

Generating regression tests for software migration

Franck Fleurey, Benoit Baudry, Alain Nicolas, Erwan Breton, Jean-Marc Jézéquel

▶ **To cite this version:**

Franck Fleurey, Benoit Baudry, Alain Nicolas, Erwan Breton, Jean-Marc Jézéquel. Generating regression tests for software migration. [Research Report] RR-6971, INRIA. 2008, pp.46. inria-00399493

HAL Id: inria-00399493
https://inria.hal.science/inria-00399493

Submitted on 30 Jun 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Generating regression tests for software migration

Franck Fleurey — Benoit Baudry — Alain Nicolas — Erwan Breton — Jean-Marc Jézéquel

N° 6971

June 2009



ISSN 0249-6399 ISRN INRIA/RR--6971--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Generating regression tests for software migration

Franck Fleurey* , Benoit Baudry† , Alain Nicolas‡ , Erwan Breton‡ , Jean-Marc Jézéquel§

Thème : Model-Driven Engineering
Équipe-Projet Triskell

Rapport de recherche n° 6971 — June 2009 — 43 pages

Abstract: Software modernization projects consist of the redesign of a legacy application and its migration to a novel platform. The validation of the migration step is a major concern since it has to check the exact preservation of the functionalities of the legacy system. Regression testing can be used to perform this validation. However, in most migration projects the specifications and test cases for the legacy application are obsolete. In this context, producing and running the tests can represent more than 50% of the overall migration cost. Model-driven migration is based on the reverse engineering of models of legacy systems for modernization. In this paper, we report on an experience where these models where used to derive functional test scenarios. Based on these models, we have defined several test criteria to qualify the obtained test scenarios. The models and criteria for test generation were developed for the migration of a large-scale banking application.

Key-words: software migration, model-driven engineering, model-based testing, test generation, regression testing

This work was done in the context of the INRIA industrial post-doc of Franck Fleurey at Sodifrance in 2007-2008.

* SINTEF, Oslo, Norway – franck.fleurey@sintef.no
† INRIA, Rennes, France – bbaudry@irisa.fr
‡ Sodifrance, Nantes, France – {ebreton, anicolas}@sodifrance.fr
§ University of Rennes, Rennes, France – jezequel@irisa.fr

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Generating regression tests for software migration

Résumé : Software modernization projects consist of the redesign of a legacy application and its migration to a novel platform. The validation of the migration step is a major concern since it has to check the exact preservation of the functionalities of the legacy system. Regression testing can be used to perform this validation. However, in most migration projects the specifications and test cases for the legacy application are obsolete. In this context, producing and running the tests can represent more than 50% of the overall migration cost. Model-driven migration is based on the reverse engineering of models of legacy systems for modernization. In this paper, we report on an experience where these models where used to derive functional test scenarios. Based on these models, we have defined several test criteria to qualify the obtained test scenarios. The models and criteria for test generation were developed for the migration of a large-scale banking application.

Mots-clés : software migration, model-driven engineering, model-based testing, test generation, regression testing

1 Introduction

As development techniques, paradigms and platforms evolve far more quickly than domain applications, software modernization is a constant challenge to software engineers. Modernization projects usually consist of the redesign and refactoring of a legacy application and its migration to a novel platform. Since migration is usually more than a simple translation from a platform to another, it is impossible to reach full automation. Thus, it is necessary to thoroughly validate the produced system to ensure that it behaves as the original system. As observed in several works, the cost of validation in migration projects can reach 70% of the total cost [22, 7].

The validation of the migration is based on a set of reference test scenarios that are relevant of the behaviours of the legacy system. These test scenarios should capture all relevant behaviour of the legacy system, and include the expected results. Then, the migrated system is valid if all these tests pass on this new version of the system: this indicates that the migrated system has the same behaviour as the legacy system. The main reason for the prohibitive cost of validation is the lack of accurate and updated specifications and reference test scenarios for the legacy system. This implies that the reference tests have to be generated at the time of the migration, and that this generation can only be based on the running legacy system and on the knowledge of the legacy applications users. This leads to an expensive process in which both legacy application domain experts and testers of the migration team have to be involved.

For example, let us consider how Sodifrance, the company with which we have collaborated for this work, deals with the generation of reference test scenarios. Currently, it asks its customers to produce the reference tests for the legacy applications. The production of good test scenarios is very costly both for Sodifrance and for its customer because it requires a number of iterations between the domain experts who write the tests and the migration team who checks their coverage. Most of this process is manual and the communication between the domain experts and the migration team is difficult because of their different expertises: domain experts know how to use the application but do not know how it is made whereas the developer can only give feedback at the code level. This time and effort must be spent since the reference test scenarios are a critical part of the agreement between the company and its client: they specify the minimal quality required by the client for the migrated system.

Two important factors have to be considered to reduce the cost of the validation: techniques and test criteria to assist the systematic generation of test scenarios, and models that can be used without involving the application domain experts. These techniques should assist the testers in such a way that they do not need to understand precisely all the business concerns in order to create relevant test scenarios or to improve the quality of existing scenarios. We can notice that one benefit of this very particular testing process is that a complete oracle is available for free: the legacy system can provide the expected output for any input scenario.

The objective of this work is to propose a testing technique that would allow Sodifrance to:

1. Create the reference tests required to develop and validate the migrated application without the help of domain experts (i.e. without involving the client).

2. Provide regression tests to the client together with the migrated application. These tests are valuable to the client as they can be used for any further evolution of the application.

This document is organised as follows. Section 2 presents in detail the model-based approach developped by Sodifrance for software migration. Section 3 presents a set of test criteria based on the models used for migration. Section 4 details how the approach can be completed using structural test generation techniques. Finaly, section 5 concludes this report.

2 Software migration using MDE

Positioned from the mid 80's on IT services dedicated to Banks and Insurance Companies, Sodifrance has developed a strong legacy modernization expertise based on software solutions to industrialize transformation projects. Since 1994, Sodifrance has adopted and promoted model-driven engineering (MDE) approaches for modernization projects. It has industrialized model-driven techniques for reverse-engineering, code analysis and transformation and for representing and manipulating information systems. These solutions allow the company to propose efficient and profitable solutions for migration and modernization of software legacy systems.

This chapter presents the model-driven migration process developed at Sodifrance. This process includes automatic analysis of the existing code, reverse engineering of abstract high-level models, model transformation to target platform models and code generation. We detail the different meta-models and transformations that are produced for the automation of these steps. We also discuss what artefacts can be directly reused and which ones need to be adapted from one project to another. Sodifrance has developed a tool suite for model manipulation called Model-In-Action (MIA) that is used as a basis for automating the migration.

2.1 Model-driven migration process

The constant evolution of software technology leads to continuous migrations of software components. These projects may be motivated by different reasons such as the obsolescence of a technology, the pressure of users, or the need to build a single coherent information system when merging companies. Most of the time software migration is achieved through the full re-development of the legacy application. Model-driven software development offers an opportunity for increasing the automation in software migration.

The full automation of migration is difficult to achieve not only because of the distance between the legacy platform and the new platform but also in order to ensure the quality of the new application. Most of the time, the objective of migration is not to simply "compile" the legacy application to a new platform but to create a new version of the application using state of the art development techniques. This is necessary to ensure the maintainability of

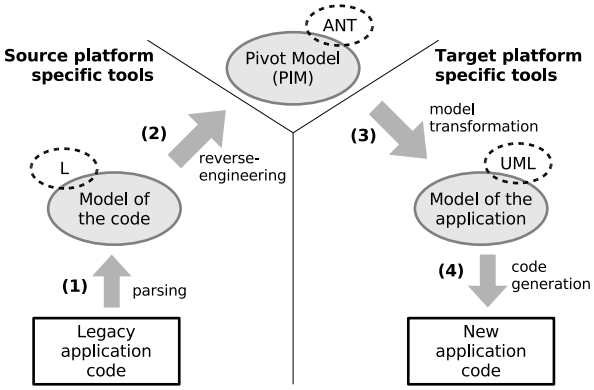


Figure 1: Model-driven migration priciple

the new application and to leverage the latest technologies in terms of graphical user interfaces, distribution and mobility.

In the following, section 2.1.1 first presents the general process developed by Sodifrance for model-driven migration, section 2.1.2 discusses the automation of the process and section 2.1.3 details how this process is adapted in practice along the phases of a migration project.

2.1.1 Migration general process

Figure 1 presents the general process developed by Sodifrance for model-driven migration. This process is mainly divided in four steps.

The first step is the parsing of the code of the legacy application, to build a complete model of the code of the application. This step can be divided into two stages: first a parser builds an abstract syntax tree from the code and, then this syntax tree is processed by a transformation to build an actual model that conforms to the meta-model of the legacy language. During the second stage, all the symbols such as types, variables or function calls are resolved and properly bound to the appropriate model elements. This is a necessary step to allow for a efficient analysis of the legacy system. The meta-model denoted L on figure 1 corresponds to the meta-model of the legacy application implementation language.

The second step is a reverse-engineering from the code model to a platform independent model. The role of this step is to abstract high-level views from the model of the code. This step is implemented by model transformations from the legacy language meta-model (L) to a pivot meta-model. The pivot meta-model used by Sodifrance is a platform independent meta-model called ANT which contains packages to represent:

- Static data structures (close to the UML class diagram).
- Actions and algorithms (it includes an imperative action language).
- Graphical user interfaces and widgets.
- Application navigation.

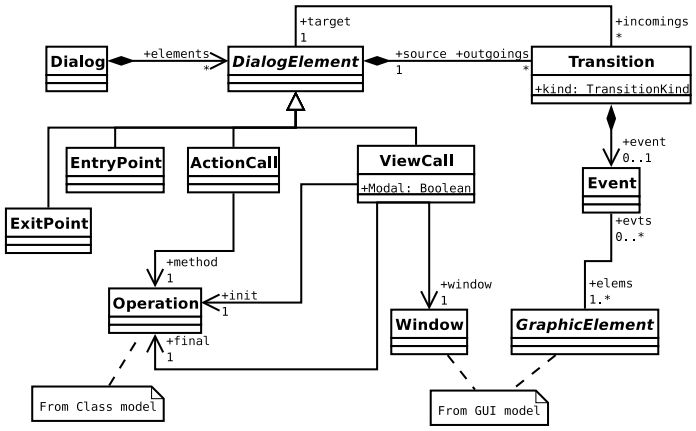


Figure 2: Excerpt of the ANT navigation meta-model

The navigation is the most high level view of the ANT meta-model. Figure 2 shows an excerpt of this meta-model. It connects dialog elements which correspond to GUI forms, transitions between forms and their GUI events with operations in the class model.

All ANT views have to be created through model transformations from the model of the code of the legacy application. In order to be able to create high-level views, such as a model of the graphical user interface of the legacy application, the model transformations have to rely on a knowledge of the libraries of the legacy platform and on coding conventions (or code patterns introduced by tools) that were used during the development of the legacy application. This is the reason why, even if the legacy platforms for several migration projects are similar, the legacy code must be carefully studied in order to properly adapt the migration tools to every single project.

The third step is the transformation of the ANT model into a platform specific model of the application. This step is implemented using model transformations from the ANT meta-model to the UML meta-model. These transformations are design transformations which refine the platform independant views of the pivot model to fit the target platform. Again at this stage, it is important to adapt the transformation to meet the requirements of every customer. This issue is discussed with more details and illustrated on a specific project in section 3.3.

The last step is the generation of the code of the new application from the platform specific model. To implement this step, Sodifrance uses template-based text generation tools in order to be able to easily customize code generation acording to the customers requirements. The specific tools used by Sodifrance for the implementation of model-transformations and code generation are presented section 2.2.

2.1.2 Automation in the migration process

To reduce the cost of migration the goal is to achieve an optimum automation in the migration process. However, this should not impact the quality in terms of

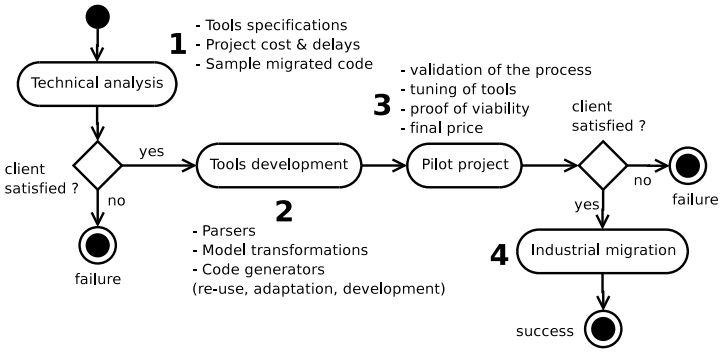


Figure 3: Model-driven migration project phases

design, performances or maintainability of the resulting application. Since the legacy application is fully-executable and the target platform is usually powerful enough, one could argue that the migration should be completely automated. It is theoretically possible: it would be the equivalent of writing a compiler for the legacy language that targets the new platform.

