CHAPTER 9

MODEL-BASED TEST-CASE GENERATION IN ESA PROJECTS

Stefan Mohacsi
Armin Beer

Stefan Mohacsi and Armin Beer describe their experience in using model-based testing (MBT) for the European Space Agency (ESA). Their team developed a test automation framework that took significant effort to set up but eventually was able to generate automated tests very quickly when the application changed. This chapter includes an excellent return-on-investment calculation applicable to other automation efforts (not just MBT). The team estimated break-even at four iterations/releases. The need for levels of abstraction in the testware architecture is well described. The application being tested was ESA's Multi-Mission User Information Services. The multinational team met the challenges of automation in a large, complex system with strict quality requirements (including maintainability and traceability) in a waterfall development—yes, it can work! If you are thinking of using MBT, you will find much useful advice in this chapter. A mixture of inhouse, commercial, and open source tools were used by the team. See Table 9.1.

9.1 Background for the Case Study

The members of the Siemens Support Center Test have many years of practical experience in testing large and safety-critical projects. In cooperation with Siemens

Table 9.1 Case Study Characteristics

Characteristics	This Case Study
Application domain	European Space Agency earth observation services
Application size	>500,000 LOC
Location	Vienna, Austria; Frascati, Italy; Darmstadt, Germany
Lifecycle	Traditional
Number on the project	>100 working for various contractors of ESA
Number of testers	5
Number of automators	3
Time span	Test automation for SCOS-2000 started in 2004, for MMUS in 2008; both projects are still ongoing.
Dates	2004 onward
Tool type(s)	Inhouse, commercial, open source
Pilot study undertaken	No
ROI measured	Yes, projected payback after 4 cycles
Successful?	Yes
Still breathing?	Yes

Space Business, they created a test automation framework for the European Space Agency (ESA) that has been successfully applied in a number of projects.

Key areas of our experience include:

- State-of-the art processes, technologies, and methods for testing complex systems.
- Automated testing technologies and tools—commercial, open source, and inhouse.
- Process improvement projects in software testing.
- Research software testing techniques in cooperation with universities.

In this case study, the application of model-based testing (MBT) and a test-case generator (TCG) in projects of the European Space Agency (ESA) is presented. This chapter has four main sections. Firstly, we examine MBT and TCG and their application in software development. The next section describes the nature of the project, the challenges, and the applied framework. This is followed by an outline of the experience and lessons learned after the application of the framework in ESA projects. The final section presents a summary and brief outline of future plans in the areas of MBT and TCG.

9.2 Model-Based Testing and Test-Case Generation

In our experience of developing complex systems, test cases are mainly created manually and are based on experience rather than systematic methods. As a consequence, many redundant test cases exist and many aspects of an application remain untested. Often, the effort for the implementation and maintenance of these test cases is higher than expected, and test completion criteria are too coarse. MBT and TCG helped us to improve the quality of the system specification and its associated test cases and made it possible to complete the project on time and within budget.

Issues to be solved in this area are as follows:

- Which kind of model notation can be used as a basis for deriving test cases?
- Many specifications using popular notations like Unified Modeling Language (UML) are not formal enough for automatically generating complete test cases because a lot of information is contained in plain text annotations. While UML can be extended to achieve the required level of formality, this approach typically aims at code generation rather than test generation and involves considerable complexity and effort.
- Formal models like finite-state machines are often very large and difficult to apply.
- A trade-off between formalization and practical suitability needs to be found.
- Which modeling and test-case-generation frameworks are available, and how useable are they?
- How efficient is test automation in terms of reducing test costs, and how effective is it in assuring high product quality?
- How effective are the test cases in detecting defects?

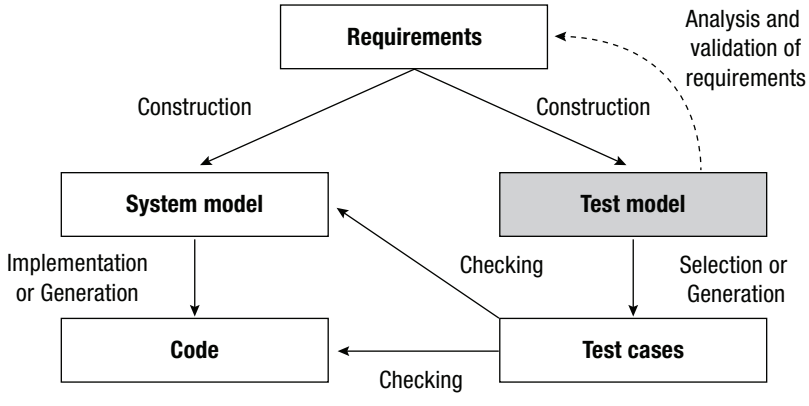
As depicted in Figure 9.1, the requirements are the source for the construction of both the system and the test models. The two models have to be separate because no deviations would be detected if both the code and the test cases were generated from the same model.

