning documents to specification and design documents to even source code. Alternatively, if the specification and design are represented as verifiable models (i.e., abstractions) bearing precise semantics, then automated model analysis can be considered. Model analysis is used here as a general term to refer to any form of analysis of model properties. For example, one may want to verify whether deadlocks are possible in a concurrent design [57]. Model analysis methods range from formal and exhaustive (e.g., model checking [31]) to search-based heuristics[1] [31]. Executable code can be tested, that is, executed in some controlled and systematic fashion in order to ensure that it properly implements its specifications. This chapter will focus on two types of verification: model analysis using search heuristics (simply referred to below as model analysis) and testing. The main reason is that, in the foreseeable future, these options are believed to be the only ones scalable to large systems.

Testing

Testing usually has a number of complementary objectives. It must, of course, be effective at triggering failures and therefore detecting faults. But it is also important that the testing process be automated and repeatable. It must be automated to be cost-effective, given the typical complexity of software systems, and it must be repeatable so that a precise understanding of the fault can be gained to verify that it was properly corrected. It must also be helpful in locating faults in the source code once a failure is observed, i.e., the fault localization problem [55]. Finally, it must be systematic in order to associate an expected level of dependability with a specific testing strategy.

[1] This form of analysis is not to be confused with model checking, which, as further discussed below, is an important approach to model analysis that has been getting increasing attention in recent years.

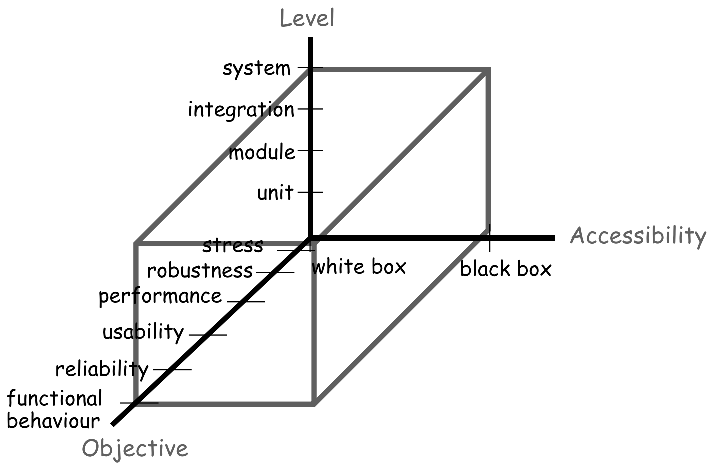

Figure 28.1 Dimensions of software testing.

There are many different testing activities in a typical software development process, as illustrated in figure 28.1. Such activities typically differ according to the type of faults they aim to find and the phase of verification during which they can be applied. Testing can focus on the verification of single units or components in isolation (i.e., unit or component testing). It can target the interactions of components or (sub)systems, which is usually referred to as integration testing, or it can encompass the testing of entire systems, whether on development or deployment platforms, which can be distinct and very different for embedded systems [57]. Test techniques are typically classified according to three main categories. Black-box techniques rely exclusively on some representation of the specification of the system under test (SUT): They do not use internal information regarding design or source code. White-box techniques rely on structural information obtained, for example, through source code analysis. Techniques that rely on partial information about the system's internal details, such as design documents, are often referred to as grey-box techniques. These different testing techniques are complementary, since they are used to target different types of faults during different verification activities [42]. One practical issue is to determine the appropriate combination of techniques in a given development context.

When testing is based on models of the system's behaviour or structure, the terms *model-based* and *model-driven* are typically used. The main idea, in the context of model-driven development (MDD) [51, 44], is to exploit the early models of the system's specifications and design to automate (part of) the test case generation process. The development models are transformed into test models that can then be exploited by specific test generation algorithms to satisfy certain coverage requirements [61]. The notions of test model and coverage criteria are illustrated in 28.2, where the test model could, for example, be a state machine model of the SUT and the coverage criterion could require that all transitions in the state machine be

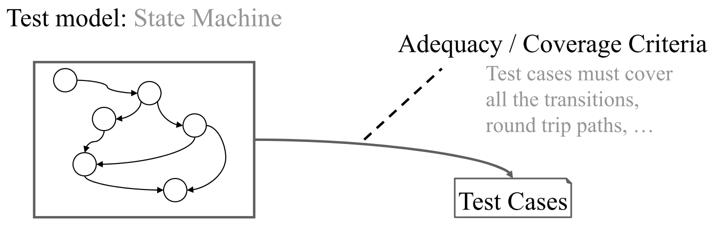

Figure 28.2 Deriving test cases from test models and coverage criteria.

covered by test suites. Based on a test model and an objective coverage criterion, mechanisms can be devised to automatically derive test cases, although this is not always an easy endeavour. The goal is then to check the conformance of a system implementation with a dedicated model representation, which is itself derived from specification and design information. Failures can be due to a fault either in the model or in the implementation. Models have also been shown to be useful in addressing the well-known oracle problem, which is the way the verdict of each test execution will be determined, since test case execution results can be compared against relevant information in the model.

Model Analysis

When the behaviour, structure, and other properties of the software are modeled during the specification and design stages, such models can be analysed to verify various relevant properties early in the development cycle, long before any testing can be performed. There are different ways to go about this, each differing in their level of formality, practicality, and scalability. One important field of research is model checking [31], which can be defined as "the process of checking whether a given structure is a model of a given logical formula". For example, the structure can be a finite state machine and the logical formula can be expressed in propositional logic or temporal logic. Such formulas may target safety, temporal, or concurrency properties. The advantage of the model-checking approach is that it systematically explores all reachable states (at a certain level of abstraction) of a system, thus providing strong confidence about whether certain properties hold. One of the challenges of model checking, however, is that on industrial-scale problems, it often faces a combinatorial explosion problem for which many solutions are still currently being investigated [31]. Another practical issue is that many model checkers require the use of modelling languages that are far away from current development practice and not always easy to use on large-scale problems, for example, temporal logic. The approach to model analysis focussed on in this chapter is aimed at being more scalable and is based on evolutionary or meta-heuristic search techniques [16]. In this situation, a fitness or objective function is defined to represent the violation of a property and a guided but random search technique is used to traverse the space of possible behaviours of the system being verified. The point is to uncover problems without having to explore the entire state space of the system. Such an approach does not guarantee that a property will never be violated but through the guided search it optimizes the chances of finding property violation occurrences if there are any. Furthermore, approaches based on search heuristics are usually not as demanding as model checking in terms of the level of formality required for the model input. This is also expected to facilitate the integration of model analysis with the other activities of software development. Examples will be discussed below.

State of the Art, State of Practice, and Problem Definition

The point here is not to provide a comprehensive overview of the state of the art in software testing. For this, the reader is referred to the several reviews on the subject [23, 49, 37]. What follows is, instead, a subjective assessment reflecting the author's experience trying to bridge research and practice and apply or tailor research results in practical situations.