However, as stated in the previous section, migration, and especially in the context of modernization, is more than just creating an executable version of the application on top of the new platform. The goal is to design the application for the new platform in order to make it more efficient, more reliable, easier to maintain or easier to extend than the legacy application. In practice this means that the new code should respect the coding standards and best practices of the target platform languages, it should take into account the specific requirements related to the software development process used by the customer company, there should be models for the new application, etc.

In the migration process implemented by Sodifrance the first two steps (as presented on figure 1) are usually completely automated, i.e. all the information from the legacy system is represented in the pivot model. This is to concentrate the manual effort on the transformation from the pivot model to the new application and avoid having to deal manually with the legacy code as a whole. If some elements of the legacy code cannot fit properly in the pivot model, these elements are captured as notes or tags and presented to the developer when the corresponding parts of the application are transformed or generated.

To maximize the efficiency of the migration process, the tasks that are left to the developer have to be clearly identified and the developer should be provided with all the information he or she needs. This is taken into account in the design of the transformations and code generators. For example in the case of a Java code generator, *TODO* directives can be generated for every piece of code that requires manual inspection, re-factoring or completion. This *TODO* directive can contain the kind of work that has to be done and references to the model elements that are relevant to it. The *TODO* directives are summarized into a task list which gives the developer a clear view of what has to be done.

2.1.3 Migration project phases

Prior to the actual migration and implementation of the new application, the design, the implementation and the validation of a project specific migration process must be completed. This includes the parsing of legacy languages, reverse engineering transformations, high-level design of the new application and mappings between the structures of the legacy application and the concepts of the target platform. All these tasks require some effort due to their complexity and their overall influence on the migration project. In the project structure used by Sodifrance, as represented on figure 3, there are three project phases before the actual migration can start.

The first phase represented on figure 3 is a technical analysis. Its objective is to study the legacy platform, define the target platform and specify the tools that are needed by the migration process. This phase is crucial for the migration project. It is used to estimate the effort that would be required for the development of the tools and the total effort that would be required for the migration. At the end of the technical study a total contractual price is proposed to the customer. During the technical study a small component of the legacy application is usually migrated using generic tools and manually completed to match the code that would be produced using the final tools. This serves as a test for the tool specifications and as a demonstration of the resulting code the customer can expect. If both the price proposed by Sodifrance and the quality of the migrated code are satisfactory to the customer, the project can carry on.

The second phase represented on figure 3 is a tool development phase. The objective is to develop all the tools that have been specified for the migration process. Most of the time the tools do not have to be developed from scratch but are rather re-used or adapted from previous projects. However, most of the time even if the language is the same, the language version and the coding style might be different and require some adaptation.

The third phase represented on figure 3 is a pilot project. The objective of the pilot project is to validate and fine tune the migration process and the tools it uses. It also serves as a demonstration of the viability of the process and allows measuring its efficiency precisely. During this phase, a component of the legacy application is used as a benchmark for the migration process. This component has to be chosen to be as representative as possible of the components of legacy application. In practice the development of the pilot project is truly a testing and debugging phase for the migration tools. For this reason it is usually a lot longer than the migration of a comparable component once the migration process is fully-functional. At the end of the pilot project, the customer is provided with a final price for the project and has a sample of how the new application would look like.

Projects seldom have to stop after the pilot project: the actual migration usually starts shortly afterwards. The preparation of a model-driven migration process can be quite long (the three phases described previously usually require around 6 months to complete but can last up to a year on specific projects such as the one described in section 3.3), but once the process is up and running, the migration rate can be far more rapid than with any competing techniques. This is discussed in section 3.4, but before that, the next section presents the model-driven engineering tools used by Sodifrance to practically implement model-driven migration.

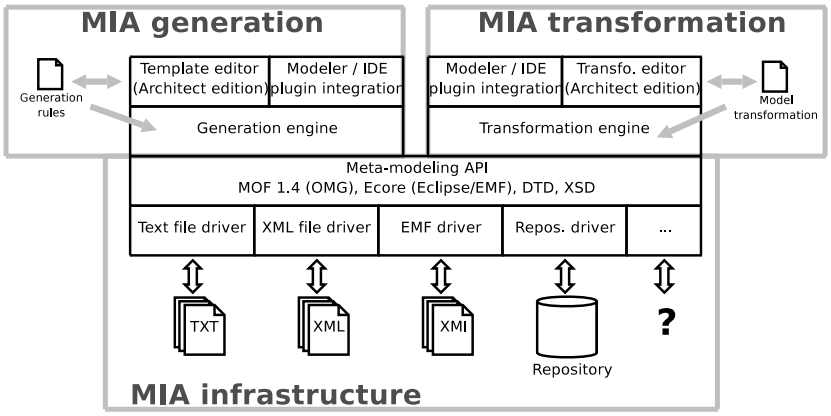


Figure 4: Model-In-Action tool suite architecture

2.2 Model-In-Action (MIA) tool suite

Implementing the migration process presented in the previous section requires advanced, scalable and reliable tools for model transformation and code generation. For both the needs of migration project and development projects, Sodifrance has developed Model-In-Action (MIA) [23], a suite of model-driven engineering tools. This section gives a quick overview of these tools.

Figure 4 presents a simplified architecture diagram for the MIA tools. One of the essential requirement for a company like Sodifrance is to be able to adapt to any specific modeling technology used by their clients. In the design of MIA this has been taken into account by creating a generic modeling platform that can connect through various drivers to existing repositories and modelers. On top of this generic modeling layer the suite is composed of two main products: *MIA-Transformation* for model-to-model transformation and *MIA-Generation* for code generation. Each of these tools is divided in three types of components:

- Core engines for model transformations and code generation. These components are on top of the meta-modeling API and do not have any user interface. They are responsible for the execution of model transformations and code generators.
- Development environments for model transformations and code generators (MIA Architect environments). These environments are used by software architects to design and implement the model transformations and code generators required by MDE projects.
- User environments for model transformation and code generators (MIA developer environments). There are not only standalone versions of these tools but also plug-in versions that integrate directly in the IDEs and modelers of the software developers.

MIA-Transformation is a rule-based model-to-model transformation engine. A model transformation is defined by a set of rules defined between some input meta-models and some output meta-models. Each rule is composed of three elements:

- A context: it corresponds to the set of declared variables and parameters.
- A query: it is an expression that calculates the set of model elements to be processed by the rule.
- An action: it can be a creation, a modification or a deletion of model elements and is performed for each model element returned by the query.

When using MIA-Transformation, alternative languages may be used for expressing transformation rules. MIA-Transformation includes both a fully declarative language (close to the declarative form of QVT) and an imperative language. The two languages can even be mixed in a single transformation rule: the query can be written using the declarative language and the action implemented imperatively. In addition, as rule based transformation has some limitations, it is possible to define transformation services in Java and use them in transformation rules.

MIA-Generation is a template based model-to-text transformation engine. The idea of MIA-Generation is to attach text generation scripts directly in meta-models in order to define how each model element should be generated. There are two kinds of scripts:

- Templates that textually describe the piece of code to be generated.
- Macros that allow more complex operations such as string handling or model navigation.

The macros are defined directly in Java and can be called from the template. The fact that the generation scripts are directly attached to the meta-model makes MIA text generators easy to understand, adapt and maintain. In addition, the generation engine can keep track of the execution of each generation script and the text it has produced. This provides the developer with all the information required to tune or fix a code generator.

2.3 Migration of a large-scale banking application

This section reports on how the migration process described in section 2.1 is applied in the context of a large-scale banking application. The migration of this application is part of the modernization of the information systems of a French bank[1]. The objective of the project was to migrate a mainframe system made of around a million lines of code to J2EE in order to ease the maintenance and future evolutions of the system. The overall system is composed of:

- 42 applications (for a total of 800 forms and 7500 events)
- 99 prints and exports using Cristal Report
- 990 server services
- 20 batch processes

[1] For confidentiality reasons, and for the protection of Sodifrance customers, this chapter does not provide specific details on the migrated application

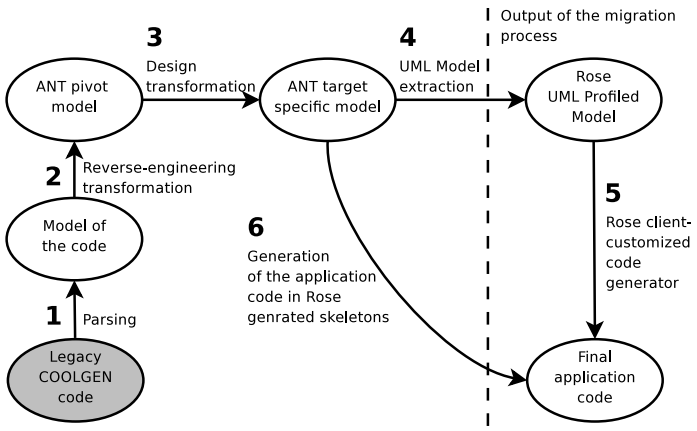


Figure 5: Banking application migration process

Sodifrance (and their model-driven migration approach) was chosen by the bank for the migration of this system not only because of the quality assurance provided by the use of automation but also for pricing reasons. After an initial study of the project by Sodifrance and several competing companies, the price proposed by Sodifrance was significantly lower than the price of any brute-force re-development strategy (out-sourced or not). In the following, section 2.3.1 presents the customer's requirements and the migration process that has been developed, section 2.3.2 details the project schedule and section 2.3.3 discusses the problem of the validation of the migrated application.

2.3.1 Specific requirements and migration process

For the modernization of its information system both the servers and the client applications of the bank had to be migrated. The whole legacy application had been developed using the COOLGEN IDE. COOLGEN provides an intermediate programming language and produces executable application by compiling this language to a combination of C code and COBOL code. For the modernization of the system, the servers had to be migrated to plain COBOL because the code generated by COOLGEN was difficult to maintain and had some performances issues. The 42 client applications had to be migrated from COOLGEN to J2EE web applications. The applications and the servers would communicate through a COBOL/Java middleware. The following focuses on the migration of the 42 client applications from COOLGEN to J2EE.

An important requirement of the customer for this project was the strict respect of its internal development standards. All the new applications developed by this bank are generated from Rational Rose UML models. All the models conform to a UML profile developed by the bank itself and specific code generators are used. As a result of the migration process the bank expected to be able to round-trip between models and code using its usual profiles, tools and code generators. The model-driven migration process had to be adapted to take this specific requirement into account.

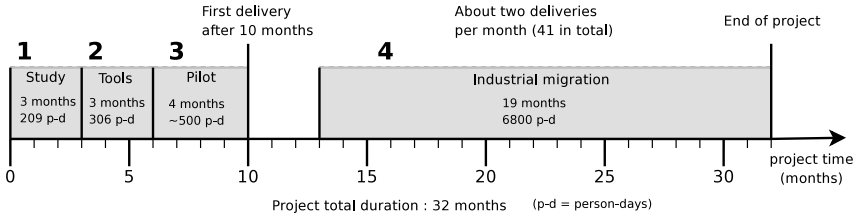


Figure 6: Banking application migration time schedule and cost breakdown

Figure 5 presents the migration process that is applied to each of the 42 applications of the legacy system. Steps 1 and 2, which correspond to the parsing and reverse-engineering of the application, are similar to the two first phases of the general process presented in section 2.1. These two phases produce an ANT model of the legacy application which includes all the information contained in the code of the applications (windows, widgets, statements, expressions). Step 3 (also quite similar to the third step of the general process) does the mapping between the source architectural concepts and the target ones to produce a complete platform specific model of the target application.

Steps 4, 5, and 6 of the process presented figure 5 are specific to the banking system migration and designed to produce customer-specific synchronized UML models and source code for the target application. Firstly, step 4 is a model transformation that extracts a UML-profiled model from the ANT application model. The elements of the target application, such as statements, that do not fit in the UML-profiled model are ignored. Then, step 5 consists in using the regular code generator used in all the development projects of the bank to produce code skeletons from the UML model. In regular projects these skeletons have to be filled manually but here the role of step 6 is to automatically generate the final application code in the code skeletons. The manual phase of the migration can then be carried out: the model transformations and the code generators have left notes in the UML model and comments in the code wherever a manual migration task has to be accomplished.

2.3.2 Project time schedule and cost breakdown

This section details the organization and cost breakdown for the banking system migration. The overall project required a total of 9315 days of work including 7815 for the migration of the 42 client applications. As discussed in section 2.1 any model-driven migration project has several mandatory initialization phases to design a specific migration process and adapt or develop the required tools. Figure 6 presents the scheduling and the cost (in terms of days of work) for each phase of the banking system migration project.