The test model's required level of detail depends on the requirements of the TCG and the planned types of test (e.g., functional or randomized tests). This approach guarantees the traceability from the requirements to the test cases, which is a prerequisite for an objective assessment of the test coverage and the product quality.

9.2.1 Model-Based Testing with IDATG

MBT and TCG are implemented by the tool IDATG (Integrating Design and Automated Test-Case Generation). IDATG has been developed since 1997 by the Siemens Support Center Test in the scope of a research project in cooperation with universities and the Softnet Austria Competence Network. Over the years, the functionality has continuously been expanded and the tool has been applied in numerous internal and external projects. Today, IDATG is a commercial tool and closely integrated with the Siemens test management tool SiTEMPPO.

IDATG consists of a set of visual editors for MBT specification and a powerful TCG that produces complete test scripts for various tools. Within the IDATG notation, we can distinguish different layers of abstraction that significantly increase the maintainability of the tests. The workflow for IDATG is shown in Figure 9.2.



Source: P. Liggesmeyer: Software Qualität; 2. Aufl., Spektrum Verlag; 2009

FIGURE 9.1 Construction of a test model and generation of test cases

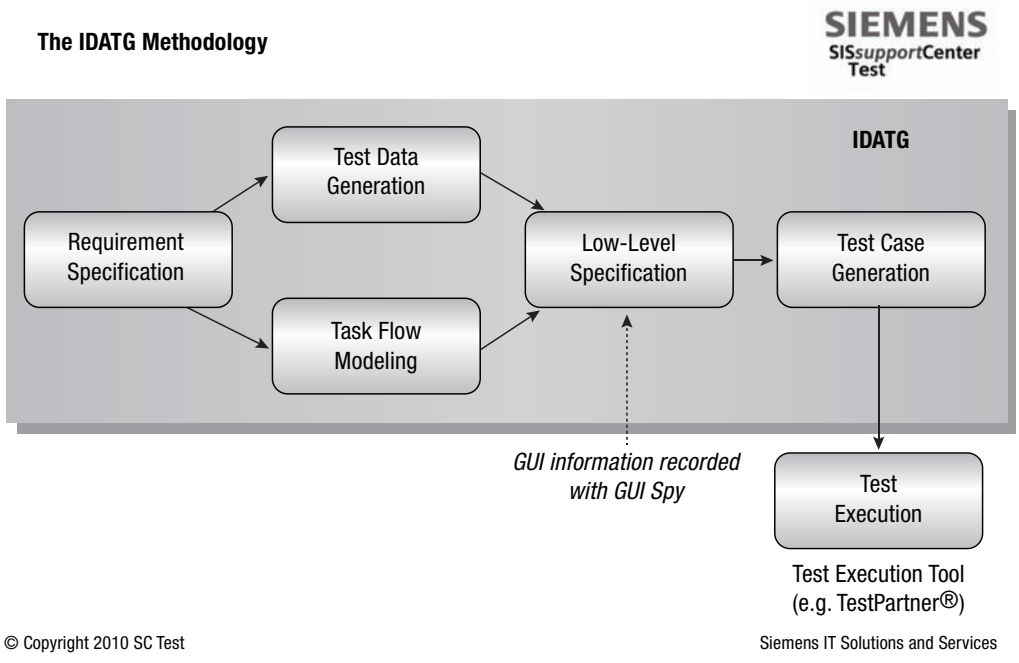


FIGURE 9.2 Workflow for using IDATG

9.2.1.1 Task Flow Models in IDATG

In IDATG, the following types of objects are organized in a hierarchical tree:

- Use case tasks that represent typical use scenarios or any other test scenarios that might be of interest.
- Building blocks that represent frequently used sequences of test steps and can be used as parts of a use case task. Similar to functions in a programming language, building blocks can be parameterized. Furthermore, building blocks may comprise other building blocks.

The *task flow model* of a task or building block is a directed graph that shows the associated sequence of steps. If there is more than one possible way to fulfill a task, the graph has multiple paths.

The steps of a task flow model may either represent *atomic actions* or refer to a *building block* that is itself composed of several steps.

In this way, it is possible to define a complex task as a sequence of simpler ones. This technique significantly reduces the effort for test maintenance, because a change only affects one building block instead of every single test case.

Good Point

Well-structured testware has layers of abstraction to keep different types of change restricted to as few elements as possible.

For example, for GUI testing, an atomic action can be the user input applied to a GUI object (clicking a button, entering text into a field). For non-GUI testing, typical actions are calling a shell or Python script or executing a remote command.

Apart from the test instruction, a step may also have preconditions, a timeout value, a criticality, and expected results. Preconditions are used to express semantic dependencies and may refer to building block parameters or IDATG variables (so-called designators). For example, the building block `Login` in Figure 9.3 expects the string parameters `User` and `Password` as well as a Boolean parameter `Anonymous`. Depending on the value of `Anonymous`, the generation algorithm chooses the upper or lower path. An example for such a precondition would be `(#Login:Anonymous# = FALSE) AND (#Login:User# != "")`.

