PhD-FSTC-2016-13
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defense held on 22/04/2016 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

by

DANIEL DI NARDO

MODEL-BASED TEST AUTOMATION STRATEGIES FOR DATA PROCESSING SYSTEMS

DISSERTATION DEFENSE COMMITTEE

DR-ING LIONEL BRIAND, Dissertation Supervisor
Professor, University of Luxembourg

DR-ING YVES LE TRAON, Chairman
Professor, University of Luxembourg

DR FABRIZIO PASTORE, Vice-Chairman
University of Milano-Bicocca, Milan, Italy

DR ROBERT FELDT, Member
Professor, Blekinge Institute of Technology, Karlskrona, Sweden

DR HÉLÈNE WAESELYNCK, Member
Professor, LAAS - Laboratory for Analysis and Architecture of Systems, Toulouse, France

Abstract

Data processing software is an essential component of systems that aggregate and analyse real-world data, thereby enabling automated interaction between such systems and the real world. In data processing systems, inputs are often big and complex files that have a well-defined structure, and that often have dependencies between several of their fields. Testing of data processing systems is complex. Software engineers, in charge of testing these systems, have to handcraft complex data files of nontrivial size, while ensuring compliance with the multiple constraints to prevent the generation of trivially invalid inputs. In addition, assessing test results often means analysing complex output and log data. Complex inputs pose a challenge for the adoption of automated test data generation techniques; the adopted techniques should be able to deal with the generation of a nontrivial number of data items having complex nested structures while preserving the constraints between data fields. An additional challenge regards the automated validation of execution results.

To address the challenges of testing data processing systems, this dissertation presents a set of approaches based on data modelling and data mutation to automate testing. We propose a modelling methodology that captures the input and output data and the dependencies between them by using Unified Modeling Language (UML) class diagrams and constraints expressed in the Object Constraint Language (OCL). The UML class diagram captures the structure of the data, while the OCL constraints formally describe the interactions and associations between the data fields within the different subcomponents.

The work of this dissertation was motived by the testing needs of an industrial satellite Data Acquisition (DAQ) system; this system is the subject of the empirical studies used within this dissertation to demonstrate the application and suitability of the approaches that we propose.

We present four model-driven approaches that address the challenges of automatically testing data processing systems. These approaches are supported by the data models generated according to our modelling methodology. The results of an empirical evaluation show that the application of the modelling methodology is scalable as the size of the model and constraints was manageable for the subject system.

The first approach is a technique for the automated validation of test inputs and oracles; an empirical evaluation shows that the approach is scalable as the input and oracle validation process executed within reasonable times on real input files. The second approach is a model-based technique that automatically generates faulty test inputs for the purpose of robustness testing, by relying upon generic mutation operators that alter data collected in the field; an empirical evaluation shows that our automated approach achieves slightly better instruction coverage than the manual testing taking place in practice. The third approach is an evolutionary algorithm to automate the robustness testing of data processing systems through optimised test suites; the empirical results obtained by applying our search-based testing approach show that it outperforms approaches based on fault coverage and random generation: higher coverage is achieved with smaller test suites. Finally, the fourth approach is an automated, model-based approach that reuses field data to generate test inputs that fit new data requirements for the purpose of testing data processing systems; the empirical evaluation shows that the input generation algorithm based on model slicing and constraint solving scales in the presence of complex data structures.

Acknowledgements

This dissertation represents the culmination of a journey I started four years ago. A great many people have contributed to its publication and I would like to express my gratitude to all those who helped make it a reality.

I would like to thank my supervisor, Lionel Briand, for all his encouragement and support throughout my PhD studies. I am grateful to have had the opportunity to have worked with and learned from one of the top academics in the field.

I would like to thank my co-supervisor, Fabrizio Pastore, for his almost daily advice on how to conduct research and write good research papers. I never ceased to be amazed by his postive energy, stamina and work ethic.

I would like to thank my initial co-supervisor, Nadia Alshahwan, for helping me to get started with my research and for encouraging me to trust in my abilities.

I would like to thank SES for supporting my work and providing the case study system that was the subject of my empirical studies. In particular, I would like to thank Vincent Masquelier, Tomislav Nakić-Alfirević, David Valcárcel Romeu, and Marko Pecić for providing me with technical assistance and guidance related to the case study system. I would additionally like to thank Raul Gnaga, Frank Zimmer, and Romain Cloos for their valuable feedback and assistance throughout my PhD.

I would like to express my gratitude for all the friendships I have formed with my colleagues in the Software Verification and Validation Lab. My colleagues not only provided me with helpful advice along the way but also contributed to a warm work environment.

Finally, I would like to thank my mother and father for their lifelong support and for always encouraging to me to pursue my dreams.

Supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03 and FNR 4082113).

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface.
APID Application Process Identifier.
AST Abstract Syntax Tree.

CADU Channel Access Data Unit.
CCSDS Consultative Committee for Space Data Systems.
CFG Context-Free Grammar.
CPU Central Processing Unit.
CRC Cyclic Redundancy Check.

DAQ Data Acquisition.

EMF Eclipse Modelling Framework.
ESA European Space Agency.

GPSR Global Positioning System Receiver.
GUI Graphical User Interface.

HTML HyperText Markup Language.

ISP Instrument Source Packet.

JVM Java Virtual Machine.

MBT Model-Based Testing.
MDE Model-Driven Engineering.
MPDU Multiplexing Protocol Data Unit.
MSI MultiSpectral Instrument.

NASA National Aeronautics and Space Administration.

OCL Object Constraint Language.
OMG Object Management Group.

RAM Random-Access Memory.
RQ Research Question.
RSA Rational Software Architect.
RT-STPS Real-time Software Telemetry Processing System.

SAR Synthetic Aperture Radar.

SBSE Search-Based Software Engineering.
SBST Search-Based Software Testing.
SES Société Européenne des Satellites.
SUT System Under Test.

UML Unified Modeling Language.

VCDU Virtual Channel Data Unit.
VCID Virtual Channel Identifier.

XML Extensible Markup Language.

Chapter 1

Introduction

1.1 Context

This dissertation presents a set of approaches based on data modelling and data mutation to automate the testing of data processing systems. The work presented in this dissertation has been done in collaboration with Société Européenne des Satellites (SES) [SES, 2016], a world leading satellite operator, based in Luxembourg. Data processing software is an essential component of systems that aggregate and analyse real-world data, thereby enabling automated interaction between such systems and the real world. Examples are search engines that return stock quotes [Yahoo!, 2016], web applications that show real-time airplane positions [FlightRadar24, 2016], and phones able to translate in real time the words of a road sign [Schroeder, 2010].

In data processing systems, inputs are often complex files that have a well-defined structure, and that often have dependencies between several of their fields. Software engineers, in charge of testing these systems, have to handcraft complex data files, while ensuring compliance with the multiple constraints to prevent the generation of trivially invalid inputs. In addition, assessing test results often means analysing complex output and log data.

Complex inputs pose a challenge for the adoption of automated test data generation techniques because the adopted techniques should be able to deal with the complex constraints that exist between data fields; for example, random data generation approaches cannot be easily adopted because they are likely to produce test cases that are immediately rejected by the system and, therefore, are not effective in testing diverse system behaviour. An additional challenge regards the automated validation of execution results. Software engineers often take advantage of the fact that outputs are saved in log files, which can be processed to validate results. Validation is partially automated through scripts developed for each test input. Scripts check for specific output messages expected for a given input (i.e. the test oracles are written manually); such an approach requires a high development effort per test case and should the system specifications change, the scripts must be reassessed to ensure they are still valid.

In our approach, we propose to model the input and output data and the dependencies between them by using Unified Modeling Language (UML) class diagrams and constraints expressed in the Object Constraint Language (OCL), following a specific methodology. The UML class diagram cap-

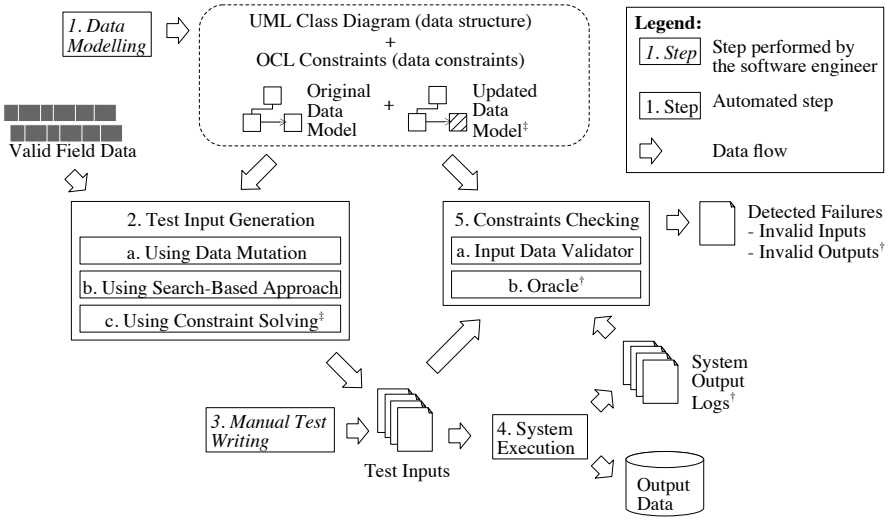


Figure 1.1. Overview of the different approaches developed for this dissertation.
†Constraints checking validates test input data only, if no system output logs are available.
‡Test input generation using constraints solving is only applied when new data requirements have been modelled.

tures the structure of the input, configuration and output data in a precise and detailed way, while the OCL constraints formally describe the interactions and associations between the data fields within the different subcomponents. The UML class diagram and the OCL constraints are the data model of the system. To build these models, the requirements and the domain knowledge of the system stakeholders is used without the need for access to the source code. This makes the approach black-box and, therefore, well suited for conditions commonly found in industry where parts of systems are outsourced or subcontracted, such as the case with our industry partner SES. We make use of an SES satellite Data Acquisition (DAQ) system to motivate the work within this dissertation as well as to demonstrate the application and suitability of the approaches that we propose.

The main objective of the data model in this project is to facilitate an end-to-end automated testing process. This includes input data validation, oracle checking, test input generation, and also includes activities for test suite optimisation (i.e. selection of the minimal number of test cases that allows for the achievement of the testing goals).

Fig. 1.1 presents an overview of the different approaches presented in this dissertation to enable automated testing and how they fit into the development process. Data modelling (step 1) must first be performed manually to create a data model that corresponds to the System Under Test (SUT). This data model is used to drive the various automated techniques proposed within this dissertation. Given a data model and valid field data, we can automatically generate system test inputs (step 2). Our techniques generate system test inputs: by using generic mutation operators (step 2a), by additionally using a search-based approach (step 2b), and by using constraint solving (step 2c)—this approach is used in cases where the available field data is out of date with respect to the system implementation; in this case, additional modelling must be performed to capture the updated data model. Our solution can also work with manually written test inputs (step 3). Given test inputs, the SUT can be executed (step 4) to generate system output logs. Given the test inputs, system output logs, and the data model, constraints checking (step 5) can be performed to check the validity of the input data (step 5a) and to act as a test oracle (step 5b); the constraints checker reports any detected failures.

1.2 Research Contributions

In this dissertation, we addressed the challenges of testing data processing systems. Specifically, we make the following contributions:

1. A precise data modelling methodology, dedicated to modelling the structure and content of complex input/output data stores (e.g. files), and their relationships for systems where the complexity lies in these elements, such as DAQ systems. This contribution has been published in a conference paper [Di Nardo et al., 2013] and is discussed in Chapter 3.
2. A technique for the automated validation of test inputs and oracles based on the the application and tailoring of Model-Driven Engineering (MDE) technologies in the context of data processing systems. This contribution has been published in a conference paper [Di Nardo et al., 2013] and is discussed in Chapter 4.
3. A model-based technique that automatically generates faulty test inputs for the purpose of robustness testing, by relying upon generic mutation operators that alter data collected in the field. This contribution has been published in a conference paper [Di Nardo et al., 2015b] and is discussed in Chapter 5.
4. An evolutionary algorithm to automate the robustness testing of data processing systems through optimised test suites. This contribution has been published in a conference paper [Di Nardo et al., 2015a] and is discussed in Chapter 6.
5. An automated, model-based approach that reuses field data to generate test inputs that fit new data requirements for the purpose of testing data processing systems. This contribution has been submitted for publication in a journal [Di Nardo et al., 2016] and is discussed in Chapter 7.
6. For each of the techniques, an empirical evaluation aimed at assessing its suitability in the context of data processing systems.

1.3 Organisation of the Dissertation

Chapter 2 contains a description of the data processing system, our case study system, whose testing requirements motivated the work and the proposed approaches of this dissertation.

Chapter 3 describes the model-based methodology proposed to capture the structure and content of the complex input and output data associated with data processing systems and used to drive the test related approaches in this dissertation. It also evaluates the feasibility of applying the modelling methodology that captures the structure and content of complex input/output data.

Chapter 4 describes an approach to support the validation of test inputs and the checking of test oracles in the context of data processing systems. An empirical evaluation is performed on an industrial data processing system to validate the applicability and scalability of our proposed approach.

Chapter 5 describes a model-based technique that automatically generates faulty test inputs for the purpose of robustness testing, by relying upon generic mutation operators that alter data collected in the field. The approach uses UML stereotypes and OCL queries to configure the mutation operators to implement a fault model of the SUT. An empirical evaluation is performed on an industrial data processing system to evaluate the effectiveness of our proposed approach.

Chapter 6 describes an evolutionary algorithm to automate the robustness testing of data processing systems. The approach makes use of four fitness functions (model-based and code-based) that enable the effective generation of robustness test cases by means of evolutionary algorithms. An extensive study is performed to observe the effect of fitness functions and configuration parameters on the effectiveness of the approach using an industrial data processing system as case study.

Chapter 7 describes an automated, model-based approach to modify field data to fit new data requirements for the purpose of testing data processing systems. The approach uses a scalable test generation algorithm based on data slicing that allows for the incremental invoking of a constraint solver to generate new or modified parts of the updated field data. An industrial empirical study is performed demonstrating (1) scalability in generating new field data and (2) coverage of new data requirements by generated field data in addition to a comparison with expert, manual testing.

Chapter 8 discusses related work.

Chapter 9 provides a description of the core software implementations developed for the empirical studies of this dissertation.

Chapter 10 summarises the thesis contributions and discusses perspectives on future work.

Chapter 2

Motivating Industrial Case Study

This section introduces the industrial data processing system that motivated our research and that we used to evaluate the approaches presented in this dissertation. Our case study system is a DAQ system developed at SES [SES, 2016], a world leading satellite operator, to collect and process satellite data. The particular system we consider has been developed to process the transmission data for the European Space Agency (ESA) Sentinel series of satellites [ESA, 2016]. We will refer henceforth to this system as SES-DAQ.

There are multiple Sentinel mission types; each mission type is intended to provide different Earth observations (e.g. climate or vegetation). The satellites related to each mission type have specialised instrumentation. Accordingly, the content of the data transmitted for the different missions varies.

Fig. 2.1 gives a high level overview of the SES-DAQ. The system receives satellite transmissions and processes them according to configuration settings. It first checks for transmission errors. If the

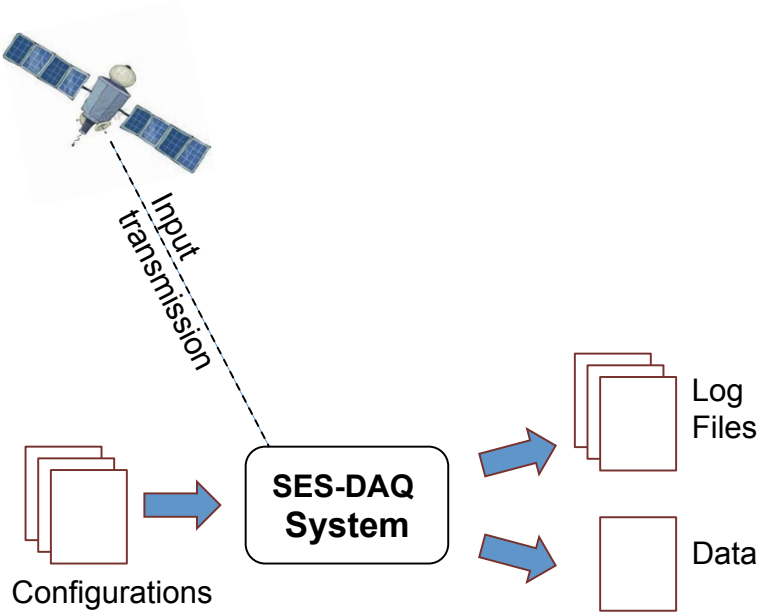


Figure 2.1. High-level overview of the data acquisition system that receives satellite transmissions.

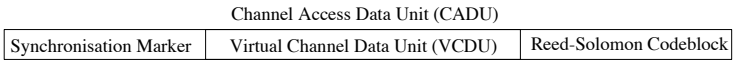


Figure 2.2. Channel access data unit (CADU) [CCSDS, 2011]. A satellite transmission consists of a sequence of CADUs.

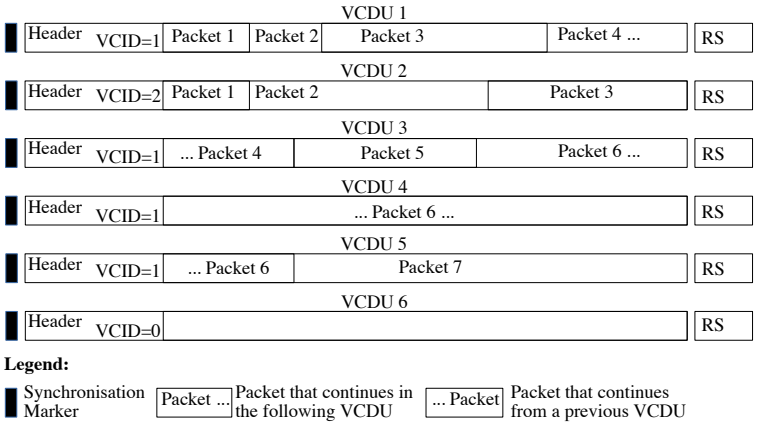


Figure 2.3. A simplified example of the transmission data processed by the SES data acquisition system. The keyword *VCID* indicates the virtual channel each *Virtual Channel Data Unit (VCDU)* belongs to; RS is the Reed-Solomon Codeblock.

transmission appears valid then the system extracts the actual packets of data to later recombine them into the original data. The system outputs the data received as well as log files capturing errors and events of interest that occurred in the transmission.

2.1 Background on Satellite Transmissions

Satellite transmissions are a good example of data with a complex structure and multiple relationships among its fields. The Consultative Committee for Space Data Systems (CCSDS) is tasked with developing standards pertaining to space related communications [CCSDS, 2016].

Satellites communicate with ground systems via one or more physical channels in one or both directions. A physical channel corresponds to transmitted bitstreams of data, for example, transmitted from a satellite to a ground (i.e. DAQ) system. Bitstreams are transmitted as a sequence of Channel Access Data Units (CADUs) [CCSDS, 2011]. Fig. 2.2 shows a single CADU. Each CADU consists of: a Synchronisation Marker, a specific bit pattern used to identify the start of the CADU; a Virtual Channel Data Unit (VCDU) that contains the actual data being transmitted (for a particular virtual channel); and a Reed-Solomon Codeblock, used for error correction.

Fig. 2.3 shows a simplified example of a satellite transmission processed by SES-DAQ. Each transmission consists of a sequence of *VCDUs* [CCSDS, 2006] (each VCDU is contained within a CADU). Each *VCDU* contains a *Header* and a packet zone that contains a sequence of *Packets* [CCSDS, 2003]. Each *VCDU* is preceded by a *Synchronisation Marker* and followed by a Reed-Solomon Codeblock. The *VCDUs* in a transmission may belong to different virtual channels; a unique Virtual Channel Identifier (VCID) number identifies each virtual channel. *VCDUs* can be active (i.e. they transmit data) or idle (i.e. they do not transmit anything). A special VCID is used to transmit idle data. Fig. 2.3 shows a transmission with six *VCDUs*: four belonging to virtual channel 1; one belonging to

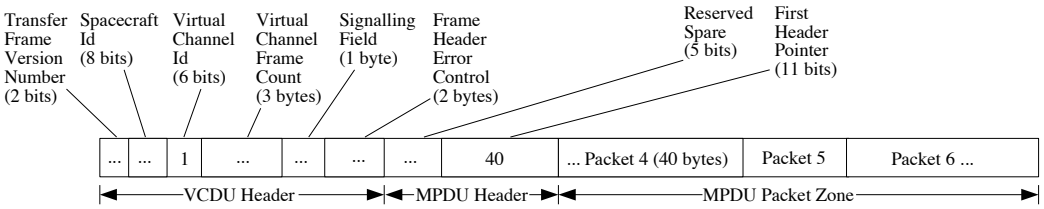


Figure 2.4. Structure of a Virtual Channel Data Unit (VCDU) [CCSDS, 2006]. The structure begins with six fields that make up the VCDU header, followed by the two-field Multiplexing Protocol Data Unit (MPDU) header, followed by the MPDU packet zone. The example shown in the figure corresponds to VCDU 3 in Fig. 2.3.

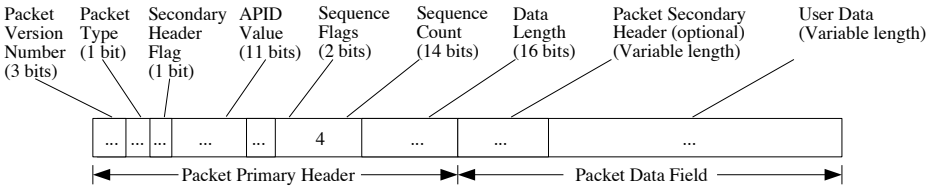


Figure 2.5. Structure of a space packet [CCSDS, 2003]. The structure begins with seven fields that make up the primary packet header, followed by the packet data field (an optional packet secondary header and user data). Note: APID, Application Process Identifier.

virtual channel 2; and one belonging to virtual channel 0, which indicates idle data.

Each *VCDU* has a predefined length. *Packets* can have varying lengths and they may span across multiple *VCDUs*. *Packet 4* in Fig. 2.3 spans over *VCDU 1* and *VCDU 3*, while *Packet 6* spans over *VCDU 3*, *VCDU 4*, and *VCDU 5*.

Fig. 2.4 shows the structure of a *VCDU* in more detail. A *VCDU* consists of a *VCDU Header*, a *Multiplexing Protocol Data Unit (MPDU) Header*, and an *MPDU Packet Zone*. Some of the fields of the *VCDU* are characterised by complex constraints. For example, the *Virtual Channel Frame Count* is used to keep an ordering between *VCDUs* that belong to a same channel: the *Virtual Channel Frame Count* of a *VCDU* must be greater by one than the *Virtual Channel Frame Count* of the previous *VCDU* on the same virtual channel.

A more complex constraint pertains to the field *First Header Pointer*, whose value depends both on the *VCDU* type and on the transmitted data. The *First Header Pointer* contains an offset to the beginning of the first *Packet* that starts in the packet zone of the *VCDU*. In idle *VCDUs*, the *First Header Pointer* should be filled with an expected idle bit pattern. In active *VCDUs*, the *First Header Pointer* should contain the offset to the beginning of the first new *Packet*. For example, Fig. 2.4 shows that in *VCDU 3* the *First Header Pointer* is set to 40 because the first 40 bytes in the *Packet* sequence of *VCDU 3* belong to the end of *Packet 4*, whose transmission started in *VCDU 1*. The first *Packet* starting in *VCDU 3* is thus *Packet 5*, which starts at byte 40 of the packet zone.

There are constraints on other data fields as well. Fig. 2.5 shows the structure of a space *Packet* (or Instrument Source Packet (ISP)); an *ISP* consists of a *Packet Primary Header* and a *Packet Data Field*. The 16-bit field *Data Length* is used to specify the length of the *Packet Data Field*: *Data Length* is equal to the number of octets in *Packet Data Field* minus 1.

2.2 SES-DAQ System

The SES-DAQ system is a high-speed, DAQ system that is developed and tested by two independent teams. It is a good example of a data processing system dealing with complex input and output data, written in Java. The system has 53.5 k lines of Java code (cyclomatic complexity is 8700) and runs on a 6-core Central Processing Unit (CPU) server with 6 GB of Random-Access Memory (RAM).

The system accepts as input one or two binary files that correspond to physical channel data (i.e. a sequence of CADUs). Each CADU contains a VCDU that contains one or more packets of data (packets can span multiple VCDUs). Error-correction information is stored at all levels of the data as all communications with the satellite are unidirectional, so the receiver cannot ask the sender to resend erroneous data.

Several configuration files are used to define how the input file should be processed (e.g. the valid Application Process Identifier (APID) values for the packets on a given virtual channel); these files are in the Extensible Markup Language (XML) and text formats. Four different log files report on the results of processing the input file (e.g. two of the logs report any errors occurring at the VCDU and the packet processing levels); the output logs are in the XML format.

Ongoing refurbishments are planned for the system for the foreseeable future.

2.3 Testing of SES-DAQ

The test suite for the system is currently created manually by highly experienced engineers, with domain expertise. There are 32 test cases that are approved by the client for the validation of the system, which address the SES-DAQ system requirements related to data acquisition and processing. These test cases use synthetic input data; some represent valid transmissions, while others represent typical patterns of events and errors in such systems.

These test cases check that the system performs as expected and captures these events and errors in the output log files. For example, a test case could simulate a transmission error where one of the counters is not in sequence. The input file for this test case contains an out of sequence counter and the expected result is that the output log file reports this event.

Each test case consists of a small input transmission file that is six CADUs in length (< 12 kB) and the relevant configuration files. The execution of test cases is currently automated using Maven 2 [Maven, Apache, 2016] (a project management tool that supports automated test execution) and Groovy [Groovy Community, 2016] (a Java-like Java Virtual Machine (JVM) dynamic language). The expected result is checked by using hard-coded assertions and checksum comparisons to previous execution results.

By examining the manually created system test cases and related documentation, we derived a fault model. Fault models are used to capture the specific faults that can occur for a given *behaviour description*, meant to satisfy a given specification [Pretschner et al., 2013]; given a fault model, test cases can be derived. In the case of SES-DAQ, the fault model pertains to the errors that can occur in the format of the satellite transmission input data. Transmission errors (i.e. data faults) may be due to

Table 2.1. Fault Model of SES-DAQ.

Fault	Description
Duplicate CADU/ISP	The same CADU/ISP appears twice in the transmission.
Missing CADU/ISP	A CADU/ISP is omitted during transmission.
Wrong Sequence	CADUs/ISPs are sent out of order.
Incorrect Identifier	Several transmission data fields have fixed values, for example fields identifying the transmitting satellite. Hardware/software errors may assign incorrect identifiers.
Incorrect Checksum	Hardware/software errors may result in an incorrect checksum for an ISP or VCDU header.
Incorrect Counter	Counters are used to track ISP or VCDU ordering. Hardware/software errors may assign incorrect counter values.
Flipped Data Bits	Physical channel noise may flip one or more bits in the data transmission.

Note: CADU, Channel Access Data Unit; ISP, Instrument Source Packet; VCDU, Virtual Channel Data Unit.

either the physical channel (atmospheric noise or hardware antenna errors can fall into this category) or software failures at the system level (e.g. the miscalculation of header values). Table 2.1 shows a fault model for SES-DAQ.

2.4 System Testing Challenges

The complexity and constraints of satellite transmission data make the testing of SES-DAQ extremely complex and expensive. Handcrafting test cases is expensive and error-prone. Consider a software engineer who needs to check if SES-DAQ properly detects errors in the presence of a wrong sequence number in the *Virtual Channel Frame Count* of a *VCDU*. The software engineer will need to generate multiple test cases (i.e. multiple faulty input data) in order to cover all the combinations of the input data features: the number of channels in a transmission (one, two, or more), the presence of idle channels, the type of *VCDU* affected by the fault (idle or active), and the presence of *Packets* spanning across multiple *VCDUs*. The testing process is not only complicated by the number of the generated test cases, but also by the complexity of the constraints that must hold. In fact, to generate data affected by a single fault, the software engineer must create data with a wrong *Virtual Channel Frame Count* but with other fields that conform to all the other constraints; for example, the proper values for the field *First Header Pointer* must be set.

The execution of the system test suite is currently automated; however, the expected results are checked by using hard-coded assertions and comparisons of checksums to previous execution results. Consider that the SES-DAQ system specification might change over time concerning the treatment of existing data. For example, an input that previously generated an error in the log, might be considered acceptable in a subsequent release. System changes require a reassessment and possible refactoring of the Groovy scripts related to testing that serve as test oracles (one for each individual test case).

Additionally, the satellite transmission specification related to the SES-DAQ is subject to updates. For example, as new Sentinel missions are deployed, new packet types must be accommodated within the bytestream data. During development, the packet specifications might be subject to change requiring frequent updates to any representative test input data created for testing during the software refurbishment process.

The amount of time and resources allocated for testing are limited; raising the client's confidence in the system is crucial. This raises the need for an automated test generation and oracle checking approach and also emphasises the importance of automatically generating effective test cases. Other systems in the company have similar characteristics and, therefore, any methodologies and tools developed for this system can be applied in future to several other products.

Chapter 3

Modelling Methodology

Given the challenges of our case study context, we are motivated to devise a modelling methodology to support the test automation of data processing systems having complex input and output structures, with complex mappings between the two. We pursued a model-driven strategy for the various approaches proposed within this dissertation.

By modelling the input and output data structures and constraints, we are effectively creating a specification of the system data. In fact, capturing data specifications could already be a practice in place at a given company. This chapter describes how data modelling should be conducted such that the data model can also be used for testing automation.

We use this modelling methodology to support automated input validation (e.g. to check the validity of the manually written test inputs for the system). Similarly, we support test oracle automation (e.g. to check that the resulting system output resulting from a test input is correct).

As the writing of meaningful tests is a time consuming endeavour, we support automatic test generation. By extending the modelling methodology, we make use of data mutation operators that work on existing valid field data files to create faulty test inputs. In the case where a system is updated to handle new data requirements, we support automatic input generation techniques to create meaningful test inputs conforming to the newly defined data structures.

To achieve our objectives, a modelling notation is not sufficient: a precise methodology supporting the modelling objectives is necessary. A modelling methodology is a practical way to provide more precise semantics to the selected modelling notation in a specific context, in our case UML class diagrams. We initially devised a modelling methodology whose goal is to capture the structure and constraints of system related data (i.e. input, configuration, and output data).