The first preliminary phase of the project is the technical study. In the case of the banking system it took 3 months and required a total of 209 days of work (which represents about 2.5% of the project effort). Then, the tool development phase and the pilot project took 7 months to complete and required an approximate effort of 800 days of work (around 10% of the project effort). For the pilot project, a representative client application has been chosen among the 42 application that had to be migrated. The delivery of the pilot project

occurred 10 months after the beginning of the project and after about 12% of the project effort has been spent.

The important investment and delay before the first delivery is specific to model-driven migration. Moreover, because the preliminary tasks are difficult to parallelize and because the developers need to have a global view of the project to accomplish these tasks, using a large team of developers cannot really help reducing the duration of preliminary work. The developer team for these tasks have to be small (3 to 8 developers for most Sodifrance projects) and should include experts of the source platform, experts of the target platform and model transformation experts.

On the banking application the industrial migration of the 41 remaining application started 3 months after the end of the pilot project. This phase required a total of 19 months to complete. During this industrial phase of the project the migration is performed in parallel by three independent teams of around 15 developers each. Sodifrance migrates three applications at a time, and, during the 19 month period of the industrial migration an average of 2 deliveries are made per month. Contrary to the project preliminary phases that require a small developer team, the industrial migration duration can easily be shortened by increasing the number of developers.

2.3.3 Validation and quality assurance

Even with the use of automation, since there is still a significant part of the work done by hand, the migrated application has to be carefully validated in order to check its correctness, performance and integration in its new environment. In practice this is achieved thanks to a strict non-regression testing process. This test process is costly for the customers because they have to provide test cases together with the legacy applications and they have to perform acceptance testing [2]. It is also costly for Sodifrance who perform unit testing for the new application and uses the test cases provided with the legacy application to do regression testing. In the case of the banking application the total testing cost is around 3500 days of work (around 1000 days for unit testing and 2500 for regression testing). This represents 45% of the total project cost.

2.4 Discussion

This section compares *model-driven migration* with brute-force *re-development* migration strategies. The most significant difference between the two approches is the significative *preliminary tasks* required by model-driven techniques. This section especially discusses the influence of these preliminary tasks on the schedule and cost breakdown of migration projects and shows that for projects over a critical size, the model-driven approach is more profitable than re-development.

Complete re-development has some advantages over automated migration. Firstly the development process is similar to the development of any application except that it has a fixed and non-ambiguous specification. This allows using efficient software engineering techniques which is reassuring and unsurprising to the customer. Secondly, the target application can easily be re-designed, re-factored and adapted to the new platform. Thirdly, evolutions to the legacy

[2] The numbers provided in this section do not include this cost. Only the cost for Sodifrance is taken into account

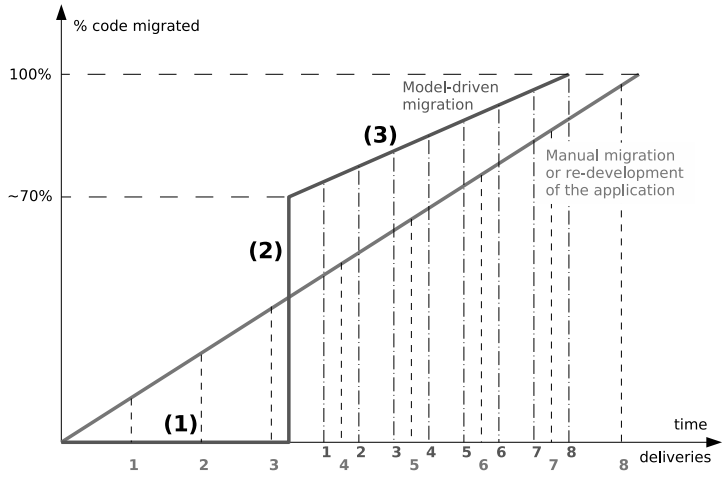


Figure 7: Migrated code percentage in function of time

application can be taken into account in the design of the new application. All these advantages, most of the time combined with out-sourcing to cut workforce cost, allow full re-development to be a common option for modernization.

In this context, thanks to model-driven migration, Sodifrance has managed to provide a comparable quality of service at lower prices to its customers on a number of modernization projects.

2.4.1 Migration time schedule

Figure 7 compares re-developpment and model-driven migration with respect to the percentage of code migrated over time. For re-development the model we use is linear: the components of the legacy application are re-developed one by one. For model-driven migration the process is a little different: during the first stage of the project (1) the objective is to develop the tools that will be used to partially automate the migration. During this first stage no code of the new application is produced at all but once the tools are fully functional they typically allow generating about 70 percent of the final application code (2). The actual migration can then begin (3), each component of the legacy application is manually completed and delivered to the customer.

The most important difference between the two approaches is the first phase of the model-based process which is an investment in specific tools that will make the migration faster. One of the drawback of model-driven migration is that for an initial period of time, no final code is produced and thus nothing can be delivered to the customer. In the example of figure 7 the legacy application has been divided in 8 components. Using a re-development strategy, the first component can be delivered to the customer just after the beginning of the project. On one the hand the first component is delivered after quite a long period of time: using model-driven migration, when the first component is delivered, 3 components have already been finished with re-development. But on the other hand, using model-driven migration, once the production of the new

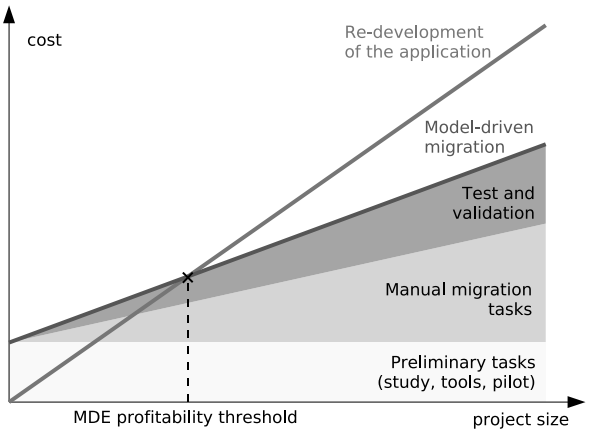


Figure 8: Project cost in function of it size

application has started, the delivery rate can be faster than for re-development. Eventually, the delivery of components developed using the model-driven approach can catch up with the delivery of re-developed components (this is the case for components 7 and 8 on the figure).

In the case of the banking application described previously, the preliminary tasks of the migration project required 10 months which represents about a third of the total project duration. From an economical point of view, more than 10% of the total migration cost was spent on these preliminary tasks. The next sub-section discusses the profitability of this investment.

2.4.2 Migration cost repartition

To be profitable, the model-driven migration process must be applied on legacy applications that have a sufficient size. Indeed, the effort that has to be invested for developing migration tools mostly depends on the complexity of the input and output platforms but not on the volume of code that has to be migrated. Figure 8 presents an estimation of the cost of a migration project using both model-driven migration and re-development.

In the case of re-development the cost is directly proportional to the size of the legacy application. In the case of model-base migration, there is a fixed initial cost related to the development of tools which is complemented by a linear cost corresponding to manual migration efforts. The gradient of the function corresponding to model-driven migration is lower than the gradient for re-development because using migration tools reduces the manual effort that has to be provided.

A general profitability threshold for model-driven migration cannot be estimated accuratly because it really depends on the ratio of tools that have to be developed for each project. In practice, the experience of Sodifrance on model-driven migration shows that the profitability threshold for MDE in the context of migration is quite low. Even for projects that require about 1000 days of work, the initial overhead of developing tools pays off. On the migration of the large banking application described previously, Sodifrance estimated that the

cost of re-development whould have been around twice the price of model-driven migration.

2.4.3 Benefits and limitation of model-driven migration

The primary advantage of model-driven migration is to partly automate the migration process. As discussed in the previous sections this allows Sodifrance to significantly lower the prices and duration of migration projects. This is the reason why Sodifrance is often chosen over competing companies that propose full re-development.

The second advantage is to allow for reuse between migration project. This is another element that allows cutting the cost of migration. All the transformations and tools that have been developed for a migration projects can be adapted to future project that have similar input or output platforms.

The first limitation of model-driven migration is a commercial limitation related to the cost and time consumed by preliminary tasks. In the project presented in section 3.3 the first delivery of migrated code occured after 10 months. This is a commercial issue because after the begining of the project the customer has to wait for a long time without seeing the progression of the project. To mitigate this issue, possible solution is to works in close collaboration with the IT department of the customer and, if possible, to include members of the customer company in the development team.

The second limitation is related to the cost of testing. This is not specific to model-driven migration but is a general problem in software migration. For the banking application discussed previously, testing represents 45% of the total migration cost. This cost does not include the cost of the production of regression tests and acceptance testing which are the responsability of the customer [3]. One of the reasons of the important cost of testing tasks is that, for most migration project, they are mostly handled manually. In the same way model-based techniques has been applied to smartly automate repetitives migration tasks, Sodifrance is now studying model-based regression testing using meta-modeling languages such as Kermeta [15] to reduce the cost of testing.

2.5 Related works

Software modernizaton has been identified by the OMG as an important application field for model-driven architecture. The Architecture-Driven Modernization (ADM) is an OMG task force dedicated to this topic [18] that aims at building standard metamodels and tools for software modernization. Reus et al. in [20] propose a MDA process for software migration that is quite similar to ours. They parse the text of the original system and build a model of the abstract syntax tree. This model is then tranformed into a pivot language that can be translated into UML. A prototype automates parts of this process using ArcStyler [17]. Bordbar et al. [3] propose a model-based approach for maintenance of data-centric systems. Their MDA approach improves the evolution and maintenance of databases in applications developed with java and modelled with

[3] The production of the tests is a cost that has to be taken into account by the client. However, this cost is usually far lower than the cost of providing the complete specifications required for full re-development.

UML. In [27], Zou transforms legacy code towards object-oriented languages. Parts of this process are implemented with automatic program transformations.

Another type of works related to the study presented in this chapter concerns feedbacks from industrial projects that have applied model-driven approaches. In the last two editions of the MoDELS conference, two studies gave such feedback. In [1], Baker et al report on the significant improvements in productivity and reliability gained with MDE techniques and also present the remaining issues to profit more from those approaches. In [24], Staron presents a study on the requirements for the adoption of MDE in software industries. The chapter reports on the observations of two companies that tried to MDE in their development process. In [10] M1_Global_solution compares MDE and off-shore development. They conclude that MDE tool increased developer productivity by over 50 percent and advocate a combination of MDE for automatic production of a large part of the system and off-shore development for the parts that need to be manually developed.

2.6 Conclusion

This chapter has precisely presented the model-driven migration and modernization process developed by the Sodifrance company. We have detailed the process and the tools that automate this process. The chapter has also dicussed the benefits introduced by MDE in terms of reuse and automation, and also the issues that are introduced to fully benefit from reusable transformations and generators. Finally we have showed that, even if the process is not fully automated and requires manual adaptation from one project to the other as well as manual implementation of some parts of the final application, it is still viable compared to manual re-development.

Even if model-driven engineering is already economically profitable for migration, there are still some important challenges that need to be tackled. A major issue in terms of human effort is testing. Today, regression test is used to validate the migration. However, the production of efficient regression test cases is manual, ad-hoc and difficult to evaluate. Future work consists in adding, in the reverse-engineering phase, a step to reverse a model for high-level control flow in the application in order to eveluate test coverage at use-case level. Moreover, unit and integration test for the migrated code is also very expensive. A possible solution here could consist in generating test objectives when generating the code. The next two chapters discuss these issues in detail.

3 Model-based test generation

3.1 Introduction

As development techniques, paradigms and platforms evolve far more quickly than domain applications, software modernization is a constant challenge to software engineers. Modernization projects usually consist of the redesign and refactoring of a legacy application and its migration to a novel platform. Important parts of a modernization process can be automated, e.g. using model-driven engineering techniques for reverse engineering, model to model and model to code transformations. However, because migration is more than a simple trans-

lation from a platform to another, it is impossible to reach complete automation. Thus, it is necessary to thoroughly validate the produced system to ensure that it behaves as the original system. As observed in several works, the cost of validation in migration projects can reach between 50% and 70% of the total cost [22, 7].

This validation can be performed using regression test suites. This approach consists first in selecting an efficient test suite for the legacy system. This test suite must be composed of all meaningful usage scenarios for the legacy system and should cover the requirements of the system. This suite must also associate expected results for each input data or scenario. Once such a test suite is available it provides a relevant representation of the behaviour of the original system and can be used to test the new version of the system.

A crucial issue for the development of such an approach is to be able to select an efficient and meaningful test suite for the original system. This selection is difficult because there are usually neither models nor up-to-date specifications for the system to migrate. The business scenarios are only known by the domain experts who use the system intensively. However, these experts cannot evaluate the efficiency of their scenarios as regression test cases. Thus it is necessary to assist the generation of test scenarios with techniques for coverage measurement.