It is also important to define the effects of each step. For instance, the value of a variable may change and thus affect the precondition of a subsequent step. For example, after the Login block, there might be a step with a precondition `#UserRole# = "Admin"`. In this case, it is necessary to specify to which value the variable `#UserRole#` is set in the `Login` block.

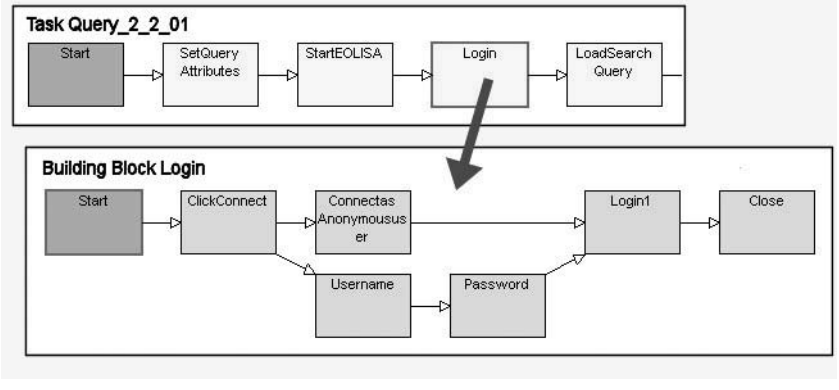


FIGURE 9.3 IDATG task flow (example)

Before the GUI is available, the task flows can already be modeled, with the assignment of the GUI objects at a later stage. As soon as a GUI prototype is available, the GUI layout, including the properties of all GUI objects, is recorded with the built-in IDATG GUI Spy directly from the screen. The steps of the task flows can then be assigned to specific GUI objects simply by pointing at the recorded screenshot. IDATG also offers a semiautomated auxiliary function for reassigning the steps after changes.

9.2.1.2 Test Data Generation

Apart from test sequences that can be expressed with a graph, it is also important to define the input data that should be used for the tests. Because it is usually impossible to test a component with every possible input combination, various methods are used to reduce the number of test inputs while still revealing a high number of program defects. IDATG provides a number of powerful features for creating this input data as well as expected results and for including it in the task flows. One example is the CECIL (cause-effect coverage incorporating linear boundaries) method, a powerful combination of the cause and effect, multidimensional equivalence class, and boundary value methods.

9.2.1.3 Test Case Generation

IDATG generates valid test cases based on the task flows and semantic conditions defined by the user. The test cases include the expected results and steps to check them against the actual results. The user can adjust the test depth by choosing between several coverage criteria (e.g., coverage of all steps, connections, data). Once a set of test cases is generated, it can be exported in various formats, such as XML; scripts for QuickTest Professional (QTP), TestPartner, SilkTest, or WinRunner; or as plain text. The produced test scripts do not require any manual processing but are immediately executable.

9.3 Our Application: ESA Multi-Mission User Services

ESA's Multi-Mission User Services (MMUS) infrastructure provides services for earth observation (EO) users for satellite planning, cataloging, and ordering of EO products. Supported services include user and project management, product catalog search and ordering, mission planning, online information, and documentation services. Various interfaces are available to the user, such as a Java standalone application (see Figure 9.4) and a web interface.