This chapter highlights the following research contribution:

1. A precise test modelling technology, dedicated to modelling the structure and content of complex input/output data stores (e.g. files), and their relationships for systems where the complexity lies in these elements, such as DAQ systems. The modelling methodology is based on UML class diagrams and the OCL, and gives semantics to the notation through a practical methodology dedicated to the model-based testing of data processing systems.

2. An empirical evaluation on an industrial data processing system to validate the applicability and scalability of our proposed approach.

We devised a methodology to model the input and output data of a system together with their dependencies using UML class diagrams and OCL [OMG, 2015] constraints. Our first objective was to automate test input validation and oracles. The input/output models capture the structure of complex input, configuration and output data in a precise way, including constraints that formally describe the interactions and associations between their different subcomponents. To build the input/output models, the requirements and domain knowledge of the system's stakeholders is elicited, without the need to access the source code, making the approach black-box. The modelling process is in practice iterative; it might be necessary to adjust the model when defining the constraints to add an attribute or operation that is necessary for the constraints.

Simply modelling a system's data is not sufficient. In order to provide automation that fulfils the goals of our modelling methodology, we must be able to take the real-world data associated with a system (e.g. the binary and text files of interest) and convert these into instantiated objects (based on our UML representation) in order to perform the various testing related activities that we propose (i.e. input validation, oracle checking, and test input generation).

Our examples in this chapter focus on the modelling of the satellite binary transmission file. Appendix A provides additional details on the modelling of the configuration files and output files associated with the SES-DAQ.

The chapter proceeds as follows. Section 3.1 presents the different applications of the modelling methodology we propose for the testing and validation of data processing systems. Section 3.2 describes the steps that we propose to model the input and output data structures and Section 3.3 describes how we define the constraints. Section 3.4 describes how UML stereotypes can be used to augment the modelling methodology to support the use of automated generic data parsers. Section 3.5 gives an overview of other enhancements made to the data model to enable automated test input generation. Section 3.6 presents the empirical results obtained. Finally, Section 3.7 concludes the chapter.

3.1 Modelling Methodology Applications

Working with the domain experts and engineers, we identified four applications of this modelling methodology, namely test design, test oracle, specifications refinement, and run-time verification.

Test design

Due to the complexity of the input file structure and dependencies between fields, writing test cases for such systems might be challenging. The model and constraints can help the tester design test cases: Artificial input files created for testing can be debugged using the model and constraints.

Test oracle

The constraints on the input/output can be used to automate the oracle. The output log files can be very large and complex making manual checking of the expected output time-consuming and impractical. An automated oracle could reduce the oracle cost allowing testers to execute a larger number of test cases in the time allocated for testing.

Specifications refinement

Real transmission files can be tested on the model and input constraints to validate the model itself, which is a specification of the system. As previously mentioned, in systems such as DAQ systems, it might be hard to get detailed specifications of the expected input files early in the development of the system. Moreover, in practice such specifications tend to change, often in implicit ways, during development and testing. Therefore, applying real transmission files to the model could be a way to partially reverse engineer or refine these specifications. The real transmission files are expected to be valid; therefore, violations of the input constraints and failures when loading the transmission file into the model might indicate faults in the model and constraints.

Run-time verification

In a similar way, using real transmission files while focusing on the constraints that map the input to the output could help to identify faults. In real transmission files, the expected output is unknown (i.e. which events should be captured in the output logs). However, input/output constraint violations could indicate faults in the specifications of the system or unhandled events.

3.2 Capturing Data Structures

According to our methodology, the software engineer needs first to decide what needs to be modelled (i.e. what files or attributes). For each of the files that are selected for modelling, the first step is to understand the structure of the file and decide what information needs to be captured. A general guideline is to capture every element in the files that is needed for automated testing. For example, if some fields can only be assigned a limited number of predefined values, this needs to be captured in the model as an enumerated data type or a property that captures the allowed range. Class diagrams, the most common type of UML diagrams, are then created that capture the structure and the information that we decide to include.

The structure of the file is modelled using a tree structure of classes and composition relationships combined with regular associations and generalisations. The structure of the class diagram can be dictated by either the logical or physical structure of the component we want to model. For example, the fields of a file can be logically divided into subgroups that help understand the purpose of each part of the file. On the other hand, some files are divided into subparts physically; for example, if the file is structured using XML.

Figs. 3.1 and 3.2 depict a simplified, sanitised model of the input data of our case study system. These models are the structural representation of the CADU bytestream data (shown in Figs. 2.2, 2.4, and 2.5). Fig. 3.1 shows that satellite transmission data (*TransmissionData*) is composed of one

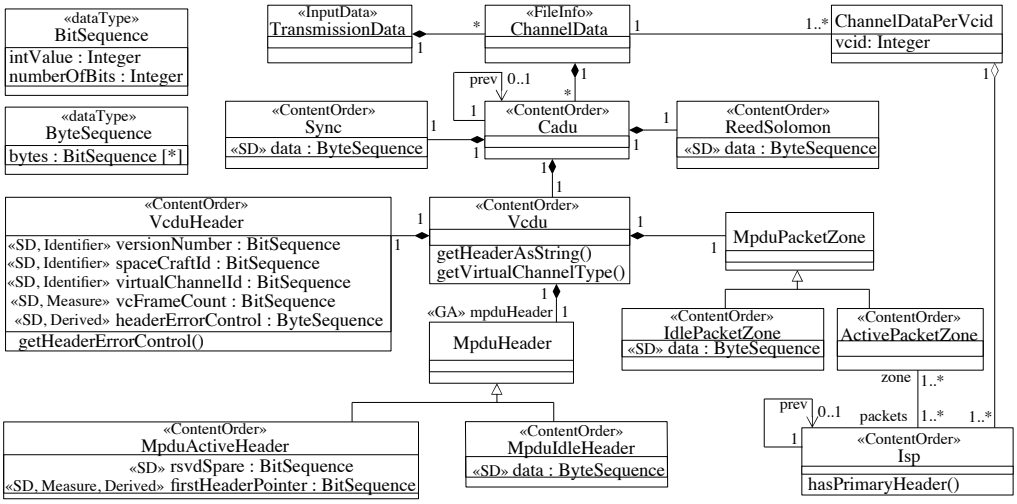


Figure 3.1. Simplified model example for the input Channel Access Data Unit (CADU) data in the case study system. Note: SD, StreamData; GA, GeneralisedAttribute.

or more physical channels (*ChannelData* elements), which in turn are composed of a sequence of CADUs (*Cadu* elements). Fig. 3.2 shows the representation of the space packets (*Isp* elements) associated with the MPDU packet zones of an active virtual channel (*ActivePacketZone* elements, in Fig. 3.1). This model will be used as an example throughout this section to illustrate the different modelling elements that we use.

Each field that is composed of subfields is represented in the model by a class. The relationship between classes is modelled as follows:

1. A containment, which is the relationship between a field and its subfields, is represented in the model by a composition (an edge with a black diamond). For example, in the model in Fig. 3.1, the field *Cadu* is composed of *Sync*, *Vcdu* and *ReedSolomon* data.
2. In some cases, a field can be one of several alternative fields (i.e. the field has multiple different definitions but only one of them can be true at a time). This case is represented in the model by a generalisation (edge with an unfilled triangle). For example, in Fig. 3.1, the field *MpduPacketZone* (a superclass) can be one of two subtypes: an *IdlePacketZone* or an *ActivePacketZone* (modelling either an idle bit pattern or the presence of actual data, respectively).
3. If a field can have several instances of the same subfield, this is represented in the model by a *multiplicity* on the edge that connects the field to its subfields. A multiplicity is a number or range representing the minimum and maximum number of times the subfield can appear. For example, a *ChannelData* instance of Fig. 3.1 can contain zero or many *(*)* instances of *Cadu*.

Leaf fields are fields that do not have any subfields and hold an actual value. These fields are represented in the model by class attributes having non-class type values. Multiple related leaf fields may be contained within a same class; for example, *versionNumber*, *spaceCraftId*, *virtualChannelId*, *vcFrameCount*, and *headerErrorControl* together form a *VcduHeader*. The choices influencing the creation of classes and attributes can also be determined by readability or visibility considerations. For example, the synchronisation bytes of a *Cadu* could have been included as an attribute *sync* within the *Cadu* class. To emphasise their importance, the synchronisation bytes are rather contained within a distinct class, *Sync*, within the attribute *data*. Note that modelling a field as a class or attribute does

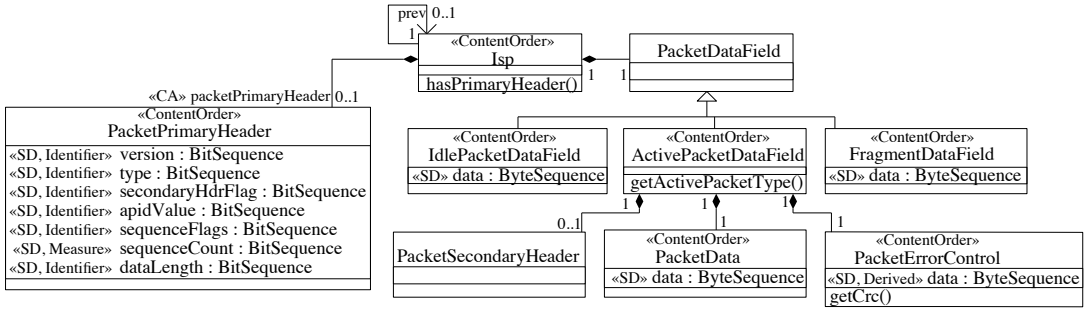


Figure 3.2. Simplified model example for the input Instrument Source Packet (ISP) data in the case study system. Note: SD, StreamData; CA, ConditionalAttribute.

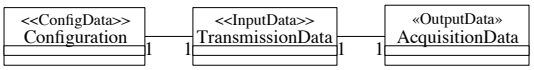


Figure 3.3. Associations between input, configuration and output class diagrams.

Table 3.1. Mapping of file information items to class elements.

File Item	Model Element	Example
Field	Class	ChannelData
Leaf Field	Attribute	spacecraftId
Containment	Composition	Cadu is composed of Sync, Vcdu, and ReedSolomon
Alternative Subcomponents	Generalisation	MpduPacketZone can either be an IdlePacketZone or a ActivePacketZone
Optional/Multi Subcomponents	Multiplicity	One ChannelData instance can have zero or many instances of Cadu
Dependency	Association	The association between TransmissionData and Configuration
Computation	Operation	getHeaderErrorControl

not affect the final test validation process or oracles.

Dependencies between fields or different files are represented in the model by associations (edges). These associations help in navigating the model to precisely specify a field when defining constraints. For example, Fig. 3.3 illustrates the associations between the input class diagram (Fig. 3.1) and the configuration (Fig. A.1 in Appendix A) and output (Fig. A.2 in Appendix A) class diagrams.

Finally, in some cases we need to define operations on some fields. These operations are not part of the data, as opposed to other items in the class diagram; they correspond to standard computations, normally provided by existing libraries, used to compute or check field values. These computations are represented in the model as an operation in the class that represents the relevant field. For example, the field *headerErrorControl* in the class *VcduHeader* has a value (calculated using *versionNumber*, *spaceCraftId*, and *virtualChannelId*), used for error detection and correction, that is generated using a complex, but standard, algorithm provided by a library that can not be expressed in OCL. Therefore, the operation *getHeaderErrorControl()* was created to allow testers to get the valid value of the Header Error Control— useful when performing test input validation (i.e. to check that the value of *headerErrorControl* is correct) or for validating that the correct output has been captured in the logs in the presence of this error.

```
1   context VcduHeader inv:
2
3   let
4       config : XsveRtStps =
5           self.vcdu.cadu.channelData.transmissionData.configuration.rtStpsConfig.xsveRtStps
6   in
7
8       config.vcduConfig->exists(x | x.vcid = self.virtualChannelId.intValue)
9       or
10      self.virtualChannelId.intValue = config.idleVcid
```

Figure 3.4. Example of a constraint on input and configuration used to validate test cases.

```
1   context Cadu inv:
2
3   let
4       frameCount : Integer = self.vcdu.vcduHeader.vcFrameCount.intValue,
5       prevFrameCount : Integer = self.prev.vcdu.vcduHeader.vcFrameCount.intValue,
6       transData : TransmissionData = self.channelData.transmissionData
7   in
8
9       self.vcdu.vcduHeader.virtualChannelId.intValue <>
10          transData.configuration.rtStpsConfig.xsveRtStps.idleVcid
11      and
12      not self.prev->isEmpty()
13
14      and
15
16      if prevFrameCount < 16777215
17          then frameCount <> prevFrameCount + 1
18      else prevFrameCount = 16777215 and frameCount <> 0
19      endif
20
21      implies
22
23      transData.acquisitionData.vcduReportData.vcduReportBody.vcduEvent
24          ->exists(i | i.eventType = VcduEvents::VIRTUAL_COUNTER_JUMP
25                  and i.currentVcduFrame.header = self.vcdu.getHeaderAsString() )
```

Figure 3.5. Example of a constraint on input, configuration and output data used to automate the oracle.

Table 3.1 provides a summary of the mapping between elements in the files and class elements. Appendix A provides additional details pertaining to the modelling of XML and text file data.

3.3 Capturing Data Field Constraints

Restrictions on data fields and relationships (i.e. dependencies) between different fields are modelled using OCL, a logical choice when using UML. OCL is widely supported by modelling engines, such as ECore [The Eclipse Foundation, 2013].

Constraints can be divided into two main groups based on their function: constraints on the inputs (including configurations) and constraints that capture relationships between the input and output.

The constraints on the inputs are used to check that input data is valid. For example, a constraint

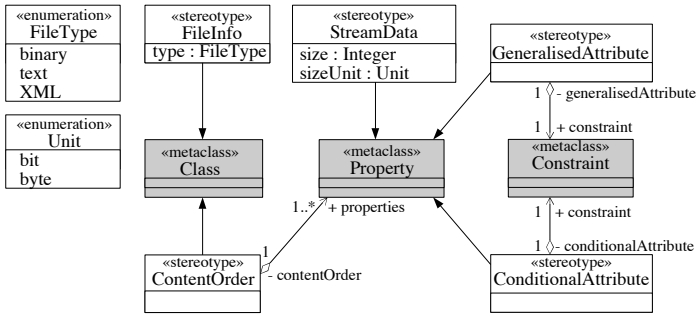


Figure 3.6. Portion of the custom profile that extends the UML metamodel to support automated parsing. The shaded elements in the figure represent UML metaclasses.

on the input data in Fig. 3.1 is that the value of *virtualChannelId* can only be one of the values defined in the configuration file. The OCL code in Fig. 3.4 precisely expresses this constraint; the field *virtualChannelId* has to exist in the list of *vcid* entries in the configuration file (line 8) or be equal to the *idleVcid* (line 10).

The constraints that define relationships between the input and output are used to verify that the result of executing a test case matches what is expected—acting as a test oracle. The OCL code in Fig. 3.5 expresses a constraint of this type; if the values of *vcFrameCount* are not in sequence for a certain *virtualChannelId* (lines 16-17), or in the case that *vcFrameCount* has reached its maximum allowed value (*16,777,215*) and the value of the next *vcFrameCount* is not zero (line 18), then a *VIRTUAL_COUNTER_JUMP* event should be recorded in the relevant output log file (*VcduReportData*, lines 23-25).

3.4 Augmenting the Model for Automated Parsing

The automation of the approaches proposed in this dissertation requires that we load the system data into memory as an instance of the data model. We therefore require the use of data parsers and object instantiators to create instantiated models that correspond to our data models. Creating custom parser software to support the parsing of specific data files can be an incredible burden on software engineers, especially in cases where specifications and file formats change frequently. Therefore, to minimise implementation effort, we propose the use of generic data parsers that are guided by extensions to our modelling methodology. In general, we believe that generic parsers could be implemented for different data types to support the adoption of the methodology in different contexts.

For example, we developed a parser that processes bytestreams of data and one that processes text files. In the specific case of the SES-DAQ, which receives as input the transmission data saved on the filesystem as bytestreams, we developed a generic bytestream parser that processes stereotypes and OCL expressions that were created to augment the data model to enable automated parsing (further details on the parser tool created for this project can be found in Chapter 9).

We enhanced the methodology to support the automatic data parsing of binary satellite transmission files by introducing four stereotypes: *«ContentOrder»*, *«StreamData»*, *«GeneralisedAttribute»*, and *«ConditionalAttribute»*. Fig 3.6 shows the portion of our UML profile where these stereotypes are defined.

```
1  context Vcdu::getVirtualChannelType() : VirtualChannelType body:
2
3      if self.vcduHeader.virtualChannelId.intValue =
4          self.cadu.channelData.transmissionData.configuration.rtStpsConfig.xsveRtStps.idleVcid
5        then VirtualChannelType::IDLE
6      else VirtualChannelType::ACTIVE
7      endif
```

Figure 3.7. OCL query to determine whether a virtual channel is active or idle.

The stereotype *«ContentOrder»* extends (via the arrow with the solid black triangle) the metaclass `Class` (i.e. the stereotype can be applied to the classes of our data model). When applied to a given class, this stereotype captures the parsing order of the attributes of the underlying class via the containment denoted by the role *properties*, part of the directed aggregation association between *«ContentOrder»* and `Property`. Note that the `Property` metaclass represents UML attributes. For example, the *«ContentOrder»* stereotype assigned to the class *Vcdu* in Fig. 3.1 indicates that the order in which the fields (i.e. attributes) of the class *Vcdu* appear in the bytestream is as follows: *vcduHeader*, *mpduHeader*, and *mpduPacketZone*.

The stereotype *«StreamData»* extends the metaclass `Property` (thus, the stereotype can be applied to the attributes of our data model). When applied to a given attribute, this stereotype enables software engineers to specify the number of bits or bytes a particular data field (represented by the attribute) occupies. The stereotype *«StreamData»* specifies (via its attributes) the *size* (an Integer) and the *sizeUnit* of the transmitted data (i.e. a bit or byte). For example, in Fig. 3.1, the *«StreamData»* stereotype attached to attribute value *virtualChannelId* sets the size of the represented data field to 6 bits; similarly, the *«StreamData»* stereotype assigned to attribute value *headerErrorControl* sets the size of the represented data field to 2 bytes. The bit representation of the data is then translated according to the format declared in the data model (a BitSequence or ByteSequence).

The stereotype *«GeneralisedAttribute»* extends the metaclass `Property`. This stereotype is associated with an OCL query (denoted by the association between stereotype *«GeneralisedAttribute»* and metaclass `Constraint`) that will be used during bitstream parsing to choose which class to instantiate from a given generalisation of classes (e.g. from a base class and its specialisations). For example, in Fig. 3.1, the *«GeneralisedAttribute»* stereotype applied to the attribute *mpduHeader* has an OCL query, *getVirtualChannelType()* defined in the context of the *Vcdu* class. This OCL query, presented in Fig. 3.7, returns a value that is used to determine whether an instance of *MpduActiveHeader* or *MpduIdleHeader* should be created and then assigned to the attribute *mpduHeader*.

The stereotype *«ConditionalAttribute»* extends the metaclass `Property`. This stereotype has an OCL query (denoted by the association between stereotype *«ConditionalAttribute»* and metaclass `Constraint`) that will be used during bitstream parsing to determine whether an optional data field should be created. For example, in Fig. 3.2, the *«ConditionalAttribute»* stereotype applied to the attribute *packetPrimaryHeader* has an OCL query, *hasPrimaryHeader()* defined in the context of the class *Isp*; the OCL query *hasPrimaryHeader()* returns a Boolean value that indicates whether or not an instance of *PacketPrimaryHeader* should be created and assigned to the attribute *packetPrimaryHeader*.

This section dealt specifically with the stereotypes necessary to parse the satellite bytestream data. There are other file types that we need to parse to be able to instantiate the entire data model. The

parser solution we developed (presented in Chapter 9), in fact, is able to call one of three generic parsers we developed to parse the files related to the SUT. Accordingly, we introduce a fifth stereotype, *«FileInfo»*, also shown in Fig 3.6, that the tool uses to determine how to proceed with parsing.

The stereotype *«FileInfo»* extends the metaclass `Class`. The stereotype is applied to each class within the data model that represents the root of the modelled data corresponding to the contents of a file. Stereotype *«FileInfo»* has one attribute, *type*, that indicates whether the data file associated with the modelled data is: (1) binary, (2) text, or (3) XML. For example, the first option, binary, refers to the transmission binary data covered in this chapter; in this case, *«FileInfo»* is applied to the class *ChannelData* in Fig. 3.1 to designate that the contents of the data having the root class *ChannelData* are of type binary.

Examples of the other two file type options, XML and text, can be found in Appendix A.

3.5 Other Enhancements to the Data Model

3.5.1 General support for automation

Given that we can define and instantiate our data model using the modelling methodology that has been augmented to support data parsing, some of tools designed to implement the approaches of this dissertation further required that we classify the subcomponents of the model instance according to the types of data they represent. Accordingly, we introduced the *«InputData»*, *«OutputData»*, and *«ConfigData»* stereotypes. Fig. 3.3 shows the application of these three stereotypes to the data model of the SES-DAQ.

3.5.2 Support for test input generation via data mutation

We enhanced the methodology to support automatic data mutation by using four UML stereotypes: *«InputData»*, *«Identifier»*, *«Measure»*, and *«Derived»*. Data mutation is used to support automatic test input generation. An overview of these stereotypes is given in Chapter 5.

3.5.3 Support for test input generation via constraint solving

This dissertation includes an approach for automatic test input generation in the presence of updated data models, and the methodology supports this. In particular, we introduced the stereotype *«Replacement»* to support test input generation in the presence of replaced classes. Details pertaining to the *«Replacement»* stereotype can be found in Chapter 7.

3.6 Empirical Evaluation

We performed an empirical evaluation of the modelling methodology aimed at addressing its scalability with respect to the effort involved in the modelling process. The modelling process must be scalable in practice, and only a realistic empirical evaluation can help us to make this assessment. The empirical evaluation aims to respond to the following research question:

RQ: How much effort is needed to produce the data model and constraints for a real system?

The size of the created model and constraints for an actual, representative data processing system is a surrogate measure for modelling effort, since actual effort is largely dependent on skills and experience and therefore not a generalisable measure. Of course, determining if the size of a model is acceptable is subjective. An assessment must therefore be made in light of typical system models and based on the experience with project engineers.

3.6.1 Subject selection

This empirical evaluation makes use of the SES-DAQ system described in Chapter 2.

3.6.2 Approach application and data collection

To perform the empirical evaluation, we first studied the SES-DAQ system using design and test documents and held several modelling sessions with the system testers and developers. The model and constraints for the system were created, in an iterative manner, following the modelling methodology defined in Sections 3.2 and 3.3.

For this evaluation, IBM Rational Software Architect (RSA) was used to create the UML class diagrams and the OCL constraints. Alternatively, there are free software packages available that could also be used (e.g. Papyrus UML).

3.6.3 Results

In this section, we discuss the results of the empirical evaluation to answer our research question. As mentioned before, the size of the model and constraints can be a surrogate measure to estimate the effort needed to follow our modelling methodology in a specific context. We report model size rather than modelling time because the time needed for modelling is largely dependent on the person's domain knowledge and expertise in modelling and in OCL, and is therefore expected to vary from context to context.

We count and report the number of classes, attributes, associations (which include compositions) and generalisations of our case study system in Table 3.2. The results show that the size of the model, with 68 classes overall, seems acceptable given typical system model sizes and the size of the SUT (53.5 KLOC). Of course, as mentioned before, such an assessment is subjective. Therefore, we also rely on the feedback of the system's developers and testers when they were presented with the final model and constraints; since the system is developed by third parties, the modelling methodology allows for a high-level view of input and output constraints. The feedback we received is that the size of the model and constraints are reasonable compared to the benefit of defining the constraints used for test validation and oracles.

We classify each constraint in the model of the case study system based upon the location(s) of the referenced system data (i.e. the location(s) of the constrained attributes within the model). There are two distinct categories for the constraints: (1) those involving only input and configuration data (the constraint of Fig. 3.4 is in this category), and (2) those involving input, configuration and output data (the constraint of Fig. 3.5 is in this category). The constraints of the first category are used for test validation, while those of the second category are used for the oracle. For each of the categories, we report the number of constraints, the total number of clauses in all constraints, the number of

Table 3.2. Size of the input, configuration and output models that were created for the case study system.

File	Classes	Attributes	Associations	Generalisations
Input	36	156	17	4
Configuration	9	30	6	1
Output	23	132	15	0
Total	68	318	38	5

Table 3.3. Information about constraints for the case study system classified by the files to which they apply.

File	# of Constraints	# of Clauses	# of Operations on Collections	# of Iterative Operations
Input/Configuration	27	84	20	7
Input/Configuration and Output	22	125	17	29
Total	49	209	37	36

operations on collections (e.g. $indexOf$) and the number of iterative operations (e.g. $forAll$). The results show (in Table 3.3) that the number of constraints is similar for the two categories of constraints (27 and 22). The results also show that the complexity, represented by the total number of clauses and iterative operations in constraints, mainly lies in the constraints that are used as the oracle (125 clauses compared to 84 and 29 iterative operations compared to 7).

Iqbal et al. show that they can obtain models of similar sizes for similar systems [Iqbal et al., 2012]. More specifically, they created models using UML extended with the Modeling and Analysis of Real-Time Embedded Systems (MARTE) profile to conduct four industrial case studies. Due to the differences in the modelling methodologies and on the problems addressed and the domains, we cannot directly compare the model compositions and sizes to that of this study. However, considering the size metrics reported gives us confidence that the size of our model and constraints is reasonable. One of their case studies used modelling notations similar to our own; the related model contained a total of 71 classes and 16 OCL constraints (as well as other elements). Similarly, Sabetzadeh et al. use SysML to model a real safety-critical SW/HW interface [Sabetzadeh et al., 2011]. Their design consisted of 194 elements having 186 relations and 57 attributes.

Answer to RQ:

The results show that *the size of the model and constraints is reasonable compared to typical system model sizes*. The data model created for this empirical study contained a total of 68 classes, 318 attributes, and 49 OCL constraints. Most significantly, *the cost of modelling was considered acceptable by the SES-DAQ system's engineers*, especially compared to the benefit of defining the constraints used for test validation and oracles.

3.6.4 Threats to validity

In this section, we discuss the threats to validity in this study following the standard classification of threats [Wohlin et al., 2000].

External threats

The external threats are related to the choice of case study system and the ability to generalise results. Although we only used one system in the empirical evaluation, it is representative of data processing systems. Our results might only be relevant in this application domain; nevertheless, this domain is important and widely used. Many other types of data processing systems have similar characteristics as they process very complex inputs, detect input errors, and extract data. In other words, our approach might also be applicable to any system where the complexity lies in the input, output and their mappings.

Construct threats

For studying scalability, we used the size of the model and constraints. As we mentioned before, we chose the size of the model and constraints instead of modelling effort because the latter can vary greatly based on the modeller's expertise. We reported all elements of the model and constraints that can be counted. Such data, which is representative of what can be approximately expected in practice, can then be used in context to estimate modelling effort.

3.7 Conclusion

In this chapter, we presented our proposed modelling methodology for capturing the data associated with data processing systems; this data typically has complex input and output structures having complex constraints between their fields. Many systems share these characteristics, including DAQ systems common in the satellite communications industry.

This dissertation, as further reported in our discussion of related work (in Chapter 8), is the first to provide a modelling methodology and an automation strategy dedicated to the automated testing of data processing systems. The results of our empirical evaluation show that the approach is scalable. The data model and constraints required to specify the input and output structures and their contents, as well as their dependencies, of a real data processing system (a satellite DAQ system) are of reasonable size and complexity. This data model was developed in collaboration with project engineers and no practical issues were raised regarding the modelling process.

Chapter 4

Automatic Test Input Validation and Oracles

This chapter presents a technique able to validate complex test inputs, and automate a test oracle (i.e. checking that the system outputs are valid given the inputs processed).

This technique is based on the data modelling methodology presented in Chapter 3 and complements the automatic test generation techniques presented in this dissertation. In the presence of automatically generated test cases, in fact, it is particularly important to have automated oracles. In general, test automation techniques may lead to thousands of test cases whose results might be difficult to verify by hand by software engineers; thus, a technique for test oracle automation is required.

In addition to this, the technique presented in this chapter can be used independently from the test input generation approaches presented in this dissertation. In particular, it can be used as an additional tool to validate system outputs in the presence of manually written test cases, to validate generic executions directly in the field, and to validate test inputs (i.e. to check if software inputs comply with specifications).

Since there may be changes or misunderstandings related to the purported specifications of the system, this is a way to validate both the provided input files and models. Furthermore, if artificial input files are created for testing purposes, their complexity makes them error-prone; these input files must therefore be checked.

This chapter highlights the following research contributions:

1. The application and tailoring of MDE technologies to support the validation of test inputs and the generation of test oracles in the context of data processing systems.
2. An empirical evaluation on an industrial data processing system to validate the applicability and scalability of our proposed approach.

This chapter is organised as follows: Section 4.1 presents an overview of the approach. Section 4.2 presents the empirical evaluation on a real industrial system together with a discussion of the results. Finally, Section 4.3 concludes the chapter.

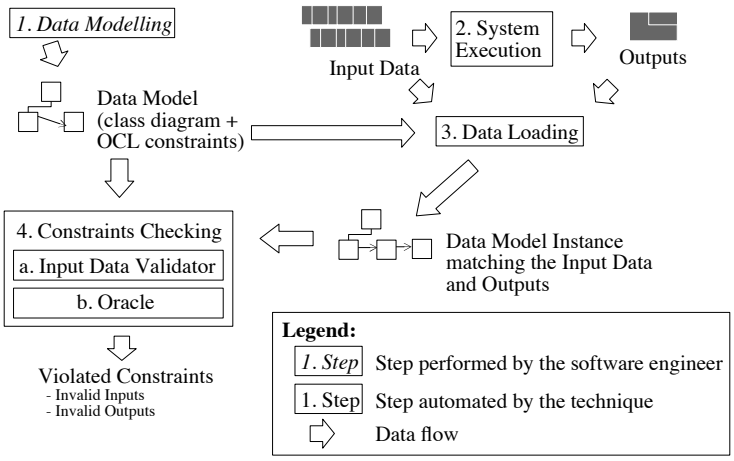


Figure 4.1. Steps for test input validation and test case oracle evaluation.

4.1 Description of the Approach

The approach works in four steps: data modelling, system execution (optional), data loading, and constraints checking. Fig. 4.1 provides an overview of the approach.

4.1.1 Data modelling

Data modelling refers to the activity performed by software engineers by following the technique described in Chapter 3 to create a *Data Model*. Data modelling consists in the definition of a class diagram that models the input and output data of the SUT. The diagram is augmented with a set of OCL constraints that capture properties of the input data, and relationships between input data and output data.