Software-testing research has been a focus of attention for more than three decades, though the number of researchers in the field has grown exponentially in the last decade. This stems from the recognition that verification activities take up to 50 per cent of resources on typical development projects and far more in the context of safety-critical development. Furthermore, several recent papers have shown that current verification practices are far from satisfactory [14].

A large body of work exists on testing techniques based on control and data flow analysis of the source code [31]. Such white-box approaches have, however, shown practical limitations in terms of their scalability, e.g., when dealing with millions of lines of code, and automation. For example, the identification of infeasible control flow paths is generally an undecidable problem. It was suggested that in the context of testing larger components or systems, white-box testing could only be used to indicate what parts of the system lacked coverage once black-box testing was applied, thus leading to the refinement of black-box test suites [42]. As a result, in practice, white-box testing is used in its simplest form, such as statement coverage and, more rarely, edge/branch coverage. Very few tools [5, 35] go beyond these simplest strategies for code coverage, despite decades of research on the topic.

Another important body of work in the testing area concerns mutation analysis or fault-based testing [61]. The main idea is to define so-called mutation operators to automatically modify the SUT in small ways and generate large numbers of program mutants, i.e., programs with one incorrect change. Test suites are then refined until all mutants trigger at least one failure. The motivation is to expose weaknesses in the test data and ensure that all parts of the SUT are exercised during testing. The main problem with mutation testing is that typically many mutants are equivalent from a functional standpoint to the original program and identifying equivalent mutants is once again an undecidable problem. Second, even on small programs, very large numbers of mutants can be generated and this therefore requires valid sampling strategies to select a representative subset of mutants. Last, whether test suites that effectively detect mutants are also effective at detecting real faults is still in question, although recent evidence suggests that this is the case [33]. As a result, despite substantial research over the last two decades, to the knowledge of the author, only limited industrial application of fault-based testing has been reported.

Another area of intense research has been the use of state machine models to test communication protocols [49]. This research started three decades ago with the seminal article by Chow [59] on the W method. This research focussed, to a great extent, on finding strategies to traverse the finite state machines to guide testing and on automatically deriving distinguishing sequences of inputs to determine the resulting state of test sequences, thus helping determine whether the communication protocol conformed to its state machine specification. Though this body of work has impacted the verification of communication protocols, state-based testing in other areas is still a very rare practice. One reason is that very little is known regarding the cost-effectiveness of various test strategies based on state models [47]. Recent experiments [45] have shown that certain strategies are far from effective, even for small software components. Another reason is that the practice of state modelling in software development is still infrequent–with perhaps the exception of certain areas in embedded systems where it is required by standards–in part because it is not appropriate for every type of system but also because it is rather complex in large components or subsystems.

A large body of work addresses the problem of regression testing. The goal is to be able to minimise and prioritise regression test cases on every release [43, 36, 28]. This is important, since with the rise of incremental development, several releases of a software product are typically released every year. Since regression test suites that check whether unchanged functionality has not been broken by new changes can be quite large, re-running all regression test cases can be impractical or even impossible. This is where selection and prioritisation techniques come in. Most of these are based on source code control flow, data flow, and change analysis. Most existing studies are based on small, artificial programs and very little is known about the conditions under which such regression test techniques are beneficial. The gains in many cases seem to be rather small and test case selection often leads to faults remaining undetected. Though prioritisation is more promising, empirical results based on realistic conditions (i.e., real releases, changes, and systems) are rare. As a result, despite at least two decades of work on the subject, the results of academic research in regression testing are scarcely applied in practice and there is no comprehensive commercial tool supporting most techniques.

Other areas of intense research that have failed to transfer to industrial practice include testing based on logical specifications [19] and the use of combinatorial designs, especially in the context of testing software with many configuration parameters [60].

In recent years, with the maturation of the Unified Modeling Language (UML) [24, 50] as a standard modelling language for software, practical and model-based verification techniques have received increasing attention. Indeed, for the community working on model-based verification, UML has brought a number of advantages. It addressed a wide range of modelling requirements and came with an increasingly sophisticated set of tools and open-architecture technologies to help automate analysis and testing based on models. The UML is the modelling language used in the context of the Model Driven Architecture®(MDA®) standard supported by the Object

Management Group^{TM}(OMG^{TM})[2], a large consortium of software industry leaders. Furthermore, the language can be used at different levels of formality and extended and tailored according to needs into so-called profiles. Also UML 2.0 contains many features that support the modelling of large-scale, complex systems. Though it can be used at different levels of rigour, UML has tried to bring together best practices from many world experts and past successful methods and a real attempt to make modelling practical and scalable has been made through the work around MDA. Recently, UML was even extended to define a UML testing profile [15] whose main goal was to provide a way to model testing information, e.g., test cases, suites, and harnesses, using also the UML and its associated tools, thus facilitating the development of test tools and the interchange of test data among them. Other profiles exist, for example, in the areas of real-time and embedded systems [39] and safety-critical software [38]. Such profiles, as standards approved by the OMG, can be used to develop models that are then appropriate as a source of information for automating verification, either through model analysis or model-based testing. The work on using UML models to automate testing is, however, still fragmentary and, as discussed in [32], most techniques are superficially defined and not automated and validated to an extent that makes them interesting solutions to consider.

At the same time that MDA was evolving as a standard, a new field of research emerged focussing on hard, long-standing test automation issues: evolutionary testing. The basic idea is to transform an automation problem into a search/optimisation problem. Evolutionary search techniques, also referred to as meta-heuristic search techniques, are used to solve typically hard search problems in large search spaces [16]. They have shown to be effective for a variety of testing problems, such as automating the generation of test suites to achieve code coverage [58]. More recently, these techniques have been used to address non-functional testing problems [52, 7] such as execution time deadlines and safety properties. Despite promising results, however, the scalability and effectiveness of evolutionary testing to address realistic test automation problems still remain to be investigated [40].

A recurring problem in past testing research is related to the scalability and cost-effectiveness of the proposed test techniques. How can one gain sufficient confidence in complex software systems when limited time and resources are available? There is obviously no perfect solution to this dilemma. As further developed below, however, there are reasons to believe that certain types of model-based analysis and testing strategies, supported by evolutionary search techniques, can be combined to address large-scale software verification in a cost-effective and scalable manner.

Although mostly an academic exercise at this point, model checking is the most common approach to model analysis and consists in verifying that a system or, rather, a model thereof, complies with a property, for example related to safety or concurrency, by exhaustively exploring all its reachable states [31]. It therefore requires that system models, for example state machines, and property models such as temporal logic formulas, be developed to be applicable. These models must comply with the specific notation used by the selected model checker (e.g., Promela in SPIN

[2] www.omg.org

[31]). Despite many claims, its practice, however, is still very rare and very little evidence exists to show that it can scale up to real verification problems and whether this is applicable in most software development contexts. The main problems stem from its underlying principle of exhaustively searching a state space and, though a number of approaches have been investigated to alleviate this problem, no generally satisfactory solution has been devised.

28.3 Requirements for Model-based Verification

The purpose of this section is to clearly identify the requirements to address the problems stated in the previous section. This will help us structure our discussion in the next section and provide clearer arguments to support the proposed research directions.