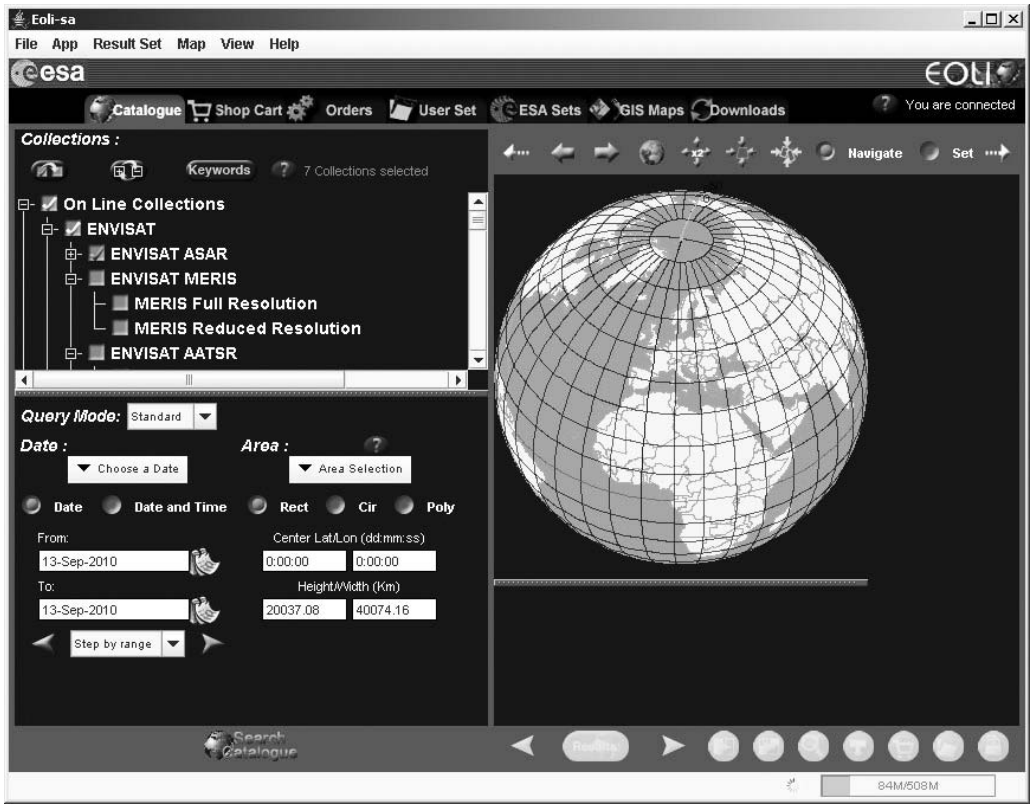


FIGURE 9.4 GUI of Multi-Mission User Services

Apart from the development and maintenance of the operational system, an important activity is the assembly, integration, and operational validation (AIV) of MMUS for major software versions ("evolutions"). The main tasks of the AIV team are:

- Setup and maintenance of an offsite integration and validation environment that is representative of the operational target environment.
- Installation and configuration of the MMUS applications in the AIV environment.
- Creation and maintenance of test specifications.
- Integration testing and system testing of software deliveries, changes, and related documentation.

- The responsibility of the AIV activities has been outsourced to Siemens. An objective of ESA was that the services should improve over time in order to continue to satisfy existing and new requirements. The analysis of key performance indicators (KPIs) and the technology evolution in the marketplace had to be taken into account. Innovation and use of advanced technologies, particularly for test automation, were major issues. An explicit goal of ESA was to reach a test automation level of at least 75 percent, meaning that 75 percent of all test steps had to be executed by an automation tool. In this project, a number of test steps, for example, editing database entries on a distant system, must be done manually.

9.3.1 Testing Approach for MMUS

A number of challenges were encountered in testing the MMUS applications:

- High complexity of the distributed MMUS system.
- Reducing the effort for test maintenance and regression testing.
- Poor testability—many GUI objects could not be properly identified by automated test tools.
- Lack of experience with virtualization required for the setup and maintenance of multiple test systems with different configurations.

9.3.1.1 Key Strategies

The following list gives the key strategies that were adopted to meet the challenges:

- Early involvement of the AIV team in document reviews to ensure early detection of requirement and design defects.
- Systematic test case design and generation using the model-based IDATG approach to guarantee an adequate level of test coverage and reduce the effort for test maintenance.
- Good cooperation with the developers resolved the testability issues. Following a clear naming convention, a unique ID was assigned to each GUI object by the developers.

Good Point

Developer cooperation is needed to build in automatability and testability.

- Strict test entry and exit criteria. An efficient integration test can take place only if the components have reached a certain quality level. For instance, a quick incoming inspection of the most important functions ("smoke test") was introduced, and one of the test entry criteria was "No 'critical' or 'blocking' defects are discovered during the smoke test." Likewise, the test of the completed system is only possible if the integration has been successful. An example for a test exit criteria is "All test cases with priority 'high' or 'medium' have been executed with result 'passed.'"
- Compliance with International Software Testing Qualifications Board (ISTQB) standard terminology (e.g., to avoid using the same terms with different meanings).

9.3.1.2 Test Framework

The software environment for the test activities consists of a set of powerful yet affordable tools. It is the result of testing experience in numerous projects at ESA, such as automated regression testing for the Spacecraft Operating System (SCOS)-2000 (see Mohacsi 2005 and 2006) as well as in the telecommunications, railway, insurance, and banking domains. All of these were complex projects where distributed teams worked together to deliver multiple releases of software systems to their clients. Figure 9.5 gives an overview of the AIV environment.

- For the crucial task of storing mission data and test management information, the commercial tool SiTEMPPO (http://at.atos.net/sitemppo) is used. It is not only used for ESA projects but applied in a wide range of domains, such as automotive, health care, and the public sector. Its flexibility and adaptability to new missions is a valuable advantage.
- The main component for designing the test model and generating test cases is IDATG. From the abstract model defined in IDATG, the tool produces scripts both for GUI testing and for testing over other interfaces.
- For GUI testing, *MicroFocus TestPartner* is used. Test scripts are generated by IDATG and imported automatically into the TestPartner database.
- In the course of ESA's SCOS-2000 project, we recognized the need for a generic test execution tool that can perform tests over arbitrary interfaces (e.g., Common Object Request Broker Architecture [CORBA], TCP/IP). Because no such solution was available, Siemens developed the tool *Test Commander*, which receives test scripts generated by IDATG and executes them by sending commands over a set of test drivers and checking the results. The tool has a simple user interface, is written in Tool Command Language and employs the GUI toolkit (a combination called Tcl/Tk), and is therefore completely