4.1.2 System execution

During the system execution step, input data (e.g. a test case input) is executed against the SUT. In our context, input data consists of input files containing the information modelled with the given data model of the system. In our experiments with SES-DAQ, we focus on the processing of input satellite transmission files. In principle, other data formats (e.g database entries) could be used as inputs. System outputs, in our context, files that fit the data model, are saved for further processing (i.e. to identify failures).

This system execution step is optional. In the case a software engineer is interested only in validating the content of input data (i.e. a test input, or a generic input of the program), system execution is not required since system outputs are not used by the technique. The test outputs generated by the system are processed to obtain an automated test oracle.

For the SES-DAQ, the manually created test suite, a collection of representative *Transmission Data* (described in Section 2.3), is executed. For each test case, we save the resulting *System Output Logs*.

4.1.3 Data loading

The input data and (optionally, if it was executed against the system) its corresponding output data are loaded as an instance of the given data model.

In the case of SES-DAQ, we load the contents of the transmission data, configuration files, and output logs associated with each executed SES-DAQ test case. The details of how this is implemented are provided in Chapter 9.

4.1.4 Constraints checking

Once the model has been instantiated, the input data and input/output data constraints can be automatically checked. More specifically, since we use OCL to express data constraints, we can rely upon existing OCL libraries to check if an instance of the data model (i.e. an instance of the class diagram) invalidates some of the given constraints.

We distinguish between two constraints checking components, the *Input Data Validator* and the *Oracle*. The former identifies invalid data in input data; the latter acts as an oracle and identifies outputs that invalidate constraints for the given input data (i.e. failures).

The *Input Data Validator* calls the input and configuration OCL constraints to validate the input data. Invalid constraints correspond to an input that does not fit the specification. An invalid constraint is reported to software engineers as a problem that might reflect one of the following three scenarios:

1. The input is wrong, which means that there is a failure in the component that generated the input; this allows, for example, to identify failures in the components working in the field that generated such data.
2. The input is wrong by design. Another context of use is that of test design; software engineers in fact can use the *Input Data Validator* to determine if manually written inputs are valid, or if they contain just the expected data faults they intended (test inputs, in fact, by design may contain erroneous data to test the capability of the SUT to cope with wrong inputs).
3. In the presence of field data known to be valid, the component allows for the identification of errors in the data model (i.e. in the data specification).

Similarly, the *Oracle* calls the constraints that map the input and configuration data to the output data to validate the results of executing the SUT. Invalid constraints mean that there is a failure. Depending on the constraints, invalid constraints may identify missing, invalid, or malformatted outputs (e.g. having a missing field value) given the input. For example, in the case of SES-DAQ, input/output constraints capture the outputs expected in the presence of an invalid input (e.g. as in Fig. 3.5). The presence of an invalid constraint indicates that, in the presence of an invalid input, a corresponding error message was not reported by the system. In the case of the constraint given by Fig. 3.5, in the event that two VCDUs on a same virtual channel are swapped, the constraint detects that *vcFrameCounts* are out of order and checks that the corresponding error(s) are reported in the relevant output log; if no such error(s) are logged, then a constraint violation is reported.

4.2 Empirical Evaluation

We empirically evaluated the effectiveness of the approach presented in this chapter by applying it on the SES-DAQ system. To be useful, our proposed approach must be scalable and applicable in practice, and only a realistic empirical evaluation can help us assess such criteria. To this end, we defined two research questions:

RQ1: Is the approach scalable in terms of execution time? How long does it take to validate input data and apply the oracle?

The validation of input data and applying the oracle have to be fast enough and scale effectively as file sizes increase, at least within realistic ranges.

RQ2: Are the model and constraints accurate in practice in validating input data and applying the oracle? Are they accurate in uncovering issues, if any, in the real transmission files, the SES-DAQ system, or the specifications of the system?

Since our validation is based on models of the input, output and their mappings, the main aim of our investigation is to determine whether the level of abstraction of the models is not an impediment to accurate input validation and oracles. This research question aims to validate the technique in two contexts: (1) when it is used to validate test inputs and test outputs, and (2) when it is used to validate field data. For our approach to be effective for test input validation, it needs to correctly identify the constraint violations that are expected for each test input and not trigger false positives by indicating violations in perfectly correct input files. On the other hand, if constraint violations on the input occur when validating real transmission files, this may indicate either problems in the specifications (e.g. implicit changes) or bugs in the implementation.

4.2.1 Subject selection

This empirical evaluation makes use of the SES-DAQ system described in Chapter 2. The technique is particularly useful for SES-DAQ because it needs black-box testing; in fact, this DAQ system is developed and tested by two independent teams with limited or no knowledge of the underlying software code.

There are 32 manually written test cases available (each test input is ($<$ 12 kB) in size), as described in Section 2.3. We know the expected behaviour of this acceptance test suite. For our evaluation, we also use a real representative transmission file, provided by the SES, which is larger in size ($\approx$ 2 GB, a realistic size for such files in our context) and can be used to further validate our approach and in particular assess its scalability. The file is expected to be correct and should only trigger violations if either our specifications (models) or the SES-DAQ system are incorrect; in practice, this usually happens when the client changes the specifications of the system during development and testing, thus leading to violations that require implementation changes.

4.2.2 Approach execution and data collection

To perform the empirical study, we use the model and constraints for the system that were created for the empirical study in Section 3.6. We consider two sets of input data: (1) the 32 manually written

test inputs for the system and (2) multiple real satellite transmission files of incrementally larger sizes up to 2 GB, obtained by sampling from the real representative transmission file.

Each of the input data files considered for the empirical evaluation was executed against the DAQ system to obtain its corresponding test output log files.

We validated the input and configuration files for each of the 32 test cases with our tool using the created model and constraints. We then processed the output log files for each test case using our tool to apply the oracle. The same process was repeated for the real transmission files.

To respond to RQ1, we analysed the relationship between execution time and file size. For each test input and real transmission file that was processed, we recorded the execution times of the three main subprocesses for each: Instantiating the model, validating the input data (checking the input and input/configuration constraints) and applying the oracle (checking the input/output and input/configuration/output constraints). We report the time for each process individually to be able to identify which process is more time-consuming and could benefit from optimisation. The processing time for input data validation and applying the oracle is expected to depend on both the input file size and content (e.g. the number of constraint violations) of the input and output files. Investigating input validation scalability therefore entails studying the relationship between processing time and the size of the input file. One particular issue of interest was the shape of the relationship between processing time and input file size. A linear relationship would clearly suggest our approach is scalable whereas an exponential relationship would limit its range of applicability. The processing time of oracles is expected to be more complex to study as it depends on both the input and output files' size and content. Assessing whether the execution time for applying the oracle is acceptable with representative transmission files is nevertheless important.

To respond to RQ2, the log files that were produced by our tool were examined to analyse if the tool behaved as expected in finding constraint violations. For the 32 manually written input data files, we expected violations of the input and input/configuration constraints that reflected the errors of the files containing invalid data; when we applied the oracle, we expected that no violations would occur as the SUT was deemed to be compliant with the system test suite. Though violations were not expected in the real transmission files, in practice the client could have changed the specifications during development, thus leading to violations that required changes to the SES-DAQ system implementation.

All experiments were executed on a MacBook Pro with an Intel Core i7 CPU running at 2.2 GHz with 8 GB RAM.

4.2.3 Results

In this section, we discuss the results of the empirical evaluation to answer our two research questions.

4.2.3.1 RQ1: Execution time

Table 4.1 reports the execution time for the 32 test cases in the acceptance test suite. For each process, we report the minimum, maximum and average execution time over all 32 test cases. The results show that execution time is a maximum of 940 milliseconds, which makes it feasible to validate the input

Table 4.1. Execution times of the approach on the acceptance test cases.

Operation	Execution Time (ms)		
	Min	Max	Avg.
Model Instantiation	684	845	762
Test Input Validation	1†	56	41
Oracle	0†	39	31
Total	685	940	834

†One of the 32 test cases considers an empty input.

and apply the oracle for all 32 test cases in less than 30 seconds. This is not surprising as the test cases required by the client are small in size.

We investigated the relationship between the size of the input file and execution time that we obtained by analysing processing times for different sizes of real transmission files. Real transmission files can be much larger than test case files: The largest test case input file has 6 *CADUs*, while the real transmission files we used can have up to one million *CADUs*.

Figs. 4.2, 4.3 and 4.4 show the result of our analyses. For each figure, the *y*-axis shows execution time while the *x*-axis is the file size in number of *Cadu* elements. Note that each *CADU* is approximately 2 kB in size. Each curve is based on seven roughly equally spaced data points of randomly selected real transmission files. Of course, as discussed earlier, we do not expect input file size to be the only factor that affects performance, however, it is expected to be the main factor.

Applying the oracle is the most time consuming process in our approach. For the largest file size (2 GB), applying the oracle takes 49.94 minutes compared to 1.35 minutes for instantiating the model and 2.46 minutes for validating the input file. Recall from Table 3.3 and the discussion in Section 3.6.3 that the constraints used for the oracle are more complex than the constraints used for test case validation. Therefore, it is expected that checking those constraints would take more time than checking the input validation constraints.

We noticed that execution time for the first data point is higher than expected for each of the three processes. This might be caused by the constant time overhead that is required for starting up and initialising the tool.

The three graphs clearly show that the relationship between size and execution time is approximately linear, with a maximum of 53.7 minutes in total to perform all three processes for one million *Cadu* elements (2 GB). Such execution times are acceptable in practice since practitioners do not need instant results. These execution times enable testers, for example, to run batch jobs overnight, which is a common practice for large scale testing in the industry. The linear relationship with input file size indicates that much larger files can be handled in the future.

When we compared the execution time results of real transmission files to those of the test cases in the acceptance test suite, we noticed that while applying the oracle is the most time consuming process for the real transmission files, we did not observe the same for the acceptance test cases. This might be caused by the fact that each test case was designed to violate only one constraint, while the real transmission files exhibited a large number of constraint violations caused by implicit changes by

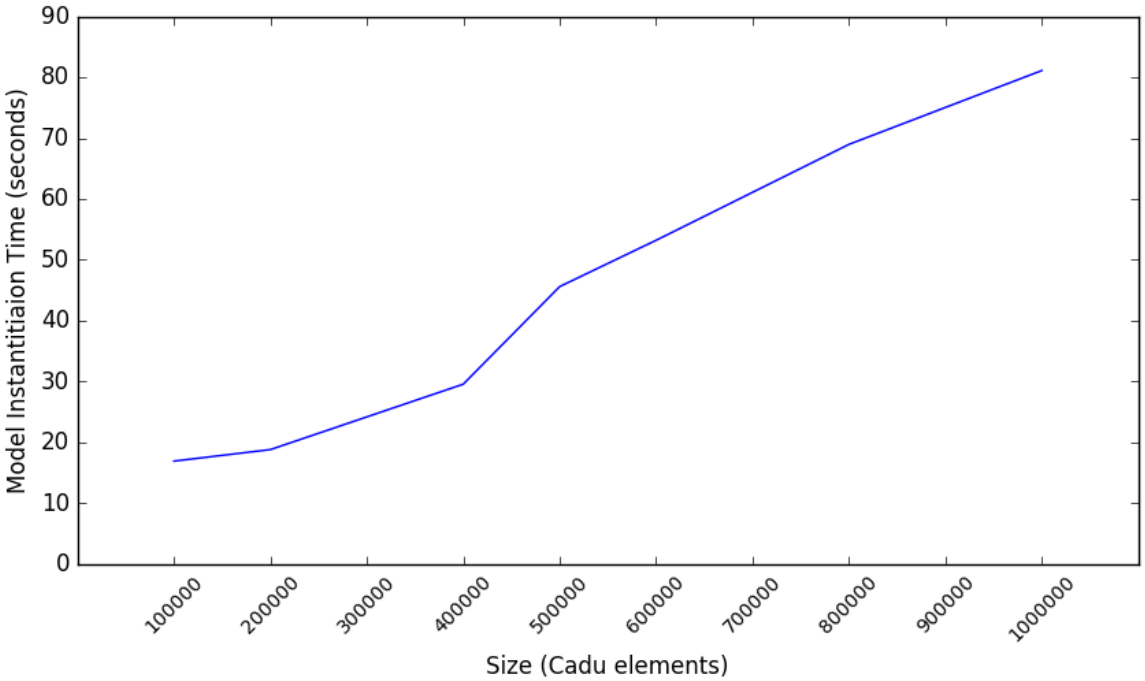


Figure 4.2. Model instantiation time versus the input file size.

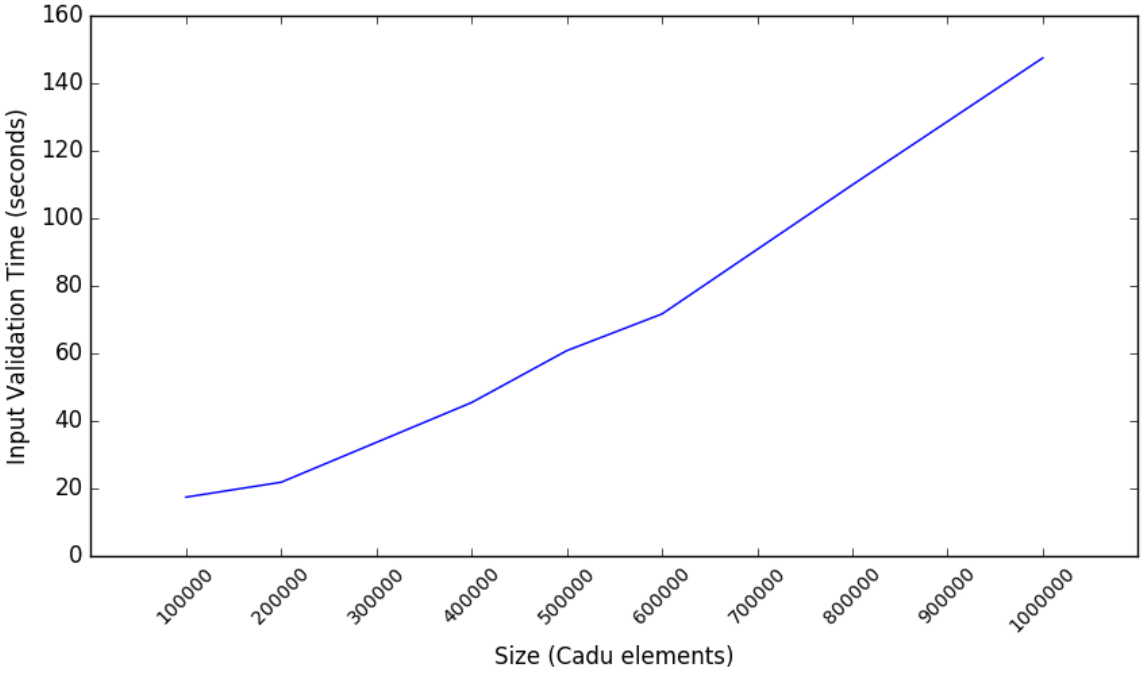


Figure 4.3. Input validation time versus the input file size.

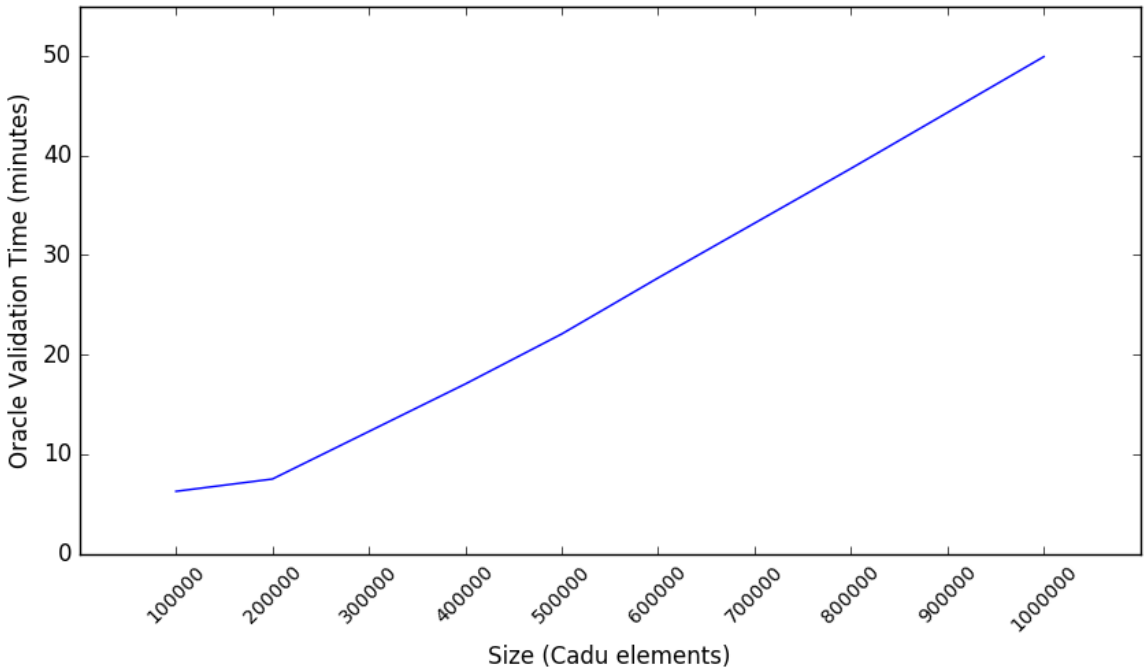


Figure 4.4. Time needed to apply the oracle versus the input file size.

the client to the specifications, which were not yet reflected in the system.

Answer to RQ1: The results show that *our approach is scalable in terms of execution time*. The input data validation and oracle execution time on real transmission files is manageable in practice, with less than 3 and 50 minutes for input and oracle constraints, respectively. Since oracle constraints are more complex, it is not surprising that they take more time to be checked. Furthermore, the linear relationship between the size of the input file and execution time makes it possible to potentially process much larger files.

4.2.3.2 Additional remark on execution time

We observed that the performance of our approach is also dependent on the quality of the OCL constraints. For example, if OCLs contain iterative operations, validation takes longer. As an effective solution to deal with this problem, we found that using additional associations in the model could help.

For example, when the constraint initially developed to check for the VIRTUAL_COUNTER_JUMP event (in Fig. 4.5) was executed, the performance of the tool suffered a considerable degradation as the number of *Cadu* elements in the input file increased. This was caused by the nested loops used when expressing the constraint. To solve this problem, we modified the model slightly and rewrote the constraint in a way that avoids the constructs that negatively affect performance. We added a custom association (i.e. *prev* in Fig. 3.1) that links every *Cadu* element to the previous *Cadu* element having the same *virtualChannelId*. In this way, we avoid the nested loops when checking the constraint, which greatly improves performance. The OCL code in Fig. 3.5 shows the newer version of the constraint after modification. Note that the context of the original constraint is *ChannelData*, while the context for the new constraint is *Cadu*. The new constraint will be checked by the tool for each *Cadu* element while the old constraint needs to be checked only once for each input file. Nevertheless, the new constraint still performs better in terms of execution time.

```
1  context ChannelData inv:
2
3  let
4      vcduHeaders : Sequence(VcduHeader) = self.cadu.vcdu.vcduHeader,
6      transData : TransmissionData = self.transmissionData
5  in
6
7      vcduHeaders->forAll(x, y : VcduHeader |
8          y.virtualChannelId.intValue <> transData.configuration.rtStpsConfig.xsveRtStps.idleVcid
9
10         and
11
12         vcduHeaders->indexOf(x) < vcduHeaders->indexOf(y)
13         and
14         x.virtualChannelId.intValue = y.virtualChannelId.intValue
15         and
16         vcduHeaders->forAll(z : VcduHeader |
17             not (
18                 vcduHeaders->indexOf(x) < vcduHeaders->indexOf(z)
19                 and
20                 vcduHeaders->indexOf(z) < vcduHeaders->indexOf(y)
21                 and
22                 x.virtualChannelId.intValue = z.virtualChannelId.intValue
23             )
24         )
25
26         and
27
28         if x.vcFrameCount.intValue < 16777215
29             then y.vcFrameCount.intValue <> x.vcFrameCount.intValue + 1
30         else x.vcFrameCount.intValue = 16777215 and y.vcFrameCount.intValue <> 0
31         endif
32
33         implies
34
35         transData.acquisitionData.vcduReportData.vcduReport.vcduReportBody.vcduEvent
36             ->exists(i | i.eventType = VcduEvents::VIRTUAL_COUNTER_JUMP
37                         and i.currentVcduFrame.header = y.vcdu.getHeaderAsString() )
38     )
```

Figure 4.5. Example of a constraint on input, configuration and output data used to automate the oracle prior to making changes to enhance performance. The updated, more efficient, version is that of Fig. 3.5. Note: The constraint was originally developed using slightly different data types; for consistency, we show an updated version here that is compliant with the UML model of Fig. 3.1.

4.2.3.3 RQ2: Accuracy

For our approach to be effective, the tool has to be able to identify all constraint violations in the input and accurately apply the oracle without reporting any false positives. For each of the 32 test cases, we know from the test documentation what violations of the input constraints should be reported. We expect no violations of the oracle constraints because the system is correct with regard to the test suite based on the manually written system tests.

When we validated the 32 test cases using our tool, we found that every expected violation of input/configuration constraints was correctly reported with no false positives. We also found, as expected, that no violations of the oracle constraints were reported. Because all the constraints are checked on each test case, any number of test cases can be added to the test suite without the need to manually check the expected output or write assertions for the new test cases (the current practice). This reduces the resources and effort needed to test the system more extensively.

When we validated the real transmission files, we found that in some files many input constraints were violated. Investigating these violations revealed that some of the specifications of the input file were changed by the client but had not yet been implemented in the system. For example, a constraint on *ISPs* (modelled in Fig. 3.2) specifies that their length should be a multiple of four. This constraint was violated in the real transmission file as the client decided to remove this restriction causing a violation to be reported for each *packet* that violated this constraint. This change was discovered by SES through running the real transmission on the DAQ system and then manually examining the output log files. This shows that our input data validation approach could help identify changes in specifications without the need to execute the transmission file on the system, which takes considerably more time and effort than input data validation due to the need to analyse many test case failures, as opposed to a subset of violated constraints.

Answer to RQ2: The results of our empirical evaluation show that *our approach is accurate in validating test cases and applying the oracle*. The results also show that our approach is able to identify implicit changes in specifications (from the client) of the input file and the DAQ system without the need to execute any test cases on the DAQ system.

4.2.4 Threats to validity

Internal threats

The internal threats to validity in this study are related to the test cases and transmission files used to evaluate the system. We used all the test cases and transmission files provided by the system testers to avoid experimenter bias.

External threats

The external threats are related to the choice of case study system and the ability to generalise results. Although we only used one system in the case study, it is representative of data processing systems. Our results might only be relevant in this application domain; nevertheless, this domain is important and widely used. Many other types of systems have similar characteristics as they process very complex inputs, detect input errors, and extract data. In other words, our approach might also be applicable to any system where the complexity lies in the input, output and their mappings.

Construct threats

For studying scalability, we used the execution time of the test validation and applying the oracle processes. Though execution time depends to some extent on the content of the files, and not just their size, the used transmission files are not only real but representative and, in any case, we are mostly interested in the order of magnitude of the processing, not exact figures.

4.3 Conclusion

In this chapter, we evaluated our proposed automated test validation and oracle approach for data processing systems with complex input structures and mappings between input and output. Many systems share these characteristics, including DAQ systems common in the satellite communications industry. In such systems, generating valid test inputs is challenging and checking the output manually is time-consuming and error-prone. Requirements change on a regular basis and the input and output files are large and complex. Our approach is driven by models of the input and output structure and content, and to support it, we defined a specific modelling methodology using UML class diagrams and OCL constraints as a notation. We developed a tool to automate our approach and evaluated the approach on a real industrial DAQ system. Though there is substantial work in the area of Model-Based Testing (MBT) (as reported in related works, Chapter 8), none of the existing approaches match the needs for DAQ systems and other systems with similar characteristics.

Our empirical evaluation showed that we were able to properly validate input data and automate the test oracle by correctly identifying failing and successful test inputs, as well as effectively validating real transmission files. Additionally, the results of our empirical evaluation show that the approach is scalable as the input and oracle validation process executed within reasonable times on real transmission files for a real satellite DAQ system. Though input files can be validated in a few minutes, it can take up to 50 minutes for applying the oracle. The main reason is that transmission files in our case study generated large numbers of violations due to changes to specifications from the client that had yet to be addressed. Furthermore, oracle constraints are more expensive to check as they are significantly more complex, though such durations are fine in practice since such oracle validation processes can be run in batch mode, at night for example. Furthermore, results on existing test suites show that the level of abstraction of the model is not an impediment to the accurate evaluation of oracles. In terms of scalability, the relationship between execution time and input file size is linear, suggesting that much larger files can be handled in the future using our approach.

Chapter 5

Automatic Test Input Generation

In this chapter, we focus on the problem of testing the correct behaviour of data processing systems in the presence of faulty input data. The proper identification of faulty input data is particularly important in our case study since a satellite transmission is often subject to alterations that may introduce faults, such as channel noise, which should be properly handled to prevent the generation of erroneous results. This problem generalises to other types of data processing systems. For example, web crawling engines, which process complex data structures (i.e. web pages) and need to determine the presence of faulty data (e.g. tags) not properly closed (a common mistake of novice web developers and bloggers).

Most model-based approaches that relate to the problem above focus on generating data structures for unit testing [Boyapati et al., 2002, Senni and Fioravanti, 2012]. In addition, the few approaches that focus on system testing and generate faulty data are based on Context-Free Grammars (CFGs) and, as a result, cannot generate data that presents complex relationships between the data fields [Hoffman et al., 2009, Zelenov and Zelenova, 2006].

The approaches that are particularly appealing for automatically testing data processing systems for input data faults are the ones based on the adoption of mutation operators that alter existing test inputs to generate faulty data [Shan and Zhu, 2009, Bertolino et al., 2014, De Jonge and Visser, 2012].

The main limitation of these approaches is that the mutation operators are specific for the particular kind of input data used by the SUT. In contrast, we would like software engineers to define a fault model, capturing their experience of the domain, and then provide a way to tailor generic mutation operators to implement this fault model on a specific data model of inputs and outputs.

This chapter highlights the following research contributions:

1. A model-based technique that automatically generates faulty test inputs, by relying upon generic mutation operators that alter data collected in the field.
2. The use of UML stereotypes and OCL queries to configure the mutation operators to implement a fault model of the SUT.
3. An empirical evaluation on an industrial data processing system to evaluate the effectiveness of our proposed approach.

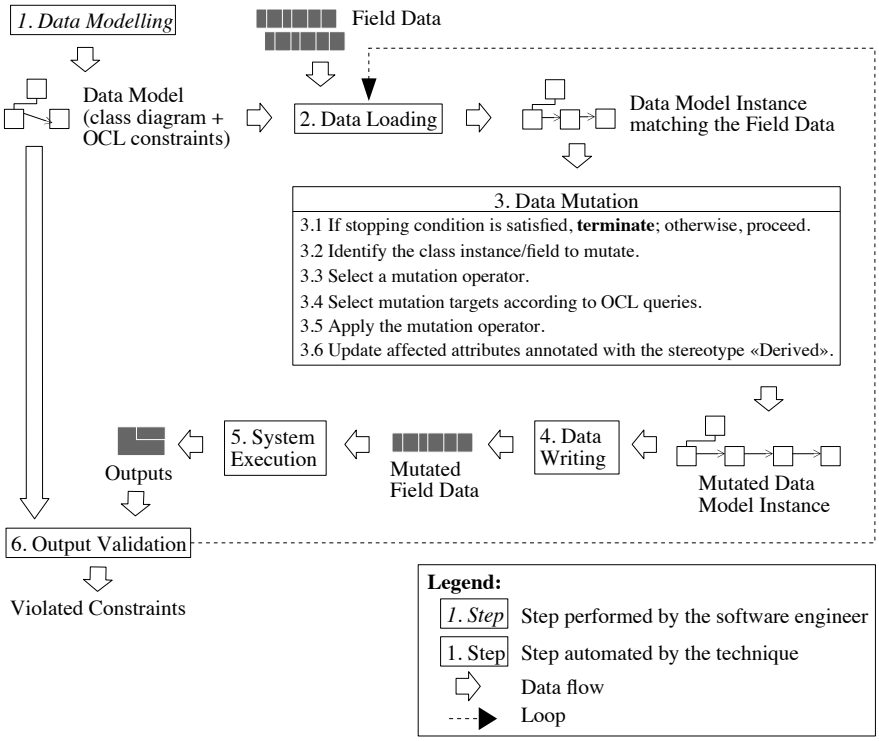


Figure 5.1. Steps for generating complex and faulty test data.

The remainder of the chapter proceeds as follows. Section 5.1 describes the data mutation technique presented in this chapter. Section 5.2 details the data mutation operators that are used to mutate field data. Section 5.3 describes how mutation operators can be configured to comply with the fault model of the SUT. Section 5.4 describes how we enhanced the modelling methodology to enable automatic data mutation. Section 5.5 provide details about the data mutation strategies considered in this chapter. Section 5.6 presents the empirical results obtained. Finally, Section 5.7 concludes the chapter.

5.1 Description of the Approach

The mutation approach requires two inputs: field data and a data model (i.e. a class diagram modelling input and output data, annotated with UML stereotypes and constraints, which in our case are written in the OCL). Field data is used to generate new test inputs by means of six different data mutation operators that alter existing data on the basis of its data model. The data model drives the usage of the generic mutation operators in order to implement the fault model specified for the SUT. The characteristics of the fault model are captured by means of UML stereotypes and OCL expressions. UML stereotypes are used to select the data fields to mutate, the kind of mutation operators to apply, and the data fields to update after mutation to preserve data invariants. OCL expressions are used to configure mutation operators by selecting the targets of the mutation through appropriate OCL queries.

Further, the technique makes use of automated test oracles that, in our context, consist in the detection of faulty inputs that remain undetected by the SUT. Oracles work by checking OCL constraints that capture the relationships between faulty inputs and expected outputs, hereafter input/output con-

straints. These constraints are relatively simple to define, since they indicate the output expected (e.g. an error message) in the case of invalid input data.

The approach works in six steps: data modelling, data loading, data mutation, data writing, system execution, and output validation. Fig. 5.1 provides an overview of the approach.