Testing

R1. To be cost-effective and scalable, test techniques must be automated. This must include both the automation of test case and oracle generation. For regression testing, both test case selection and prioritisation must also be supported.

R2. The user requirements for a test technique must be realistic. It must account for what can be realistically expected given the complexity of systems, the skills and education of software engineers, international standards, and the maturity of supporting technology in the foreseeable future.

R3. Systems must be designed to be testable if any testing technique is to be cost-effective. Methodologies for supporting the design and assessment of testability must be provided. Testability is typically defined as reflecting two dimensions [29]: observability and controllability. One must be able to observe the state of the SUT and set it in a state appropriate for preparing the execution of test cases. This typically entails that built-in test interfaces be provided when designing software components and subsystems [22]. Another important aspect is that the design of a system must enable a rational integration strategy by allowing stepwise component and subsystem integration while minimising the need for stubs or mocks [21].

Analysis of Early Specification and Design Artefacts

R4. Though the analysis of all interesting properties is unlikely to be fully automated in practice, effective decision support should be provided to facilitate the analysis of large specification and design models. The involvement of the analyst must be minimised as well as the amount of information that must be processed to inspect the artefact and achieve a decision.

R5. Analysis techniques, just as for testing, must be realistic in terms of the inputs required from the analyst. This entails that the modelling notation used and underlying technology be carefully selected to be usable for large systems, be supported by effective, extensible, and open architecture tools, and account for international modelling standards. For modelling to be scalable, such notation should effectively support hierarchical, partial, and incremental modelling, the definition of model aspects (cross-cutting concerns at the modelling level), and the automated consistency and completeness checking of different modelling views (e.g., different UML diagrams).

28.4 Moving Forward

This section will outline what is considered to be an ambitious yet realistic research approach to converge towards cost-effective engineering solutions for the verification of software systems within the framework of MDD. The choices made will be explicitly linked to the requirements stated previously.

Model-Based Test Generation and Context Simulation

The point of model-based test generation is to exploit specification or design information for the purpose of automating test case generation (R1) according to systematic strategies and thus check the conformance between an implementation and its specification and design. In order to provide effective and scalable test automation, there is little alternative to model-based testing. Indeed, adequate abstract representations (models) of the SUT must be defined to support the automated derivation of test cases and oracles based on explicitly defined test strategies. Such automated support would be difficult to conceive based on source code analysis[3] or informal, and therefore ambiguous and probably incomplete, textual documentation. With test-ready system models, or in short, test models, test automation is bound to be limited to test execution and replay. On the other hand, as discussed in the section about choice of modelling paradigm on page 428, the test modelling requirements must be realistic so as to be applicable in practice in the context of large and complex software systems. A balance must be struck between the effort invested in modelling and the benefits achieved through verification automation (R2).

Figure 28.3 presents an overview of the activities and artefacts involved in model-based testing. As one can see, the approach involves four parts, respectively related to test modelling from specification artefacts[4], exploiting the test models for test case generation, generating the executable test harness (e.g., scripts) for a targeted platform, and running the test cases on it. Automating test oracles is also a very significant, practical problem throughout the software industry and is discussed in the next subsection, since it implies specific techniques.

[3] White-box testing has been shown not to scale up, as source code static analysis leads to numerous difficulties on non-trivial programs, for example, infeasible execution paths. Furthermore, a great deal of relevant information is difficult to reverse-engineer from source code, for example, states and their invariants.

[4] We can also focus on design artefacts, depending on the purpose and level of testing, though in the context of object-oriented analysis and design (OOAD), design models are refinements of analysis (specification) models. We will only refer to specifications in the remainder of the text.

Let us now examine the main features of Figure 28.3. Test modelling requires devising an analysable model for the specific purpose of test automation from system specifications. If such specifications are informal, then the modelling process is mostly manual, though it can still be supported by a specific modelling methodology and tool. For example, one may derive some form of UML state machines from the informal specification of a control or reactive system. The model then has to be checked for consistency and completeness. Traceability information between the specifications and the test model must be saved so that changes to the specifications can later be more easily accommodated by changes in the test model. The second phase is then to exploit the test model to generate test requirements (e.g., paths to cover all transitions in a state machine [22]), then test cases once the test requirements have been validated (e.g., all transition paths are feasible), and then possibly prioritise test cases if the resulting test suites are large. Prioritisation is usually based on a risk model and can be achieved through some form of optimisation strategy. The risk model can be based, for example, on how safety critical the functions triggered by test cases are (e.g., transition paths) or how error prone the executed components are (e.g., based on historical data or complexity measurements [17]). Last, any model-based testing tool needs to be coupled with the available test script generator and test execution environments on the specific development or deployment platform.

In the context of embedded real-time systems, it is crucial to perform as much verification as possible on the development platform. Running test cases on the deployment platform may be vastly expensive and in many cases dangerous and difficult to set up. To be able to run, say, a control system on a development platform, its environment needs to be simulated. For example, the behaviour sensors and actuators, external systems, or even users need to be emulated as if the system under test were actually running in its deployment environment. One interesting approach to be investigated is the use of UML modelling and extensions to model the system *context* [57] so as to be able to generate the emulation code automatically. One can, based on appropriate models, replace the drivers' code with emulation code having identical interfaces. For example, state machines describing devices can be transformed into code emulating these devices. Using the same modelling technology for modelling the context and the system offers many practical advantages. It must be determined whether, in the context of embedded systems, a UML profile, such as the Profile for Schedulability, Performance, and Time Specification (SPT) [39], could be used to model all relevant properties of a context (e.g., time properties of sensors). The principles of such an approach are illustrated in figure 28.4. Another interesting opportunity to investigate is then to use the context models to drive system testing by adapting a guided random testing approach [3]. Random test generation can be supported by different forms of guidance in order to generate tests automatically in a way that achieves proper coverage of the system and provides sufficient evidence of its reliability.

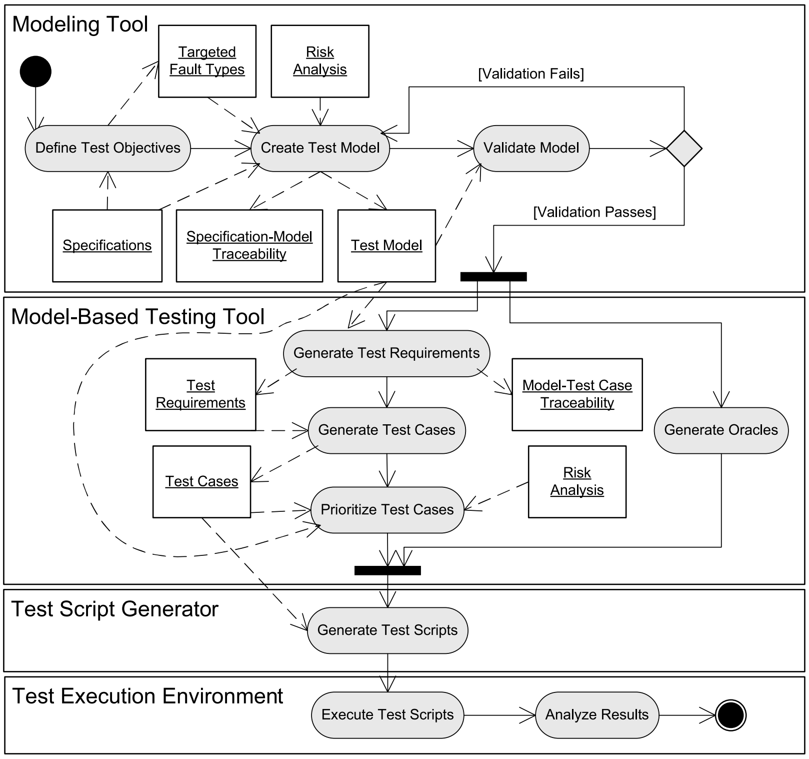

Figure 28.3 A high-level view of model-based testing.