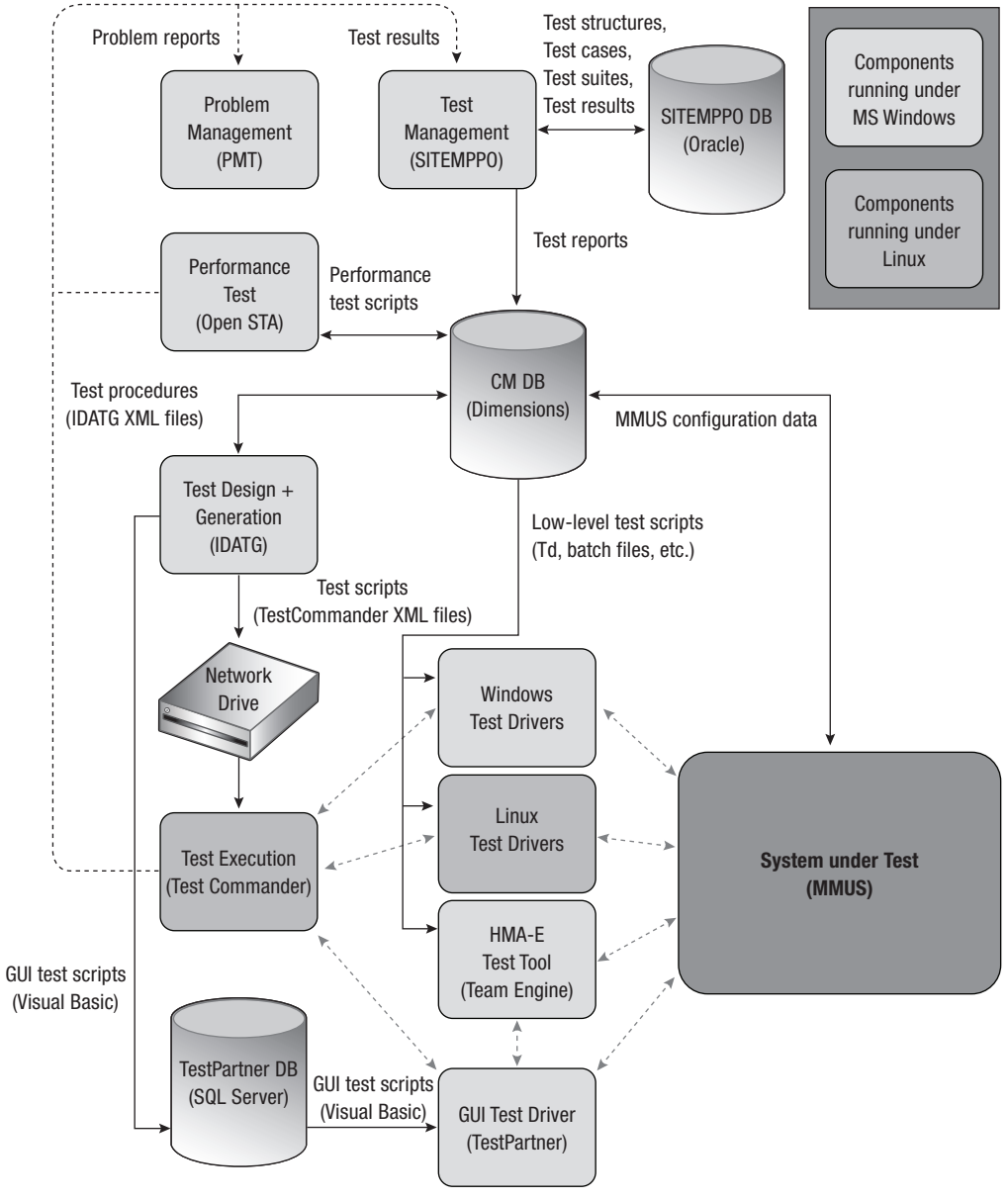


FIGURE 9.5 Overview of AIV environment

platform independent. About one person-year was spent on its development. Because the tool belongs to ESA and it is very easy to plug in new test drivers, the tool could easily be adjusted and reused for the test of MMUS.

Tip

Implement levels of abstraction by making reusable elements that are as generic as possible.

- The only system-specific parts are the *test drivers* required by Test Commander and *stubs or simulators* to replace parts of the system that have not yet been integrated.
- The ESA-specific tool *Team Engine* is used for generating SOAP (Simple Object Access Protocol) messages. The tool can either be used standalone or run automatically via its web GUI using TestPartner.
- For performance testing, finding an affordable tool is particularly hard. In our experience from previous projects, a reasonable solution is the open source tool *OpenSTA*, which provides sufficient functionality for an end-to-end performance test.

9.3.1.3 Typical Test Scenario

One of the core functions of MMUS is the search for images taken by EO satellites. The user not only can search the catalog for existing images but also may issue queries for "potential" ones (i.e., images that could be taken in the future when a satellite passes over a certain area).