5.1.1 Data modelling

Step 1, *data modelling*, refers to the activity performed by software engineers by following the technique described in sections 3.2 and 3.3. Data modelling consists in the definition of a class diagram that models the input and output data of the SUT. The diagram is augmented with a set of OCL constraints that capture properties of the input data, and relationships between input data and output data. To support data mutation, the model is additional augmented with mutation related stereotypes that are described in Section 5.4.

5.1.2 Data loading

Step 2, *data loading*, loads a chunk of field data into memory, in the form of objects that we refer to as a *field data object*. Data loading depends on the kind of data processed by the SUT. In practice, a tool can load the field data as an instance of the modelled class diagram produced by the software engineers (e.g. Chapter 9 describes our related implementation for the SES-DAQ system).

5.1.3 Data mutation

In Step 3, *data mutation*, the approach mutates the loaded field data using mutation operators (described in Section 5.2) according to configuration settings (described in Section 5.3) that allow for the application of a specific fault model. For the approach of this chapter, only a single mutation is made per test input.

Data mutation consists of the following activities: *the identification of the classes and fields to mutate*, *the selection of a mutation operator*, *the selection of a mutation target (an instance of the class or field to mutate) according to OCL queries*, *the application of the mutation operator*, *and lastly the updating of the attributes of the mutated data object annotated with the stereotype «Derived»*. For the study in this chapter, mutation operators are selected according to two strategies, *Random* and *All Possible Targets*, both detailed in Section 5.5.

For the case of SES-DAQ, we developed a mutation toolset as described Section 9.3.2.

5.1.4 Data writing

Step 4, *data writing*, writes the mutated field data object back to the format processed by the SUT, thus generating what we call *mutated field data*. In practice, a tool similar to the one used for *data loading* can be used (e.g. Chapter 9 describes our related *data writing* implementation for the SES-DAQ system).

5.1.5 System execution

In step 5, *system execution*, the approach executes the SUT using as input the mutated field data. Once the mutated field data has been written in the format processed by the SUT, the SUT is executed and the outputs are collected for the validation step. For example, in the case of SES-DAQ, we collect the output logs generated by the system for each mutated field data passed as input.

5.1.6 Output validation

In step 6, *output validation*, the approach identifies violations of the input/output constraints as indicators of software failures. Input/output constraints capture the system output, usually an error message, expected in case of a faulty input. The approach loads the output of the system and determines the presence of violated constraints. This step makes use of the oracle approach presented in Chapter 4.

In addition to violated constraints, the approach reports contextual information like the name of the operator applied on the original data, the target entity, and a portion of the original and mutated data in the neighbourhood of the mutation. Our experience indicates that this contextual information is useful to better understand the fault.

It is worth mentioning that the approach reports failures only when an input/output constraint is violated—that is, when a faulty input data does not lead to the generation of an expected output (i.e. an error message). This implies that if a data mutation does not generate faulty data, a false positive is not going to be reported by the technique.

Steps 2 to 6 are repeated until the data mutation step determines that a stopping condition is reached. Section 5.5 details the stopping conditions for the two different mutation selection strategies.

5.2 Data Mutation Operators

The approach mutates input data by applying six mutation operators on the data loaded into memory (i.e. on the field data object). The identification of the mutation operators to apply and the selection of the data fields to mutate is guided by stereotypes used in the data model. Section 5.4 describes how we enhanced the modelling methodology presented in Chapter 3 to enable automatic data mutation.

The stereotype *«InputData»* is used by the software engineer to annotate the classes that model input data, to distinguish them from the configuration and output data classes, which do not need to be mutated. The two stereotypes *«Identifier»* and *«Measure»* are used to annotate class attributes; data that correspond to an attribute tagged by one of these stereotypes is mutated by applying a specific mutation operator or, otherwise, by applying an operator that simply flips bits. The stereotype *«Derived»* is used to annotate class attributes that need to be updated after certain mutations in order to prevent trivial inconsistencies; Section 5.3 provides additional details about the role of this stereotype.

We defined six mutation operators: three working at the class instance level (Class Instance Duplication, Class Instance Removal, Class Instances Swapping), and three working at the attribute level

(Attribute Replacement with Random, Attribute Replacement using Boundary Condition, Attribute Bit Flipping).

The six mutation operators do not include operators for the generation of class instances from scratch (e.g. we do not include an operator whose goal is to create a new class instance and add it to a sequence). This choice mainly depends on the fact that the generation of a portion of a complex test input from scratch would require a detailed specification of the characteristics of such inputs, for example by means of a grammar. One of the benefits of the approaches presented in this dissertation is that they do not require software developers to provide a complete specification of the format of the input data.

The following subsections detail the six mutation operators. For each operator, an informal description is provided along with an example that refers to the input data of SES-DAQ (refer to Figs. 3.1 and 3.2).

5.2.1 Class Instance Duplication (CID)

Description: The operator *Class Instance Duplication* duplicates an instance of a class belonging to a collection of elements. This operator copies a randomly chosen instance of a class in a collection and then inserts it at a random position in the collection. This operator simulates unexpected data in a collection.
Example: For the SES-DAQ, this operator can be applied to the containment associations between the classes *ChannelData* and *Cadu*, and between the classes *ChannelDataPerVcid* and *Isp*. In both cases, the duplicated data generated by this operator simulates a transmission error.

5.2.2 Class Instance Removal (CIR)

Description: This mutation operator deletes a randomly selected instance of a class from a collection of elements.
Example: For the SES-DAQ, this operator can be applied to the containment associations between the classes *ChannelData* and *Cadu*, and between the classes *ChannelDataPerVcid* and *Isp*. The removal of an instance of class *Cadu*, for example, simulates a transmission error that may lead to either missing or broken packets; in this case, when processing erroneous data created with this mutation operator, SES-DAQ should report a *VIRTUAL_COUNTER_JUMP* error as indicated by the constraint in Fig. 3.5.

5.2.3 Class Instances Swapping (CIS)

Description: Swaps the positions of two randomly chosen instances of a class in a collection of elements.
Example: For the SES-DAQ, this operator can be applied to the containment associations between the classes *ChannelData* and *Cadu*, and between the classes *ChannelDataPerVcid* and *Isp*. The effect of swapping two packets belonging to the association between the classes *ChannelDataPerVcid* and *Isp* simulates the presence of transmission data sequence errors.

5.2.4 Attribute Replacement with Random (ARR)

Description: This mutation operator replaces the value of an identifier attribute in an instance of a class with a different, randomly chosen, value. In principle, all the attributes of a class can be replaced with randomly chosen values, but in the general case a randomly generated value is not necessarily erroneous. We are interested in mutations that lead to errors; for this reason, we introduced the UML stereotype *«Identifier»* that allows software engineers to indicate which attributes are used as identifiers, and thus can be mutated according to the ARR operator. The *«Identifier»* stereotype enables software engineers to specify a numeric range for the random value to generate.
Example: This mutation operator can be applied to all the attributes annotated with the stereotype *«Identifier»*. For example, a random mutation of the attribute *versionNumber* belonging to an instance of class *VcduHeader* simulates an invalid frame version, which should be reported by the software in the error logs.

5.2.5 Attribute Replacement using Boundary Condition (ARBC)

Description: This mutation operator changes the value of an attribute according to a boundary condition criterion. This operator is particularly useful for mutating attributes that should be bound within a range, these attributes are usually measures. We thus introduced the UML stereotype *«Measure»* to annotate the attributes that belong to this category. This stereotype enables software engineers to indicate the minimum and maximum values allowed for the annotated attribute. The mutation operator generates (up to) six values according to traditional boundary testing strategies: minimum value, minimum value plus/minus one, maximum value, and maximum value plus/minus one. The operator ensures that the generated values are in the range representable with the data type (e.g. unsigned bytes cannot represent negative values).
Example: This operator can be applied to all the attributes annotated with the UML stereotype *«Measure»*. For the SES-DAQ, this operator can be applied to the attribute *vcFrameCount* of class *VcduHeader*.

5.2.6 Attribute Bit Flipping (ABF)

Description: This operator randomly selects an attribute that corresponds to transmitted data and alters the value of a randomly selected bit. This mutation operator is particularly effective for introducing errors in attributes that cannot be designated as an *«Identifier»* or a *«Measure»*. The operator works by flipping a single bit of an attribute.
Example: This mutation operator can be applied to the attribute *data* of class *PacketData* of the SES-DAQ. The attribute *data* is a byte array: the mutation of one of its bits simulates the presence of a realistic transmission error that should be identified thanks to the presence of a Cyclic Redundancy Check (CRC) error detection code (contained in the field *data* of class *PacketErrorControl*).

5.3 Configuring Mutation Operators to Apply a Specific Fault Model

Although the mutation operators proposed in Section 5.2 are generic, we combine the use of UML stereotypes and OCL constraints to enable software engineers to configure the mutation process to

Table 5.1. Mapping between the SES-DAQ fault model of Table 2.1 and mutation operators and configurations.

Fault	Mutation Operator	Configuration
Duplicate CADU	Class Instance Duplication.	«InputData»†
Duplicate ISP	Class Instance Duplication.	«InputData»†, «Derived»
Missing CADU	Class Instance Removal.	«InputData»†
Missing ISP	Class Instance Removal.	«InputData»†, «Derived»
Wrong CADU Sequence	Class Instances Swapping.	«InputData»†
Wrong ISP Sequence	Class Instances Swapping.	«InputData»†, «Derived», Query to select packets
Incorrect Identifier	Attribute Replacement with Random.	«Identifier», «Derived»‡
Incorrect Checksum	Attribute Replacement with Random.	«Identifier»
Incorrect Counter	Attribute Replacement using Boundary Condition.	«Measure», «Derived»‡
Flipped Data Bits	Attribute Bit Flipping.	–

†This stereotype need only be applied to the root input container class. ‡If necessary.
Note: CADU, Channel Access Data Unit; ISP, Instrument Source Packet.

generate data that matches the specific fault model of the SUT.

The technique presented here simply requires that software engineers annotate the attributes and the classes of the data model with stereotypes that specify the nature of these attributes/classes. Mutation operators are then applied to the data according to the stereotypes used. Table 5.1 shows the mutation operators and the corresponding configurations (i.e. stereotypes and queries) that enable the specific faults of the SES-DAQ fault model (Table 2.1). The mutation operator *Attribute Bit Flipping* does not require the application of a specific stereotype; it is applied on all the attributes not annotated with other stereotypes.

Data mutation may lead to the generation of inconsistent data containing multiple errors that do not enable proper system testing. In particular, it might lead to the generation of trivial faults that do not comply with the given fault model or that mask the intended effect of the mutation operator, thus preventing the possibility to cover the whole fault model.

In the running example, a trivial fault may mask the effect of a mutation operator applied to one of the following attributes (shown in Fig.3.1) of class *VcduHeader*: *versionNumber*, *spacecraftId*, and *virtualChannelId*. Any change to the values of one of these attributes makes the value of the check symbols (used for error-correction) within attribute *headerErrorControl* inconsistent. The attribute *headerErrorControl* of class *VcduHeader* is calculated by applying the Reed-Solomon algorithm [Wicker and Bhargava, 1999] against the bit fields that correspond to attributes *versionNumber*, *spacecraftId*, and *virtualChannelId*. For example, mutating the attribute *virtualChannelId* leads to an inconsistent (wrong) value for the attribute *headerErrorControl*. In practice, this may prevent testing the effect of a wrong value for the attribute *virtualChannelId* since the software first verifies the correctness of the check symbols. To properly evaluate the effect of a fault in one of the three attributes of class *VcduHeader* that are protected by the check symbols, the value of the attribute *headerErrorControl* should be recalculated (i.e. considering the updated mutant value assigned to

```
1   context Isp::cis() : Set(Isp) body:
2
3   let
4       packet1 : Isp = Isp.allInstances()->any(true),
5       vcid : Integer = packet1.mpduActivePacketZone->first().vcdu.vcduHeader.virtualChannelId.intValue,
6       packet2 : Isp =
7           packet1.channelDataPerVcid->select(cd | cd.virtualChannelId = vcid).isp
8               ->select(p | p <> packet1)->any(true)
9   in
10
11      Set { packet1, packet2 }
```

Figure 5.2. OCL query for the swapping of two packets.

attribute *virtualChannelId*).

Inconsistent data might also be caused by mutation operators that target class instances. For example, the swapping of packets (represented by the class instances of *Isp* in Figs. 3.1 and 3.2) that belong to two different virtual channels may lead to the generation of VCDUs that contain packets with invalid APIDs (i.e. inconsistent data). The goal of the swapping operator is, rather, that of generating sequences of packets received in a wrong order. The class instances swapping operator should thus be applied only to instances of the class *Isp* that belong to the same virtual channel.

To preserve data consistency, we enable software engineers to configure the behaviour of mutation operators by means of OCL queries and UML stereotypes.

OCL queries are used to enable software engineers to specify the characteristics of the object instances on which the mutation operators can be applied. By default, a mutation operator is applied on randomly chosen objects in the absence of an OCL query that configures its behaviour.

Fig. 5.2 shows an example of an OCL query that regulates the swapping of instances of class *Isp*. Line 1 in Fig. 5.2 indicates that the OCL query controls the application of the operator *Class Instances Swapping (CIS)* on class instances of type Isp; the query returns a set containing the Isp instances to be swapped (line 11). Line 4 shows the OCL expression that selects the first parameter (this query simply indicates that any Isp instance might be used as first parameter of the *CIS* operator). Lines 5 to 8 indicate how the second parameter of the *CIS* operator should be selected. The query identifies the virtual channel (i.e. *vcid*) that the Isp instance, *packet1*, belongs to (line 5), and then selects a different Isp instance, *packet2*, that belongs to this same virtual channel (lines 6 to 8).

We defined a UML stereotype, *«Derived»*, that enables software engineers to specify which attributes need to be updated after certain mutations in order to prevent trivial errors. The stereotype requires that software engineers specify the name of a method that is invoked at runtime by the mutation framework to regenerate the value of the annotated attribute. The implementation of this function should be provided by the software engineer and we expect to find the method in the class path. Such implementations should match methods present in the SUT, which are expected to be unit tested.

Fig. 3.1 shows that the attribute *headerErrorControl* of class *VcduHeader* has been annotated with the stereotype *«Derived»*. In this case, the attribute is associated with a utility function named *Util.reedSolomon* (a method available in the SES-DAQ implementation) that recalculates the check symbols (e.g. following the mutation of a *spacecraftId* field).

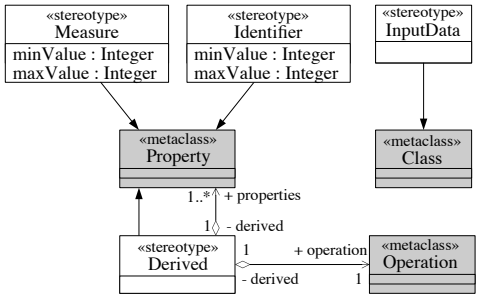


Figure 5.3. Portion of the custom profile that extends the UML metamodel to support automated data mutation. The shaded elements in the figure represent UML metaclasses.

5.4 Augmenting the Model for Automated Data Mutation

We enhanced the modelling methodology to enable automatic data mutation by introducing four stereotypes: *«InputData»*, *«Identifier»*, *«Measure»*, and *«Derived»*. Fig 5.3 shows the portion of our UML profile where these stereotypes are defined.

The stereotype *«InputData»* extends the metaclass `Class`. The stereotype is applied to the classes that model input data, to distinguish them from other data class types (i.e. *«ConfigData»* or *«OutputData»*), which do not need to be mutated. The stereotype need only be applied against the root container class of the corresponding input data (e.g. the TransmissionData class in Fig. 3.1).

The stereotypes *«Identifier»* and *«Measure»* both extend the metaclass `Property` (i.e. they can be applied to the attributes of the data model). Each stereotype has two Integer attributes, *minValue* and *maxValue*. Data corresponding to an attribute annotated by one of these stereotypes is mutated by applying a specific mutation operator. These stereotypes specify (via their attributes) the *minValue* and the *maxValue* (both Integers) that define a range for the values generated by the corresponding mutation operators. For example, in Fig. 3.1, the attribute *spacecraftId* of class *VcduHeader* is annotated with the stereotype *«Identifier»* and the attribute *firstHeaderPointer* of class *MpduActiveHeader* is annotated with the stereotype *«Measure»*. Section 5.2 details how mutation operators are applied for the given stereotypes.

The stereotype *«Derived»* extends the metaclass `Property`. This stereotype is applied to attributes whose data needs to be updated after certain mutations in order to prevent trivial inconsistencies (e.g. in Fig. 3.1, the attribute *headerErrorControl* of class *VcduHeader*). When applied to a given attribute, this stereotype captures the attributes of our data model (via the containment denoted by the role *properties*) to monitor to detect if they have been mutated; the stereotype also designates, via the role *operation*, the method for updating the value of the given *«Derived»* attribute. Section 5.3 provides additional details about the role of this stereotype.

5.5 Data Mutation Strategies

The technique generates a *mutated field data object* by applying the mutation operators described in Section 5.2 on the *field data object* (recall that this is an instance of the data model that corresponds to a specific data chunk). Although in principle all the elements of the *field data object* could be mutated, thus obtaining hundreds of mutants even from a small portion of the input data, the elements

```
Require: fieldData, a reference to the field data
Require: model, the data model
Require: mutants, the number of mutated versions to generate
 1: created ← 0
 2: while created < mutants do
 3:     dataObj ← loadData(fieldData)
 4:     operator ← randomlySelectMutationOperator()
 5:     target ← randomlySelectMutationTarget(operator,model)
 6:     opInputs ← identifyOpInputs(dataObj,target,operator)
 7:     if opInputs ≠ null then
 8:         dataObj ← operator.execute(dataObj,opInputs)
 9:         dataObj ← updateData(dataObj,model)
10:         executeSystem(dataObj)
11:         created ← created + 1
12:     end if
13: end while
```

Figure 5.4. Algorithm for applying the Random mutation strategy.

to mutate should be carefully selected to reasonably limit the test execution time.

In this chapter, we consider two simple strategies to select the elements to mutate: *Random (RND)*, and *All Possible Targets (APT)*. Chapter 6 focuses on elaborating and evaluating more sophisticated strategies, including the possibility of applying multiple mutation operators to a same data chunk.

5.5.1 Random data mutation strategy

The *RND* mutation strategy randomly selects a mutation operator and randomly applies it to one of the elements where it can be applied. The algorithm in Fig. 5.4 shows how the *RND* mutation strategy works. The algorithm iterates until enough mutated data objects have been generated (line 2). New data is loaded at every iteration (line 3), this is done to sample multiple parts of the field data, thus increasing the possibility of covering all the different types of inputs. After selecting the mutation operator (line 4), the algorithm randomly selects the mutation target (line 5). Depending on the mutation operator, the mutation target will be either an attribute of the data model, or a class instance or class instances that belong to a collection.

In line 6, the algorithm selects the class and attribute instances to mutate (i.e. the inputs for the mutation operator). This is done by invoking function *identifyOpInputs*. The logic behind function *identifyOpInputs* is simple: if there is an OCL query for the selected operator, function *identifyOpInputs* identifies the inputs of the operator by executing the OCL query. Otherwise random inputs are selected. Line 8 executes the mutation operator. Line 9 updates the data by executing the methods associated to all the affected attributes annotated with the stereotype *«Derived»*. Function *executeSystem* invoked in line 10 executes the activities required to test the system with the mutated data: data writing, system execution, and output validation.

5.5.2 All Possible Targets data mutation strategy

The *APT* mutation strategy ensures that an instance of each class or attribute of the data model is mutated at least once by each of the mutation operators that can be applied to it. Fig. 5.5 shows the algorithm for applying the *APT* strategy. The algorithm first identifies all the possible pairs $\langle target, operator \rangle$ that result from the composition of every possible target (attribute or association in the data model) with the operators that can be applied to it (line 2). To maximise data diversity, the algorithm ensures that each mutation operator is executed on a different sample of the data (line 4).

```
Require: fieldData, a reference to the field data
Require: model, the data model
Require: maxAttempts, the maximum number of mutation attempts
 1: attempts ← 0
 2: targetPairs ← retrieveTargetsToMutate(model)
 3: while targetPairs.size() > 0 and attempts < maxAttempts do
 4:     dataObj ← loadData(fieldData)
 5:     i ← 0
 6:     mutated ← false
 7:     while i < targetPairs.size() and !mutated do
 8:         attempts ← attempts + 1
 9:         ⟨target, operator⟩ ← targetPairs[i]
10:         opInputs ← identifyOpInputs(dataObj, target, operator)
11:         if opInputs ≠ null then
12:             dataObj ← operator.execute(dataObj, opInputs)
13:             dataObj ← updateData(dataObj, model)
14:             executeSystem(dataObj)
15:             targetPairs.remove(i)
16:             mutated ← true
17:         end if
18:         i ← i + 1
19:     end while
20: end while
```

Figure 5.5. Algorithm for applying the All Possible Targets mutation strategy.

The algorithm then looks for the first pair $\langle target, operator \rangle$ that can be applied on the loaded data (lines 7 to 19): the algorithm iterates on the list of pairs $\langle target, operator \rangle$ until it is able to identify a set of inputs that allows for the application of the operator on an instance of the target class. Once an operator is found, the algorithm generates the mutated data and invokes the SUT (lines 12 to 14), and removes the pair $\langle target, operator \rangle$ from the list of pairs to process. The algorithm continues until all the pairs are processed or the maximum number of attempts is reached.

5.6 Empirical Evaluation

The goal of our evaluation is to determine whether the mutation approach presented in this chapter can automatically achieve equivalent or better coverage than manually written test cases. Though test cases are derived from requirements and library and dead code tend to make any interpretation of coverage difficult, in practice, in many organisations coverage is an indication of test completeness.

5.6.1 Subject selection

We empirically evaluated the effectiveness of our mutation approach, using the strategies presented in this chapter, by applying it on the SES-DAQ system. To assess our approach, we compare it to the SES-DAQ manual test suite (described in Section 2.3).

5.6.2 Approach execution and data collection

To perform the empirical study, we created a model of the SES-DAQ system data according to our modelling methodology (the same used for the empirical study of Chapter 3). We added the appropriate stereotypes (described in Section 5.4) and configured them (following the methodology of Section 5.3) to fit the SES-DAQ fault model. The complexity of the input data model is clearly indicated by its size, thus illustrating how complex it can be in practice to manually generate input files for testing: 82 classes with 322 attributes and 56 associations. The overall effort required to tailor the

Table 5.2. Test suite coverage results.

Strategy	Coverage (bytecode)		
	Minimum	Maximum	Average
SES Manual Test Suite	–	–	22,820 (70.9%)
RND Test Suite Generation	22,550 (70.1%)	23,060 (71.7%)	22,899 (71.2%)
APT Test Suite Generation	23,226 (72.2%)	23,374 (72.7%)	23,283 (72.4%)

Note: In total, SES-DAQ has 32,170 bytecode instructions. RND, random data mutation strategy; APT, all possible targets data mutation strategy.

generic mutation operators to the fault model of SES-DAQ has been limited: only four OCL queries and two attributes annotated with the stereotype *«Derived»* were necessary. Finally, the definition of input/output constraints also required limited effort: 23 input/output constraints were necessary to automate oracles so as to validate the entire system.

To generate faulty input data, we randomly sampled chunks (50 CADUs in size) of a large transmission file containing field data provided by SES and mutated them. The size of the transmission file is 1 million CADUs (or about 2 GB), containing 1 million VCDUs belonging to four different virtual channels. We generated test cases by applying both the RND and APT test generation strategies against the field data chunks. For APT, we stopped when all attributes were covered and this led to the generation of 43 test cases. To better compare the two strategies, we generated 43 test cases for RND as well. Note that, in our context, since test execution is entirely automated, the number of executed test cases is not a concern, within reasonable limits. All the generated inputs included faulty data matching our fault model. The two main costs are the cost of generation and verification of test results. Manual testing is expensive in both respects. RND is quick in generating test cases whereas APT is slower, due to differences in generation algorithms as described above. In both cases, we automate the test oracle the same way. What we hope to achieve with APT is a more systematic and predictable strategy in terms of code coverage, as reported below.

Both APT and RND carry a degree of randomness (i.e. the sampling of the elements to mutate). To account for such randomness, we evaluated RND and APT with 10 different test suites generated independently, averaged their results, and accounted for their random variation in our analysis.

To measure the coverage of test cases we used EclEmma, a Java code coverage tool [Mountainminds, 2006]. We collected information about the number of bytecode instructions covered. This is the finest coverage granularity available in EclEmma since it captures the coverage of the subexpressions in branch conditions. Since SES-DAQ includes third party components, we narrowed the measure of the coverage to 341 classes that SES software engineers identified as being involved in data processing (in total, 32,170 bytecode instructions) and that were developed by them.

5.6.3 Results

Table 5.2 shows the results of the experiments. The manual system test cases implemented by SES cover 71% of all instructions, amounting to a total of 22,820 bytecode instructions covered. APT covers on average 23,283 instructions, that is, 72% of all instructions. RND performs similarly to the SES test suite, since it covers 22,899 instructions on average. We inspected the instructions covered by the three test suites and we noticed that APT covers the same instructions as the test

cases implemented by SES plus hundreds of instructions related to the handling of faulty data cases, not accounted for by the manual test suite. Such a difference may sound small, but these uncovered instructions may address critical faults.

To further compare APT and RND, we report that the minimum and maximum number of instructions covered by the test suites generated applying APT are 23,226 and 23,374, respectively. As a comparison, the RND test suites cover a minimum and a maximum of 22,550 and 23,060 instructions, respectively. For some of the runs, RND actually covered fewer instructions than the SES test suite. This clearly shows that in general APT performs better than RND. Other more advanced strategies will be investigated in Chapter 6.

In other words, in a completely automated fashion, our approach covers more than 70% of all instructions, which is on par with industry standards for system testing coverage [Cornett, 2016], and the APT approach consistently executed more instructions than expensive, manual testing, based on the expertise of experienced engineers. Such automation is a source of significant savings in a context where several releases of the SUT are produced every year, over a typical lifespan of a decade or two.

5.7 Conclusion

In this chapter, we presented an approach to automatically test software systems that process complex, structured data, which might contain faults. Though many data processing systems fit this description, a typical example is a satellite DAQ system. Automated testing is particularly important in such a context as each test case is highly complex and expensive to produce manually since it is a large, structured, and complex input file. This problem is even more acute in a context of frequent system changes, when test cases need to be reviewed and possibly updated/replaced to remain consistent with new requirements, and when one needs to run frequent regression testing.

Though many MBT approaches have been reported in the literature (see Chapter 8), they do not target data processing systems and are, for the most part, focused on testing compliance with behavioural system models or generating data structures for unit testing.

The approach proposed in this chapter, receives field data and a data model describing its structure and content. The approach applies generic mutation operators on field data to generate faulty data and exercise the robustness of the data processing system. UML stereotypes and OCL queries are used by software engineers to configure mutation operators to implement a fault model of the SUT, which captures domain experience in terms of typical faults encountered in input data. This also prevents the generation of trivially invalid data.

An industrial case study performed with an already deployed DAQ system, for which frequent versions are released every year, shows that our approach achieves slightly better instruction coverage than the manual testing taking place in practice, based on domain expertise.

Chapter 6

Search-Based Robustness Testing

This chapter presents an advanced approach for creating effective robustness test suites based on data mutation. Robustness is “the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [ISO/IEEE, 2010]. Robustness testing of data processing software in the presence of invalid inputs is of particular importance because of the impact unexpected failures may have. For example, data faults in the transmission of an airplane position may crash airplane tracking applications, while malformed HyperText Markup Language (HTML) pages may lead a web crawler to report stock market collapses and cause panic to end users [McCarthy, 2001].

Robustness testing of data processing software is often complicated by the complex structure of the input data. A well-known example is an HTML page that contains many blocks, some of which are kept hidden or contain dynamic information. Similar complexity characterises other kinds of processing systems; for example, the SES-DAQ system developed by our industrial partner SES to process satellite transmissions (see Chapter 2). When performing robustness testing, software engineers need to handcraft complex data structures where valid and faulty values need to be inserted while taking care to preserve all the relationships among the data fields. Handcrafting huge amounts of complex data is particularly time consuming and error prone.

In this chapter, we tackle the more general problem of generating minimal robustness test suites, with high fault revealing power, for data processing systems. When dealing with robustness testing, a single test generation criterion, for example the coverage of the SES-DAQ fault model (described in Section 2.3 and implemented in Chapter 5) is not enough. In practice, Chapter 5 presents a technique that checks if basic test input sanitisation functionalities are implemented in a software. In this chapter, we deal with the more complicated problem of testing software robustness. Multiple factors must be considered; for example, the presence of multiple data faults in the same input, the coverage of functional specifications in addition to fault models, and the generation of a minimal number of test cases. Satisfying multiple criteria when generating robustness test suites may easily lead to combinatorial explosion and specific techniques able to deal with scalability issues are required.

When addressing complex problems where there is a large space of candidate solutions, and one wants to choose a solution that maximises some chosen criteria, metaheuristics are a plausible solution [Luke, 2013]. Metaheuristic algorithms, such as evolutionary algorithms, identify optimal solutions for a problem by iteratively building candidate solutions and by testing them to identify the

one that best achieves the objectives. At each iteration, new candidate solutions are built by means of a tweak operation that is applied on a copy of a candidate solution. The tweak operation allows the algorithm to explore the search space looking for an optimal solution.

Testing techniques based on metaheuristic search focus mostly on unit testing [Ali et al., 2010], while techniques that tackle testing at the system level either address the problem of testing non-functional properties such as execution time [Afzal et al., 2009], or deal with the problem of testing systems where the costs of testing do not depend on complex input data structures, such as in the case of embedded systems working with input signals [Iqbal et al., 2015].

In this chapter, we propose a model-based evolutionary algorithm that relies upon a data model and a set of data mutation operators to build system test suites for data processing systems that optimises multiple objectives. The evolutionary algorithm uses data sampling and data mutation operators to generate new test inputs, and relies upon four different model-based and code-based fitness functions to evaluate how well each test input contributes to a proper robustness test suite. Model-based fitness functions exploit the data model to generate test cases that cover important aspects of the behaviour of the system by ensuring the coverage of all the different types of test inputs processed by the system, the presence of different types of data faults, and the possible violations of the constraints among test inputs. The code-based fitness function has the goal of achieving the maximum structural coverage of the SUT.

This research contributions in this chapter are:

1. An evolutionary algorithm to automate the robustness testing of data processing systems.
2. The defining of four fitness functions (model-based and code-based) that enable the effective generation of robustness test cases by means of evolutionary algorithms.
3. An extensive study of the effect of fitness functions and configuration parameters on the effectiveness of the approach using an industrial data processing system as case study.