Deriving Test Oracles from Models

One of the hardest problems in test automation is the oracle problem (R1). When a large number of test cases are generated and run, it is absolute necessary to automate the generation of test verdicts. This is the role of test oracles. Because models capture the expected system behaviour, such information can be used to check the conformance of an implementation with its specification. In UML, behaviour can be modeled in different ways and at various levels of details. State machines capture states, their invariants, and their transitions and are particularly suitable for certain types of systems and components, as in the embedded systems domain. Interaction diagrams capture possible component interaction patterns and can be useful during integration testing to check whether observed interactions (e.g., from execution traces) are consistent with expectations. Various types of constraints, defined, for example, with the Object Constraints Language (OCL)—a component of the UML—can be checked at run time, though it is important to consider the execution overhead incurred and its consequences in real-time distributed systems. Examples of types of constraints include operation contracts, class invariants, and safety properties. Research has been carried out to capture execution traces in distributed systems [34]

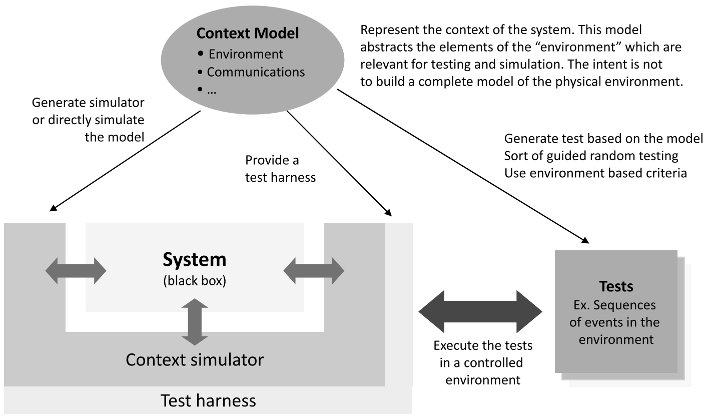
Context Model
• Environment
• Communications
• ...
Represent the context of the system. This model abstracts the elements of the "environment" which are relevant for testing and simulation. The intent is not to build a complete model of the physical environment.
Generate simulator or directly simulate the model
Provide a test harness
Generate test based on the model
Sort of guided random testing
Use environment based criteria
System
(black box)
Context simulator
Test harness
Execute the tests in a controlled environment
Tests
Ex. Sequences of events in the environment

Figure 28.4 Context modelling and simulation for test purposes.

and abstract sequence diagrams from these. One challenge, however, is to obtain models at similar levels of abstraction as design models from execution traces to facilitate comparisons. Studies have assessed the effectiveness of contract assertions as oracles [13] and the necessary level of detail and density of oracles [12] to achieve effective fault detection results. The effectiveness of using state invariants as oracles has also been investigated and has shown in many cases to be insufficient by itself [21, 18]. Overall, very little empirical research and scant results exist regarding the cost-effectiveness of various oracle strategies, whether for model-based testing or in general. Effective model-based oracle strategies and their empirical assessment are therefore an important research endeavour. One particularly difficult area of investigation is the automation of oracles for detecting quality-of-service problems, for example, related to response time, throughput, or security.

Choice of Modelling Paradigm

Ideally, the same modelling paradigm (notation, process) and technology should be used for system modelling and test modelling. In practice, this facilitates integration of specification, design, and testing activities. Designers and testers can then "speak" the same "language" and use common development platforms and technologies. This is of high practical importance, since the lack of collaboration and communication between design and testing teams is a notoriously common problem. As a result, the adopted modelling paradigm needs to account for the needs of all stakeholders, including analysts, designers, and testers.

In the foreseeable future, MDA and UML will remain the de facto international standards for MDD (R2 and R5). They will continue to evolve under the control of the OMG and will be increasingly supported by open-source or open architecture technologies such as Eclipse-based modelling platforms [25]. These tools typically enable the definition of new profiles (e.g., for specific verification purposes) and plug-ins to automate model analysis and testing (R1 and R4). These characteristics are important because they enable the use of common environments for analysts, designers, and testers. Second, by supporting the definition of profiles in a UML context, they enable the extension of design models for the purpose of deriving test models amenable to automation. Through plug-in mechanisms, modelling and development environments can then be extended to automate model-based testing. Many useful UML profiles have already been defined and approved by the OMG, such as profiles for real-time and concurrent systems–Schedulability, Performance, and Time Specification (SPT) [39] and Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [48]–quality of service [38], and testing [15]. All development activities can then be centred around one model repository that is exploited for various purposes, such as the generation of specification and design documents, code generation, test case generation, and model analysis. Furthermore, substantial research is currently underway to adapt the concepts of aspect-oriented development to the UML modelling realm (R5), thus facilitating the definition of cross-cutting properties, for example, safety or security [2]. The current definition of an executable subset of UML [44] is also expected to facilitate model analysis automation.

Testability

In order to decrease the cost of testing and facilitate its automation, providing ways to assess and improve the testability (R3) of architectures and designs is an important endeavour. Work has already been reported on the analysis of dependencies among components in order to identify integration hotspots and devise optimal integration orders so as to minimise testing efforts [21]. Such dependency analyses can be conveniently based on UML models (e.g., class and sequence diagrams, OCL constraints) or extensions providing more detailed information about dependencies. Testability measurement frameworks have also been proposed to help assess testability based on fault injection [41] and object-oriented designs [29]. More research, however, is still required in this important area to guide designers towards testable architectures and designs. In particular, field studies must be carried out to really understand what the most problematic testability factors are. Case studies are also required to understand the cost-benefit relations of various approaches to improve observability, for example, the cost of developing built-in test interfaces and the various trade-offs that can be made in terms of the granularity of information flowing through such interfaces. For example, should concrete object states be made visible or are abstract state assertions sufficient? Also, how does one effectively deal with observability in the context of distributed systems with distributed state information?