The main challenge for testing has been the huge number of possible search options:

- Large number of satellites, most of them carrying various sensors that support different image modes
- Date and time parameters (both for past and future)
- Geographical parameters (search area)
- Satellite and sensor-specific parameters like orbit, track, swath, and cloud coverage

To handle this complexity, our goal was to model the basic test sequence to be as flexible and maintainable as possible. The general scenario of a catalog search test case is shown in Figure 9.6.

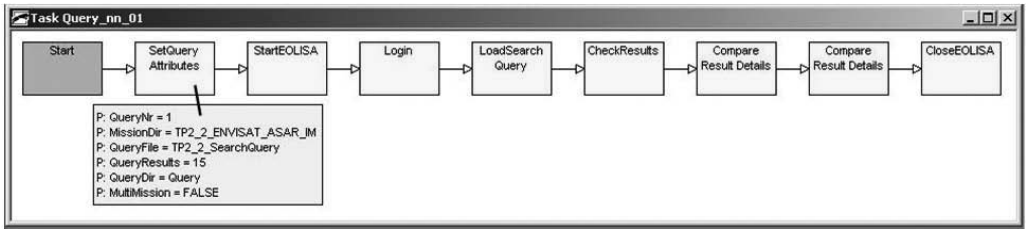


FIGURE 9.6 Test scenario catalog search

The block `SetQueryAttributes` is used to define the parameters of the test case. These include the directories and files for the test data, the number of expected query results, and a Boolean attribute indicating whether the query covers more than one mission. A previously prepared query file containing the search options is loaded from the path specified by the test case's parameters. From the set of query results, some representative items are checked in more detail, including an image comparison. The block `CompareResultDetails` appears once for each compared item and receives the index of the item as a parameter. In our example, we are comparing two items.

During the generation, the blocks are expanded and replaced by atomic steps. The average length of a completed catalog search test case is about 120 steps. The basic scenario can be reused with different parameters very easily:

- Create a copy of the scenario.
- Change the parameters of the first step and the `CompareResultDetails` blocks.
- Prepare the query file and the expected results (reference data).

In this way, a large number of test cases could be created in a very short time. In case of changes to the system (e.g., different login process), only a building block had to be adapted for generating a complete set of updated automation scripts. The same approach was used for other test scenarios, such as the ordering of high-resolution satellite images.

Good Point

Use a standard template to add additional automated scenarios with minimum effort.

9.4 Experience and Lessons Learned

The experience of using MBT has been very positive. In this section, we describe the benefits achieved and the return on investment (ROI) calculations for MBT, as well as some problems and lessons learned.

9.4.1 Benefits

The benefits of this solution have been the following:

- Implementing MBT with reusable building blocks significantly improves the maintainability (details can be found in Section 9.2.1.1).
- Traceability of requirements throughout all test-related documents (test model, test cases, test reports, defect reports) allows an objective assessment of the test coverage and the product quality. If neglected, important aspects of the system could remain untested.
- High defect detection potential of advanced TDG methods like CECIL. A number of case studies have shown that by applying the CECIL method, up to 30 percent more defects can be detected than with ordinary cause–effect analysis.
- Gaps and defects in the system model are discovered at an early stage because of the early involvement of the test team in reviews and the creation of a formalized test model.

9.4.2 ROI of Model-Based Testing

In this section, we focus on the ROI activities for test case creation, maintenance, and execution and compare the necessary efforts for a manual versus a model-based approach. Our study covers a basic set of 180 test cases for the catalog search and ordering scenarios. The total number of test cases created for MMUS was much larger. The values for the manual approach are based on experiences from testing previous MMUS versions and have been extrapolated accordingly.

Tip

Calculating ROI is important to do but doesn't have to be exact or complete to be useful and valid.

9.4.2.1 Test Creation

In the context of the MMUS project, the aim of test case creation was to achieve 100 percent requirements coverage and to include all satellites and sensors in the tests.

The textual description of a *manual* test case in a document, including the preparation of appropriate test data, took about 1 hour. The basic test set consisted of 180 test cases, so *180 hours* are required.

On the other hand, the creation of a *test model*, including building blocks, variables, conditions, and so on, took about 2 weeks (80 hours) for each of the two main scenarios. However, once a scenario exists, it can easily be copied and reused with different parameters (1 hour per test case, including test data preparation). Thus, the overall effort for the creation of the test model was 2 × 80 hours + 180 × 1 hour = *340 hours*.

Note that there were more usage scenarios than these, but we decided to start the automation with the two most important and promising ones.

Good Point

It is important to start with a limited set of tests but ones that will give real value when automated.

9.4.2.2 Preparation of the Test Automation Framework

Apart from the test case creation, several additional activities were required for automating the tests:

- Acquisition, installation, and configuration of test tools: *60 hours*
- Tool training for testers: *40 hours*
- Testability analysis of the user interface, solving testability problems: *200 hours*
- Obtaining all necessary information required to create a detailed test model: *160 hours*

All in all, about *460 hours* were required to prepare the test automation framework.