The chapter proceeds as follows. Section 6.1 presents a list of challenges that need to be addressed when building a search algorithm for robustness testing. Section 6.2 provides an overview of the evolutionary algorithm that we designed to generate robustness tests. Section 6.3 describes how data mutation is adopted to generate new test inputs during the search. Section 6.4 presents the heuristic functions used to evaluate candidate solutions. Section 6.5 describes the seeding strategy integrated into the algorithm. Section 6.6 details how we automate the execution and validation of the generated test suites. Section 6.7 presents the empirical results obtained. Finally, Section 6.8 concludes the chapter.

6.1 Challenges for the Search-Based Generation of Robustness System Tests

Search algorithms are useful when addressing complex problems where there is a large space of candidate solutions, and one wants to choose one that optimises some chosen criteria. A typical example in software engineering is test data generation, in which a tester might want to find test data that maximises code coverage or triggers new failures.

There are many different kinds of search algorithms, including genetic algorithms, which have been widely applied in many fields. Not all search algorithms perform well on all types of problems. Each problem can have special characteristics that are better exploited by some search algorithms, whereas others might struggle.

In this chapter, we identify four main challenges that need to be addressed when designing an algorithm for the automatic generation of a system test suite: (1) configuring the algorithm to properly find a tradeoff between *exploration* and *exploitation* of the search landscape, (2) building a *tweak operation*, (3) defining effective *fitness functions* for the problem under investigation, and (4) integrating effective *seeding techniques* (i.e. techniques that speed up search by exploiting knowledge about the input domain).

One of the main discerning characteristics among search algorithms is the tradeoff they have between *exploration* and *exploitation* of the search landscape. On one hand, some algorithms (e.g. hill climbing) put more emphasis on the exploitation of the search landscape. This means that, given an evaluated solution, they will only look at "close" solutions to see if any small change could improve the chosen criterion. On the other hand, other algorithms put more emphasis on the exploration of the search space. Typical examples are population-based algorithms, in which a diverse set of solutions is maintained to consider different areas of the search landscape at the same time, in case one area turns out to feature more fitting solutions than the others.

In the context of robustness testing, exploration is very important since it enables the construction of test suites with a high diversity of test cases. In the case of SES-DAQ, for example, one wants to generate test inputs that include *IdlePacketZones* and test inputs that include *ActivePacketZones* at the same time (these fields are modelled in Fig. 3.1). However, exploitation leads to covering invalid inputs that include a specific set of data faults. For example, SES-DAQ might be able to properly process an invalid input containing faulty data that breaks the virtual channel frame count constraint of Fig. 3.5, but it may crash in the presence of both a broken constraint and a duplicated packet. To satisfy this case, the algorithm must be able to exploit the search space by both breaking the constraint of Fig. 3.5 and by duplicating a packet.

Tweak operations play an important role in guiding the search algorithm towards the exploitation of the search landscape; they modify existing solutions to generate new candidate solutions. When defining tweak operations, the characteristics of the domain should be carefully considered. For example, when a solution is represented in a binary format, a tweak operation could just flip one or more bits. However, flipping bits on a complex data transmission file would likely result in meaningless or trivially wrong input data. Therefore, one would need to exploit the information in the data models to automatically derive better tweak operations, for example by applying our model-based data mutation approach (detailed in Chapter 5).

In the context of robustness system testing, one still wants to generate faulty input data, but in a nontrivial way (e.g. by including multiple faults in a same test input). However, there is a limit on the number of faults that can be included in a same input. Although having multiple mutations that affect a same test input might help stress the robustness of the system, a number of faults that is too high might lead to inputs that are trivially recognised and discarded by the SUT.

Another very important aspect for the success of a search algorithm is the definition of a *fitness*

```
Require: fD, the field data used to generate test inputs
Require: dM, the data model used to drive tweaking
Require: budget, the maximum proportion of the search space that needs to be visited
Require: p_field, probability of sampling a new individual from the field data
Require: p_mutation, probability of mutating an individual just after sampling it
Require: p_seeding, probability of using seeding to sample from the field data
Require: minSize, the minimum size of a test input
Require: maxSize, the maximum size of a test input
Require: maxMutations, the maximum number of mutations for a same test input
Ensure: archive, the archive containing the minimised robustness test suite
 1: total = 0
 2: while total < budget do
 3:     if ( (archive.individualAvailable(maxMutations) = false)
 4:      OR (random() < p_field) ) then
 5:         ind = sampleNew(fD, dM, minSize, maxSize, p_seeding)
 6:         if random() < p_mutation then
 7:             mutate(ind)
 8:         end if
 9:     else
10:         ind = archive.sampleACopy()
11:         mutate(dM, ind)
12:     end if
13:     if ( improving(ind, archive) then
14:         archive.add(ind)
15:         for prev in archive do
16:             if subsume(ind, prev) then
17:                 archive.remove(prev)
18:             end if
19:         end for
20:     end if
21:     total = total + ind.size
22: end while
```

Figure 6.1. An evolutionary algorithm for robustness testing.

function, used to evaluate how close a solution is to optimising a chosen criterion. Such a function is problem dependent, and effort must be made to design a proper one. Ideally, one would like to exploit as much domain information as possible, but this might lead to computationally expensive fitness functions. The more time consuming the fitness function, the fewer solutions can be evaluated by a search algorithm in the same amount of time. In the case of model-based testing, this tradeoff can be very critical, as fitness functions calculated on models can be much quicker to compute than ones calculated from test case executions (e.g. using structural coverage).

Finally, in the presence of complex search spaces, the quick identification of a proper solution is often aided by the adoption of techniques that exploit knowledge about the search space to build solutions that improve the effectiveness of the search algorithm; these are known as *seeding techniques*. Different seeding techniques have been adopted in the context of software testing; for instance, a well-known approach used by techniques that generate inputs for unit testing is the reuse of constant values taken from the source code of the SUT [Fraser and Arcuri, 2012]. However, smart seeding at the system level is not as simple as that, and poses new challenges; in our case, since our test generation approach is black-box, these challenges are related to how we can exploit the information of our system data model data to improve the search-based test data generation.

6.2 Evolutionary Data Mutation Algorithm

To address the problem of generating test cases to stress the robustness of software at the system level, we propose a novel evolutionary algorithm based on an archive. Fig. 6.1 shows the algorithm.

The evolutionary algorithm uses data models to automatically generate test inputs. Such data models are designed according to the data modelling methodology introduced in Chapter 3. Test cases are generated by using the data mutation approach presented in Chapter 5.

In our context, a solution to the search problem consists of a test suite that effectively tests the capability of the software to handle invalid data. A test suite is a collection of test inputs (i.e. data files conforming to a data model) to be processed by the SUT. During the search, test inputs are generated, and those are then aggregated to form a final test suite to give as output to the user. In our context, the solution to the search problem (i.e. the test suite) cannot be represented using a fixed size data structure because the size of the test suite (i.e. the number and size of the test cases it contains) cannot be known a priori.

We did not directly use a traditional search algorithm (e.g. a genetic algorithm or a hill-climbing algorithm) due to the special characteristics of the addressed problem. For example, in system level testing, each test case execution can be computationally very expensive. So, a traditional genetic algorithm that works on a population would likely be too computationally expensive to use. Furthermore, special care would be needed to design a crossover operator that generates valid offspring. On the other hand, hill-climbing algorithms put emphasis on the exploitation of the search landscape; this is achieved by using a tweak operation that iteratively improves a single solution. In our case, it is hard to envision a single tweak operation that works at the test suite level and allows for the building of a minimised test suite that contains test inputs with high diversity.

Our customised evolutionary algorithm is based on the use of an archive of test cases, initially empty. Archives have been used in prior work related to multi-objective search algorithms (e.g. [Parks and Miller, 1998, Knowles and Corne, 1999]). The archive plays the important role of guiding the algorithm towards the exploitation of the search landscape like hill climbing, while maintaining at the same time some characteristics of population-based algorithms. Like hill climbing, the algorithm improves only a single test input at each iteration, but uses the archive to keep a collection of the best test inputs found so far (i.e. the test suite). Furthermore, our algorithm keeps solutions in the archive that are different from each other to maximise exploration, like population algorithms. The size of the archive can vary during the search and the test suite is minimised by keeping in the archive only the best individuals that contribute to overall fitness of the whole test suite (i.e. the archive). Finally, the individuals in the archive are tweaked one at a time, thus exploiting the search landscape and creating new individuals that improve the overall fitness of the test suite.

Our novel algorithm addresses all the challenges presented in Section 6.1. The tradeoff between exploration and exploitation is controlled by configuration parameters that regulate: (1) the probability of tweaking an individual from the archive versus generating a completely new individual (parameter p_{field} in Fig. 6.1), (2) the probability of working with correct test inputs versus the use of test inputs that contain at least one fault (parameter $p_{mutation}$ in Fig. 6.1), and (3) the maximum number of data faults a test input may contain (parameter $maxMutations$ in Fig. 6.1). Tweaking operations are implemented by means of mutation operators described in Section 5.2, while specific fitness functions have been developed and are described in Section 6.4. The probability of seeding is controlled by the parameter $p_{seeding}$ (further details are given in Section 6.5).

At the first iteration, the archive is empty, and a new random individual needs to be sampled. Generating new input data completely at random would result almost certainly in trivially wrong

data. An alternative is to sample according to some specific rules, if those can be defined for the addressed problem domain (e.g. a grammar in the testing of parsers). In our case, we used a different approach that relies on the sampling of field data. In the case of industrial data processing systems, we can have access to very large amounts of existing valid field data. If not already available, a large field data pool can be constructed, and then used by the evolutionary algorithm to sample from.

New individuals are sampled from the available field data by means of the function *sampleNew*, which randomly selects and returns a chunk of the available field data (Line 5). The function *sampleNew* receives as input two integer values, *minSize* and *maxSize* that indicate the minimum and maximum size of the data chunk to be sampled. Since in general an input for a data processing system does not have a size that is fixed a priori, we leave it to the software engineers to decide the range of the input size according to their domain knowledge; for example, in the case of SES we choose the values 1 and 500 for the minimum and maximum values, respectively (where these values represent the number of CADUs).

The evolutionary algorithm incrementally builds a test suite by keeping only those individuals in the archive that contribute to improving the overall combined fitness of all the currently stored test cases, which will form the final test suite. The algorithm generates test inputs by applying the technique presented in Chapter 5, that is by sampling chunks of data from the field data and by mutating these chunks to generate possibly faulty inputs. Unlike the strategies considered in Chapter 5, in this chapter we consider the possibility of applying multiple mutations to a same test input. Each test input is thus represented in terms of the offset from the beginning of the original field data file, the length of the sample, and a list of the mutations that have been applied to the sample.

The algorithm keeps exploring the search space until a given stopping condition is reached (Line 2). Since our algorithm focuses on the generation of test inputs for data processing systems, we express the stopping condition in terms of the amount of data processed to generate test cases, that is the sum of the size of all the test inputs generated during the search. At each iteration, the algorithm increments the counter of the data processed (Line 21). In the specific case of SES, we measure the size of a single test input in terms of the number of CADUs that it contains, but different measurement units (meaningful for a given domain) may be used for different systems.

At each iteration, the algorithm works by tweaking an individual (i.e. a test input). Each individual is created by sampling either the field data or the archive; this choice is driven by a probability value, the parameter p_{field} in Fig. 6.1, which indicates the probability of sampling a new individual from the field data (see Lines 3 and 4).

If no individuals are available in the archive, the algorithm samples the field data (see the condition *archive.individualAvailable() = false* in Line 3). This happens in two situations, when the archive is empty (i.e. on the first search iteration) or when all the individuals in the archive have already been mutated a maximum numbers of times. Software engineers can specify the maximum number of mutations that can be applied to the same test input. This is done to avoid trivially invalid inputs. Although in principle a test input can be mutated an infinite number of times, the presence of too many mutations (i.e. data faults) on the same test input, might transform the test input into a trivially invalid input that is easily detected by the data processing system and does not help to extensively test its robustness.

Lines 6 to 8 show that the parameter $p_{mutation}$ regulates the probability of mutating an individual just after creating it, that is, the probability of working with individuals that contain at least one data mutation. Lines 10 and 11 show that every time the algorithm samples a copy of an individual from the archive it applies a mutation to it. This is done to create a copy that differs from the original one. This tweaking operation is further described in Section 6.3.

Lines 13 and 14 show that an individual is added to the archive only if it improves the overall fitness of the archive. Similarly, the algorithm removes any individual already present in the archive that is subsumed by the last one added (see Lines 15 to 17). Individuals that are subsumed by new ones can be safely removed from the archive because they do not contribute to the overall fitness of the archive. Section 6.4 describes the assessment procedure adopted to measure how individuals contribute to the fitness of the archive.

6.3 Tweaking by Means of Data Mutation

The search algorithm tweaks individuals by applying the data mutation operators described in Section 5.2. There are six mutation operators that can be applied to an individual: Class Instance Duplication, Class Instance Removal, Class Instances Swapping, Attribute Replacement with Random, Attribute Replacement using Boundary Condition, and Attribute Bit Flipping. To apply a mutation operator to an individual, the algorithm loads into memory the data chunk corresponding to the test input as an instance of the data model (the tool described in Section 9.3 was used to implement this feature).

Each mutation operator can be applied only to a specific set of targets (our data mutation operators are configured, as described in Section 5.3). This is done to avoid making mutations that will result in the generation of trivial data faults and to ensure conformance with a domain-specific fault model. Software engineers specify the targets of each mutation operator by using appropriate stereotypes in the UML class diagram (data model). These stereotypes indicate the elements that can be mutated and the operators that can be applied to them. To mutate an individual, the algorithm randomly picks a mutation operator, identifies a possible target for the operator on the current individual, and applies the operator on the target. For example, during the generation of test inputs for SESDAQ, the algorithm may randomly choose the operator *Attribute Replacement with Random*. It then selects one of the attributes that can be mutated according to that operator, for example the attribute *sequenceCount* of class *PacketPrimaryHeader* (modelled in Fig 3.2). Finally, it identifies a specific instance to mutate, which means that it changes the value of the attribute *sequenceCount* of one of the *PacketPrimaryHeader* instances contained in the current test input.

In case a selected operator cannot be applied on a given individual, another operator is randomly selected; this can happen, for example, if the algorithm selects the attribute *sequenceCount* for replacement with random, but the current test input contains only *IdlePacketZones* (which according to Figs. 3.1 and 3.2 does not contain any *PacketPrimaryHeader* instances).

6.4 Assessment Procedure

We identify four objectives that should be fulfilled to effectively stress the robustness of the software:

- O1: include input data that covers all the classes of the data-model;
- O2: include data faults such that all the possible faults of the fault model have been covered;
- O3: cover all the clauses of the input/output constraints;
- O4: maximise code coverage.

Each of the four objectives captures how well the test inputs cover some specific targets, respectively, the classes of the data-model, the faults of the fault model, the clauses of the input/output constraints and the code instructions. Each objective defines a set of *targets* (e.g. instructions in code coverage) that the algorithm aims to cover. A given objective is fully achieved by a test suite if each of its targets is covered by at least one test input of the test suite. A portion of the targets for SES-DAQ along with their coverage for three test inputs are shown in Table 6.1 (covered targets are marked with an X). We describe below each objective, and how fitness improvements and subsumption can be defined in terms of these objectives.

Objective O1 ensures that the test suite includes test inputs that cover all the classes of the data model. Each class of the data model is univocally represented by an objective target. A test input covers a class if it contains at least one instance of the class. Table 6.1 shows that input *I3* covers, among others, classes *ActivePacketZone* and *IdlePacketZone*; input *I1* covers only class *ActivePacketZone* (there are no idle packets in *I1*).

Objective O2 ensures that an instance of each class and attribute has been mutated at least once by each mutation operator that can be applied to it (to generate test inputs covering all the faults of the fault model). Our algorithm generates faulty data (i.e. new test inputs) by applying mutation operators on instantiated field data objects. Since the attributes and classes of the data model can be mutated in different ways by applying different mutation operators, for each test input we keep track of which mutation operator has been applied to a specific class/attribute. Table 6.1, for example, shows that input *I1* contains at least one instance of a *VcduHeader* whose *vcFrameCount* has been mutated with the operator *AttributeReplacementWithRandom*, while input *I3* contains both a *CADU* with a deleted packet (operator *ClassInstanceRemoval* is marked as being applied on an instance of class *Isp*) and a *VcduHeader* whose *versionNumber* has been replaced with a random value (see operator *AttributeReplacementWithRandom* marked for the attribute *versionNumber*).

Objective O3 ensures that every clause of the input/output constraints has been exercised. Input/output constraints are expressed in the form of implications. The left hand side of the implication captures the characteristics of the input under which a given output is expected. The right hand side captures the characteristics of the expected output (see Fig. 3.5).

To measure how well the test suite stresses the conditions under which a given output is generated, it is enough to focus on the clauses contained on the left side of the implication (i.e. clauses defined over the characteristics of the test input). For each clause, we aim to have at least one test input that causes the clause to be true and another that causes the clause to be false. Each clause is thus associated to two different targets for objective O3 that trace whether the clause is true/false at least once in the input. Table 6.1 shows some targets derived for the constraint in Fig. 3.5. Table 6.1 shows that input *I1* has at least one *VcduHeader* whose *vcFrameCount* (*frameCount* in the constraint) does not correspond to the previous *vcFrameCount* (*prevFrameCount* in the constraint) plus one (this is the effect of the mutation operator *AttributeReplacementWithRandom* applied to the attribute *vcFrameCount*). The same clause can be both true and false within the same test input. This

is the case of input I1 that, in addition to having a *VcduHeader* with the invalid *vcFrameCount*, also includes *VcduHeaders* with valid *vcFrameCounts* (i.e. in the constraint, having a *frameCount* equal to *prevFrameCount* plus one).

It is noteworthy that, by focusing on the input clauses, we can measure objective O3 without the need to execute the SUT (i.e. without generating an output for a given test input). This makes the search algorithm scale even when the execution of the test cases is particularly time consuming.

Objective O4 aims to maximise the structural coverage of the source code. This is one of the means adopted by software engineers to ensure that all the implemented features have been tested at least once. Each instruction in the system is an objective target. In our implementation, we measure coverage using EclEmma, a toolkit for measuring Java code coverage [Mountainminds, 2006].

The main limitation of measuring the structural coverage of system test cases is that it requires the execution of the SUT. This may slow down the overall search process considerably and prevent the generation of results in practical time. Furthermore, in certain contexts, for example systems deployed on dedicated hardware, structural coverage might not be easily calculated in practice. For this reason, the empirical study presented in Section 6.7 aims also to determine to which extent objective O4 is subsumed by other objectives.

Our algorithm works with objectives that are not conflicting and aims to maximise the coverage of all targets. Therefore, the algorithm does not rely upon the computation of Pareto fronts, a solution adopted by others (e.g. [Zitzler et al., 2001]).

Our algorithm adds to the archive only test inputs that improve the overall fitness (i.e. a test input must cover at least one target not covered by the other inputs in the archive). Furthermore, the algorithm removes from the archive any test inputs subsumed by new test inputs. A test input i' subsumes an input i'' if, and only if, i' covers all the targets covered by i'', and either i' covers at least one target not covered by i'' or the size of i' is smaller. For example, given an archive that contains inputs *I1* and *I2*, our algorithm creates input *I3* by tweaking a copy of input *I2* (i.e. by deleting a class instance). *I3* is added to the archive because it covers target *Isp::ClassInstanceRemoval* (not covered by *I1* and *I2*). Given that *I3* subsumes *I2* the algorithm will then remove *I2* from the archive thus minimising its size.

6.5 Input Seeding

To further improve search results, we developed a novel model-driven seeding strategy that guides the search towards the identification of a diverse and complex set of test inputs.

To stress diversity in the data, the algorithm aims to generate test inputs that cover all the available data types (see objective O1 described in Section 6.4). Given that some of these data types may occur rarely in the field data, it might be highly improbable to cover these types by means of random sampling. To guarantee the coverage of all the data types, it is enough to know the locations of the different data types. For example, within a field data sample of SES-DAQ, we may have idle packets only in a very small number of the VCDUs of the bytestream. Having the location information makes

Table 6.1. Assessment of three inputs of SES-DAQ.

	Objective Targets	Test Inputs		
		I1	I2	I3
Objective O1	TransmissionData	X	X	X
	Sync	X	X	X
	VcduHeader	X	X	X
	MpduIdleHeader		X	X
	MpduActiveHeader	X	X	X
	IdlePacketZone		X	X
	ActivePacketZone	X	X	X
	Isp	X	X	X
	...			
Objective O2	VcduHeader.versionNumber::AttributeReplacementWithRandom		X	X
	VcduHeader.vcFrameCount::AttributeReplacementWithRandom	X		
	Isp::ClassInstanceRemoval			X
	Isp::ClassInstanceDuplication			
	Isp::ClassInstancesSwapping			
	...			
Objective O3	True : prevFrameCount < 16777215	X	X	X
	True : frameCount <> prevFrameCount + 1	X		
	True : prevFrameCount = 16777215			
	True : frameCount <> 0	X	X	X
	...			
	False : prevFrameCount < 16777215			
	False : frameCount <> prevFrameCount + 1	X	X	X
	False : prevFrameCount = 16777215	X	X	X
	False : frameCount <> 0			
	...			
O4	SesDaq.java:Line 10	X	X	X
	SesDaq.java:Line 11	X		
	...			

it easier for the search algorithm to load multiple data chunks containing idle packets; otherwise, the chance of loading idle packets is low when only resorting to random sampling.

To stress complexity our algorithm looks for test inputs that contain instances of two or more subclasses belonging to the same generalisation. In the case of SES-DAQ, this corresponds to the case of an input containing a transition between two alternate data types (e.g. from idle to active packet zone). These complex inputs are interesting for robustness testing because one can assume that handling heterogeneous data zones might be more prone to processing failures than homogeneous ones.

Our seeding strategy works by first processing the field data to build a *seeding pool* that contains data chunks that are useful to stress both diversity and complexity. To maximise diversity, we identify, for each class *S* that is a subclass of a generalisation: (1) data chunks that contain *at least* one instance of the subclass *S*, (2) data chunks that contain *only* instances of the subclass *S* (i.e. data chunks that contain instances of *S* but that do not contain instances of other classes belonging to the same hierarchy of *S*). To maximise complexity, we identify, for each pair of classes that belong to a generalisation, data chunks that contain at least one instance of each class in the pair. In the case of SES-DAQ, this ensures that we test scenarios where two different data types are processed in sequence (e.g. idle and active packets). Table 6.2 shows a list with the characteristics of the data chunks we identify for the SES-DAQ data model in Figs. 3.1 and 3.2.

The *seeding pool* can be used when a new individual is sampled. When seeding is enabled, the

Table 6.2. List of the characteristics of the input data used to drive seeding for SES-DAQ.

Only MpduIdleHeader instances are included
At least one MpduIdleHeader instance is included
Only MpduActiveHeader instances are included
At least one MpduActiveHeader instance is included
Only IdlePacketZone instances are included
At least one IdlePacketZone instance is included
Only ActivePacketZone instances are included
At least one ActivePacketZone instance is included
Both MpduIdleHeader(s) and MpduActiveHeader(s) are included
Both IdlePacketZone(s) and ActivePacketZone(s) are included
Only ActivePacketDataField instances are included
At least one ActivePacketDataField instance is included
Only IdlePacketDataField instances are included
At least one IdlePacketDataField instance is included
Both IdlePacketDataField(s) and ActivePacketDataField(s) are included

algorithm selects one of the chunks in the seeding pool. In the case of SES-DAQ, this is done by first selecting one on the characteristics listed in Table 6.2, and then by loading a data chunk that presents such characteristics. Software engineers can tune the use of seeding by means of a parameter for the search algorithm that indicates the probability of applying seeding when sampling a data chunk from the field data ($p_{seeding}$ in Fig. 6.1).

Higher values of $p_{seeding}$ guarantee that all the characteristics are covered, at the expense of a free exploration of the search space (which may lead to the sampling of complex test inputs not identified by predefined seeding characteristics).

6.6 Testing Automation

The evolutionary algorithm generates a minimised robustness test suite that is kept in an archive. The test suite consists of a set of test inputs that can be executed against the data processing system SUT. The oracle relies on the modelling methodology introduced in Section 3.3.

After the execution of a test case, the oracle simply loads the test input and the test output as an instance of the data model, and checks if the OCL constraints of the data model are satisfied. Unsatisfied constraints indicate the presence of a failure (i.e. unexpected or missing output) and are reported to the software engineers. Similarly, crashing executions are reported.

6.7 Empirical Evaluation

We performed an empirical evaluation in order to respond to the following research questions:

- **RQ1:** How does the search algorithm compare with random and state-of-the-art approaches?
- **RQ2:** How does fitness based on code coverage affect performance?
- **RQ3:** How does smart seeding affect performance?
- **RQ4:** What are the configuration parameters that affect performance?
- **RQ5:** What configuration should be used in practice?

6.7.1 Subject of the study

As subject of our study we considered the industrial SES-DAQ system (introduced in Chapter 2). SES-DAQ is a good example of a data processing system dealing with complex input and output data, written in Java (having 32,170 bytecode instructions). The same data model developed for the evaluation of Chapter 5 was used. The data model of SES-DAQ includes 82 classes with 322 attributes and 56 associations. As an input for our approach, we considered a large transmission file containing field data provided by SES, the same adopted for the empirical evaluation in Chapters 4 and 5. The size of the transmission file is 1 million CADUs (or about 2 GB), containing 1 million VCDUs belonging to four different virtual channels.

6.7.2 Experimental settings

To answer our research questions, we carried out a series of experiments. Since our search algorithm depends on several parameters, we evaluated several possible configurations.

The *minSize* and *maxSize* parameters (i.e. the minimum and the maximum size of a test input, measured in CADUs) were fixed to 1 and 500, respectively. We used three different values for p_{field}: 0.3, 0.5, and 0.8. For $p_{mutation}$, we considered the values: 0.0, 0.5, and 1.0. For *maxMutations*, we used: 1, 10, and 100. For $p_{seeding}$, we used two values, 0.0 and 0.5, which means that in one case the seeding strategy was not used, while in the other case the seeding was applied with a 50% probability every time a new input was sampled.

In order to give the algorithm some degree of freedom when exploring the search space, we do not consider the case in which inputs are selected exclusively according to the smart seeding strategy. Finally, we considered cases with and without the code coverage fitness function. This led to $3 \times 3 \times 3 \times 2 \times 2 = 108$ different configurations.

The search budget (in the case of SES-DAQ, the number of CADUs inspected when building new inputs during search) might vary from project to project. For this reason, we evaluated each of the 108 configurations of the algorithm on five different search budgets from $50,000$ to $250,000$ CADUs (in steps of $50k$). $250,000$ CADUs correspond to one fourth of the transmission file used for building test inputs. This led to $108 \times 5 = 540$ different configurations of the search algorithm.

Because search algorithms have a random component, to take into account the effects of such randomness on the final results, each of the experiments was repeated five times with a different random seed. This led to a total of $540 \times 5 = 2,700$ runs of the algorithm. Because each run could take between ten and thirty-five hours, we used a large cluster of computers to run these experiments [Varrette et al., 2014].

6.7.3 Cost and effectiveness metrics

We want to assess and compare cost-effectiveness among automatically generated test suites. Code coverage is used as a measure of test effectiveness as it helps assess how complete the test suite is from a structural and functional standpoint. We measure the code coverage in terms of the number of bytecode instructions covered by the test suite by using EclEmma [Mountainminds, 2006]. Maximising code coverage within time constraints is often an objective among system testers. Even small

Table 6.3. Comparison between the best search algorithm configuration and random search. (Seeding disabled. Code coverage fitness enabled.)

Budget	Configuration	Coverage (Avg/Min/Max)	# Tests (Avg/Min/Max)
50k	Best: r=0.5,m=1,n=100	**23424.4** / 23407 / 23448	**28.4** / 19 / 32
	BO: r=0.5,m=1,n=100	**23424.4** / 23407 / 23448	**28.4** / 19 / 32
	Rand: r=1,m=1,n=1	23386.8 / 23341 / 23424	43.2 / 38 / 46
100k	Best: r=0.5,m=1,n=100	**23487.8** / 23461 / 23577	**31.6** / 25 / 35
	BO: r=0.5,m=1,n=100	**23487.8** / 23461 / 23577	**31.6** / 25 / 35
	Rand: r=1,m=1,n=1	23436.8 / 23428 / 23458	52.0 / 50 / 57
150k	Best: r=0.5,m=1,n=100	**23502.0** / 23471 / 23577	**34.0** / 30 / 38
	BO: r=0.5,m=1,n=100	**23502.0** / 23471 / 23577	**34.0** / 30 / 38
	Rand: r=1,m=1,n=1	23453.4 / 23438 / 23480	57.8 / 55 / 64
200k	Best: r=0.5,m=0.5,n=100	**23519.6** / 23464 / 23618	**34.6** / 28 / 38
	BO: r=0.5,m=1,n=100	23513.4 / 23476 / 23579	36.0 / 31 / 41
	Rand: r=1,m=1,n=1	23465.8 / 23449 / 23490	60.2 / 57 / 66
250k	Best: r=0.5,m=1,n=10	**23538.6** / 23463 / 23631	38.4 / 31 / 43
	BO: r=0.5,m=1,n=100	23515.2 / 23480 / 23579	**36.4** / 33 / 40
	Rand: r=1,m=1,n=1	23482.6 / 23452 / 23499	62.4 / 60 / 69

improvements in coverage can help exercise important corner cases. For example, in our case study, additionally covered instructions often turned out to be critical blocks of code including exception handling and critical scenarios. We also compare test suites with respect to their size because, given two test suites having identical coverage, one would prefer the smaller one, entailing a lower testing and debugging cost. The size of a test suite is measured in terms of the number of test inputs in the test suite. Smaller test suites are of practical importance as, in many systems, system testing must be performed on actual deployment hardware or a dedicated, realistic testing platform, which requires some degree of tuning or simulation to run test cases. Access time to such platforms can also be limited.

6.7.4 RQ1: How does the search algorithm compare with random and state-of-the-art approaches?

The effectiveness of a search-based algorithm highly depends on the nature of the problem to solve. For example, if the solution space of the problem is flat, that is if the fitness function does not provide any gradient, then a search-based algorithm might perform even worse than a random approach.

RQ1 aims to evaluate the usefulness of the search-based algorithm proposed in this chapter by comparing it with a random algorithm and with a simple model-based algorithm.

However, a direct comparison with a trivial random generation approach would not bring any useful result. For example, test inputs containing random data may be trivially invalid and useless for extensively testing the system. For this reason, we use as baseline for the comparison the random algorithm employed in Chapter 5; the algorithm (shown in Section 5.5.1) samples a portion of the field transmission file, randomly selects a mutation operator and applies it to one of the elements where it can be applied.