Search-Based Model Analysis of Non-Functional Properties

Once models of the systems are available during the specification or design stages, they can be exploited to analyse relevant properties related to non-functional aspects of the system such as safety, security [10], availability, and performance [53]. Such models, in the context of the previous discussion, would typically be expressed with one or several of the existing UML profiles or rely on a newly defined, domain-

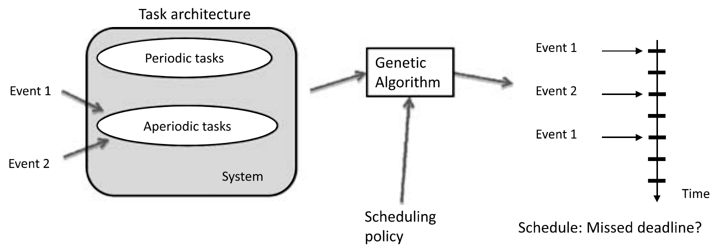

Figure 28.5 Searching for deadline misses.

specific profile. Model analysis is complementary to testing since it can help identify problems early on in the development process, long before any code is available for testing. As described previously, one very active field is that of model checking, which is based on a systematic exploration of the state space of a system as represented by a formal model, for example, expressing real-time properties in temporal logic. The approach adopted here, which will be explored further in the future, is different and likely to be more scalable and applicable, especially in the context of evolving UML and MDA standards. Based on adequate profiles of the UML, a search-based approach to the analysis of non-functional properties of software designs is adopted. What this means is that any non-functional property analysis may be expressed as a search problem and meta-heuristic search algorithms, like evolutionary algorithms, may be used to explore the space of execution scenarios defined by the system model. For example, based on a model of the task architecture of a concurrent real-time system, one can search whether possible scenarios can trigger response time or concurrency problems, such as deadline misses or deadlocks. This is illustrated in figure 28.5, where a task architecture describing tasks, their deadlines, interdependencies, and estimated execution times is provided as input to a genetic algorithm (GA). Assuming a specific scheduling policy, the search is then guided towards schedules (e.g., task seeding times) that minimise the difference between estimated task completion times and their deadlines. The task architecture information can come, for example, from UML design models using the MARTE profile.

Recent work on this topic, using dedicated GAs as a search mechanism, has yielded promising results [52, 7] and has been shown to be very effective, for example, when compared to similar model-checking studies. A search approach does not, of course, guarantee that any property holds in a design. It merely indicates that, if no violations are found, they are unlikely to occur. This is also the case, however, with a model-checking approach in practice, since models can be erroneous anyway and heuristics must be adopted to help improve the scalability of the state exploration [31].

One advantage with a search-based approach is that it does not attempt to perform an exhaustive and systematic exploration of the state space but, rather, searches, in a random but guided way, for specific problems, such as the violation of safety or concurrency properties. In addition, as opposed to model checkers requir-

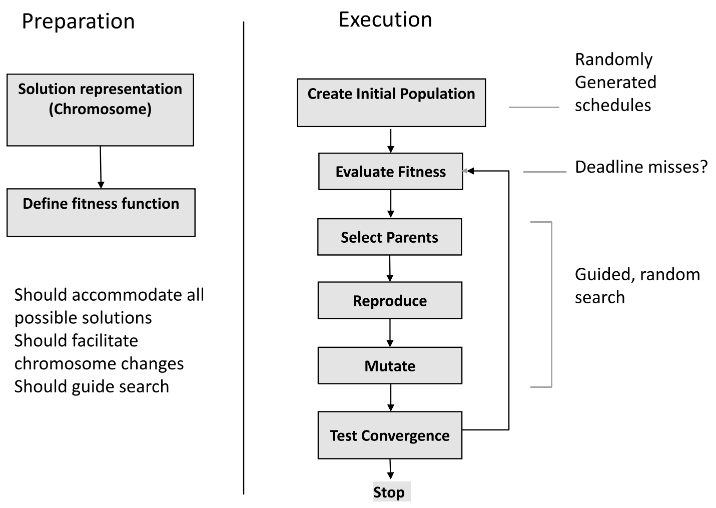
Preparation
Execution
Solution representation (Chromosome)
Define fitness function
Should accommodate all possible solutions
Should facilitate chromosome changes
Should guide search
Create Initial Population
Evaluate Fitness
Select Parents
Reproduce
Mutate
Test Convergence
Stop
Randomly Generated schedules
Deadline misses?
Guided, random search

Figure 28.6 Using GAs to search for deadline misses.

ing the definition of properties in formal logical expression, for example, temporal logic [31], a search-based approach has shown to work in combination with UML models and their extensions [52, 7]. It is important that a model analysis technique avoid or minimise the amount of additional modelling that must be performed, in addition to what is necessary for design and analysis purposes (R5). Ideally, design models, for example, in UML, should be reused or, at worst, augmented to enable analysis. For instance, recent work [7] on detecting deadlocks and starvation problems relied on the SPT and MARTE UML profiles and obtained very encouraging results. This implies, of course, that an appropriate solution representation (defining the search space) and fitness function be defined to effectively guide the search. This may, however, turn out to be impossible for some non-functional properties and clearly identifying the opportunities and limits of such an approach is part of the needed research. As an example, for GAs a number of important decisions have to be made that can potentially impact the efficiency and effectiveness of the search, as illustrated in when searching for deadline misses in real-time systems. In the preparation stage, solutions (e.g., schedules) have to be represented as “chromosomes”, a fitness function must be defined based on an effective search heuristic, and several parameters of the GA must be set appropriately (e.g., mutation and cross-over rates). During the execution stage, the GA then generates a random initial population of chromosomes and then iteratively modifies them through generations of successive populations in what constitutes a random guided search.

Because there is no guarantee that a search mechanism will find property violations, search algorithms need to be carefully assessed through empirical studies, showing that the search is indeed effective at finding problems and scalable to realistic models [40]. Another challenge is to define or adapt appropriate UML profiles or other domain-specific modelling languages to capture all the required information for the search to be effective, but in a way that is consistent with engineering practices, based on international standards and well supported by tools. A technology requiring improbable inputs is not likely to ever transfer to practice.

Empirical Studies of Model-Based Analysis and Testing

Regardless of the specific choice they entail in terms of the test model, coverage criteria, and oracle, all verification techniques that scale up are inherently heuristic: They do not guarantee the detection of faults. Their main point is to be systematic in the way the system or model is exercised and verified so as to obtain a predictable result in terms of fault detection and focus on certain types of faults. Software engineers are therefore confronted with making the difficult decision of choosing a set of test and analysis techniques—possibly different across various test phases—which will fit within their budget and time constraints and that are likely to be effective at detecting faults early in the development process. For example, Briand et al. [45] used simulation to assess and compare various test strategies based on state machines in terms of fault detection. The results showed significant variations across SUTs due to their real-time characteristics, among other things.

It is common to refer to the cost-effectiveness of a technique as the fault detection obtained over the cost of applying it or, even better, the effort saved by fault detection minus the effort of detection. Since one cannot analytically assess or compare the cost-effectiveness of various testing techniques, it is natural to resort to empirical studies. Such studies should investigate the following categories of questions:

- What cost and fault detection rates can be expected from using a verification technique?
- How do alternative techniques compare in terms of both cost and fault detection rates?
- Is it beneficial to combine two or more verification techniques? Are these techniques complementary in terms of fault detection?