9.4.2.3 Test Execution

The average time required for executing a test case manually is about 45 minutes, most of which is dedicated to checking details of the actual results against

the expected ones. In each test case, about 100 individual data checks have to be performed.

The total *manual* execution time of a complete test cycle is 45 minutes × 180 test cases = 8100 minutes = *135 hours*.

Automated execution with TestPartner is much faster, because checking the data is a matter of a few seconds. Even if we take into account that sometimes the test execution is halted by unforeseen events and has to be repeated, the average execution time does not exceed 4 minutes for any one test case.

The total *automated* execution time of a complete test cycle is 4 minutes × 180 test cases = 720 minutes = *12 hours*.

9.4.2.4 Test Maintenance

Maintaining textual test case descriptions can become quite tedious because changes of the system under test often affect a huge number of test cases. This is particularly difficult if the semantics have been changed and make it necessary to adapt the sequence of the test steps. On average, about 15 minutes are required per test case for maintenance before each test cycle.

The total *manual* maintenance time of a complete test cycle is 15 minutes × 180 test cases = 2700 minutes = *45 hours*.

While the maintenance of automated test scripts for a capture/replay tool can be a nightmare, the maintenance of an efficiently designed *test model* is far easier. Instead of having to adapt every single test case, only a few details of the model have to be changed. Subsequently, all test cases are generated afresh from the updated model. The whole process takes no longer than *5 hours*.

9.4.2.5 Overall Effort

The overall *manual* test effort can be calculated as follows:

Creation + Execution + Maintenance = 180 hours + (135 hours × number of test cycles) + [45 hours × (number of test cycles – 1)]

(Note that maintenance is only required on the second and subsequent test cycles.) Likewise, the overall test effort for *MBT* automation is:

Creation + Automation + Execution + Maintenance = 340 hours + 460 hours + (12 hours × number of test cycles) + [5 hours × (number of test cycles – 1)]

For the first test cycle, we get 315 hours for the manual approach and 812 hours for the model-based one. After about four test cycles, we reach the breakeven point of ROI with 855 hours versus 863 hours. In the course of the project, the difference becomes increasingly significant: The effort for 20 test repetitions reaches 3,735 hours versus 1,135 hours. This is summarized in Table 9.2 and Figure 9.7.

9.4.3 Problems and Lessons Learned

No matter how good an approach is, problems are encountered and lessons are learned along the way. These are some of the things we learned through our experience.

- Creating a test model requires detailed design documents. However, such documents often do not provide the required level of detail. Involvement of the test team in early reviews can help to mitigate this problem.
- Testability issues should be considered at an early stage. All test objects have to provide appropriate interfaces that allow automated test tools to identify them, send commands, and verify the results. "Design for Test" is already state-of-the-art in hardware development but very often neglected in the software domain.
- The creation of the test model requires considerable effort. If only a few test repetitions are planned, MBT might not pay off. On the other hand, MBT can significantly reduce the effort for each new test repetition. Therefore, it has to be calculated for each project when and if the ROI can be reached.

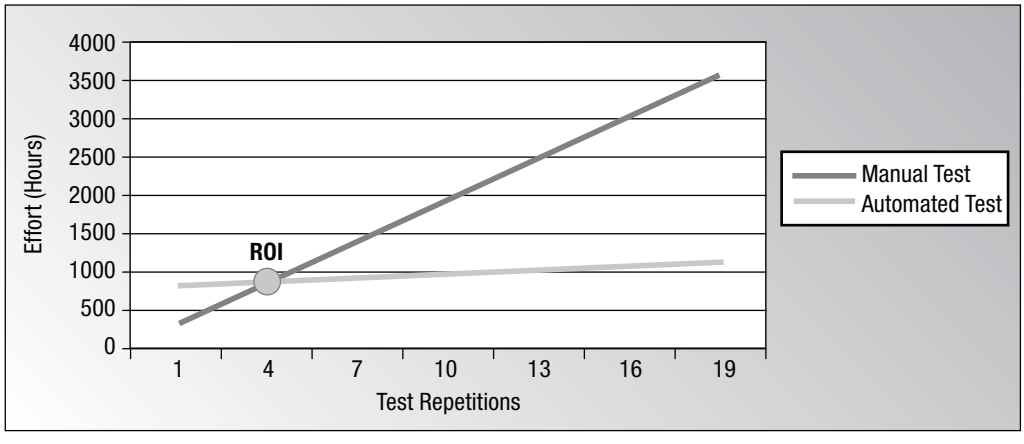


FIGURE 9.7 ROI of model-based test automation

Table 9.2 ROI Calculation

Activity	Manual Hours	Model-Based/Automated Hours
Test creation	180 test cases @ 1 hr: 180 hr	Creation of 2 scenarios @ 80 hr: 160 hr 180 test cases @ 1 hr: 180 hr
	Manual test creation total: 180 hr	Automated test creation total: 340 hr
Preparation of framework	N/A	Acquisition and installation of tools: 60 hr Tool training: 40 hr Testability analysis and improvement: 200 hr Obtaining information for test model: 160 hr
		Framework preparation (automation) total: 460 hr
Test execution (per cycle)	180 test cases @ 45 min: 135 hr	180 test cases @ 4 min: 12 hr
Test maintenance (per cycle)	180 test cases @ 15 min: 45 hr	2 scenarios @ 2.5 hr: 5 hr
Total Effort	**Manual Hours**	**Model-Based/Automated Hours**
1 test cycle	315 hr	812 hr
2 test cycles	495 hr	829 hr
3 test cycles	675 hr	846 hr
4 test cycles	855 hr	863 hr
5 test cycles	1035 hr	888 hr
10 test cycles	1935 hr	965 hr
20 test cycles	3735 hr	1,135 hr