The algorithm employed in Chapter 5 does not support the generation of test inputs of variable size; furthermore, it does not minimise the generated test suite. To address these limitations and

perform a fairer comparison, we execute an improved version of the random algorithm employed in Chapter 5 by using our search algorithm with a specific configuration: *minSize* and *maxSize* are set to the same values as the search algorithm; $p_{field} = 1$, to generate a new test input at each iteration by sampling the field data; and $p_{mutation} = 1$, to always mutate the sampled test input (as was done in Chapter 5). The value chosen for $maxMutations$ is irrelevant because we generate a new test input at each iteration (because of $p_{field} = 1$).

To compare with a simple model-based approach, we consider the results achieved with the *All Possible Targets (APT)* approach proposed in Chapter 5. The APT mutation strategy (shown in Section 5.5.2) ensures that each class or attribute of the data model is mutated at least once by each of the mutation operators that can be applied to it.

Table 6.3 shows the comparison of the search algorithm with random search. Columns *Budget* and *Configuration* report the search budget and the best configurations found, column *Coverage* reports the average, minimum and maximum coverage achieved by the test suite, and column *# Tests* reports the average, minimum and maximum number of test inputs in each test suite. For each search budget we identified the best configuration out of the 54 configurations of the search algorithm with seeding disabled (*Best* in Table 6.3). Furthermore, we identified the best configuration on average over all the search budgets (*BO* in Table 6.3). The comparison with *BO* is fairer since the configuration indicated as *BO* is not optimised for a specific search budget, but is a stable, good overall configuration. For each configuration we report the probability of random sampling (r), the probability of applying mutation when sampling (m), and the maximum number of allowed mutations in a test (n). The best values for maximum coverage and minimum test suite size appear in bold. Our comparison did not include configurations with seeding because it is an optimisation of the search algorithm. The impact of seeding is addressed in *RQ3*.

The search algorithm presented in this chapter provides better results than both the random approach and the APT algorithm presented in Chapter 5. APT achieves an average coverage of 23,283 instructions, which is less than the coverage obtained with the search and random approaches (see Table 6.3). This is mostly because APT focuses on the coverage of the model and stops after sampling many fewer CADUs (at most 20,000 CADUs, while the lowest search budget is 50,000). In summary, the search algorithm presented in this chapter generates significantly less test inputs while achieving better coverage. This mainly results from the adoption of fitness functions that help minimise the test suites by keeping only useful test inputs in the archive.

Results also show that the search algorithm achieves better coverage than the random approach. The difference in coverage ranges between 37.6 and 56 instructions. Though the difference in coverage might not appear large, as discussed earlier, even small increases in coverage might exercise important corner cases. Once again, the search algorithm generates significantly smaller numbers of test inputs (e.g. 57.8 versus 34 on average for a 150k budget).

Our conclusions hold even considering the random variation across runs, which is small, as shown by the Min and Max values appearing in Table 6.3.

Table 6.4. Comparisons between best search algorithm configurations based on whether code coverage is employed in the fitness evaluation (column 'Code'), and on whether smart seeding is activated (column 'Seeding').

Budget	Code	Seeding	Configuration	Coverage	#Tests	#Mutations
50k	False	0.0	Best: r=0.5,m=1,n=100	23361.4	**17.0**	4.8
	True	0.0	Best: r=0.5,m=1,n=100	23424.4	28.4	3.6
	False	0.5	Best: r=0.5,m=1,n=10	23417.2	21.0	4.0
	True	0.5	Best: r=0.5,m=1,n=10	**23428.4**	34.2	3.2
	True	0.5	BO: r=0.3,m=0,n=10	23401.8	27.0	4.3
100k	False	0.0	Best: r=0.3,m=1,n=10	23404.4	**16.8**	8.2
	True	0.0	Best: r=0.5,m=1,n=100	**23487.8**	31.6	4.9
	False	0.5	Best: r=0.5,m=1,n=10	23442.2	21.0	6.4
	True	0.5	Best: r=0.3,m=0,n=10	23487.0	33.2	5.6
	True	0.5	BO: r=0.3,m=0,n=10	23487.0	33.2	5.6
150k	False	0.0	Best: r=0.8,m=1,n=100	23418.4	28.2	4.0
	True	0.0	Best: r=0.5,m=1,n=100	23502.0	34.0	6.0
	False	0.5	Best: r=0.5,m=1,n=100	23447.4	**23.4**	7.5
	True	0.5	Best: r=0.3,m=0,n=10	**23528.2**	35.6	6.5
	True	0.5	BO: r=0.3,m=0,n=10	**23528.2**	35.6	6.5
200k	False	0.0	Best: r=0.8,m=1,n=100	23426.0	28.0	4.7
	True	0.0	Best: r=0.5,m=0.5,n=100	23519.6	34.6	6.7
	False	0.5	Best: r=0.5,m=1,n=100	23456.0	**23.2**	9.2
	True	0.5	Best: r=0.3,m=0,n=10	**23551.0**	37.2	7.0
	True	0.5	BO: r=0.3,m=0,n=10	**23551.0**	37.2	7.0
250k	False	0.0	Best: r=0.8,m=1,n=100	23433.2	28.6	5.4
	True	0.0	Best: r=0.5,m=1,n=10	23538.6	38.4	7.1
	False	0.5	Best: r=0.5,m=1,n=100	23461.8	**23.6**	10.3
	True	0.5	Best: r=0.3,m=0,n=10	**23554.4**	37.2	7.4
	True	0.5	BO: r=0.3,m=0,n=10	**23554.4**	37.2	7.4

6.7.5 RQ2: How does fitness based on code coverage affect performance?

To answer *RQ2*, Table 6.4 shows, for each search budget, the best configurations (*Best*) with and without code coverage fitness (*Code*), with seeding enabled and disabled (*Seeding*). Furthermore, Table 6.4 also reports the configuration that performs better on average over all the search budgets (*BO* in Table 6.4).

Enabling code coverage fitness results in higher coverage but it comes, however, at the expense of a significantly larger test suite. All configurations in Table 6.4 with higher coverage enable coverage fitness whereas all the ones with smaller test suites do not. For small search budgets, the difference in coverage when enabling coverage fitness is small, thus suggesting that relying on model information is enough, not requiring test execution for generating test cases.

6.7.6 RQ3: How does smart seeding affect performance?

Table 6.4 shows that smart seeding has a positive effect on cost-effectiveness when the search budget is above 150k. In these cases, smart seeding is always part of the configurations that achieve the highest coverage or the lowest number of test cases.

6.7.7 RQ4: What are the configuration parameters that affect performance?

For each of the 108 configurations, we calculated their average coverage over all the search budgets (thus considering 25 test suites for each configuration). We ranked these configurations based on their average coverage. A detailed analysis showed that coverage fitness was enabled in the top 15 configurations, and never by the worst 15, thus showing its importance to guide the search. The effect of seeding is however much less visible as it depends on other parameters.

Different parameters have different effects depending on the selected search budget. For example, Table 6.4 shows that, for small search budgets (i.e. search budgets including at most 100,000 CADUs) search achieves better results when more focused on exploitation (i.e. having a low probability of random sampling, like 0.3 and 0.5). Table 6.4 also shows that with higher search budgets, in the absence of coverage fitness or seeding, putting more emphasis on the exploration of the search landscape (i.e. using a 0.8 probability of random sampling) pays off. This result is expected since, with a higher search budget, random sampling allows for the sampling of much of the field data transmission file, roughly one fourth.

Further, Table 6.4 shows some interesting side effects. For search budgets above or equal to 150,000, using either seeding or code coverage decreases the need to explore the search landscape (probability of random sampling decreasing from 0.8 to 0.5). If both are used, even less exploration is needed (probability of random sampling equal to 0.3). This phenomenon can be easily explained for smart seeding, as it does provide more diverse and useful samples in the search landscape. In the case of code coverage, this phenomenon occurs because some rare inputs contribute to code coverage but not to other search objectives. Enabling code coverage fitness prevents the algorithm from discarding rare inputs that contribute to code coverage once they are found. In the absence of code coverage fitness, such rare inputs can be easily missed if there is low variety in the test inputs stored in the archive. In the case of higher values of exploration, there is going to be higher variety in the archive, which increases the probability of having those rare inputs.

For completeness, Table 6.4 reports also the average number of mutations per test input (column *#Mutations*). Although the presence of multiple mutations (i.e. data faults) in a same input may trigger hard-to-detect, complex failures, it could also complicate debugging. Table 6.4 shows that on average the number of mutations is low compared to the maximum allowed (e.g. 5.4 versus 100 for a budget of 250k), thus making eventual debugging operations easier[1].

6.7.8 RQ5: What configuration should be used in practice?

The best overall configuration (see Table 6.4) is using $p_{field} = 0.3$ (small probability of sampling a new test data at random instead of reusing the ones already in the archive), $p_{mutation} = 0$ (do not mutate new inputs immediately when sampled), and $maxMutations = 10$. Furthermore, it does use seeding and the code coverage fitness function.

For each search budget, we ran experiments only five times per configuration, due to the high time cost of running them. On one hand, this is useful to get a general picture of cost-effectiveness trends among different parameter configurations. On the other hand, it makes it harder to compare

[1] To further simplify debugging our implementation keeps track of the list of mutations applied on each test input and a reference to the mutated element.

Table 6.5. Statistical comparisons of best overall (BO) configuration against best with no seeding, best with no code coverage fitness function, and random with code coverage.

Budget	Configuration	Coverage	$\hat{A}_{12}$	p-value
50k	BO	23401.8	-	-
	BO no seeding	23424.4	0.32	0.403
	BO no code	23417.2	0.40	0.676
	Rand with code	23386.8	0.64	0.530
100k	BO	23487.0	-	-
	BO no seeding	23487.8	0.56	0.835
	BO no code	23442.2	0.92	0.037
	Rand with code	23436.8	0.96	0.022
150k	BO	23528.2	-	-
	BO no seeding	23502.0	0.68	0.403
	BO no code	23443.4	1.00	0.012
	Rand with code	23453.4	0.94	0.027
200k	BO	23551.0	-	-
	BO no seeding	23513.4	0.80	0.144
	BO no code	23448.6	1.00	0.012
	Rand with code	23465.8	1.00	0.012
250k	BO	23554.4	-	-
	BO no seeding	23515.2	0.80	0.144
	BO no code	23450.4	1.00	0.012
	Rand with code	23482.6	1.00	0.012

two specific configurations, as the randomness of the algorithm does introduce some degree of noise. Is the best found configuration really better than the others? To address this issue, one could run more experiments just on a subset of configurations of interest. For example, in our case, we are interested in what is the best overall configuration, how it differs when seeding and code coverage fitness functions are or are not used, and how it compares with random search. However, even with just five runs, we obtained statistically significant results regarding our research questions, as reported in Table 6.5.

Following the guidelines in [Arcuri and Briand, 2014], we used the Wilcoxon-Mann-Whitney U-test to check statistical difference quantified by the Vargha-Delaney standardised effect size. For large search budgets, code fitness has the strongest effect (e.g. for a 250k budget, *BO* is statistically significantly better than *BO no code*, with an effect size of 1.00). Also for large budgets, random yields statistically and practically worse results than search (e.g. for a 250k budget, *BO* is statistically significantly better than *Rand with code*, with an effect size of 1.00). But for low budgets, no statistically significant results are visible, which can be explained by low statistical power resulting from lower effect size (less search) and a small number of observations.

6.8 Conclusion

Building a minimal robustness test suite for data processing systems, with high fault revealing power, is complicated by multiple factors: the complex structure of the test inputs that present several constraints among their data fields, the need for generating a set of inputs that covers both the functional specifications and the data faults captured by a given fault model, and the possibility to have multiple data faults in a same input.

We designed a novel evolutionary algorithm that addresses these challenges by: generating complex test inputs by means of data mutation, relying upon model-based and code-based fitness functions, and identifying optimal test suites by managing the tradeoff between the exploration and the exploitation of the search landscape thanks to a set of configuration parameters. The fitness functions capture aspects that are relevant for robustness testing, that is, how well each input covers the structure of the input data, the fault model, the functional specifications, and the structure of the system.

Empirical results obtained by applying our search-based testing approach to test an industrial data processing system show that it outperforms previous approaches (introduced in Chapter 5) based on fault coverage and random generation: higher code coverage is achieved with smaller test suites.

Furthermore, we show that although a fitness function that includes code coverage is essential to maximise the coverage of the generated test suite, fitness functions based on models alone can achieve good coverage results, while significantly reducing test suite size. This is of practical importance as test generation is much quicker and often more practical when no test execution is required. Finally, we identified a best configuration for our search algorithm that returns better results regardless of the search budget; this configuration facilitates the application of our approach and includes smart seeding, which turns out to be a key feature in improving search results.

Chapter 7

Testing of New Data Requirements

When testing data processing systems, software engineers often take advantage of the availability of a huge quantity of real world data to perform system level testing. For example, when developing a web crawler, software engineers can rely upon existing web pages to verify the robustness[1] of the system. The approaches of this dissertation presented so far automatically generate test inputs by modifying existing field data. However, in the presence of new requirements, where there is a need to deal with new data formats, software engineers may no longer have the benefit of having existing real world data with which to perform testing. Typically, new test inputs that comply with the new format would have to be written.

This situation is very common, especially in industry, where requirements are continuously changing. For example, the approach of this chapter was motivated by the needs of SES. The SES-DAQ (data acquisition system for satellite transmissions, introduced in Chapter 2) has been developed for the ESA Sentinel series of satellites. The first of the Sentinels is already in orbit and more Sentinel satellites will be launched in the coming years. Real transmission data for the first of the Sentinel mission types is available for testing the data acquisition system. For other Sentinel mission types, real transmission data is not yet available. Additionally, during the development process it is not uncommon for the transmission data specifications to continue to change. Hence, an approach that supports the automatic generation of valid synthetic transmission data files is necessary to ensure that the DAQ system can be thoroughly tested throughout development.

In practice, testing data processing systems involves the handcrafting of inputs, which requires the creation and editing of large and complex data structures saved in binary files. Furthermore, given available time and resources, test files should be as realistic as possible in terms of size and content, as large sizes will stress the system more and are more likely to reveal faults. The size and complexity of the test inputs makes this process error-prone and expensive, especially in the presence of changing requirements that force software engineers to modify or rewrite already defined complex test inputs.

Most existing approaches for the automatic generation of test inputs cannot be used because they require extensive specifications in the form of CFGs, which cannot capture all the complex relationships between data fields. A few approaches that use extended grammars to capture such relationships

[1] Robustness is “the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [ISO/IEEE, 2010].

exist (e.g. [Xiao et al., 2003]); however, these approaches are not based on a generic language to express such relationships and are limited in the types of relationships they can express.

Constraint solvers that process constraints expressed using the OCL language [Ali et al., 2013], Alloy [Anastasakis et al., 2007], or constraint programming [Cabot et al., 2008] can be used to generate test inputs from scratch. However, existing approaches do not scale in the presence of numerous constraints and complex and highly structured data, as visible for Alloy in our empirical study. An additional limitation of these approaches is that they require software engineers to model all the constraints on input data, which may require a lot of time.

To limit the modelling effort, other approaches generate test inputs by mutating existing field data [Shan and Zhu, 2009]. Our own approach, presented in Chapters 5 and 6, that generates test inputs by mutating available field data represented by a data model is not applicable when existing field data do not comply with new data requirements.

In this chapter, we propose an approach that modifies existing field data to generate test inputs for testing new requirements. The approach combines data modelling and constraint solving. The approach scales in the presence of complex and structured data, thanks to both the reuse of existing field data and the adoption of an innovative input generation algorithm based on slicing the model into parts; it reuses field data for parts unaffected by requirements changes, and then iteratively updates parts that are affected, by using data generated by means of constraint solving.

The contributions presented in this chapter are:

1. An automated, model-based approach to modify field data to fit new data requirements for the purpose of testing data processing systems.
2. A scalable test generation algorithm based on data slicing that allows for the incremental invoking of a constraint solver to generate new or modified parts of the updated field data.
3. An industrial empirical study demonstrating (1) scalability in generating new field data and (2) coverage of new data requirements by generated field data in addition to a comparison with expert, manual testing.

The chapter is structured as follows. Section 7.1 summarises the challenges of the research problem addressed in this chapter. Section 7.2 overviews the approach. Sections 7.3 and 7.4 present the details of the core contributions presented in the chapter: reuse of existing data and generation of missing or invalid data with constraint solving. Section 7.5 shows, by means of an example, how the algorithm proposed by this chapter correctly generates a complete solution. Section 7.6 discusses the empirical results obtained. Section 7.7 concludes the chapter.

7.1 Running Example

To define data requirements, we rely upon the data modelling methodology described in Chapter 3.

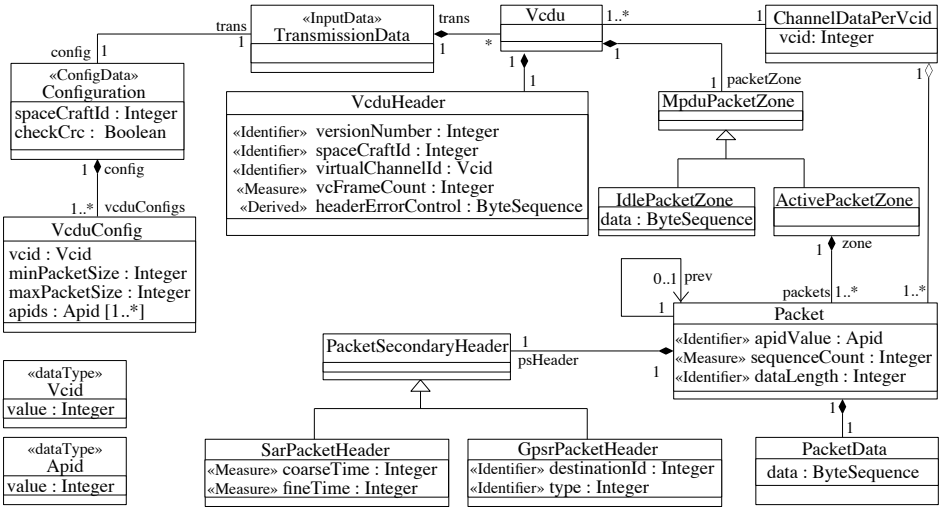


Figure 7.1. Condensed input/configuration data model for the SES-DAQ working with Sentinel-1 satellites. The model uses elements shown in the CADU-level and ISP-level input data model figures (respectively, Figs. 3.1 and 3.2) as well as elements of the configuration data model (Fig. A.1 of Appendix A).

7.1.1 Existing data requirements

For the running example presented in this chapter, we present a condensed version of the SES-DAQ Input and Configuration data models in Fig. 7.1. The condensed data model uses most of the elements shown in the CADU-level figure (Fig. 3.1) with some changes: the *TransmissionData* is directly composed of *Vcdus* and we do not consider the *MpduHeader*. With reference to the ISP-level figure (Fig. 3.2), the condensed model has a *Packet* class that is a substitute for the *Isp* and *PacketPrimaryHeader* classes; the *Packet* class also directly contains a *PacketSecondaryHeader* and a *PacketData* class. Note that in Fig. 7.1, we now define the subtypes of the *PacketSecondaryHeader* class, which is directly relevant for the running example. With reference to the Configuration Data figure (Fig. A.1), Fig. 7.1 contains the configuration elements that are directly relevant to the running example.

Fig. 7.1 shows how we model Sentinel-1 mission transmission (input) data processed by SES-DAQ. The Sentinel-1 mission is the first of several planned Sentinel missions. One Sentinel-1 satellite is already in orbit. Note that for Sentinel-1, the *PacketSecondaryHeader* has two possible values: (1) the *SarPacketHeader* is used by packets containing data generated by the Synthetic Aperture Radar (SAR) instrument and (2) the *GpsrPacketHeader* is associated with packets containing Global Positioning System Receiver (GPSR) data.

The data model also captures the structure of configuration files. In the case of SES-DAQ, the structure of configuration files is captured by classes *Configuration* and *VcduConfig*. The configuration files specify how the SES-DAQ software should process data and also define what data values are valid for received transmissions. For example, the attribute *checkCrc* of class *Configuration* indicates whether or not the software should check for packet correctness by using CRC information. A *Configuration* also contains a collection of *VcduConfig* instances, one for each valid virtual channel. Class *VcduConfig* provides run-time information characterising the expected contents of a valid virtual channel, for example: a valid VCID value, *vcid* in Fig. 7.1; and a collection of valid packet identifiers (specifically, APID values), *apids* in Fig. 7.1.

```
1  context Packet inv:
2      ( self.apidValue.value=1 and self.psHeader.oclIsTypeOf(SarPacketHeader) ) or
3      ( self.apidValue.value=2 and self.psHeader.oclIsTypeOf(GpsrPacketHeader) )
```

Figure 7.2. Mapping of packet type numbers to specific *PacketSecondaryHeader* subclasses.

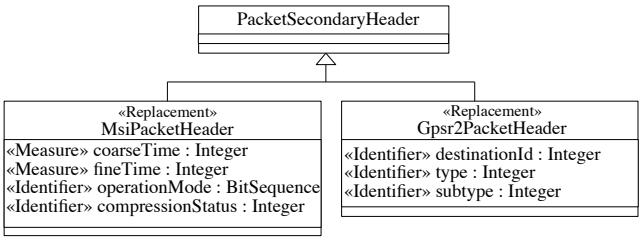


Figure 7.3. Portion of the data model for SES-DAQ that handles new data requirements specific for Sentinel-2 satellites.

```
1  context Packet inv:
2      ( self.apidValue.value=1 and self.psHeader.oclIsTypeOf(SarPacketHeader) ) or
3      ( self.apidValue.value=2 and self.psHeader.oclIsTypeOf(GpsrPacketHeader) ) or
4      ( self.apidValue.value=3 and self.psHeader.oclIsTypeOf(MsiPacketHeader) ) or
5      ( self.apidValue.value=4 and self.psHeader.oclIsTypeOf(Gpsr2PacketHeader) )
```

Figure 7.4. New OCL constraint that replaces the one in Fig. 7.2. The constraint is updated to include the new packet secondary header types.

```
1  context VcduHeader inv:
2      self.vcdu.transmissionData.configuration.vcduConfig.vcid→exists(x | x = self.virtualChannelId)
```

Figure 7.5. OCL constraint involving configuration parameters.

A constraint specific to the Sentinel-1 data model is captured by the following OCL constraint: Fig. 7.2 shows an input constraint that indicates that a *PacketSecondaryHeader* is of type *SarPacketHeader* only if the APID value of the packet is equal to 1.

7.1.2 New data requirements

New data requirements potentially result in changes to both the data model and the contents of the configuration files of the system. A change to the data model corresponds to a modification of the class diagram or the OCL constraints, while changes in configuration files consist of changes in the values assigned to configuration parameters.

Fig. 7.3 shows a portion of the data model of SES-DAQ that has been updated to process data transmitted by Sentinel-2 mission satellites. In the case of Sentinel-2, a *PacketSecondaryHeader* can be either of type *MsiPacketHeader* or *Gpsr2PacketHeader*. These two kinds of packet headers contain information that is different from the packet headers transmitted by Sentinel-1 satellites. If we compare, for example, the *GpsrPacketHeader* transmitted by Sentinel-1 and the *Gpsr2PacketHeader* transmitted by Sentinel-2 we notice that both provide information about the *destinationId*, and the *type* of the content being sent, while only the latter provides a *subtype* field that provides additional information characterising the content.

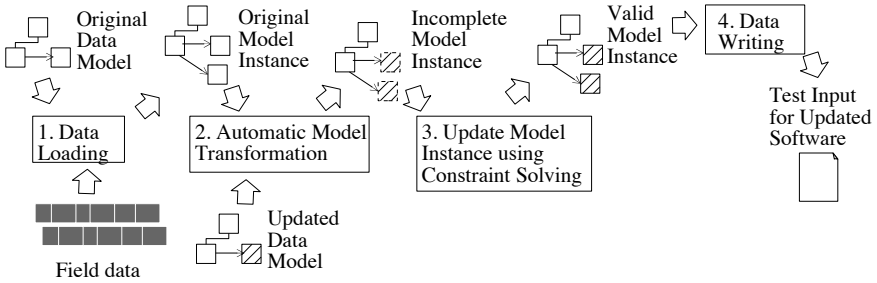


Figure 7.6. Automatic generation of test inputs for new data requirements.

Data constraints might change as well. Fig. 7.4 shows an example for SES-DAQ where the OCL constraint in Fig. 7.2 has been modified by specifying the new mappings between the packet type and the two new *PacketSecondaryHeaders*.

In addition to changes in the data model, new data requirements often imply changes in the configuration files used to run the software. Different software versions may require different configuration parameters, although changes in the content of the configuration files may not imply changes in the structure (or the related constraints) of the configuration classes captured by the data model (e.g. the configuration file for SES-DAQ has the same structure whether it is used to process Sentinel-1 or Sentinel-2 data). The configuration values to be used with a given version of the software are typically set in the field, before executing the software. When generating test inputs for the new requirements, it is thus necessary to properly set the values appearing in configuration files, because they are referenced by OCL constraints involving configuration parameters. An example is given by the constraint in Fig. 7.5 that states that the virtual channel identifier specified in a *Vcdu* header (attribute *virtualChannelId* of class *VcduHeader*) must be equal to one of the virtual channel identifiers present in the configuration file (attribute *vcid* of class *VcduConfig*).

7.2 Automatic Generation of Test Inputs for New Data Requirements

We automatically generate test inputs for new requirements by adapting existing field data. To this end, we combine model transformations with constraint solving. Model transformations enable the partial reuse of existing field data, while constraint solving allows for the generation of missing data that fulfils the updated constraints.

Fig. 7.6 shows the four steps of the approach. In Step 1 we load a chunk of field data in memory as an instance of the original data model (Original Model Instance). In the case of SES-DAQ, the process is automated by using a parser (the Data Loading component of our toolset, introduced in Chapter 9) that makes use of the modelling approach described in Section 3.4.

In Step 2 we generate an instance of the updated data model by means of a model transformation applied to the Original Model Instance. The result of the model transformation is an instance of the updated data model that is incomplete (Incomplete Model Instance); in fact, it contains only the information that can be directly derived from the Original Model Instance.

In Step 3 we generate a valid instance of the updated data model by means of constraint solving.

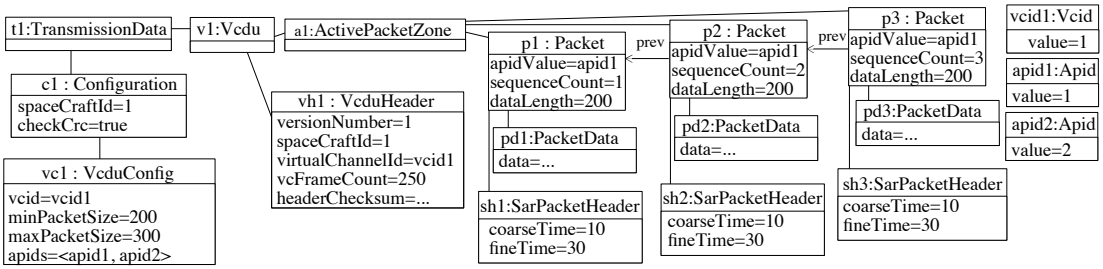


Figure 7.7. Example of an instance of the Original Data Model of SES-DAQ visualised using the object diagram notation.

As the underlying solver we use the Alloy Analyzer [Jackson, 2015]. Alloy is a modelling language for expressing complex structural constraints [Jackson, 2002], which has been successfully used to generate test inputs for testing object-oriented programs [Khurshid and Marinov, 2004]. We rely upon UML2Alloy [Anastasakis et al., 2007] to generate an Alloy model that corresponds to the class diagram and the OCL constraints of the data model.

To generate the concrete test inputs to be processed by the SUT (e.g. a binary file in the case of SES-DAQ), the content of the Valid Model Instance is written in the format processed by the SUT (Step 4). For example, to produce test inputs for SES-DAQ, we rely upon the toolset that we already used in our previous work; this toolset writes the content of the Valid Model Instance back to a file as a stream of bytes (as described in Section 9.3.3).

The following sections describe in detail Steps 2 and 3, which are the core contributions of this chapter.

7.3 Automatic Model Transformations to Generate Incomplete Model Instances

The proposed technique is able to automatically generate an incomplete instance of the updated data model in the presence of changes that alter the information provided by the data model. These changes correspond to removals, additions and replacements of classes and attributes.

The technique does not deal with model refactoring (i.e. changes that alter the structure of the data model but preserve the information provided by the data model). Model refactoring can be effectively implemented by means of model transformations [Mens and Tourwé, 2004].

To create an instance of the updated data model, the technique copies and adapts the instance of the original data model. When attributes or classes have been removed, the technique simply ignores the deleted attributes or classes when creating the copy. To deal with classes added to the data model, the technique creates an instance of each new class along with a new association instance linking the new class instance to its containing class instance. The new class instances are tagged as being *incomplete*. Similarly, the technique tags as *incomplete* the instances of classes with attributes that have been introduced in the updated data model. In the case of the replacement of classes, we rely upon a stereotype, named *«Replacement»*, that is used by software engineers in the data model to indicate that a class replaces another one. Fig. 7.3 shows that the stereotype *«Replacement»* is used for classes *MsiPacketHeader* and *Gpsr2PacketHeader*. The stereotype *«Replacement»* also enables

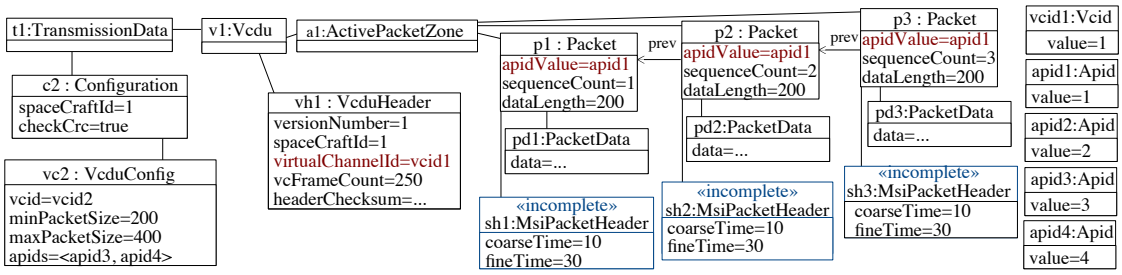


Figure 7.8. Incomplete Model Instance derived from the Original Model Instance in Fig. 7.7. Incomplete instances are blue, attribute values invalidated by modified constraints are red.

software engineers to specify, for each replacement class, the name of the class whose instances should be replaced. For example, to generate a Sentinel-2 input, class *MsiPacketHeader* replaces class *SarPacketHeader* in the field data of Sentinel-1 satellites.