In this chapter we propose to reverse abstract models from the code of the original application. These models can be used for test selection and test improvement. The reversed models provide information on the control and data flow of the legacy code. This information is used for the generation and qualification of test scenarios and to assist the improvement of test suite to select efficient test cases for regression. Based on these models, we define several criteria to quantify the coverage of test scenarios with respect to the legacy system. This work is developed in collaboration with the Sodifrance company who is specialized in the modernization of large information systems. The proposed techniques have been experiment on an industril migration project.

The chapter is organized a follows. Section 2.1 presents the software migration problem and discusses related testing issues. It then details the model-based migration approach used by Sodifrance. Section 3.2 presents the models and test criteria we propose to assist test selection. Section 3.3 describe how the proposed criteria were experimented in the context the migration of a large banking application. Section 3.4 discusses how the proposed technique contributes to the global migration testing process. Section 3.5 details related works and finally, section 3.6 concludes the chapter and gives future work directions.

3.2 A framework for selecting reference tests

The idea proposed in this chapter is to use reverse-engineered models of the legacy application in order to extract the information reaquired by the tester. In the specific case of the model-based migration process presented in the previous section the models that are generated for migration purposes can be reused. This avoids the cost of creating specific testing models direcly from the legacy code. Instead the testing models are extracted from the migration model.

In this chapter the class of application we consider are form-based application such as banking or assurance applications. These application represents the vast majority of migration projects carried by Sodifrance. The specific models and criteria we define are specific to this type of application but we believe the same

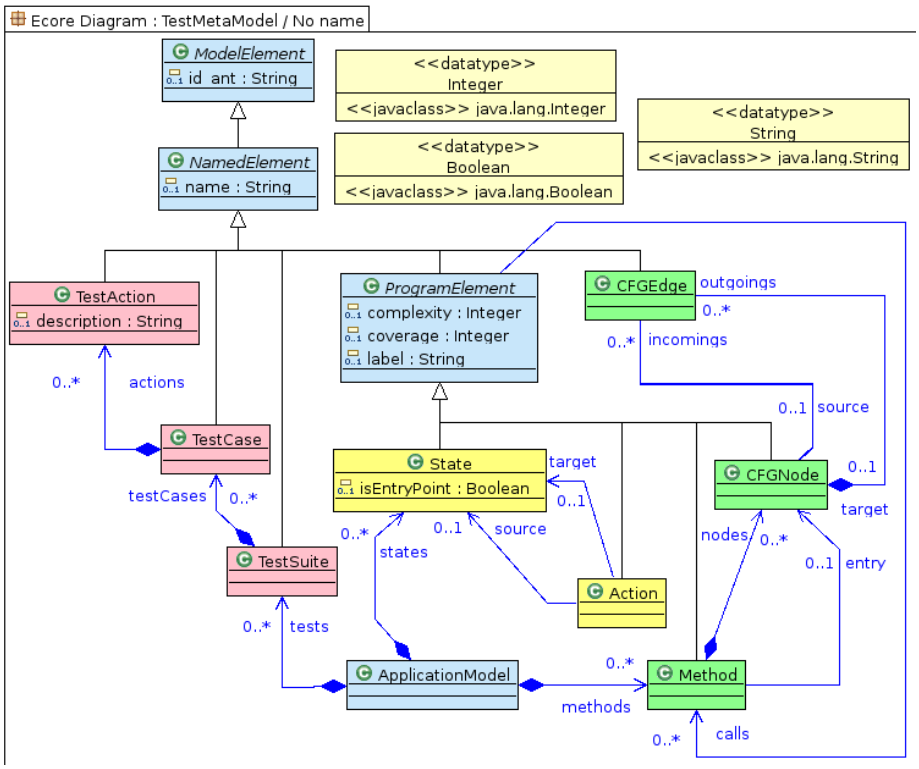


Figure 9: The ISTeCQ meta-models

kind of technique could be applied to other domains such as service oriented applications.

This section is organized a follows. We first propose a meta-model to captures the information required by the tester for selecting or improving test suites. Then, based on this meta-model, we detail two test criteria for systematicaly covering the behaviors of the application under test.

3.2.1 The ISTeCQ meta-model

Figure 9 presents the ISTeCQ (Improving Software Testing Coverage and Quality) meta-model. This meta-model is designed to provide the tester with a synthetic view of the application under test. This meta-model capture the navigation graph of the application under test and links each forms and actions with the control-flow of the undelying code. From this information it allows defining test scenarios in terms of actions in the navigation graph.

The root element for any ISTeCQ model is an instance of the class *ApplicationModel*. This object contains the forms of the navigation model of the application, the methods and the test suites of the application. The classes *State* : *Exp* and *Action* are used to represent the navigation in the application. Basically a state correspond to a form (or a screen) in the application and an action to a transition between two forms. Actions and forms are linked with the methods which

are executed when an action is performed or when a form is displayed. In ISTeCQ models, methods are represents by the class *Method*, they have a name and a corresponding control-flow graph. The classes *CFGNode* and *CFGEdge* are used to represent respectively the nodes and edges of control-flow graphs.

The classes *TestSuite*, *TestCase* and *TestAction* are used to represent test scenarios. At the level of the ISTeCQ model a test scenario is a sequence of actions in the navigation model of the application. Both actions in the navigation model and instances of class *TestAction* are identified by names (classes *Action* and *TestAction* inherit from abstract class *NamedElement*). A test case is valid only if the sequence of test actions it contains is a valid sequence of actions in the navigation model of the application. In the version of the ISTeCQ meta-model presented figure 9 the test actions just include the names of the actions to perform but these actions actually correspond to the rows of the test scenario presented in figure **??**.

3.2.2 Coverage of the navigation graph

The first criterion we propose to select test scenario is to ensure the coverage of the navigation graph of the legacy application. This criterion requires all the forms of the legacy application to be covered at least once and each transition between forms (action) to executed at least once. As the test scenarios are expressed as sequences of actions in the navigation model the criterion is easy to check from an ISTeCQ model.

The technique proposed in this chapter was designed in the context of the migration of a large banking application. The characteristics of the application and results of applying the technique on it are presented section 3.3 but in the following we use fragments of diagrams extracted from the study in order to discuss the criteria.

Figure 10 presents part of the navigation model of an application and the test coverage information. The coverage of the tests is presented with a color gradient from red to green with respect to the number of times a node or an edge is covered. The diagram shows that 1 form and 9 actions are not covered at all by the tests. Using this information the tester can improve the set of tests in order to achieve complete coverage of the navigation graph.

To be able to achieve a good coverage of the code of the legacy application this criterion has to be satisfied but it is intuitively not sufficient. In practice the complexity of an application is not homogeneously spread across its navigation models: some forms and actions are far more complex than others. The criterion defined previously only requires each element to be covered once where as complex elements should probably be covered more times than simple ones. The next section defines a criterion which formalizes this intuition.

3.2.3 Taking complexity into account

In the ISTeCQ model, the states and action are linked to the method they execute. The idea we propose is to measure the complexity of the code executed by each action in order to estimate a minimal number of times it has to be covered by the tests. As a measure of complexity we chose to use the cyclomatic complexity [11]. The cyclomatic complexity of a piece of code corresponds to the minimal number of tests required in order to be able to achieve full coverage

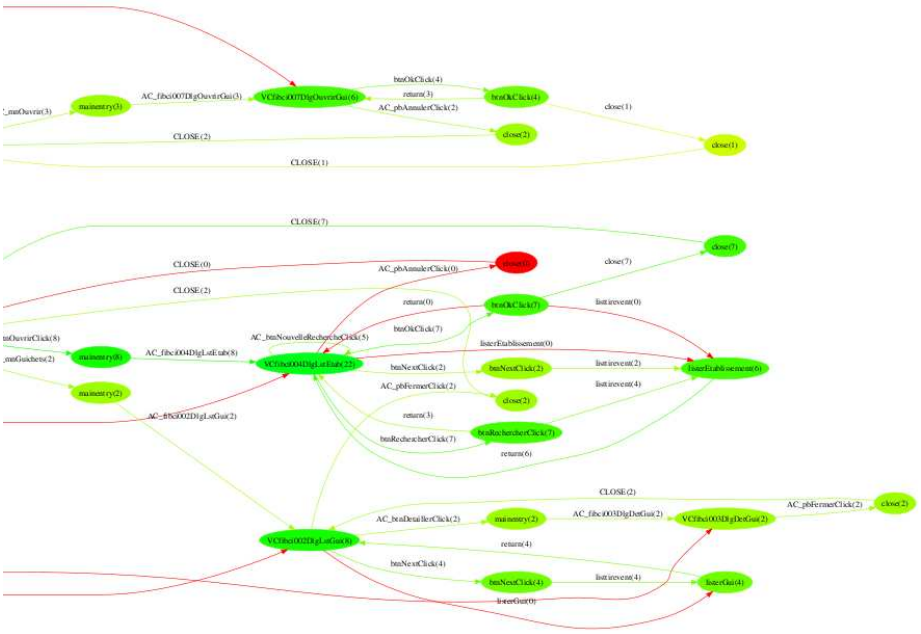


Figure 10: Test coverage of the navigation model

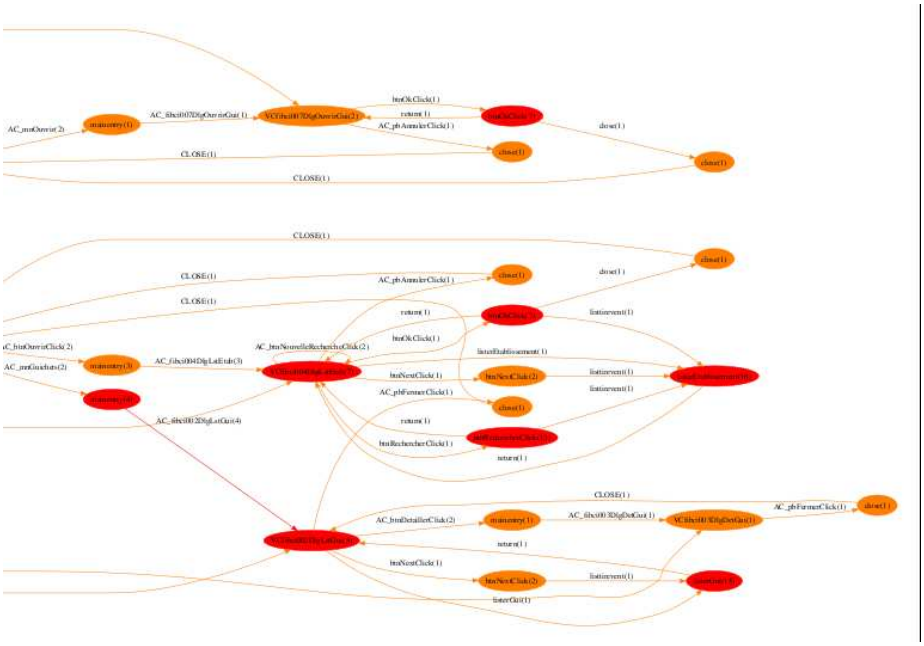


Figure 11: Navigation model with complexity information

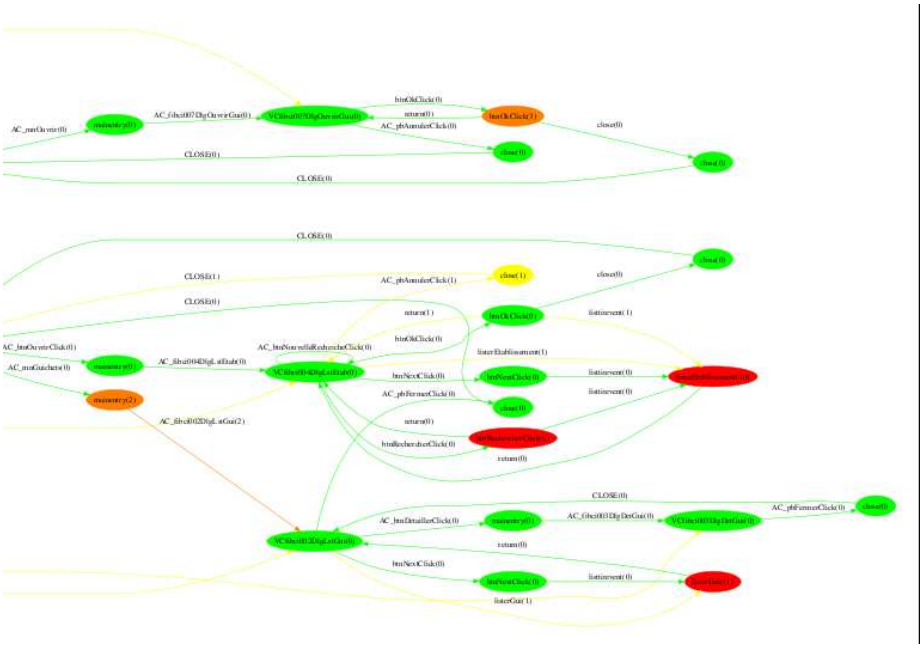


Figure 12: Test coverage of the navigation model with complexity information

of the control-flow graph branches. We can thus estimate the minimal number of tests that should cover an action by the cyclomatic complexity of this action.