- The creation of the test model requires advanced skills and should be done only by experienced test analysts. We strongly discourage any expectations that MBT or test automation could be efficiently performed by nontechnical staff or end users.

Good Point

MBT, like other technologies, can give great benefits, but only if the conditions are appropriate for its use.

9.5 Conclusion

In this section, we summarize our experience and give an idea of where we will be going in the future.

9.5.1 Summary

MBT and TCG can play an important role when it comes to improving defect detection and reducing quality costs. The prerequisites for the application of MBT are detailed requirements and skilled test case designers.

Experience from ESA projects has shown that the IDATG method is a useful approach for model-based test design. The building-block concept and the separation of sequences and data in the IDATG model help to reduce test maintenance costs significantly.

The automation framework presented here also incorporates tools for test management and defect tracking in order to allow traceability from the requirements to the test model, test cases, and defect reports. The framework is suited for testing complex applications such as MMUS and facilitates testing in the maintenance phase significantly. The efficiency of MBT and TCG is manifested by an ROI after four test cycles in the case of the MMUS project.

9.5.2 Outlook

9.5.2.1 Random Test Generation

In terms of the outlook for the future, the IDATG method presented in this chapter covers the functional test of typical use scenarios. However, reliability testing is also an important issue in respect to the development of dependable systems. To

further enhance the IDATG approach, random testing has recently been added. In our experience, random testing should not be used as a substitute for systematic testing but rather as an important supplement that is likely to discover defects that are hard to detect by other methods.

When building blocks are used as a starting point, the new method creates randomly chosen paths through and between these building blocks while still observing the semantic dependencies. In a first step, each building block is converted into an extended finite-state machine (EFSM). Then, transitions are added between all the EFSMs. Guard conditions are automatically determined from the required state of the GUI elements in the start steps of the corresponding blocks. For creating randomized test cases (or "random walks"), these sequences are used rather than completely random inputs. The conditions for these steps are checked during TCG. The generation algorithm uses a hybrid approach combining a simple depth-first search strategy with refined constraint-satisfaction mechanisms. First results indicate that this hybrid approach seems to be a reasonable trade-off between good test coverage and acceptable generation times. A working prototype has already been completed and evaluated in the course of a pilot project (see Mohacsi 2010).

9.5.2.2 Latest News

To better support the user workflow and to avoid duplication of effort, IDATG was recently integrated into the test management framework SiTEMPPO. Accordingly, IDATG will henceforth be known as "SiTEMPPO Designer."

Siemens IT Solutions and Services and Atos Origin merged in July 2011 to become Atos. The rights to the SiTEMPPO tool suite, including IDATG, are also owned by Atos. Information about the test services of our new company can be found at http://atos.net/en-us/solutions/business_integration_solutions/test-and-acceptance-management/default.htm.

9.6 References

Beer, Armin, and Stefan Mohacsi. (2008). "Efficient Test Data Generation for Variables with Complex Dependencies." IEEE ICST 2008, in Lillehamer, Norway.

ESA Earth Observation Link. (no date). Available at http://earth.esa.int/EOLi/EOLi.html.

Fraser, G., B. Peischl, and F. Wotawa. (2007). "A Formal Model for IDATG Task Flows." SNA-TR-2007-P2-03.

IDATG and SiTEMPPO homepage. Available at http://at.atos.net/sitemppo.

Mohacsi, Stefan. (2005). "A Unified Framework for API and GUI Test Automation." *Proceedings of the 6th ICSTEST*, Düsseldorf, Germany.

Mohacsi, Stefan. (2003). "Minimizing Test Maintenance Costs through Test Case Generation." *Proceedings of the 4th ICSTEST*, Köln, Germany.

Mohacsi, Stefan. (2006). "Test Automation in Space Projects." *Proceedings of the 5th QA&Test*, Bilbao, Spain.

Mohacsi, Stefan, and Johannes Wallner. (2010). "A Hybrid Approach for Model-based Random Testing." *Proceedings of the 2nd International Conference on Advances in System Testing and Validation Lifecycle*, Nice, France.

Softnet Austria Competence Network homepage. Available at www.soft-net.at/.

9.7 Acknowledgments

The authors want to thank their partners at ESA, in particular Andrea Baldi and Eduardo Gomez, for the excellent cooperation and the opportunity to work in this fascinating domain.