Empirical studies are particularly complex, however, because the cost-effectiveness of a verification technique depends on many other factors. For example, regarding testing techniques (see figure 28.7), it is not just the coverage criterion and oracle that determine cost and effectiveness:

- The test technique's degree of automation obviously affects its cost and, under time constraints, its effectiveness, since the level of coverage achieved may be less than 100 per cent. Automation includes the identification of test requirements, the generation of test drivers and test stubs (e.g., mock objects), and the implementation of test oracles.
- The types of faults present in the SUT and their probability of detection (fault profile) affect the test technique's detection capability. For example, in a study investigating the cost-effectiveness of statechart-based testing [18], we found that many faults in a concurrent cruise control could not be easily detected with just test cases generated from statecharts, since the concurrent real-time behaviour of the SUT was not part of the test model and therefore not fully exercised. As

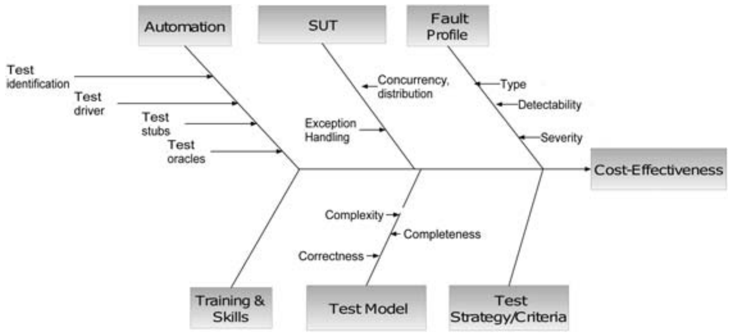

Figure 28.7 Testing cost-effectiveness factors.

a result, certain faults that could only be exercised by a certain scheduling of external events never triggered a failure.

- The SUT itself has, of course, an impact on the probability with which certain faults can be detected given a certain test technique. Concurrency, distribution, and complex exception handling partially determine not only what types of faults are present in the SUT but also how easy these faults will be to detect.
- The training and skills of testers also have a strong impact, since test techniques are usually not entirely automated and require at the very least human input. For example, test models are devised by software engineers and may vary greatly in terms of correctness and completeness. Certain techniques are entirely based on human intuition and an understanding of the system's behaviour, such as the category-partition method [9], and have been observed to yield a wide variety of results under identical conditions [8].
- Test models vary in complexity and cost. Certain techniques require complex models that entail significant cost. Furthermore, such models are likely to be incorrect or even incomplete in practice, which is an important aspect regarding how human factors can affect a technique's applicability.

Because the preceding factors can have a dramatic impact on a test technique's cost-effectiveness, empirical studies should try to control for such factors in order to ensure that any cost-effectiveness comparison among test techniques is unbiased but also representative of the targeted context of the study.

28.5 Software Verification Research at Simula

This section provides a structured overview of research objectives, recent work, and industry collaborations involving the author and his colleagues at Simula.

Research Scope

One Simula Research Laboratory's mandate is to perform industry-driven research to increase SRL's relevance and impact on actual engineering practices. In this context, and based on the discussions in previous sections, our approach to verification research can be characterised as follows:

- We assume that development and test models are expressed in the UML or extensions through profiles. Depending on the targeted aspect of verification, we may be led to define new profiles. We therefore place our work within the MDA standard proposed by the OMG in order to benefit from a rapidly growing, and often open-source, technological base and comply with the only de facto international standard regarding software modelling.
- We model not only the software being tested but also its environment. This is particularly important for embedded systems that need to be verified and tested on the development platform before undergoing the same in realistic settings, since such environment (context) models can help us generate the emulation code we need to emulate drivers and other context elements.
- Model analysis and test generation are automated through the use of metaheuristic search algorithms such as GAs. Our choice is therefore to rely on heuristic search rather than systematic and complete state exploration, such as in model checking. The main motivation is to achieve scalable solutions.
- Simulations, field studies, or controlled experiments are used to assess empirically the cost and fault detection effectiveness of verification strategies, as well as their scalability to larger models. Empirical research is therefore a major component of our activities and requires developing appropriate methodologies to empirically assess verification techniques.
- Particular emphasis will be placed on verifying non-functional properties, including safety, robustness, response time, security, and concurrency. Very little is available regarding these aspects in the context of MDD.

Related Research

To provide the background information that may help explain and exemplify the perspectives described above, recent work by the author related to this chapter can be summarised as follows.

Model Analysis

- Automated traceability of UML model refinements [27]: Traceability between analysis (domain) models and design models must be preserved so as to be able to propagate changes from one to the other. This work provides a means to automate the creation of traceability information in the context of UML model refinements.
- Impact analysis based on UML models [1]: This works provides a solution to identify the ripple effects of changes to the design across the system, based on the analysis of UML models, with a focus on class and sequence diagrams with OCL constraints.
- Model-based prediction of resource usage and load in distributed systems [4]: UML sequence diagrams, augmented with timing information, are used to pre-

dict system behaviour at the design phase in terms of network traffic, CPU, and memory usage.

- Deadlock and starvation analysis based on UML models with the MARTE profile [7]: Based on UML MARTE models (e.g., sequence diagrams), GAs are used to search for deadlocks and starvation problems in concurrent systems.
- Definition of a safety profile to support third-party safety accreditation in the aerospace domain [10]: This work is a first step towards the definition of a safety UML profile to enable safety analysis and accreditation in the context of MDA.
- Reverse-engineering UML sequence diagrams from dynamic analysis in the context of distributed Java systems [34]: This was a first step towards collecting and analysing traces in the context of distributed systems to be able to compare executions to UML design models and identify discrepancies in an automated fashion.

Model-Based Testing

- Regression test selection based on UML models [26]: Regression test selection based on source code analysis is a well-researched area. The only problem is that it is applicable only after changes have been applied to the source code. This work provides a way to assess regression test efforts and identify test cases to rerun based on changes in UML design models.
- Empirical evaluation and improvement of strategies for state machine-based testing and its combination with white-box testing [45, 18]: Simulations and series of experiments were performed to assess the cost-effectiveness of testing strategies based on state machines, how they compare to simple control flow testing, and whether the two should somehow be combined. A careful analysis also led to refinements of testing and oracle strategies in addition to practical recommendations.
- Improving state machine testing with data flow analysis [56]: Because empirical results have shown that it is not easy to select paths to test in state machines, this work investigates whether data flow information, derived from operation contracts and guard conditions in OCL, can be used to select high-fault-detection paths. The authors empirically investigate whether paths that exercise the most data flow also help detect more faults.
- Stress-testing distributed systems [54, 30]: Based on UML sequence diagrams augmented with timing information and specially designed GAs, the goal here is to stress-test distributed systems with test scenarios maximising network traffic.
- Contracts as test oracles in sequential and concurrent contexts [13, 20]: This work extends the Java Modeling Language to allow the definition of contract assertions in the context of concurrent systems. This is a necessary technology to help define test oracles for concurrent Java systems.
- Stress-testing real-time systems by generating test cases that maximise completion times for target tasks [52]: The goal was to stress-test real-time systems to maximise their chances of missing deadlines. The goal was to increase confidence so that, if such test cases were successful, deadline misses would be unlikely. Test cases are derived from UML models augmented using the SPT profile, which

has now been replaced by MARTE. Similar to concurrency problems, deadline misses, in some cases, can be directly identified from the models, without resorting to testing.