Fig. 7.8 shows the Incomplete Model Instance derived from the Original Model Instance of Fig. 7.7. Since class *MsiPacketHeader* replaces class *SarPacketHeader*, each instance of class *SarPacketHeader* in the Incomplete Model Instance has been replaced by an instance of class *MsiPacketHeader*. Each instance of class *MsiPacketHeader* has been tagged as *incomplete*.

An instance of the updated data model also often differs from an instance of the original data model by the parameter values used in the configuration file. To deal with this case, the technique automatically updates the Incomplete Model Instance to include the content of the new configuration file. To this end, the technique automatically loads the content of the configuration file into memory and replaces the instances of the configuration classes in the Incomplete Model Instance with instances that capture the new given configuration. The instance *c2* of class Configuration appearing in the Incomplete Model Instance of Fig. 7.8 replaces the instance *c1* appearing in the Original Model Instance of Fig. 7.7.

Fig. 7.8 also shows that the updates related to the *Configuration* and the *Packet* classes lead to invalid attributes. According to the OCL constraints of Figs. 7.4 and 7.5, the attribute *apidValue* of class *Packet* is expected to be equal to *apid3*, while the field *virtualChannelId* of class *VcduHeader* is now expected to be equal to *vcid2*.

7.4 Generation of Valid Model Instances

To generate a Valid Model Instance, the technique updates the Incomplete Model Instance with values generated by means of constraint solving. The technique uses constraint solving both to generate data that is completely missing from the Incomplete Model Instance (i.e. classes or attributes tagged as *incomplete*) and to replace data that no longer satisfies the constraints of the Updated Data Model.

In principle, constraint solvers can be used to automatically generate in a single run a solution that matches the shape of the Incomplete Model Instance and satisfies all of the constraints. Unfortunately, constraint solvers often present scalability issues if the data model includes multiple collections of items with constraints among their elements, which is often the case when dealing with the data models of data processing systems. For this reason, we built an algorithm, *IterativelySolve*, that,

Require: IMI, the incomplete model instance
Require: DM, the data model with the OCL constraints
Require: $rootImiClass$, the name of the class that specifies the root node of the IMI
Require: $rootConfigClass$, the name of the root class that captures the content of the configuration file
Ensure: VMI, a valid model instance generated by means of constraint solving (i.e. the test input data)

```
1:  defined ← new List()
2:  toRegenerate ← null
3:  VMI ← null
4:  slices ← depthFirstVisit(IMI, rootImiClass, rootConfigClass)
5:  odg ← buildOCLDependencyGraph(IMI, DM)
6:  repeat
7:      used ← new List()
8:      for slice in slices do
9:          IMI, used, defined, toRegenerate ← SolveSlice(IMI, odg, slice, used, defined, toRegenerate)
10:         if toRegenerate ≠ null then
11:             break
12:         end if
13:     end for
14: until toRegenerate = null
15: if IMI ≠ null then
16:     VMI ← IMI
17: end if
```

Figure 7.9. The algorithm *IterativelySolve*.

instead of generating a complete valid model instance in a single run, iteratively generates valid instances of a portion (i.e. a slice) of the updated data model, and assigns the generated values to the attributes in the updated model instance. This iteratively leads to a valid instance of the updated data model.

A slice contains a subset of the class instances that belong to a data model instance. Slices are defined by traversing a graph that corresponds to the data model instance. Let G_{DM} be a graph that corresponds to an instance of a given data model I_{DM} if it contains a set of nodes N, such that for each class instance in I_{DM} there exists a unique corresponding node n in G_{DM}, and for each pair of class instances i_{c1} and i_{c2} connected by an association, there exists an edge connecting the corresponding nodes n_{c1} and n_{c2}. We assume that the data model has a single root node r. Slices are built by means of a depth-first visit of the graph G_{DM}. A slice is a sequence of nodes that belong to the path between the root r and a leaf node n_l. Leaf nodes are identified during the depth-first graph visit and correspond to class instances without any association edges that point to class instances not yet visited. Slices contain nodes sorted according to the order in which they are traversed in the depth-first visit. We define a parent-child relationship between two nodes in a slice, $n1$ and $n2$, such that $n1 = parent(n2)$ if $n1$ and $n2$ are connected by an association link, and $n1$ was visited before $n2$. By construction, a slice cannot contain two nodes with the same parent.

IterativelySolve relies upon a constraint solver to modify the assignments of the attributes in the slices such that the constraints of the data model are satisfied. We use the expression *slice solving* to indicate the process of executing a constraint solver to identify the values to assign to the attributes of a slice in order to satisfy the constraints of the data model.

The consistency of the solution is guaranteed by the incremental nature of the algorithm: items of collections are generated assuming that previously generated items are valid.

Figs. 7.9, 7.10, and 7.11 show, respectively, the algorithm *IterativelySolve*, function *SolveSlice*, and function *EnableFactsAndSolve*, which implement the logic for the incremental solving of slices.

Require: *IMI*, the incomplete model instance
Require: *odg*, the OCL Dependency Graph
Require: *slice*, a slice generated from the IMI
Require: *used*, a list of variables appearing in the facts used to solve previous slices
Require: *defined*, a list of variables whose values have been previously defined using the results generate by the solver
Require: *toRegenerate*, a list of attributes (if any) that need to be regenerated
Ensure: *IMI*, the incomplete model instance with values updated to satisfy the constraints for each slice
Ensure: *used*, an updated list of variables used in the facts
Ensure: *defined*, an updated list of the variables defined by the Alloy solver
Ensure: *toRegenerate*, a list of attributes that have been modified in the current execution and need to be regenerated by restarting the incremental solving from the first slice

```
 1: function SolveSlice(IMI, odg, slice, used, defined, toRegenerate)
 2:     prunedODG ← pruneODG(odg, slice)
 3:     a ← generateAugmentedSlice(prunedODG, slice)
 4:     alloyModel ← uml2Alloy(DM)
 5:     instanceM, facts ← augmentAlloyModel(alloyModel, a)
 6:     if slice.processed = true then
 7:         // this is a re-execution of SolveSlice
 8:         // this slice only needs to be solved if it contains one of the variables to regenerate
 9:         if toRegenerate.contains(DefinedVars(facts)) = false then
10:             return IMI, used, defined, toRegenerate
11:         end if
12:     end if
13:     solution ← ExecuteAlloy(instanceM)
14:     if solution ≠ null then
15:         used ← SetAllFactVariablesAsUsed(facts, used)
16:     end if
17:     if solution = null then
18:         solution, used, defined, modified ←
19:                 EnableFactsAndSolve(instanceM, facts, used, defined)
20:     end if
21:     if solution = null then
22:         return null, used, defined, null
23:     end if
24:     IMI ← update(IMI, solution)
25:     slice.processed = true // simply trace that slice has been solved at least once
26:     if modified.size > 0 then
27:         return IMI, used, defined, modified
28:     end if
29:     return IMI, used, defined, null
30: end function

31: function SetAllFactVariablesAsUsed(facts, used)
32:     for fact : facts do
33:         if isConfiguration(fact) = false or isShape(fact) = false then
34:             used ← used ∪ DefinedVar(fact)
35:         end if
36:     end for
37:     return used
38: end function
```

function DefinedVar returns the variable defined in a fact (i.e. the left hand side of the assignment in the fact).

Figure 7.10. Function *SolveSlice*.

```
Require: instanceM, the alloy model that captures the content of a single slice
Require: generatedFacts, a list of facts generated from the instanceM
Require: used, a list of variables appearing in the facts used to solve previous slices
Require: defined, a list of variables whose values had been previously defined using the result generate by the solver
Ensure: solution the result generated by Alloy, or null if the formula cannot be satisfied
Ensure: used, an updated list of variables used in the facts
Ensure: defined, an updated list of the variables defined by the Alloy solver
Ensure: modified, a list of variables used in previous iterations that had been redefined to solve the current slice
 1: function EnableFactsAndSolve(instanceM, generatedFacts, used, defined)
 2:     modified ← new List()
 3:     for fact : generatedFacts do
 4:         if isConfiguration(fact) = false or isShape(fact) = false or DefinedVar(fact) ⊆ defined then
 5:             instanceM ← disable(instanceM, fact)
 6:         end if
 7:     end for
 8:     solution ← ExecuteAlloy(instanceM)
 9:     if solution = null then
10:         return solution, used, defined, modified
11:     end if
12:     for fact : generatedFacts do
13:         if isDisabled(fact) then
14:             instanceM ← enable(instanceM, fact)
15:             tempSolution ← ExecuteAlloy(instanceM)
16:             if tempSolution = null then
17:                 instanceM ← disable(instanceM, fact)
18:                 if DefinedVar(fact) in used then
19:                     modified ← modified ∪ DefinedVar(fact)
20:                 end if
21:                 defined ← defined ∪ DefinedVar(fact)
22:             else
23:                 used ← used ∪ DefinedVar(fact)
24:                 solution ← tempSolution
25:             end if
26:         end if
27:     end for
28:     return solution, used, defined, modified
29: end function
```

Figure 7.11. Function *EnableFactsAndSolve*.

IterativelySolve performs 5 main activities:

- *Slices detection:* identifies a set of slices of the Incomplete Model Instance that can be solved separately and then recomposed to obtain a Valid Model Instance.
- *Solving with Alloy:* for each slice, derives an Alloy model that is given to the Alloy Analyzer to produce assignments for the attributes that (a) satisfy the constraints of the data model and (b) reflect the actual values observed in the Incomplete Model Instance (this is implemented by function *SolveSlice*).
- *Removal of invalid values:* iteratively removes from the Alloy model the assignments that prevent the solving of slices—that is, attribute values that invalidate OCL constraints (this is implemented by function *EnableFactsAndSolve*).
- *Consistency check:* in order to generate consistent solutions, the algorithm may solve a same slice multiple times—this occurs when the solution of a slice changes a value used by previously solved slices.
- *Update of Incomplete Model Instance:* reads the values generated by the Alloy Analyzer from the Alloy solution, and copies them into the Incomplete Model Instance (implemented by function *SolveSlice*).

The last four activities are repeated for all the slices. By iteratively updating the Incomplete

Model Instance, *IterativelySolve* attempts to obtain a Valid Model Instance. The following paragraphs provide a detailed explanation of the algorithm.

7.4.1 Slices detection

IterativelySolve first identifies a set of slices of the Incomplete Model Instance by performing a depth-first visit of the Incomplete Model Instance (Line 4, Fig. 7.9). The software engineer is expected to specify the root node of the Incomplete Model Instance (e.g. class *TransmissionData* in Fig. 7.8); this is accomplished at the data model level (e.g. in Fig. 7.1, by applying the *«InputData»* stereotype on the *TransmissionData* class). Engineers also specify the name of the class that captures the contents of the configuration file (e.g. class *Configuration* in Fig. 7.8); this is also accomplished at the data model level (e.g. in Fig. 7.1, by applying the *«ConfigData»* stereotype on the *Configuration* class). This is required to avoid the generation of slices containing configuration items because configuration values are specified by the software engineer and are not to be generated by means of constraint solving. For example, in the case of Fig. 7.8, it is the software engineer who specifies the value of the attribute *spaceCraftId* in the configuration file; the constraint solver is not expected to change the given identifier. To prevent the generation of slices containing configuration items, function *depthFirstVisit* does not traverse the configuration class during the depth-first visit.

After generating the slices, *IterativelySolve* builds the OCL Dependency Graph (*ODG*, see Line 5, Fig. 7.9). An *ODG* is a directed graph whose nodes correspond to the class instances in the Incomplete Model Instance, while its edges connect all the instances that are traversed when evaluating the OCL constraints. Fig. 7.12 shows an example *ODG* built from the Incomplete Model Instance of Fig. 7.8; the figure also shows the constraints used to produce the *ODG*. Observe, for example, that nodes $p1$, $a1$, $v1$, $t1$, $c2$, and $vc2$ in Fig. 7.12 are connected by edges because they are traversed to evaluate constraint *C3*.

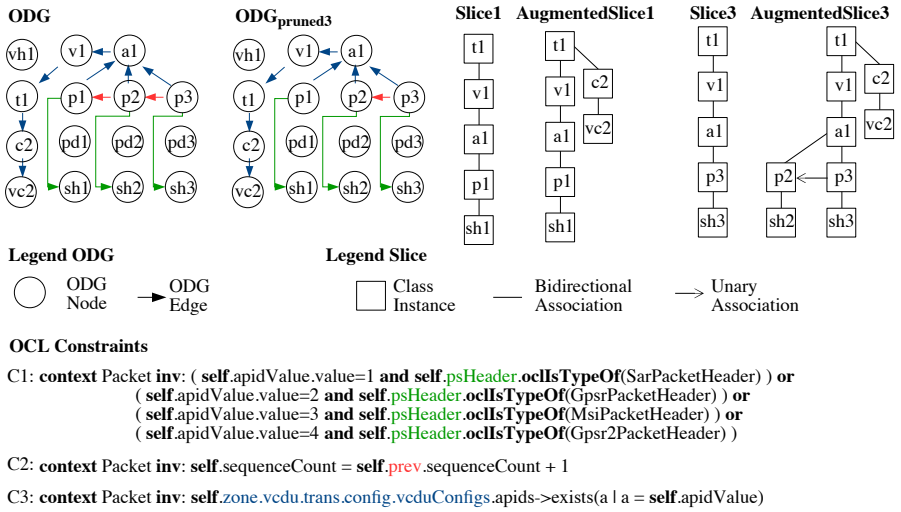
7.4.2 Solving with Alloy

Lines 6 to 14 of Fig. 7.9 implement the logic to solve the slices of the incomplete model instance; the function *SolveSlice* is invoked in Line 9 to incrementally generate a valid model instance.

SolveSlice calls the function *generateAugmentedSlice* (Line 3, Fig. 7.10) to generate an augmented slice for each slice s identified in the previous steps. The augmented slice includes all the class instances that are required to evaluate if the class instances in the slice s violate the constraints of the data model.

To build the augmented slice, the function *generateAugmentedSlice* first traverses the *ODG* to identify all the nodes that can be reached from each node in the slice s. The augmented slice contains all the class instances that correspond to the nodes traversed in the *ODG*. For example, the slice *AugmentedSlice1* in Fig. 7.12 includes the class instances $c2$ and $vc2$, which are reached when traversing the *ODG* starting from $p1$. Note that $c2$ and $vc2$ are required to evaluate the constraint *C3*.

Constraints on the items of collections may lead to a huge set of nodes that can be reached from a slice s. To generate smaller sets of reachable nodes, *SolveSlice* prunes the *ODG* (Line 2, Fig. 7.10) by removing all the edges that connect collection items with their predecessors with the exception of the items belonging to the current slice, which remain linked to their predecessor. $ODG_{pruned3}$ in



OCL Constraints

C1: **context** Packet **inv**: (**self**.apidValue.value=1 **and self**.psHeader.**oclIsTypeOf**(SarPacketHeader)) **or**
(**self**.apidValue.value=2 **and self**.psHeader.**oclIsTypeOf**(GpsrPacketHeader)) **or**
(**self**.apidValue.value=3 **and self**.psHeader.**oclIsTypeOf**(MsiPacketHeader)) **or**
(**self**.apidValue.value=4 **and self**.psHeader.**oclIsTypeOf**(Gpsr2PacketHeader))

C2: **context** Packet **inv**: **self**.sequenceCount = **self**.prev.sequenceCount + 1

C3: **context** Packet **inv**: **self**.zone.vcdu.trans.config.vcduConfigs.apids->exists(a | a = **self**.apidValue)

Figure 7.12. Example of the artefacts generated to perform slicing: *ODG*, *pruned ODG*, *slices*, and *augmented slices*. These artefacts are built from the Incomplete Model Instance of Fig. 7.8. The figure also shows the constraints used to produce the *ODG*; colours are used to show which attributes are related to the different edges in the *ODG*.

Fig. 7.12 shows the result of the pruning operation performed when processing Slice3. $ODG_{pruned3}$ contains only the edge that links $p3$ with its predecessor ($p2$), but not the edge that connects $p2$ with $p1$.

The augmented slice contains both data belonging to the original field data and incomplete data (i.e. incomplete class instances or attributes). *SolveSlice* executes the solver to generate valid class instances or attributes in place of the missing data.

SolveSlice uses UML2Alloy to generate an initial Alloy model that corresponds to the data model (Line 4, Fig. 7.10). UML2Alloy implements a model transformation that maps UML class diagrams and OCL constraints to the Alloy format. UML2Alloy generates an Alloy signature for each class and its contained attributes, facts capturing the associations between classes, and a predicate for each OCL constraint.

Given a model specified using the Alloy language, the Alloy Analyzer can generate a valid instance of the data model; however, to reuse existing field data, we need to generate a solution that also has the same 'shape' as the Incomplete Model Instance (i.e. the Incomplete Model Instance and the Alloy solution must be isomorphic). This way we can easily copy values from the Alloy solution to the corresponding Incomplete Model Instance. Function *augmentAlloyModel* (Line 5, Fig. 7.10) modifies the Alloy model in order to enforce the generation of a solution that fits the shape of the slice. Fig. 7.13 shows a portion of the Alloy model generated by function *augmentAlloyModel* from the Incomplete Model Instance of Fig. 7.8. The keyword *sig* indicates a signature; that is, a set of atomic definitions (atoms) that we use to model classes with Alloy. Signatures share similar properties with classes of UML class diagrams; in fact, they can be *abstract*, if they cannot be instantiated, and can be used to extend other signatures (see the keyword *extends*). The keyword *one* is used to indicate singletons (i.e. signatures for which only a single instance can exist). The keyword *fact* is used to indicate a property that must hold in the Alloy solution.

```
abstract sig Vcdu {
  trans:one TransmissionData,
  packetZone:one MpduPacketZone }
abstract sig TransmissionData {
  config:one Configuration,
  vcdu:some Vcdu }
abstract sig Configuration {
  trans:one TransmissionData
  checkCrc:one Bool,
  vcduConfigs:some VcduConfig }
abstract sig VcduConfig {
  config:one Configuration,
  apids:some Apid }
abstract sig Apid {
  value:one Int }
abstract sig Vcid {
  value:one Int }
```

```
abstract sig MpduPacketZone {
  vcdu:one Vcdu}
abstract sig ActivePacketZone
  extends MpduPacketZone {
  packets:some Packet }
abstract sig Packet {
  apidValue:one Apid,
  sequenceCount:one Int,
  psHeader:one PacketSecondaryHeader,
  prev:lone Packet }
abstract sig PacketSecondaryHeader {
  Packet:one Packet }
abstract sig MsiPacketHeader
  extends PacketSecondaryHeader {
  coarseTime:one Int,
  fineTime:one Int }
```

```
//declarations for class instances
one sig p2 extends Packet {}
one sig p3 extends Packet {}
one sig apid1 extends Apid {}
one sig apid3 extends Apid {}
one sig apid4 extends Apid {}
one sig c2 extends Configuration {}
one sig vc2 extends VcduConfig {}
one sig t1 extends TransmissionData {}
one sig v1 extends Vcdu {}
one sig a1 extends ActivePacketZone {}
one sig sh2 extends MsiPacketHeader {}
one sig sh3 extends MsiPacketHeader {}

//facts for data types
fact { apid1.value = 1 }
fact { apid3.value = 3 }
fact { apid4.value = 4 }

//facts for config variables
fact { vc2.apids = apid3 + apid4 }
```

```
 //facts for associations
fact { vc2.config = c2 }
fact { c2.vcduConfig = vc2 }
fact { t1.config = c2 }
fact { c2.transmission = t1 }
fact { t1.vcdu = v1 }
fact { v1.transmission = t1 }
fact { v1.packetZone = a1 }
fact { a1.vcdu = v1 }
fact { a1.packets = p2 + p3 }
fact { p2.zone = a1 }
fact { p3.zone = a1 }
fact { p2.psHeader = sh2 }
fact { s2.packet = p2 }
fact { p3.psHeader = sh3 }
fact { s3.packet = p3 }

//facts for non-config variables
fact { p2.apidValue = apid3 }
fact { p3.apidValue = apid1 }
```

Figure 7.13. Portion of the Alloy model generated to capture the part of the Incomplete Model Instance containing *AugmentedSlice3* (shown in Fig. 7.12).

To be isomorphic, the Alloy solution and the Incomplete Model Instance must share the same number and types of instances. To this end, the function *augmentAlloyModel* sets all the signatures in the Alloy model as abstract, and then creates a specialisation (i.e. a signature that extends another one), for each class instance in the augmented slice. Each specialised class is a singleton (this way we create the same exact instances observed in field data). The third column of Fig. 7.13 shows the declarations generated for the different class instances appearing in the Incomplete Model Instance; for example, the first two lines declare $p2$ and $p3$, the two instances of class *Packet* present in the portion of the Incomplete Model Instance of Fig. 7.8 that are present in *AugmentedSlice3* (Fig. 7.12). Signature *Packet* is abstract; consequently, the solver will not generate any instance of class Packet other than $p2$ and $p3$.

To preserve associations, function *augmentAlloyModel* generates a fact (i.e. a constraint) for each association between the class instances in the data model. For example, the fact '*a1.packets = p2 + p3*' in the top block of the fourth column of Fig. 7.13 indicates that *p2* and *p3* belong to the collection *packets*.

Finally, function *augmentAlloyModel* also generates Alloy facts that capture the actual values present in the Incomplete Model Instance (e.g. the facts in the bottom block of the fourth column of Fig. 7.13).[2] This is done to obtain a solution that reuses the data values observed in the original field data.

7.4.3 Removal of invalid values

Facts reflect the values observed in the field data (e.g. fact '*p3.apidValue = apid1*' in Fig. 7.13 that states that the *apidValue* of *Packet 3* is *apid1*). In the presence of updated (or new) OCL constraints that do not match the data used in the original test inputs, the solver cannot generate a solution (fact '*p3.apidValue = apid1*' breaks constraint C1). For this reason, when the solver determines that the set of given constraints is unsatisfiable, *IterativelySolve* relaxes the Alloy model by disabling the facts that prevent the identification of a solution (see function *EnableFactsAndSolve* in Fig. 7.11). The disabling of a fact is performed by adding a comment at the beginning of the line; this way facts can

[2]To minimise execution time, function *augmentAlloyModel* creates facts only for those attributes that appear in the OCL constraints.

be easily disabled and re-enabled. The disabling of facts is what enables *IterativelySolve* to replace existing values with new ones.

Function *EnableFactsAndSolve* proceeds by disabling all the facts (Lines 3 to 7, Fig. 7.11), and then iteratively enabling facts one by one to identify the ones that allow for the generation of a new solution (Lines 12 to 27, Fig. 7.11). Facts that prevent the generation of a solution are left out (Line 17, Fig. 7.11). *EnableFactsAndSolve* disables facts that capture actual values of the field data but not facts that preserve the shape of the solution. Additionally, the facts that capture configuration data (i.e. data that is not meant to be regenerated by the solver) are also not considered for removal.

For example, to create a solution from the Alloy model of Fig. 7.13, it is necessary to relax the model. In fact, this model cannot be used to generate an instance that satisfies the constraints C1, C2, and C3. In particular, constraint C1 cannot be satisfied because the *apidValue* of $p3$ is $apid1$ but its packet header is of type *MsiPacketHeader* (see the facts '*p3.apidValue = apid1*' and '*apid1.value = 1*'). *IterativelySolve* will solve this constraint only after disabling the fact *apid1 in p3.apidValue*, and will then generate a solution with *p3.apidValue* equal to *apid3*. If a solution is not found even after removing all the facts, *IterativelySolve* terminates without generating a test input (see Lines 21 to 23, Fig. 7.10). In this case, the technique simply continues the test generation process by sampling a new chunk of field data (Step 1 in Fig. 7.6).

7.4.4 Update of incomplete model instance

Once a solution is found, *SolveSlice* updates the data in the Incomplete Model Instance (see Line 24, Fig. 7.10). In particular, *SolveSlice* uses the values generated by constraint solving to update both the incomplete attributes of the Incomplete Model Instance, and any values that break the updated OCL constraints. By updating the Incomplete Model Instance, *SolveSlice* can incrementally generate a consistent solution: each augmented slice contains an item of a collection and its immediate predecessor (if any); this guarantees that the item is populated with data consistent with the previously generated item.

7.4.5 Consistency check

To enforce data consistency, when relaxing a model, function *EnableFactsAndSolve* checks if the disabled fact regards a variable that has been already observed when solving a previous slice. Lines 18 to 20 in Fig. 7.11 show that *EnableFactsAndSolve* adds to the list *modified* the names of the variables, used by previously solved slices, that have been modified during the current execution of *EnableFactsAndSolve*. If a solution for a slice is generated by changing a value used by previous slices, the algorithm restarts the solving from the beginning (see the loop in Lines 6 to 14 of Fig. 7.9). This is done to ensure that the slices already solved will still satisfy the constraints of the data model. To prevent infinite loops, *EnableFactsAndSolve* does not disable facts that define variables whose values have already been redefined by Alloy in previous iterations (see the clause '$DefinedVar(fact) \subseteq defined$' in Line 4 of Fig. 7.11).

Lines 6 to 12 of Fig. 7.10 are an optimisation. Since *SolveSlice* is executed multiple times, it should only call the Alloy Analyzer to solve slices that contain one of the variables that have been redefined or slices that have not yet been solved.

7.5 Analysis of Correctness and Completeness

This section shows, by means of an example, how the incremental solving of slices combined with the consistency check contributes to the generation of a correct and complete solution.

The algorithm proposed in this chapter, *IterativelySolve*, incrementally updates the Incomplete Model Instance by modifying the values assigned to the attributes of each slice. The Alloy Analyzer guarantees that all the values assigned to a slice satisfy the constraints of the data model.

In the unlikely case the data model instance contains only a single slice, the result given by the algorithm is correct by definition: the solver is executed once and it provides a set of assignments that satisfy all the constraints. The generated solution is trivially correct also whenever the data model instance contains only two slices without any shared variables. In this case, the final solution is the union of the two separate sets of assignments generated by the solver.

In the case of two slices with a shared variable, an incorrect solution may be generated if the values assigned when solving the second slice invalidate constraints that were true for the first slice. By means of an example, we show that, with our incremental, slice-based approach, no incorrect solution can be generated. To simplify the discussion, we model each slice by considering only the attributes belonging to the class instances in the slice, thus ignoring the associations between classes. Associations are not modified by *IterativelySolve*.

Let us take an example of a data model instance containing two slices. The two slices can be represented by the sets of variables *{b,a}* and *{c,a}*, representing class attributes, where a is shared by the two slices.

In our example we consider two simple inequalities as constraints, $C_1 : a >= b$ and $C_2 : a <= c$. Let us assume that the field data contains the values x_1, x_2, and x_3 assigned to variables a, b, and c, respectively. In our demonstration, we distinguish two cases that we identified by considering the possible valuations of the constraint $a >= b$: in *case 1*, $x_1 < x_2$, while in *case 2*, $x_1 >= x_2$.

Case 1: $x_1 < x_2$

Solving the first slice

To solve the slice *{b,a}*, our algorithm generates an Alloy model that corresponds to the formula $a >= b \; AND \; b = x_2 \; AND \; a = x_1$.

If $x_1 < x_2$, the formula cannot be satisfied. As a consequence, the algorithm will invoke function *EnableFactsAndSolve* (Line 19, Fig. 7.10). Function *EnableFactsAndSolve* starts by disabling facts that correspond to variable assignments (Lines 3 to 7, Fig. 7.11), and then solves the Alloy formula that contains the remaining facts (Line 8, Fig. 7.11). In this case, the formula contains just constraint C_1 (i.e. $a >= b$); the formula can be solved. Then the algorithm proceeds by enabling the facts one by one (Lines 12 to 27, Fig. 7.11).

After enabling the first fact, function *EnableFactsAndSolve* solves $a >= b \; AND \; b = x_2$, which results in a solution where $a = y_1$, $b = x_2$ and $y_1 >= x_2$. After enabling the second fact, the algorithm tries to solve $a >= b \; AND \; b = x_2 \; AND \; a = x_1$, which is not feasible. Function *EnableFactsAndSolve*

thus keeps the previous solution and updates the Incomplete Model Instance (Line 24, Fig. 7.10). The Incomplete Model Instance will thus contain the assignments $a = y_1$, $b = x_2$, $c = x_3$ (with $y_1 >= x_2$). The algorithm then starts solving the second slice.

Solving the second slice

The formula built by the algorithm to solve the second slice is $a <= c\ AND\ c = x_3\ AND\ a = y_1$. If $y_1 <= x_3$, the solution is immediately generated by the Alloy Analyzer in Line 13 (Fig. 7.10) and the algorithm returns with a correct model instance.

However, we are interested in understanding if the algorithm can generate an incorrect solution, which happens if the algorithm assigns to a a value lower than b. The values assigned to the model instance are updated by function *EnableFactsAndSolve*, which is executed if $y_1 > x_3$.

During the execution of function *EnableFactsAndSolve*, the fact $a = y_1$ cannot be disabled because variable a was assigned when generating data for the first slice. The formula to be solved in Line 8 (Fig. 7.11) is thus $a <= c\ AND\ a = y_1$, which leads to a solution with the assignments $a = y_1$, $c = y_2$ with $y_1 <= y_2$. This solution is returned and used to update the Incomplete Model Instance, which will then satisfy all the constraints (this is trivial since when solving *slice 2* the algorithm did not replace any value belonging to *slice 1*).

Case 2: $x_1 >= x_2$

Solving the first slice

If $x_1 >= x_2$, the field data already contains values that satisfy the formula $a >= b\ AND\ b = x_2\ AND\ a = x_1$, and the algorithm will proceed with the solving of the second slice without changing any variable values. Please note that in this case the algorithm variable *used*, which is a list data structure, is populated with all the variables appearing in the facts (see function *SetAllFactVariablesAsUsed*, Lines 31 to 38, in Fig. 7.10).

Generating data for the second slice

The formula built to solve the second slice is $a <= c\ AND\ c = x_3\ AND\ a = x_1$.

If $x_1 <= x_3$, the solution is immediately generated by the Alloy Analyzer and the algorithm returns with a correct model instance.

A more interesting case occurs when $x_1 > x_3$. In this case, function *EnableFactsAndSolve* is executed.