In practice, all the information required to compute the cyclomatic complexity of actions is available in the ISTeCQ model. The complexity of an action corresponds to the complexity of the method it calls. If the method contains calls to other methods, the control flow graphs are bound in order to compute the total complexity. In our case this can be easily done because the legacy code is COBOL code in which all calls are statically linked and these links are available in the ISTeCQ model. Once we have computed a complexity for each element of the actions, the complexity for states is estimated by the max between the sum of the complexity of the outgoing actions and the complexity of the method it directly calls.

The way the complexity of each state and action is computed ensures that the result corresponds to the minimal number of tests that can ensure complete coverage of the underlying code. Once the complexity of each element of the navigation graph is computed, it can be displayed on the navigation graph diagram in order to provide the tester with a comprehensible map of the complexity of the application. Figure 11 presents this diagram for a piece of the application of the case study. The complexity is represented by a gradient of colors from yellow to red.

Provided the complexity information, the test coverage criterion is now to covers at least a number of time corresponding to the complexity each element of the navigation graph. Figure 12 displays the value $v = complexity - testcoverage$ with a color range between green if $v \leq 0$ and yellow to red for $1 \leq v \leq max(v)$. The green elements corresponds to elements that have been

properly covered by the tests and elements from yellow to red corresponds to elements which require more tests.

It is interesting to compare figures 12 and 10 which were obtained with the same test set. By taking into account the complexity information some states of the application which were covered on figure 10 actually require more tests. On the other hand the state that was not covered on figure 10 is of a quite low complexity and thus not so critical.

To conclude, the last criterion which takes the complexity of the code into account is stronger than the previous one and seems to provide better feedback on the test coverage. Of course, both criteria are only based on models and do not provide any warranty that the actual coverage of the code will fully correspond. The next sections presents the results we obtained using these criteria on an industrial migration project and discuss how the can be used in practice.

3.3 Tools and case study

This section presents how the criteria proposed in the previous section were validated in the context of a large migration project carried by Sodifrance. This section first presents the system under migration, then details the protocol of the cases study, presents the testing tools that were developed around the ISTeCQ meta-model and finally discusses the obtained results.

3.3.1 The application under migration

The project that was used for the evaluation and experiments of the test criteria proposed in this chapter is the migration of a banking application. This migration was part of the modernization of the information systems of a French bank[4]. The objective of the project was to migrate a mainframe system made of around a million lines of code to J2EE in order to ease the maintenance and future evolutions of the system. The overall system is composed of:

- 42 applications (800 forms and 7500 events)
- 99 prints and exports using Cristal Report
- 990 server services
- 20 batch processes

Sodifrance (and their model-driven migration approach) was chosen by the bank for the migration of this system not only because of the quality assurance provided by the use of automation but also for pricing reasons. In order to validate the relevance of the ISTeCQ meta-model and the quality of the test criteria proposed in this chapter we used two applications of this system. Each of these application had to be migrated from COBOL to J2EE. They were chosen to be as representative as possible of the 42 applications of the system:

1. App. 1: 85 forms, 189 actions and 227 methods.
2. App. 2: 50 forms, 100 actions and 116 methods.

The first one is central to the system and has lots of interactions with other applications. The second one is a little smaller and dedicated to a specific task of the system. It is less coupled to the rest of the system but is representative of a number of other applications.

[4]For confidentiality reasons, and for the protection of Sodifrance customers, this chapter does not provide specific details on the migrated application

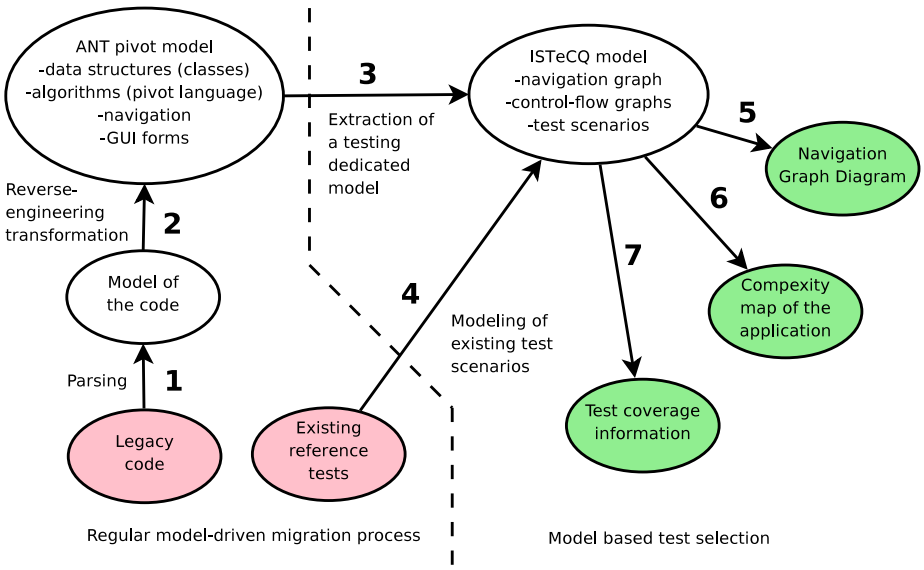


Figure 13: Test selection process

3.3.2 Case study protocol

Figure 13 presents the protocol we used in order to apply the model-based test criteria to the existing test cases. The objective of this case study is 1) to estimate the relevance of the ISTeCQ model for defining system tests and 2) to validate the test adequacy criteria we propose. To do that the idea is to create ISTeCQ models for the two applications of the baking system and to model the test cases that are provided by the customer for these two applications.

The steps 1 and 2 on Figure 13 correspond to the two initial phases of model-driven migration process (see figure 1). The parsing and reverse-engineering transformations are developed (or adapted from a previous project) at the beginning of a migration project. The developers of these transformations usually use several sample components of the application that will be migrated in order to design and validate the reverse-engineering phases. More details about these transformations and how they are developed can be found in [7]. Once these transformations are available they are used to automatically produce an ANT pivot model for each component of the application to migrate.

The ANT pivot model captures all the information from the legacy code required for migration. The role of step 3 is to extract from the pivot model a more restricted model that contains the information for test selection. This model includes the navigation graph of the application, the control-flow graphs of the actions and the ability to define test scenarios. The navigation is directly extracted from the ANT pivot models, the control flow graphs are computed from the pivot action language of ANT. Finally, if existing test scenarios are available they can be added to the model (step 4 on figure 13).

At steps 5, 6 and 7 the ISTeCQ (Improving Software Testing Coverage and Quality) model is used to generate diagrams of the application and perform

test coverage analysis. The aim of the technique is to provide an information sufficiently explicit for a tester who is not a specialist of the legacy application. Some examples of these graphs are displayed in Figure 10, Figure 11 and Figure 12 in section 3.2.

3.3.3 Tools

In order to implement the protocol described in figure 13 we developped tools around the ISTeCQ meta-model:

- An ANT to ISTeCQ transformation which extracts an ISTeCQ model from the migration ANT models.
- An ISTeCQ simulator and test editor which allows modeling and editing test scenarios in an ISTeCQ model.
- Complexity analysis algorithms which compute and export the complexity of the elements of an ISTeCQ model.
- Coverage analysis algorithms which compute and export the coverage of the tests scenario of an ISTeCQ model.

The only tool that is specific to Sodifrance projects is the ANT to ISTeCQ model transformation (step 3 in Figure 13) because it is based on the ANT meta-model. This transformation was developed using the Sodifrance Model-In-Action (MIA) tool suite. The transformation translates the MIA navigation model to the ISTeCQ navigation model and computes the control-flow graphs of the actions of the ANT model. To adapt the testing technique proposed in this chapter a similar transformation would have to be developed in order to extract and ISTeCQ model from the available models of the application to be migrated. The information should be available as it required by the migration itself.

The test editor and simulator were developed in order to ease both the definition of new test scenario and the modeling of existing ones. Figure 14 presents a screenshot of this test editor. It allows defining test suites (top left list in Figure 14) which are composed of test cases (to right list in Figure 14). As defined by the ISTeCQ meta-model, a test scenario is a sequence of actions of the application under tests. To be valid, a test case must be a valid sequence of actions with respect to the navigation model. In other words, a test scenario is valid only if it is a path in the navigation graph of the ISTeCQ model. The objective of the test editor is to assist the tester in the modeling of valid test scenario. The list of actions at the bottom-left in Figure 14 is the sequence of actions of the selected test case. The green symbol before each action means that this action is valid in the navigation model. Any invalid action would be marked in red in the editor. The set of actions at the bottom-right in Figure 14 displays all the application actions that are valid after the last action of the selected test cases. The tester can grow the test scenario by picking valid actions in this set. In practice the test editor was implemented in Java using Eclipse Modeling Framework (EMF) [4] code generators. In the case study the test editor was used for modeling the test scenario provided by the customer (step 4 in Figure 13).

The complexity and coverage analysis algorithms are the actual implementation of the test criteria proposed in this chapter. They were implemented using the Kermeta language [15]. Keremta was chosen because it is an object-oriented meta-modeling language which allow weaving semantics and transformations di-

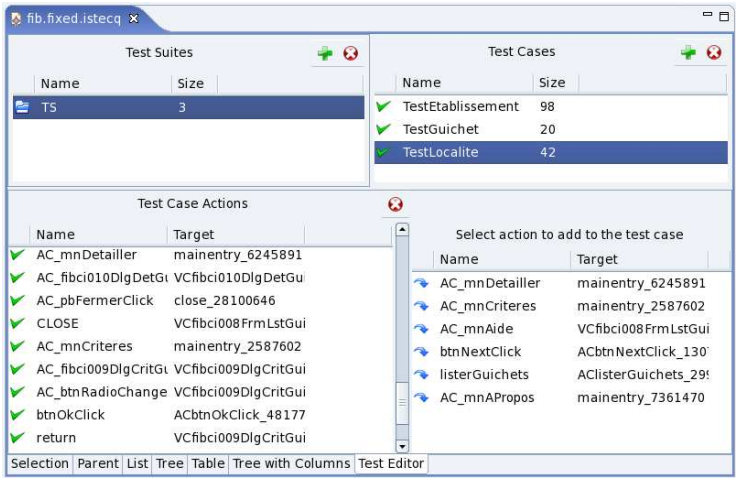


Figure 14: Screenshot of the test editor

rectly into meta-models. In practice the ISTeCQ meta-model is defined using EMF and operations written in Kermeta are weaved into it to support coverage and complexity computing. Some transformations were also added to the meta-model in order to export diagrams for complexity ans coverage. The graphs displayed in Figure 10, Figure 11 and Figure 12 in section 3.2 were obtained with these transformation. The kermeta transformation produces a textual representation of the diagram and GraphViz [8] is used to render the diagrams graphically.

3.3.4 Results

The case study has shown that there is a small distance between the functional tests provided by the customer and the test scenario that can be expressed in the ISTeCQ model. In the case of the banking application, a single developer was able to translate the existing test scenarios for each of the two applications under study in about an hour. In the resulting scenarios, each action in the ISTeCQ scenario corresponds to a line in the natural language test case provided by the customer. This is an important result since it validates the fact that the test expressed in the ISTeCQ model are at the appropriate level of abstraction to eventually replace the functional tests provided by the customer.

The second important result of the case study is the evaluation of the proposed test criteria. The criteria were computed for each of the two proposed criteria using all the functional tests provided by the customer. For both application under study the coverage results were consistent with the code coverage measurement performed in order to qualify the tests. This is an interesting result since it establishes a correlation between the coverage of the model and the actual structural coverage of the application under test.

Concerning the comparison between the simple coverage criterion and the complexity coverage criterion, the results we obtained show that taking complexity into account provides a far more precise diagnosis of what parts of the

application require more tests. The information on the complexity provides a valuable map of the application for a tester who is not a specialist of the application under test. Using this criterion allowed us to clearly exhibit weaknesses in the functional tests provided by the customer.

To conclude, the results we obtained are encouraging both for using the ISTeCQ model as a basis for testing and for the complexity coverage criterion we proposed. In order to further validate the quality of our test adequacy criteria we plan on using the ISTeCQ model for defining new test scenario. This would allow to check that 1) these test scenario are easy to translate into actual test cases and 2) that properly covering the ISTeCQ model allows improving the code coverage of the tests.

3.4 Discussion

The objective of this work is to reduce the cost of testing in migration projects by allowing Sodifrance to better analyze and improve the references tests. Ultimately the objective is to allow testers from Sodifrance to create the reference tests without the help of the customers expertise. This avoid having costly iterations between the client and Sodifrance and allows the reference tests to be sold by Sodifrance together with the migrated application. This section first details a test selection process that takes advantages of the test criteria proposed in this chapter and then discusses it benefits in terms of allowing non-specialist to write tests and in term of reducing the global cost of test generation.