- State machine-based integration testing [6]: Based on the analysis of state machines and interaction diagrams in UML models, this work attempts to devise systematic class integration testing strategies.
- Contract-based test automation of commercial off-the-shelf (COTS) components [62]: Assuming only contracts are available to describe the functionally of a COTS component, since design details are usually proprietary, this work adapts existing strategies to automatically derive adequate test suites that can be used by the component users to assess its reliability.

Industry-Driven Engineering Research in Software System Development

It is important to describe how we intend to proceed with the previously mentioned research given SRL's mandate. As opposed to many other engineering disciplines, it is rarely possible for software engineering researchers to reproduce the phenomena they are studying in the laboratory. Despite many decades of research in software engineering, the impact of academic software engineering research on engineering practice is unclear, although there is probably wide variation across different research areas. The gap between research and practice is, in our opinion, partly due to the historical specificities of software engineering, which originally branched out of computer science, which itself was initially a branch of discrete mathematics. There is therefore little engineering tradition in software engineering research and, as a result, research is far too rarely problem driven or based on precisely defined problems reflecting the reality of software system development. Though research on fundamental problems is obviously crucial, such an imbalance contrasts sharply with research in other engineering disciplines. Furthermore, because of the human and organisational factors involved in software development, software engineering cannot simply be seen as a mathematical problem. The right trade-offs have to be found between the seemingly conflicting objectives of engineering rigor, changing requirements, and tight schedules. Any solution must be scalable to large systems and teams, be compatible with the practice of incremental development, and support frequent change.

The preceding statements entail that researchers cannot fully understand the actual problems or devise and assess suitable engineering solutions, whether for verification or any other aspect of software development, if they do not work in close collaboration with industry. This is why software engineering research at SRL has focussed on industry-driven projects. A difficult question, however, is how to make such collaboration effective and productive. The practicing engineer's priority is to complete projects on schedule and not, unless there exists a clear mandate to do so, to improve engineering practices. Since software development is a collective endeavour, the task of implementing change in practices, with its training and mentoring requirements, is also far from a trivial matter and requires dedicated resources.

To make collaborative, industry-driven research effective, the verification group at SRL has adopted a research process that relies on integrating the work of doctoral candidates into the practice of industrial partners. At a high level, the first step of the process is to identify difficult, long-standing problems that our industry partners have been facing on projects. The state of the art related to the identified problem is then assessed by performing a systematic and thorough review of the scientific literature. Usually, significant gaps are found, as many of the proposed techniques are incomplete or simply not applicable as defined, for example, not scalable. A solution is then devised in context and scientifically evaluated on actual systems, using, for example, actual fault data, accounting for human and organisational factors. The last step is to generalise the solution to make it applicable to a wider context by attempting, for example, to relax some of the assumptions underlying the work.

Adopting such an approach to research presents a number of practical challenges that should be addressed. It is important to ensure that the problem selected is significant enough to constitute a doctoral thesis topic. In the current stage of maturity of software engineering practice and research, this is usually not a difficult problem. Second, long-term commitment from the industry partners is required, because such collaborations must necessarily last the duration of a thesis and cannot be interrupted without serious consequences. Working with industry partners requires understanding their problems, technology, and working processes, which entails an effort overhead not normally present in traditional doctoral work.

The following provides two illustrative examples of recent industry collaborations in which the author and his SRL colleagues were involved.

Telecom. This project took place in collaboration with an industry partner in the telecom domain. One problem identified on this project was that insufficient resources were available to thoroughly test all components on every release of a telecom product. As a result, testing was ad hoc and mostly driven by individual choices. We analysed change and fault data, as well as the source code of a number of releases and devised a prediction model to identify, on each release, where faults were more likely to be located. A testing strategy was then devised to adjust the testing intensity to the likelihood of faults in a component. This strategy was assessed and showed a 100 per cent return on investment.

Manufacturing. This project took place in the context of the development of manufacturing systems. One important problem raised was related to the safety of such systems and therefore the verification of a safety component in charge of monitoring unsafe events during the system's execution. Given the stringent safety requirements, a model-based testing approach, based on formal state models, was defined. To support change, a design strategy was elaborated to ensure traceability between the state model and the source code, thus facilitating future changes. Empirical studies are underway to assess the benefits of both the testing and design approaches.

28.6 Conclusions

This chapter addresses the verification of software systems, a highly crucial topic, given the growing importance of software throughout most economic sectors, especially in the many domains where it plays a safety- or business-critical role. This chapter argues that the current state of the art is not even close to addressing the highly challenging problems that software engineers are facing when verifying software systems. The scale and complexity of the work are so daunting that automation is a requirement. Effective automation, however, can only be achieved if systems are described by appropriate models that not only help describe their specification and design but also support verification, including both analysis of specification and design artefacts and model-based testing. With the advent of the international UML 2.0 standard and its associated MDA framework, a growing body of technology, including open-source and architecture tool platforms, has rapidly developed in recent years. This means that the introduction of MDD practices, including model-driven verification, is a much more realistic opportunity today than even five years ago.

Through carefully adapted and tailored modelling technologies, with a rigorous scientific approach, and through tight collaborations between industry and research institutions, the complexity and scale of software verification may now be tackled. This chapter highlights the need for a stronger focus on non-functional aspects of verification and addresses the use of meta-heuristic search algorithms as a practical alternative to automated model analysis and model-based testing. It also shows that model-based verification is essentially a trade-off between modelling effort in the early stages of development and automation gains in verification activities. Though devising scalable and effective solutions for model-driven verification is a significant challenge, with the rising complexity and criticality of software-based systems the use of models is bound to yield increased benefits. The benefits, costs, and scalability of the proposed solutions can only be investigated in industrial contexts and on actual systems, accounting for human, organisational, and economic factors. Such investigations are best done through carefully designed empirical studies involving both SRL researchers and their industrial partners. Such a research agenda is therefore in perfect alignment with the mandate of SRL, a leader in industry-driven, high-impact IT research.

References

[1] L. C. Briand, Y. Labiche, L. O'Sullivan, and M. Sowka. Automated impact analysis of UML models. *Journal of Systems and Software*, 79(3):339–352, 2006.

[2] R. France, I. Ray, G. Georg, and S. Ghosh. Aspect-oriented approach to early design modelling. *IEE Proceedings-Software*, 151(4):173–185, 2004.