3.4.1 Regression testing process

Figure 15 presents a global test generation process which takes advantage of the test criteria proposed in this chapter. The idea of this process is to process iteratively by using more and more detailed models of the application under test. Initially there might be an exiting test suite but this is not required. The idea is first to use the navigation model of the application and to provide the tester with some feedback on the forms and actions are not covered by the tests. Once the navigation graph is properly covered the test can move on to the next criterion which takes complexity into account. This refined criterion outlines the parts of the navigation model which requires more tests and helps the tester focusing on their testing. The benefit of these first two criteria is that they are defined on high level models and their satisfaction can be measured by model simulation.

Once the models (navigation and complexity) are properly covered the tester can use structural criteria on the code of the application to measure the actual code coverage of the tests. The model based criteria proposed in this chapter do not ensures that the code of the application will be fully covered but what is sure is that if the model-based criteria are not satisfied the code cannot be fully covered. In practice this means that the tester could be required to add a few more tests in order to fully cover the code of the application.

The benefits of the top-down approach represented Figure 15 are two-fold. Firstly, the use of high-level models of the application under test allows for non domain-specialist tester to acquire a good knowledge of the structures of the application. Secondly, the use of appropriate models at each level of the process

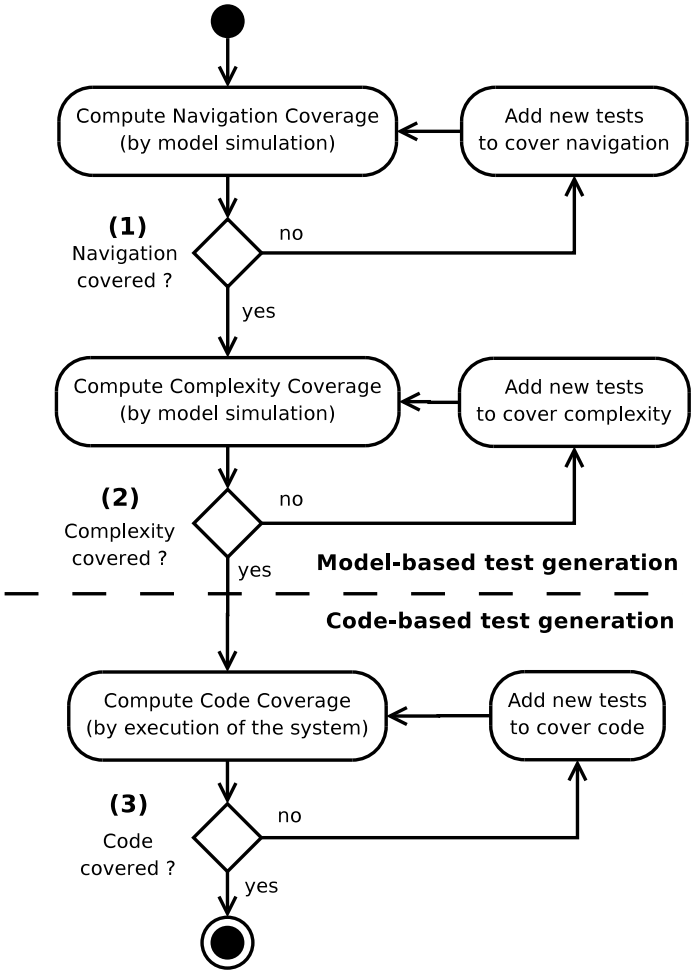


Figure 15: Global test generation process

allows reducing the cost of test generation by cutting the complexity of system under test. The two following section discusses these benefits with more details.

3.4.2 Allowing non-specialists to write tests

In migration projects the only input available is the source code of the legacy application. As the system to migrate are usually large, it is impossible (or at least far too costly) to ask someone who has no prior knowledge of the application to write system tests. This is the reason why Sodifrance has to request functional tests for the legacy application. Unfortunately these tests usually do not exist and the owner have to create them before the migration can start. As discussed in section 2.1 this is a costly process which requires a lot of iteration between domain experts and developers of the migrated system.

In that context, the use of high level models of the application (such as the navigation model and the complexity map) allows to provide the tester with a global and manageable view of the application. The intent of these models is to capture the exact information required for efficiently selecting tests. In particular, the complexity information allows a tester with no prior knowledge of the application to identify the parts of the application which require special testing care. In the case of the banking application, the high level models were sufficient for someone who had never seen the actual application to model the existing test scenarios and identify forms and events that would need more tests.

3.4.3 Reducing the cost of testing

The second benefit of the test selection process we propose is that it allows reducing the global cost of test selection. In the current approach, the only criterion used is code coverage. In practice, specialists of the application use their knowledge of the application to write tests which are evaluated in terms of code coverage by instrumenting the code of the application. Experience shows that application experts alone are not likely to achieve a good code coverage of their application. They usually have to iteratively use the code coverage information to improve their tests. This process is really costly because the application experts are not developers so they have to work together with developers in order to understand the uncovered code and figure out new test scenarios.

Using models for test selection should limit the need for studying the code of the legacy application. Figure 16 shows qualitative tendencies for the cost estimation of test generation. If a single code coverage criteria is used (Code-based test generation) the cost of testing contently grows with the coverage of the application. Using higher level models, there is an initial cost for abstracting these models but a significant portion of the application can be covered more easily using those models. The remaining part of the application has to be covered based on the code. Overall the use of model allows reducing the total cost.

In the case of model-driven migration as it is performed by Sodifrance the benefit of using models for testing is even more profitable since the models are already reverse-engineered for migration and can be re-used for testing. The initial cost of creating the models is thus negligible.

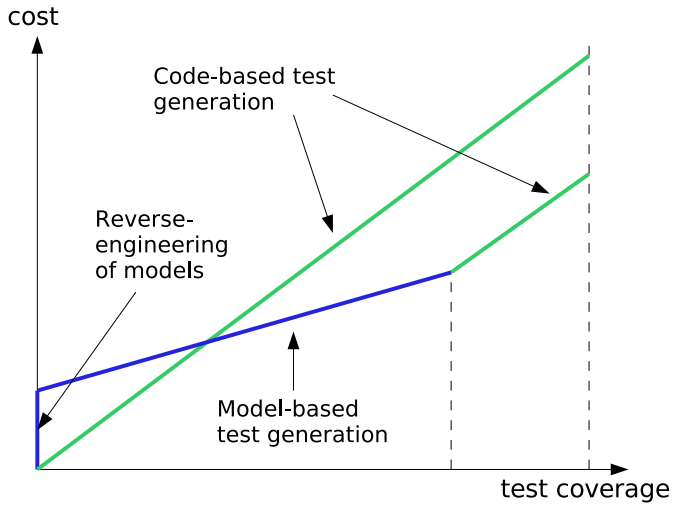


Figure 16: Reducing test generation cost

3.5 Related work

The need for efficient validation techniques in the context of migration has been identified as a crucial problem in several works. Sneed [22] observes that testing in migration projects costs between 50 to 70% of the total costs and is thus one of the major bottlenecks in migration projects. Carriere [5] manually validate the migrated system, but note as a future work that "other techniques, such as regression testing, should be explored". However, as far as we know, there is no specific work on regression testing for migration.

Several works exist for regression test selection for object-oriented programs [6, 21] or web applications [26]. These works consider a system that has been tested with a set of test cases. Then, they focus on advanced techniques to select only subset of test cases that should be reused to validate that the evolution has not introduced unexpected changes. HOwver, they do not consider the particular case of migration. The essential difference between evolutions that are considered in these works and migration is that migration is an isofunctional evolution. This means that the functionality offered by the legacy and the migrated system should be the same. The challenge for regression testing is thus not so much about selecting a subset of test cases. The major challenge is to produce test cases, form the legacy system that precisely and completely represent the system's behavior.

Another important research field that is very much related to our work is the area of model-based testing [9]. Utting and Legeard [25] identifies four main approaches known as model-based testing. The first one is the "generation of test input data from a domain model". In that case, the approach presented in this chapter is clearly model-based testing. Our work is also related to the work by Pretschner et al. [19]. In this chapter, the authors evaluate model-based testing on a large automotive case study. Their study reveals that using a model instead of textual informal documents for test generation is useful in

terms of requirements coverage. The work proposed in this chapter follows the same tracks. We want to provide tests that cover the requirements and the main scenarios on the legacy system. In order to do that and to provide efficient means for qualifying the test cases, we need a model. The model that we propose is specific to the particulat type of applications we consider: form-based applications. These applications represent an important of the large systems that are migrated (banking systems, insurance companies systems, etc.).

Memon [12], uses event-flow models for generating tests for GUI-based applications. An event-flow models is made of two elements. The first one captures each event in terms of the state in which the event may be executed, and the effects it has on the system sate. The second one is a directed graph which represents all possible sequences of events that can be executed on the application. The second part of the event-flow models is similar to the navigation model we use in this chapter. In [14], Memon et al. define test adequacy criteria for GUI testing. These criteria take the composite structure of GUI element in order to select meaningful test sequences. In [13], Memon et al. take advantage of the guards and actions associated with events in order to produce test oracles. In the context of software migration, the principle limitation of their approch is that the event-flow model cannot be reverse-engineered fully automatically.

3.6 Conclusion

Validation in the context of information system migration is crucial issue because it still represents an large amount of the total migration cost. There are few works that tackle the production of efficient test cases for migration. In this chapter we proposed to use models of the legacy application in order to produce those tests. Most of the time no models of the legacy application are available but these models can be automatically produced by revere-engineering the legacy source code.

The criteria we propose are based on a navigation model of the system under test. These criteria could be applied for any "form based" application such as the banking application presented as a case study but also on web applications. The case study allowed to verify that the navigation model is a the appropriate level of abstraction for defining system tests and that the test criteria produce valuable diagnosis for existing test scenarios.

We proposed a test generation process which leverages the models and test criteria in order to allow non domain-experts to produce system tests. This allows the developer of a migrated application to produce the regression tests required in order to ensure that the migrated application behaves just as the legacy application. Another benefit of the proposed approach is the reduction of the effort (and cost) required to produce the tests.

In future work, the criteria should be validated in terms of their quality for producing tests that achieve a satisfactory code coverage. It would be interesting to compare the fault detection rate of tests produced according to these criteria with the fault detection rate of the tests written manually by domain experts. In practice, such controlled experiments are hard to carry in an industrial context.

4 Structural test generation

Because in a migration project the code of the legacy application is available the idea is to use structural testing techniques to generate tests. There are three approaches to structural testing: static, dynamic and hybrid. Using static test generation the tests are generated without the need of executing the program under test while using dynamic or hybrid test generation the program is executed to collect more information and improve the generated test data. the following paragraphs discuss these options with respect to the testing goals of a migration project.

Static test generation for migration projects has been investigated by V.A. Nicolas in [16]. The idea of the proposed technique is to apply static test generation techniques on COBOL programs. However, this technique has a number of limitations which are clearly discussed in [16] :

- It is far from being fully automated. The tester has to provide constraints on variables, stubs for sub-programs and has to concertize test cases.
- The generated tests do not make much sense from a functional point of view. They cannot be used as guidelines by developers of the new application and cannot be sold to the client as reference tests.
- The technique is computationally too costly to apply on non-trivial programs. Studies of the code of the CDN migration project showed that a substantial amount of code could not be processed by such techniques.

These limitations were identified in [16] and a proposed way to overcome them is to use dynamic test generation.

Dynamic test generation consists not only on using static analysis of the program but also executing it to collect more information on its structure. In the literature several algorithms have been defined for that purpose. Among them, based on previous experiences, we chose the bacteriologic approach [2] because it is especially designed to generate sets of test that satisfy a coverage criterion. The use of this algorithm requires the program under test to be instrumented and executed a number of times during test generation. Its advantages are that it requires less inputs from the tester and is less computationally costly. Its drawback is that it cannot be fully automated either as it requires the tester to provide a specification of the input data of the program.

Dynamic test generation provides solutions only to some of the limitations of static test generation. In practice, to achieve a maximum automation, a combination of dynamic test generation and static analysis (hybrid approach) should be used for generating tests from COBOL programs. However these techniques have to be limited to the unit level (single COBOL programs) in order for the algorithms to execute in reasonable time. They cannot be fully automated because some of the information required for generating test cannot be inferred from the code.

This section discusses structural test generation techniques. The goal of these technique is to generate test data that covers the structure of the code of the program under test. In the literature two kind of structural test generation techniques are distinguished. The first on is static test generation which only performs static analysis on the program under test and the second on is

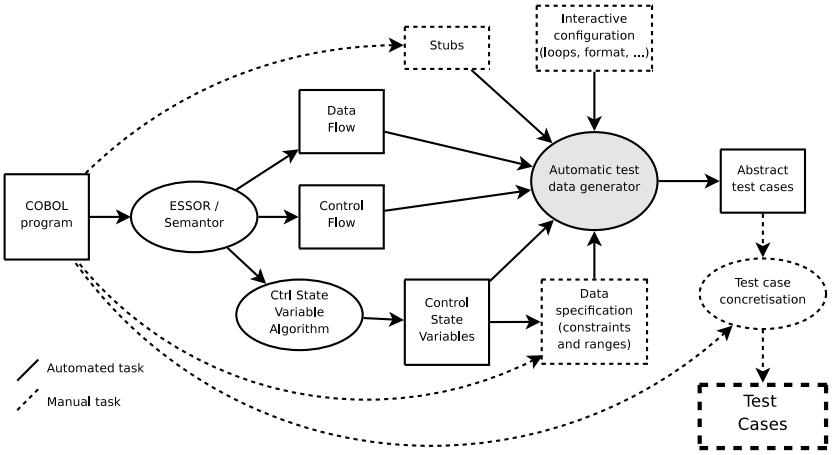


Figure 17: Generating tests from COBOL source

dynamic test generation which require executing the program under test during the test generation process. This section first recalls the results obtained by V-A. Nicolas [16] by applying static test generation on migration projects. Then we discuss how dynamic test generation could be applied in order to overcome the limitations of static test generation.

4.1 Static test generation

This section discusses the work presented in [16]. The objective of this work was to automatically generate test data for COBOL programs. Figure 17 presents the main steps of the test generation process. This process uses control flow and data flow information to extract sets of constraints on the inputs of a program in order to cover each block of code of this program.

The first limitation of this technique (as reported in [16]) is that it is costly for both the tester and in terms of computational time. The tester has to provide stubs for the program under test, semantic constraints on the variables of the program and has to concertize the test data produced by the generator. The user inputs are necessary in order to make the computational time required by the technique acceptable. The specification of semantic constraints is especially costly. During this phase the tester has to provide constraints for all the variables of the program under test. As an example, if a variable is used to represent a date, the integer representing the day should be between 1 and 31, the month between 1 and 12, etc. If a string represents a user name or an account number, the generator won't be able to find interesting values by itself in a reasonable time. In most cases the tester will have to provide at least a format or a set of possible values for string variables. To provide this information the tester has to study in details the program under test. This greatly increases the cost of test generation even if the tester can be assisted by data flow or slicing tools.

The second limitation of this test generation technique is that even with appropriate semantic constraints on all the variable of the program under test, the number of control path can be too high for the generator to produce a result

in a reasonable time. In effect, when working with sample programs coming from a real migration project we found that the control structure of some program are too complex for the generator to be applied. The number of path yields by the control structure is too high to be processed. To avoid this limitation, the tester would have to provide more information to the testing tool. This would again increase the costs of test generation.

To conclude, as it was suggested in [16], generating tests directly from the code remains a costly process. Even if the test generation itself is fully automated providing the appropriate inputs to the testing tool is costly. The main reason is that some of the information required for generating tests is not in the code and has to be inferred and formalized by the tester.

4.2 Dynamic test generation

The idea of dynamic test generation is to execute the program under test a number of times in order to collect information and select test data that meet a particular test criterion. A number of techniques have been proposed for dynamic test generation using pseudo-random algorithms such as genetic algorithms. Most of these work focuses on generating a single test data that covers a statement (or a path of the control flow graph) ofthe program under test but techniques based on the bacteriologic algorithm proposed by Baudry et al. [2] is designed to handle the generation of test suites that globally satisfies a test criterion. This section presents the application of this technique in the context of software migration.

In our context, the program under test is a COBOL program and the test criterion is code coverage. In the following we first recall the principle of the bacteriologic algorithm and then details how it can be applied.

4.2.1 Bacteriologic algorithm

The bacteriologic algorithm is an original adaptation of genetic algorithms as described in [2]. It is designed to automatically improve the quality of a set of test cases for software components. The aim of this algorithm is to generate a set of efficient test cases for a given component under test. The algorithm also takes into account the number of test cases in the generated set. Since it is specialized for test cases generation, it is more efficient than the genetic algorithm (faster convergence, easier to tune).

The general idea is that a population of bacteria is able to adapt itself to a given environment. If bacteria are spread in a new stable environment they will reproduce themselves so that they fit better and better to the environment. At each generation, the bacteria are slightly altered and, when a new bacterium fits well a particular part of the environment it is memorized. The process ends when the set of bacteria has completely colonized the environment.

Inspired by this principle, the bacteriologic algorithm takes an initial set of bacteria as an input, and its evolution consists in series of mutations (using a mutation operator) on bacteria, to explore the whole scope of solutions. The final set is build incrementally by adding bacteria that can improve the quality of the set. Along the execution there are thus two sets, the solution set that

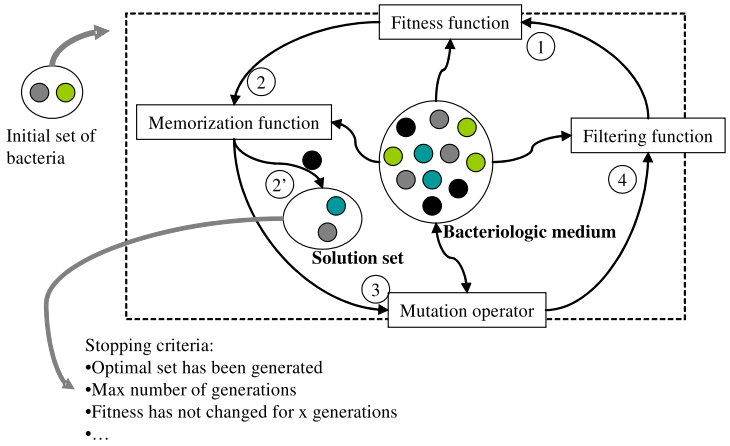


Figure 18: Bacteriologic algorithm principle

is being built, and the set of potential bacteria, that we call a bacteriologic medium.

The global process is incremental and each step is called a generation. A generation consists in four steps as described Figure 18. First, the fitness value for each bacterium in the bacteriologic medium is computed. At step 2, the bacteria that can improve the solution set are selected. At step 2' the selected bacteria are added to the solution set. Step 3 consists in applying the mutation operator. This operator selects bacteria in the medium and generates new bacteria by slightly altering the selected ones. At last, step 4 consists in deleting bad or useless bacteria from the medium. There can be several stopping criteria for the algorithm: after a number of generations, when a minimum fitness value is reached by the solution set, the fitness has not changed for a number of generations...

One particular function is applied at each step of the process given Figure 1. In the following descriptions of these functions, we call $\mathscr{B}$ the set of all possible bacteria for a given problem.

Fitness function $fitness : 2^{\mathscr{B}} \rightarrow \mathbb{R}^+$

The fitness function evaluates the quality of a solution regarding the global objective. Since a solution for the bacteriologic algorithm is a set of bacteria, the fitness function computes the fitness of a set of bacteria. However, we also need to compute the fitness of one bacterium b. This function is called relFitness and is computed relatively to the fitness of a solution set B as follows:

$$relfitness : \mathscr{B} \times 2^{\mathscr{B}} \rightarrow \mathbb{R}^+$$
$$relfitness(B, b) = fitness(B \cup b) - fitness(B)$$

Memorization function $mem : \mathscr{B} \rightarrow \{true, false\}$

This function takes a bacterium as an input and returns true if it can be memorized in the solution set. This function thus computes the relative fitness of the bacterium. If the fitness satisfies a given condition, the

function returns true and the bacterium can be memorized. For example, one condition is: a bacterium can be memorized if its fitness is greater than a given threshold (that it is called memorization threshold).

Mutation operator $mutate : \mathscr{B} \rightarrow \mathscr{B}$
The mutation operator generates a new bacterium by slightly altering an ancestor bacterium. This operator is crucial for the algorithm, since it is the one that actually creates new information in the process. We can note that by iterative applications of this operator we should explore the whole set of possible bacteria $\mathscr{B}$.

Filtering function $filter : 2^{\mathscr{B}} \rightarrow 2^{\mathscr{B}}$
This function aims at periodically deleting useless bacteria from the bacteriologic medium to control the memory space during the execution.

To fit the algorithm, a test case is modeled as a bacterium, and the four functions have to be defined according to this model. The most important functions are the fitness and mutation functions that we detail now. The fitness function computes the utility of a set of bacteria and relFitness computes the utility of a bacterium. This corresponds to the quality of a set of test cases and of a test case in our particular context.

The fitness function needs to be defined according to a target test criteria in order to measure the quality of a set of test cases. For example, an adequacy criterion can express that "all statements in the program must be executed by, at least, one test case". The quality of a set of test cases then corresponds to the proportion of statements executed by these test cases. A fitness function can be defined, based on a particular test adequacy criterion.

The second important function is the mutation function: this function actually creates new information. This function takes a bacterium as an input and slightly alters it to create a new bacterium. The actual operation for mutation on a bacterium strongly depends on the structure of the bacterium. It is not possible to define a general mutation operator.

4.2.2 Generating tests with a bacteriologic algorithm

In our context the test criterion to satisfy is code coverage. A fitness function we can use to estimate the quality a set of test cases is thus the percentage of the code of the program under test it covers. This is a suitable fitness function for the bacteriologic algorithm because that percentage can only increase if new test cases are added (i.e. the relative fitness of any test case is always positive or null). In previous works this fitness function has been successfully used for generating test suites for JAVA programs.

To be able to compute this fitness function the program under needs to be executed in order to compute the code coverage of each test case. In previous works facilities provided by the Java Debugging Interface were used to compute the code coverage of JAVA programs but in most cases the source code of the program under test will need to be instrumented in order to recall code coverage. In the case of COBOL programs it is necessary since the runtime environment does not provide services for computing code coverage.

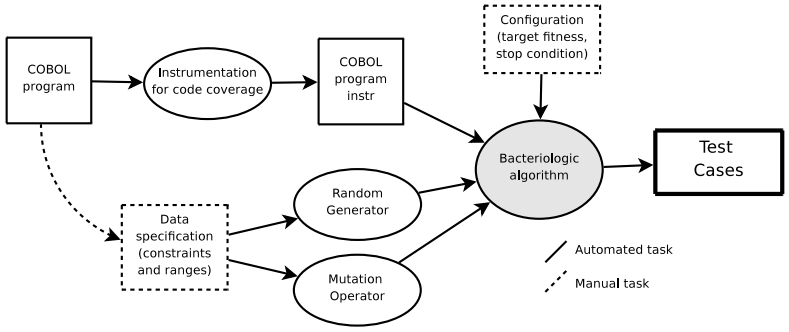


Figure 19: Using a bacteriologic algorithm for test generation

In order to apply, the bacteriologic algorithm needs an initial set of test cases to optimize and a mutation operator. In our context no test cases are available so this initial set of test cases has to be randomly generated. The mutation operator needs to be able to randomly mutate an existing test case to create an new test case. Both the random generator and the mutation operator needs to rely on a specification of the input data of the program under test.

In some cases generic operators can be defined. For instance, if the input data of a program is an integer the random generator can be any integer random generator an the mutation operator can be to add or subtract 1 to the integer. In practice such generic operators can be defined for simple types but for variables with a more complex type, a generator and mutation operator has to be specifically defined. If for instance a variable of type string is supposed to contain an account number or a date in a specific format using a random string generator won't be sufficient. In that case the tester has to provide some constraints and ranges for the variable in order for the test generator to take them into account during random generation and mutation.

Figure 19 presents the test generation process using a bacteriologic algorithm. The test generation process takes the program under and a specification of its input data as an input. The program is automatically instrumented to be able to estimate the code coverage of any test case and the specification of the input data is used to specialize a random test case generator and the mutation operator. The bacteriologic algorithm can then be executed to optimize an initial randomly generated set of test cases. during this generation phase the tester can monitor the progress of the bacteriologic algorithm and tune its stop condition. The test criterion used is code coverage but because most of the time the program under test will contain some dead code the final coverage will be less than 100%.