[3] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[4] V. Garousi, L. Briand, and Y. Labiche. A UML-based quantitative framework for early prediction of resource usage and load in distributed real-time systems. *Software and Systems Modeling (Springer)*, 2008.
[5] IBM. Rational test RealTime. 2005.
[6] S. Ali, L. C. Briand, M. J. Rehman, H. Asghar, M. Z. Z. Iqbal, and A. Nadeem. A state-based approach to integration testing based on UML models. *Information and Software Technology*, 49(11-12):1087–1106, 2007.
[7] M. Shousha, L. Briand, and Y. Labiche. A uml/spt model analysis methodology for concurrent systems based on genetic algorithms. *ACM/IEEE 11th International Conference in Model Driven Engineering Languages and Systems (MODELS 2008)*, 2008.
[8] T. Y. Chen, P. L. Poon, S. F. Tang, and T. H. Tse. On the identification of categories and choices for specification-based test case generation. *Information and software technology*, 46(13):887–898, 2004.
[9] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. 1988.
[10] G. Zoughbi, L. C. Briand, and Y. Labiche. A uml profile for developing airworthiness-compliant (rtca do-178b) safety-critical software. *International Conference on Model Driven Engineering Languages*, 2007.
[11] C. Ghezzi, M. Jazayeri, and D. Mandrioli. Fundamentals of software engineering. 1991.
[12] Y. L. Traon, B. Baudry, and J. M. Jezequel. Design by contract to improve software vigilance. *IEEE Transactions on Software Engineering*, 32(8):571, 2006.
[13] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to improve the testability of object oriented code. *Software Practice and Experience*, 33(7), 2003.
[14] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology RTI Project*, 2002.
[15] P. Baker, Z. R. Dai, J. Grabowski, Ã. Haugen, S. Lucio, E. Samuelsson, I. Schieferdecker, and C. Williams. The UML 2.0 testing profile. *8th Conference on quality Engineering in Software Technology*, Nuremberg, Germany, 2004.
[16] B. F. Jones. Special issue on metaheuristic algorithms in software engineering. *Information and Software Technology*, 43:14, 2001.
[17] N. Fenton and S. L. Pfleeger. *Software metrics: a rigorous and practical approach.* PWS Publishing Co. Boston, MA, USA, 2nd edition, 1998.
[18] S. Mouchawrab, L. C. Briand, and Y. Labiche. Assessing, comparing, and combining statechart-based testing and structural testing: An experiment. *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2007.
[19] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, 1994.
[20] W. Araujo, L. Briand, and Y. Labiche. Concurrent contracts for java in JML. *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 37–46, 2008.

[21] L. C. Briand, Y. Labiche, and Y. Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering,* 29(7):594–607, 2003.

[22] R. Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Professional, 1999.

[23] R. V. Binder. Testing object-oriented software: a survey. *Software Testing, Verification and Reliability,* 6(3–4):125–252, 1996.

[24] T. Pender *UML bible.* John Wiley & Sons, Inc. New York, NY, USA, 2003.

[25] E. Foundation. Eclipse modeling framework (EMF). May 2005.

[26] L. Briand, Y. Labiche, and S. He. Automating regression test selection based on uml designs. *Information and Software Technology (Elsevier),* 51(1), 2009.

[27] L. C. Briand, Y. Labiche, and T. Yue. Automated traceability analysis for UML model refinements. *Information and Software Technology,* 51(2):512–527, 2009.

[28] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications,* pages 312–326. ACM New York, NY, USA, 2001.

[29] S. Mouchawrab, L. C. Briand, and Y. Labiche. A measurement framework for object-oriented software testability. *Journal of Information & Software Technology,* 47(15):979–997, 2005.

[30] V. Garousi, L. C. Briand, and Y. Labiche. Traffic-aware stress testing of distributed systems based on UML models. *Proceedings of the 28th international conference on Software engineering,* pages 391–400. ACM New York, NY, USA, 2006.

[31] M. Pezzé. *Software Testing and Analysis: Process, Principles and Techniques.* Wiley, 2008.

[32] A. C. D. Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007,* pages 31–36. ACM New York, NY, USA, 2007.

[33] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering,* 32(8):608, 2006.

[34] L. C. Briand, Y. Labiche, and J. Leduc. Towards the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Transactions on Software Engineering,* 32(9), 2006.

[35] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. *The Journal of Systems & Software,* 48(2):79–89, 1999.

[36] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM),* 6(2):173–210, 1997.

[37] N. Juristo, A. M. Moreno, and S. Vegas. A survey on testing technique empirical studies: how limited is our knowledge. *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n*, pages 161–172, 2002.
[38] U. OMG. Profile for modeling quality of service and fault tolerance characteristics and mechanisms. *Object Management Group*, 2005.
[39] U. OMG. Profile for schedulability, perfomance and time specification. 2005.
[40] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of evolutionary testing. Technical report, Simula Research Laboratory, 2008.
[41] J. Voas, M. Schmid, M. Schatz, and D. Wallace. Testability-Based assertion placement tool for Object-Oriented software. *NASA*, (19980045759), 1998.
[42] B. Marick. *The craft of software testing*. PTR Prentice Hall Englewood Cliffs, NJ, 1995.
[43] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.
[44] S. J. Mellor and M. Balcer. *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.
[45] L. C. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on statechart. *Proceedings of the 26th International Conference on Software Engineering*, pages 86–95. IEEE Computer Society Washington, DC, USA, 2004.
[46] R. Conradi, P. Mohagheghi, T. Arif, L. C. Hegde, G. A. Bunde, A. Pedersen, and E. Norway-Grimstad. Object-Oriented reading techniques for inspection of UML models– an industrial experiment.
[47] L. C. Briand, M. D. Penta, and Y. Labiche. Assessing and improving state-based class testing: a series of experiments. *IEEE Transactions on Software Engineering*, 30(11):770–783, 2004.
[48] OMG. The official OMG MARTE website. 2008.
[49] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-A survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
[50] O. M. G. Uml. 2.0 superstructure specification. *OMG ed*, 2003.
[51] A. G. Kleppe, J. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.
[52] L. C. Briand, Y. Labiche, and M. Shousha. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Journal of Genetic Programming and Evolvable Machines*, 7(2), 2006.
[53] D. C. Petriu. Performance analysis with the SPT profile. pages 205–224, 2005.
[54] V. Garousi, L. C. Briand, and Y. Labiche. Traffic-aware stress testing of distributed real-time systems based on uml models using genetic algorithms. *Journal of Systems and Software*, 81(2):161–185, 2008.
[55] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM international Con-*

ference on Automated software engineering, pages 273–282. ACM New York, NY, USA, 2005.

[56] L. C. Briand, Y. Labiche, and Q. Lin. Improving statechart testing criteria using data flow information. *16th IEEE International Symposium on Software Reliability Engineering, 2005. ISSRE 2005*, page 10, 2005.

[57] H. Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications With Uml.* Addison-Wesley Professional, 2000.

[58] B. F. Jones, H. H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[59] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, pages 178–187, 1978.

[60] C. Yilmaz, M. B. Cohen, and A. A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 32(1):20–34, 2006.

[61] P. Ammann and J. Offutt. *Introduction to software testing.* Cambridge University Press, 2008.

[62] L. C. L. Briand, Y. Labiche, and M. Sowka. Automated, contract-based user testing of commercial-off-the-shelf components. *ACM/IEEE International Conference on Software Engineering (ICSE).* ACM Press, 2006.