4.3 Case Study : Virtual Meeting Server

To validate the approach detailed in the previous section, we used a Java virtual meeting server. The system is implemented in Java, it has around 80 classes and 2000 lines of code. It was chosen because its size correspond to the typical size of a COBOL program and its inputs are structured textual commands which is close to COBOL records. The study was not performed directly on

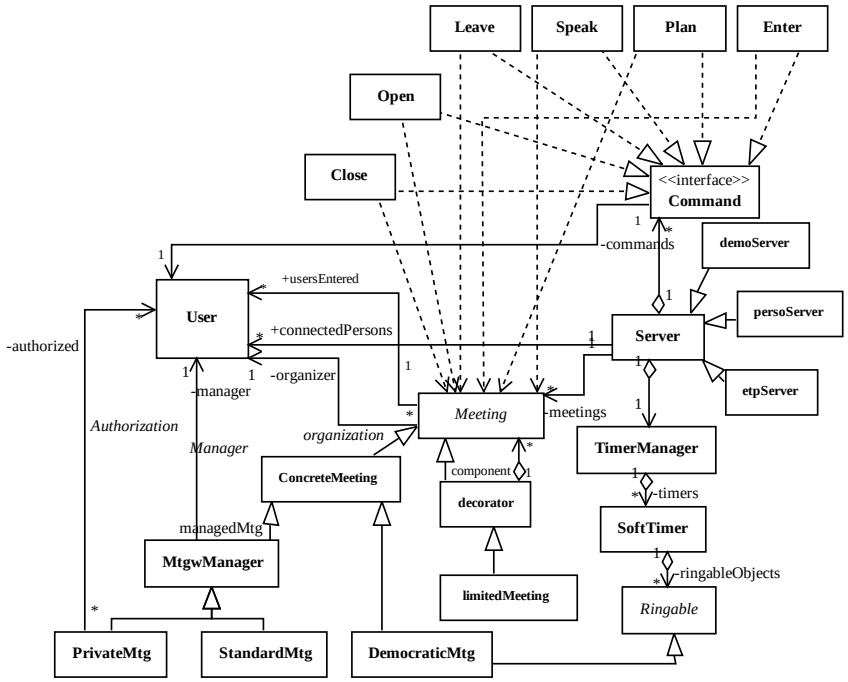


Figure 20: Simplified class diagram of the virtual meeting server

a COBOL program because of tooling issues but no theoretical obstacles have been identified to suggest that results would be different on an actual COBOL program.

Figure 20 presents a simplified class diagram of the virtual meeting server. This system allows creating different kinds of meetings (moderated or not, private or public, ...) and allows users to connect and participate to these meetings. The system handles 17 types of commands and each command has between 1 and 3 parameters. A test case for the virtual meeting server is a sequence of commands to be executed on the system.

In order to apply the test generation techniques proposed in the previous paragraph we need a random test generator and a mutation operator. In the case of the virtual meeting server these two elements are based on a random command generator. Using this random command generator, random test cases of any size can be automatically generated. To generate a new test case from an existing on, the mutation operator can simply replace a command in the existing test case by another command randomly generated. Figure 21 presents a sample randomly generated test case and the use on the mutation operator on this test case.

The static analysis of the virtual meeting server shows the the program has 376 branches. The goal of the test generator is to generate a set of tests to cover these branches. There are several parameters that can be tuned in the algorithm :

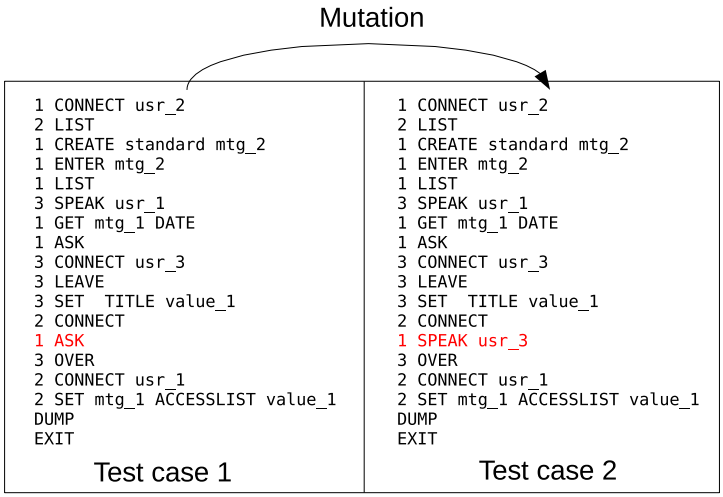


Figure 21: Test case for the virtual meeting server

Size of the test cases Each test case is a sequence of command. The size of the test cases can be fixed. In practice small test cases are better because they are more easy to understand but the test cases shoud be big enough to allow covering all the branches of the system.

Size of the population The algorithm uses a population of test cases. A large population allows more information to be kept between generation but small population allow for a faster evolution.

Mutation rate The mutation rate fixes how different are mutated test cases from an original test case. In the example of figure 21 only one command is changed but in practice more than one command can be replaced.

For all these parameters trade-offs have to be found in order to ensure the quality of the generated test cases and optimize the efficiency of the algorithm. Complete studies of the adjustments of the parameters of the bacteriological algorithm are presented in [2].

As an example, figure 22 details the execution of the bacteriologic algorithm for the virtual meeting system. The parameters used for this experiment are a test case size of 15 commands, a population of 75 test cases and a mutation rate of 15%. This execution generated a test suite composed of 18 test cases and allowed covering almost 90% of the 376 branches of the program under test.

4.4 Conclusion

The study we have done on Java program suggest that using dynamic test data generation allow overcoming some of the limitations of the static techniques that were proposed earlier. Using dynamic informations allows breaking the complexity of programs and allows handling complex data types such a strings which cannot treated by current constraint solvers. The results of the bacteriologic algorithm in previous study 4.2 and the results we obtained on the

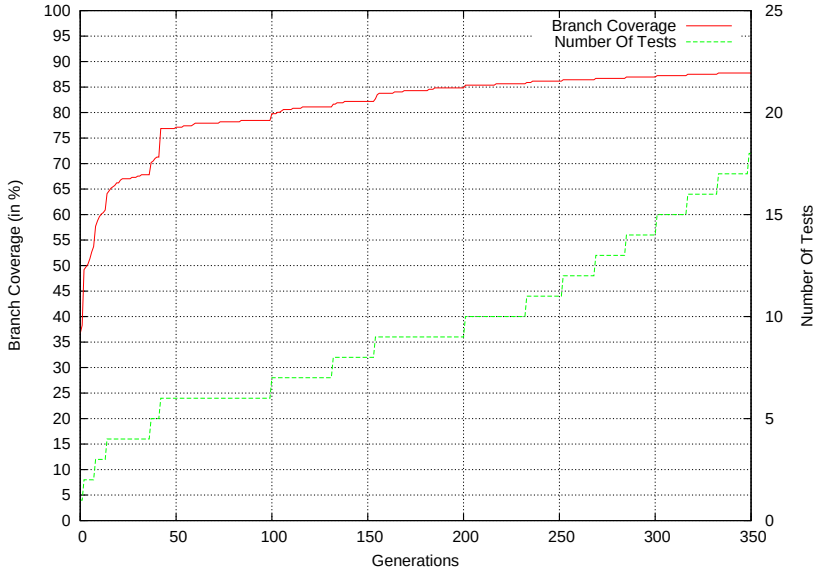


Figure 22: One execution of the test generator

virtual meeting serve suggest that it is a good candidate to fully automate the generation of a set of test cases for a program.

To apply the technique on COBOL program we do not have identified any scientific obstacles but there is a tooling issue. As long as the test generation was static they were no need to execute COBOL programs for test generation. In the case of the technique we propose the COBOL program not only needs to be statically analyzed but it also need to be executed in order to collect execution traces. This is very easy to do in modern development environment such as Java but seems to be more difficult to implement for COBOL. This is the reason why the study presented in this chapter was done on a Java program.

5 Conclusion

Even if model-driven engineering is already economically profitable for migration, there are still some important challenges that need to be tackled. The major issue in terms of human effort is testing. Regression testing is used to validate the migration but the production of efficient regression test cases is currently manual, ad-hoc and difficult to evaluate. During the post-doc we proposed techniques to reduce the testing effort. We investigated two complementary techniques: the generation of system tests from reverse-engineered models and the generation of unit tests to cover the branches of COBOL programs.

At the system level, the test criteria we proposed are based on a navigation model of the system under test. These criteria could be applied for any "form based" application such as the banking application presented as a case study but also on web applications. The case study on the CDN project allowed us to verify that the navigation model is a the appropriate level of abstraction for

defining system tests and that the test criteria produce valuable diagnosis for existing test scenarios. We proposed a test generation process which leverages the models and test criteria in order to allow non domain-experts to produce system tests. This allows the developer of a migrated application to produce the regression tests required in order to ensure that the migrated application behaves just as the legacy application. Another benefit of the proposed approach is the reduction of the effort (and cost) required to produce the tests. In future work, the criteria should be validated in terms of their quality for producing tests that achieve a satisfactory code coverage. It would be interesting to compare the fault detection rate of tests produced according to these criteria with the fault detection rate of the tests written manually by domain experts. We did not had time to carry such experiments during the post-doc.

At the unit level, we started from a static test generation techniques proposed during a previous post-doc. This techniques allowed generating test data for programs by solving the constraints on the variables of the program in order to cover all branches of the program. This techniques had two major limitations which make its use on projects impossible. The first one is scalability: it is unable to handle the complexity of typical COBOL programs. We studied the complexity of COBOL programs on the CDN projects and we have shown that a purely static test generation algorithm would require several million years to execute. The second limitation is that constraint solvers are not able to handle complex data types such as string. Basically only numeric and boolean types can be handled. To overcome these limitation, the idea we proposed is to combine the static analysis of the program under test with a dynamic test generation algorithm. The dynamic test generation algorithm actually executes in order to collect informations on its structure and on the coverage of the tests. The approach we proposed was validated on a Java case study. In future work the algorithm should be connected on COBOL tools in order to generate tests for COBOL programs.

References

[1] Paul Baker, Shiou Loh, and Frank Weil. Model-driven engineering in a large industrial context - motorola case study. In *MoDELS'05*, pages 476 – 491, Montego Bay, Jamaica, 2005.

[2] Benoit Baudry, Franck Fleurey, Jean-Marc Jezequel, and Yves Le Traon. Automatic test case optimization: A bacteriologic algorithm. *IEEE Software*, 22(2):76–82, 2005.

[3] Behzad Bordbar, Dirk Draheim, Matthias Horn, Ina Schulz, and Gerald Weber. Integrated model-based software development, data access, and data migration. In *MoDELS'05*, pages 382–396, Montego Bay, Jamaica, 2005.

[4] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. 2004.

[5] S. J. Carriere, S. G. Woods, and R. Kazman. Software architecture transformation. In *WCRE (Working Conference on Reverse Engineering)*, pages 13–23, Atlanta, GA, 1999.

[6] Pavan Kumar Chittimalli and Mary Jean Harrold. Re-computing coverage information to assist regression testing. In *ICSM'07 (International Conference on Software Maintenance)*, Paris, France, 2007.

[7] Franck Fleurey, Benoit Baudry, Alain Nicolas, Erwan Breton, and Jean-Marc Jézéquel. Model-driven engineering for software migration in a large industrial context. In *Proceedings of MODELS/UML'2007*, LNCS, pages –, Nashville, USA, October 2007. Springer.

[8] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[9] T. JÃ©ron and V. Tschaen. Test generation engine documentation. AGEDIS : Automated Generation and Execution of test suites for DIstributed component-based Software, 2004.

[10] M1 Global Solutions. Model driven software development and offshore outsourcing, 2004.

[11] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[12] Atif M. Memon. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability*, 2007.

[13] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated test oracles for guis. In *Proceedings of the 8th ACM SIGSOFT*, pages 30–39, New York, NY, USA, 2000. ACM Press.

[14] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for gui testing. In *Proceedings of the 8th European software engineering conference*, pages 256–267, New York, NY, USA, 2001. ACM Press.

[15] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML'2005*, LNCS, pages –, Montego Bay, Jamaica, October 2005. Springer. http://www.kermeta.org/.

[16] Valérie-Anne Nicolas. Conception et déloppement d'un outil de géération de jeux de test pour des applications de gestion, 1999.

[17] Interactive Objects. Arcstyler, 2007.

[18] OMG. Architecture-driven modernization, 2006.

[19] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 392–401, New York, NY, USA, 2005. ACM Press.

[20] Thijs Reus, Hans Geers, and Arie van Deursen. Harvesting software systems for mda-based reengineering. In *ECMDA-FA'06*, pages 213–225, Bilbao, Spain, 2006.

[21] G. Rothermel, Roland H. Untch, Chengyun Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[22] H.M. Sneed. Risks involved in reengineering projects. In *WCRE'99 (Working Conference on Reverse Engineering)*, pages 204–211, Atlanta, GA, USA, 1999.

[23] Sodifrance. Model-in-action tool-suite, 2007. http://www.mia-software.com/.

[24] Miroslaw Staron. Adopting model driven software development in industry - a case study at two companies. In *MoDELS'06*, pages 57–72, Genova, Italy, 2006.

[25] Mark Utting and Bruno Legeard. *Practical Model-Based Testing*. Morgan Kaufmann, 2007.

[26] Lei Xu, Baowen Xu, Zhenqiang Chen, Jixiang Jiang, and Huowang Chen. Regression testing for web applications based on slicing. In *COMPSAC 2003 (Computer Software and Applications Conference)*, pages 652– 656, 2003.

[27] Ying Zou and Kostas Kontogiannis. Migration to object oriented platforms: a state transformation approach. In *ICSM'02 (International Conference on Software Maintenance)*, pages 530– 539, 2002.



Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399