Function *EnableFactsAndSolve* disables all the facts and then re-enables them one by one. After enabling the first fact, *EnableFactsAndSolve* solves the formula $a <= c\ AND\ c = x_3$ (Line 15, Fig. 7.11), which leads to the assignments $a = y_3$, $c = x_3$ with $y_3 <= x_3$. After enabling the second fact, *EnableFactsAndSolve* tries to solve the formula $a <= c\ AND\ c = x_3\ AND\ a = x_1$, which cannot be satisfied because $x_1 > x_3$. The algorithm thus keeps the previously generated solution and adds variable a to the list $modified$ (Line 19, Fig. 7.11) to indicate that the value of variable a, which was

used to solve a previous slice (the variable belongs to the list *used*), had been modified. Variable a is also added to the list $defined$ (Line 21, Fig. 7.11).

Repetition of the main loop of *IterativelySolve*

After function *EnableFactsAndSolve* returns, *SolveSlice* updates the Incomplete Model Instance with the generated values (Line 24, Fig. 7.10) and then checks if the list $modified$ has a size greater than zero (Line 26, Fig. 7.10), which is true in this case. *SolveSlice* then returns the updated Incomplete Model Instance and the contents of the list $modified$.

Since $toRegenerate \neq null$ is true (in Line 10 of Fig. 7.9), *IterativelySolve* re-executes the loop in lines 6 to 14 in Fig. 7.9, which means that it again invokes the function *SolveSlice*. This time the Incomplete Model Instance contains the following assignments $a = y_3$, $b = x_2$, $c = x_3$ (with $y_3 <= x_3$).

Solving the first slice, second iteration

To generate data for the first slice, the solver must satisfy the formula $a >= b \; AND \; b = x_2 \; AND \; a = y_3$. If $y_3 >= x_2$, the formula trivially evaluates to true and the algorithm proceeds by solving the next slice.

If $y_3 < x_2$, *SolveSlice* invokes function *EnableFactsAndSolve*. Variable a has already been set in previous iterations, so *EnableFactsAndSolve* ends up by solving the formula $a >= b \; AND \; a = y_3$, which leads to the assignments $a = y_3$, $b = y_4$ with $y_3 >= y_4$.

Solving the second slice, second iteration

To solve the second slice, *EnableFactsAndSolve* builds the formula $a <= c \; AND \; c = x_3 \; AND \; a = y_3$, which trivially evaluates to true ($y_3 <= x_3$, according to the previous iteration of *IterativelySolve*).

The solution generated for the first slice might have lead to $y_3 >= x_2$, or to $y_3 < x_2$. If $y_3 >= x_2$, the assignments in the Incomplete Model Instance are thus $a = y_3$, $b = x_2$, $c = x_3$, which satisfy the constraints C_1 (with $y_3 >= x_2$) and C_2 (with $y_3 <= x_3$). If $y_3 < x_2$, the assignments in the Incomplete Model Instance are thus $a = y_3$, $b = y_4$, $c = x_3$, which also satisfy the constraints C_1 (with $y_3 >= y_4$) and C_2 (with $y_3 <= x_3$).

General case

The example presented in this section shows that the algorithm is able to guarantee that all the constraints are satisfied even in the presence of variables contained in slices that are redefined while solving subsequent slices.

The example has shown how *IterativelySolve* restarts the solving process from the first slice every time *EnableFactsAndSolve* redefines a variable included in a slice solved by a previous iteration. However, given that *EnableFactsAndSolve* is allowed to modify the value assigned to a variable only once, we can guarantee the termination of the algorithm, while the execution of the Alloy Analyzer gives guarantees about the correctness of the generated results.

EnableFactsAndSolve behaves the same way in the presence of one or more variables. Thus, the termination of the algorithm is guaranteed also in presence of multiple constraints working on multiple shared variables among slices.

Therefore, our iterative, slice-based approach to generating data can only ever result in a solution that is valid or it will generate no solution at all.

7.6 Empirical Evaluation

We performed an empirical evaluation to answer four research questions: the first two questions address the scalability and performance of our approach, while the remaining two questions address whether the data generated by the approach can, in fact, be used to effectively test new requirements. The research questions are:

- RQ1: Does the proposed approach scale to a practical extent?
- RQ2: How does the proposed approach compare to a non-slicing approach?
- RQ3: Does the proposed approach allow for the effective testing of new data requirements?
- RQ4: How does the use of the proposed approach compare to a manual approach?

The following subsections overview the subject of the study and the experimental setup, and describe, for each research question, the measurements performed and the achieved results.

7.6.1 Subject of the study and experimental setup

We implemented our approach as a Java prototype that: (a) relies upon the Eclipse UML2 Library for the processing of data models (i.e. class diagrams), (b) implements wrapping code to integrate UML2Alloy and the Alloy Analyzer (using the integrated SAT4J solver [Le Berre and Parrain, 2010]), and (c) implements *IterativelySolve* and all the supporting functionality.

As subject of our study, we considered SES-DAQ, the industrial data processing system introduced in Chapter 2 that processes satellite input data. Recall from Section 7.1 that the data model originally created to represent the Sentinel-1 satellite input data is updated to reflect the new data requirements of the Sentinel-2 satellite input data. SES-DAQ is a non-trivial system written in Java having 32,469 bytecode instructions (this is the successor version of the system used in the evaluations of Chapters 4 to 6). The data model initially developed for the evaluation of Chapter 5 was used. To capture the data structure, it consisted of 82 classes, 322 attributes, and 56 associations. To capture the constraints of the SES-DAQ data model we defined 52 OCL constraints (28 input constraints, 24 input/output constraints).

As an input for our approach, we considered a large transmission file containing Sentinel-1 mission field data provided by SES. The size of the transmission file is 1 million CADUs (or about 2 GB), containing 1 million VCDUs belonging to four different virtual channels.

Because of the large number of runs performed for this experiment and considering that some runs took up to 110 hours (each of which had to be repeated ten times), we used a large cluster of computers to run these experiments [Varrette et al., 2014]. To allow for a fair comparison between the

different techniques and the various file sizes considered, the experimental runs were each executed on computing nodes having the same characteristics. The experiments were run on a bullx B500 blade system [Atos, 2016] with each node having two processors (2 × Intel Xeon L5640 @ 2.26 GHz). Altogether, the experiments took over 143 days of run time to execute.

7.6.2 RQ1: Does the proposed approach scale to a practical extent?

7.6.2.1 Measurements and setup

RQ1 deals with the practical applicability of the proposed approach.

The generation of new data should be fast enough and scale effectively as file sizes increase. For this reason, to respond to *RQ1*, we applied the proposed approach to automatically generate test input files of various sizes. More specifically, we randomly sampled chunks of field data used by SES to test Sentinel-1 satellite requirements, and used those data chunks to generate inputs that cover the data requirements related to the processing of Sentinel-2 satellite data.

We automatically generated test inputs containing from 50 to 500 VCDUs, in steps of 50 VCDUs. We chose these values because of our experience with SES-DAQ. Our previous research results, in fact, show that test inputs with 50 VCDUs can be effectively used to perform conformance testing (as reported in Chapter 5). Test inputs with 500 VCDUs, instead, have been effectively adopted for robustness testing, to stress the behaviour of the software in the presence of inputs containing multiple invalid data values (as reported in Chapter 6). In general, software engineers aim to generate test files that are as realistic as possible in terms of size and content, as large sizes will stress the system more and are more likely to reveal faults. For example, in the case of SES-DAQ, larger input data files are more likely to be able to accommodate more diversity of patterns in the data and reveal faults related to the handling of large amounts of data.

For each given VCDU value, we generated ten test inputs using our approach. We measured the execution times for creating test inputs with the proposed approach; specifically, we analysed the relationship between execution time and input size.

7.6.2.2 Results

Fig. 7.14 shows a plot with the average execution time (in hours) required to generate a test input versus the number of VCDUs contained in each test input (see the curve named *Solving time*); box plots are also shown to demonstrate that the variance across runs is low in most cases. Fig. 7.14 also shows that the approach scales to a practical extent. In the case of test inputs containing 50 VCDUs, the approach requires on average 35.6 minutes to generate a single test input. In the case of test inputs containing 500 VCDUs, the approach requires on average 108.2 hours to generate a test input. A test input containing 500 VCDUs is particularly complex to generate because of the presence of multiple collections, each containing items with multiple references to other data items contained in the test input (e.g. in the case of SES-DAQ, test inputs with 500 VCDUs contain on average 24,861 class instances and 28,827 association instances). Such big inputs cannot be handcrafted by software engineers, which highlights the usefulness of our approach.

We consider the time required to generate big inputs to be acceptable in practice. The approach

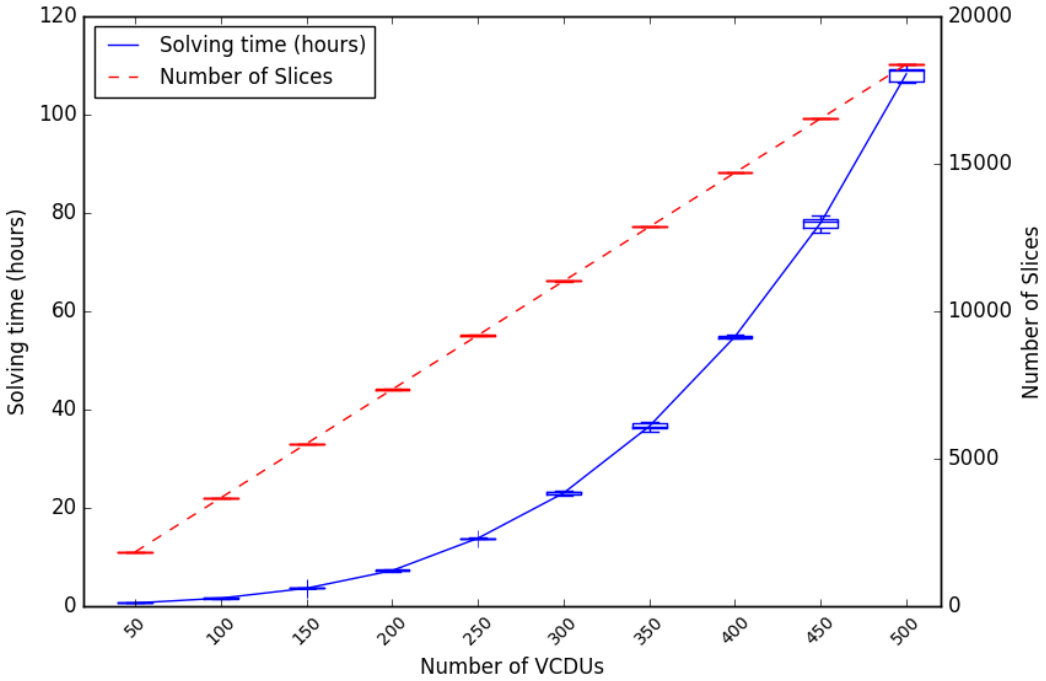


Figure 7.14. Average execution time required to generate test inputs and average number of slices versus number of Virtual Channel Data Units (VCDUs) in each generated test input. Boxplots are given for each data point. Minimum whisker value is Q1 - 1.5*IQR, maximum whisker value is Q3 + 1.5*IQR; where IQR is the interquartile range.

provides the benefit of automated test generation (i.e. no human effort is required to generate the test cases) and, furthermore, thanks to its model-based nature, does not negatively impact on the deadlines of the software testing process even if test generation may require days to complete. Given that the proposed approach requires only an updated data model and existing field data, the test input generation process can be started immediately after new data requirements are defined. Test generation can be executed while the requirements are implemented; for this reason, the generated test inputs are likely to be available before the system is ready to be tested.

The curve for solving time in Fig. 7.14 shows an exponential growth. This is mainly due to the nature of the input data processed by SES-DAQ. To better understand this behaviour we also report in Fig. 7.14 the average number of slices per test input size, and in Fig. 7.15, we show the average number of calls to function *SolveSlice* and the average number of times function *ExecuteAlloy* (i.e. the Alloy Analyzer) had been invoked during the generation of a test case. The plot in Fig. 7.14 shows that the number of slices grows linearly with the number of inputs. Recall that the iterative process restarts the slice solving loop if *EnableFactsAndSolve* alters the value of a variable used in a previous slice. Consequently, *SolveSlice* can be invoked multiple times—as was the case for our experiments—against the same slices when generating a test input. A direct consequence of this is that the average number of calls to function *SolveSlice* grows exponentially with the size of the inputs (Fig. 7.15). This trend depends on the presence of several data items shared by multiple slices, and constitutes an indirect indicator of the complexity of the input data. Although the average number of calls to function *SolveSlice* grows exponentially, the average numbers of calls to the Alloy Analyzer does not, which means that function *SolveSlice* often does not invoke the Alloy Analyzer; this is an effect of the optimisation implemented in Lines 6 to 12 of *SolveSlice* (Fig. 7.10). Therefore, the most

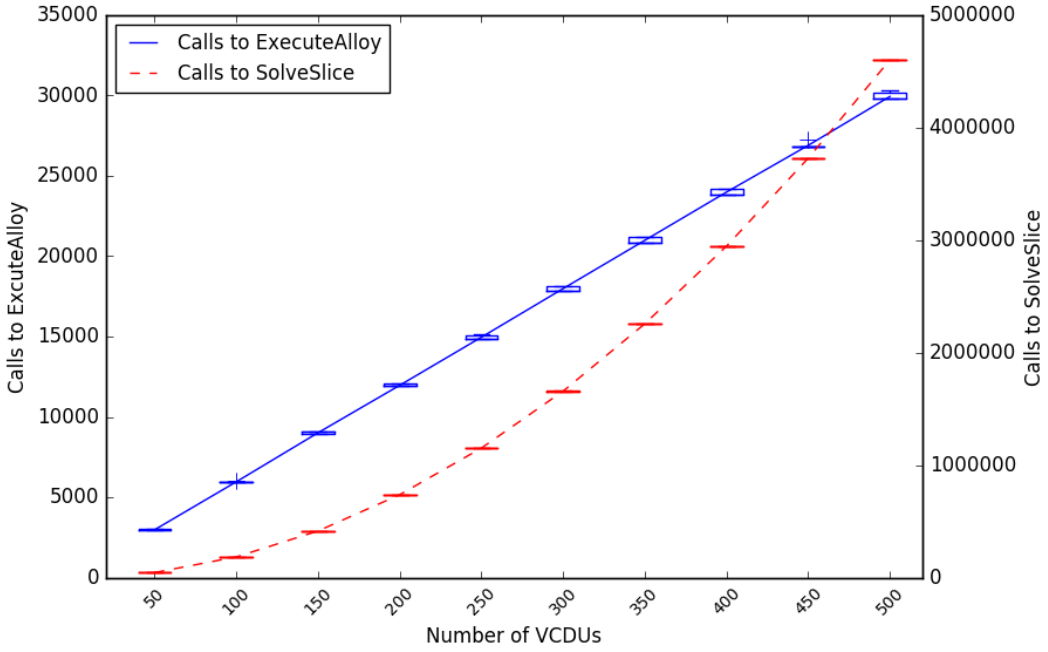


Figure 7.15. Average number of calls to functions SolveSlice and ExecuteAlloy versus the number of Virtual Channel Data Units (VCDUs) in each generated test input. Boxplots are given for each data point. Minimum whisker value is Q1 - 1.5*IQR, maximum whisker value is Q3 + 1.5*IQR; where IQR is the interquartile range.

plausible explanation for the exponential growth in solving time (Fig. 7.14) is the exponential growth in calls to SolveSlice (Fig. 7.15), which consumes computation time, even though it does not always invoke the Alloy Analyzer.

7.6.3 RQ2: How does the proposed approach compare to a non-slicing approach?

7.6.3.1 Measurements and setup

To be justified, the proposed approach should provide an advantage over a more straightforward approach that does not use slicing. To respond to RQ2, we thus compared the performance of the approach proposed in this chapter with an approach that generates test inputs from scratch without relying upon a slicing algorithm.

We built a solution that uses a modified version of *IterativelySolve* that does not apply slicing. We refer to this approach as *NonSlicingSolving*. *NonSlicingSolving* processes the entire Incomplete Instance Model to derive an Alloy model that captures the shape of the input data but not the actual values of attributes. All of the attribute values are thus generated from scratch through a single execution of the Alloy Analyzer. To compare the scalability of the two approaches, we apply them to generate test inputs containing different numbers of VCDUs and we measure the execution time required to generate new test inputs.

NonSlicingSolving does not scale; in fact, it cannot generate test inputs containing 50 VCDUs because of out of memory errors. In 10 separate executions performed to generate test inputs with 50 VCDUs, the Alloy Analyzer always crashed because of out of memory errors, even when 16 GB

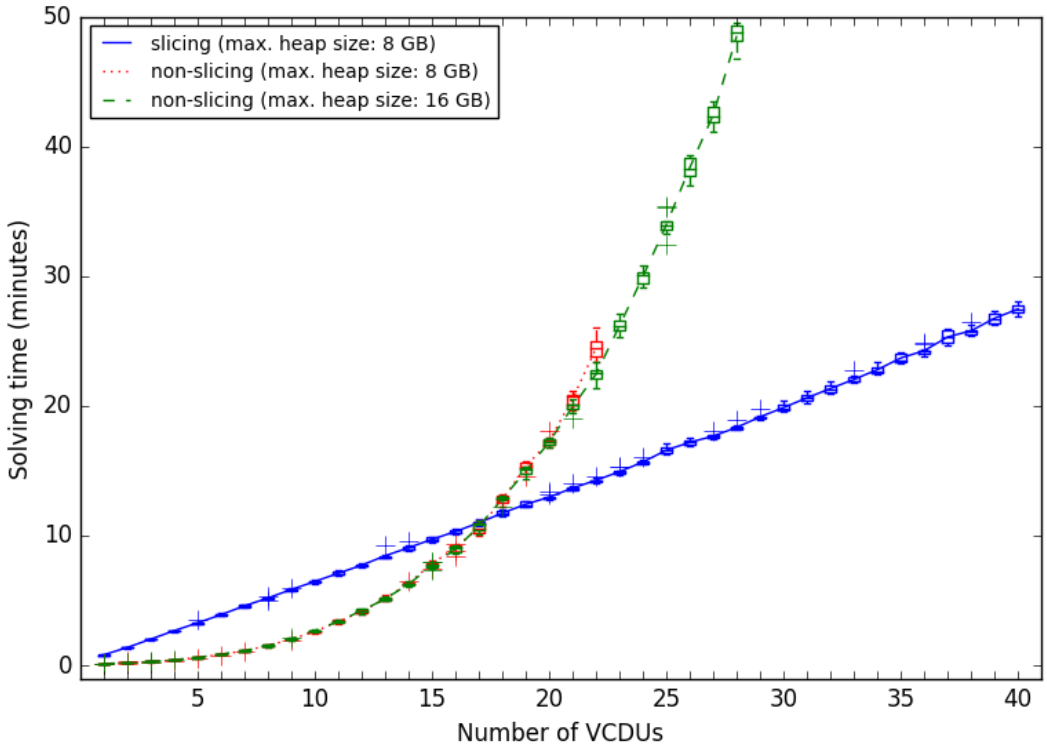


Figure 7.16. Comparison of the performance of *IterativelySolve* (that uses slicing) with a non-slicing approach. Average execution time required to generate test inputs versus number of Virtual Channel Data Units (VCDUs) in each generated test input. Boxplots are given for each data point. Minimum whisker value is Q1 - 1.5*IQR, maximum whisker value is Q3 + 1.5*IQR; where IQR is the interquartile range.

of RAM had been dedicated to the Alloy Analyzer process. *NonSlicingSolving* is thus useless for performing robustness testing; it is unable to generate sufficiently large augmented field data files.

To better compare the two approaches and study the effect of input size on execution time, we ran several experiments to generate test inputs containing 1 to 40 VCDUs. We used both *IterativelySolve* and *NonSlicingSolving* to generate 10 different test inputs for each possible input size containing from 1 to 40 VCDUs. To perform the experiment, we randomly sampled chunks of Sentinel-1 field data. Each sample contained the required number of VCDUs (i.e. 1 to 40 VCDUs), and we then applied the two approaches to generate a test input to validate Sentinel-2 requirements. As a metric of performance, we measured the solving time to generate a valid model instance for the two approaches. For the *NonSlicingSolving* approach, we studied the performance using maximum heap sizes of both 8 and 16 GB. For the approach proposed in this chapter we used a maximum heap size of 8 GB[3].

7.6.3.2 Results

Fig. 7.16 shows the obtained results for both *NonSlicingSolving* and *IterativelySolve*; the *x*-axis reports the input size measured in number of VCDUs and the *y*-axis reports the execution time taken in minutes. Fig. 7.16 shows that the *NonSlicingSolving* is more efficient for very small test inputs, but it becomes increasingly inefficient with a growing number of VCDUs. *IterativelySolve* always

[3]Recall that the approach proposed in this chapter always terminated, while *NonSlicingSolving* crashed because of out of memory errors.

performs better than *NonSlicingSolving* with test inputs containing more than 17 VCDUs. *NonSlicingSolving* shows an exponential growth, this result is in line with research indicating that the Alloy Analyzer shows an execution time that grows exponentially in the presence of relations that involve thousands of elements [Leuschel et al., 2011]. When *NonSlicingSolving* is executed using an 8 GB maximum heap size, out of memory failures begin to occur at 22 VCDUs, and no solution is possible with 23 VCDUs or more. When executing *NonSlicingSolving* using a 16 GB maximum heap size, out of memory failures begin to occur at 29 VCDUs; no solution is possible with 29 VCDUs or more.

7.6.4 RQ3: Does the proposed approach allow for the effective testing of new data requirements?

7.6.4.1 Measurements and setup

RQ3 aims to evaluate the effectiveness of the approach—that is, the ability of the approach to generate test inputs that are effective to test the new requirements of the software system.

One of the key features of SES-DAQ, our case study system, is the ability to automatically identify and discard invalid inputs (e.g. the system should be able to automatically identify and discard data units containing out-of-order packets); more importantly, the system is expected to be robust enough in the presence of invalid inputs to continue functioning without failing. For this reason, robustness testing (i.e. testing the capability of the system to deal with invalid data) plays a fundamental role in evaluating whether the software meets its requirements.

To respond to RQ3, we thus applied the approach proposed in this chapter to automatically generate new data intended for use in robustness test cases targeting new data requirements. Since robustness testing deals with the generation of invalid test data, we must adapt the proposed approach to automatically generate invalid test inputs for Sentinel-2 requirements. To this end, we integrated the proposed technique with the approach presented in Chapter 5 that generates robustness test cases by automatically mutating chunks of valid field data. We made use of our oracle approach (see Section 3.3) to determine if the output generated by the system after processing an automatically generated test input is wrong.

Fig. 7.17 shows how the technique presented in this chapter is integrated with our previously developed data mutation and oracle approaches. Fig. 7.17 shows how the technique presented in this chapter is integrated with the data mutation approach (presented in Chapter 5) and the oracle approach (presented in Chapter 4).

In practice, since field data for the new requirements are not available, we relied upon the approach proposed in this chapter to generate valid chunks of augmented field data that meet the new data requirements.

Fig. 7.17 shows that the test inputs generated by the technique presented in this chapter are given as input to the *All Possible Targets* data mutation technique presented in Section 5.5.2 that randomly selects a mutation operator and a possible target for the mutation (i.e. an attribute or a class instance) among the ones not already considered in previous iterations. The process is repeated until all the mutation operators have been applied on one instance of every attribute (or class) on which they can be applied. The generated test inputs are then executed against the software, and the OCL input/output

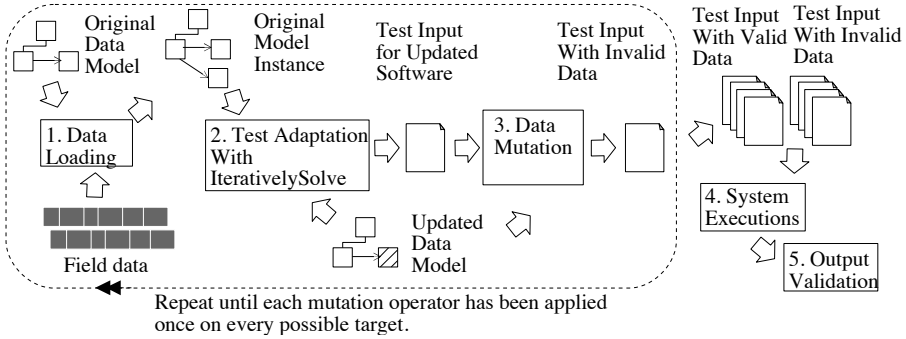


Figure 7.17. Testing process followed to respond to research questions RQ3 and RQ4.

Table 7.1. Coverage of instructions/branches implementing Sentinel-2 specific data requirements.

	S2 Instructions Covered			S2 Branches Covered			
Test suite	Avg	Min	Max	Avg	Min	Max	#Tests
Auto S1+S2	74 (77.9%)	74 (77.9%)	74 (77.9%)	9 (81.8%)	9 (81.8%)	9 (81.8%)	103.1†
Manual S1+S2	68 (71.6%)	–	–	8 (72.7%)	–	–	32

†Average value. One of the 10 test suites generated contains an additional test case for an attribute that only occurs very rarely in the field data.
Note: Auto, automatically generated test cases according to our methodology; Manual, test cases written manually by SES; S1, Sentinel-1; S2, Sentinel-2.

constraints included in the data model are used to verify if the resulting outputs are correct.

To answer *RQ3*, we used test inputs generated with the process in Fig. 7.17. Since the latest version of SES-DAQ implements both Sentinel-1 and Sentinel-2 data requirements, we created test suites containing both test cases for the original Sentinel-1 data requirements (i.e. input data generated using data mutation only) and test cases for the new Sentinel-2 data requirements (i.e. input data generated with the process in Fig. 7.17). We generated ten test suites to assess the impact of the randomness of the process. For example, different samples of field data are used; also, the same mutation operator might target different locations within the sampled transmission data. Each generated test input is 50 VCDUs in size (i.e. the same size adopted in Chapter 5).

The ten test suites were then executed against the SES-DAQ system. We relied on code coverage as a surrogate measure for fault detection effectiveness [Ammann and Offutt, 2008]. We used a code coverage tool, EclEmma [Mountainminds, 2006], to measure and evaluate code coverage in terms of bytecode instructions and branches. An analysis by [Li et al., 2013] has determined that the implementation of branch coverage in EclEmma, which measures the branches covered at the bytecode level, provides the equivalent of clause coverage (i.e. it checks that each clause of a predicate evaluates to both true and false, and, for switch statements, each branch is considered to end with a break statement). We measured the number of instructions and branches implementing Sentinel-2 data requirements that were covered by the test inputs generated with the proposed approach. Branch coverage captures the effectiveness of the test suite to spot failures that depend on specific values for certain clauses.

7.6.4.2 Results

The top section of Table 7.1 shows the results of the different test suites considered for our evaluation. Column *Test suite* lists the name of the test suites, *Auto S1+S2* is the test suite automatically generated to cover both Sentinel-1 and Sentinel-2 data requirements. Column *S2 Instructions Covered* reports the average, minimum and maximum number of bytecode instructions implementing Sentinel-2 data requirements that have been covered by the generated test suites. Column *S2 Branches Covered* reports the average, minimum and maximum number of Sentinel-2 specific branches covered by the generated test suites.

The new data requirements related to the processing of Sentinel-2 data have been implemented in 11 branches, for a total of 95 bytecode instructions. The results show that, for each automatically generated test suite, the proposed approach covers 9 (81.8%) of the branches related to the new data requirements (see column *S2 Branches Covered*). The proposed approach does not cover two branches because these branches have been written to handle a particular error that cannot be introduced through the data mutation operators of the approach presented in Chapter 5. However, the proposed approach proved to be able to cover most, 77.9%, of the instructions implementing the new data requirements in a fully automated manner. Uncovered instructions correspond to the two uncovered branches mentioned above.

7.6.5 RQ4: How does the use of the proposed approach compare to a manual approach?

7.6.5.1 Measurements and setup

In general, the costs of software testing depend on the time required to design test cases and prepare the test suites, and the time required to execute the test cases. In our approach, the former corresponds to the cost of modelling while, for manual testing, this is the cost of defining and writing test cases. As for execution time, in many situations it is sufficiently small to have no practical impact on the test process and is then of negligible importance.

Another important aspect is that, in our context, manual testing was performed by highly experienced engineers, with domain expertise. In a context where change is frequent, versions are many, and test engineer turnover is high, this expertise may not always be available. In such situations, automation brings a significant advantage as test cases can be automatically regenerated. However, changes must be made to the model and the assumption is that such changes are less expensive than reviewing and possibly changing every test case in a test suite.

In the case of SES, for example, software engineers have to handcraft test inputs (i.e. binary files) containing proper values so that they resemble a valid satellite transmission. We report here some data that shows that manually written test inputs are expensive to produce. Although each manually written test input contains only 6 VCDUs, it has a complex structure that is labour intensive to produce manually. On average, the manually written test inputs for SES-DAQ contain the equivalent of 130 class instances of the data model, and 261 attribute values. The complexity of the inputs does not depend only on the structure of the file, but also on the constraints that a test input must satisfy to be a valid input for SES-DAQ. On average, each test input contain 36 attribute values that must satisfy at least one constraint, for a total of 1152 attribute values that are constrained within the whole test suite.

This data clearly shows that manually writing and maintaining test inputs is not straightforward.

On the contrary, data modelling is a software engineering practice that can play a key role when designing the software, which means that the model required by the proposed approach may coincide with the models already produced by software engineers without any additional cost. Furthermore, in the case of engineering companies like SES, data modelling has additional benefits. SES engineers, for example, find data modelling particularly useful because it allows for the structure of data and data constraints to be characterised with a high level notation that facilitates discussions with the management of the company (usually engineers who can read class diagrams and constraints written in OCL).

Furthermore, when comparing two testing approaches, it is particularly important to take into account the fault detection effectiveness of the test cases. As for RQ3, we use instruction and branch coverage as a surrogate measure for fault detection effectiveness.

To answer RQ4, for each of the automatically generated test suites (the same as those used for the evaluation of RQ3) and the test suite written manually by SES software engineers with a high degree of domain expertise: we measured test suite size and test execution time; furthermore, to measure test suite effectiveness, we measured the branch and instructions coverage. We specifically considered the coverage of branches and instructions implementing the functionality that deals with new data requirements. The manually written test suite for SES-DAQ tests both Sentinel-1 and Sentinel-2 data requirements; there are 32 test cases in the test suite, three of which were specifically written to test Sentinel-2 data requirements.

We compared the size of the manually written SES test suite with the number of test cases in the automatically generated test suites. We also compared the average time required to execute the test suites. These measurements are useful because if execution time does not introduce important delays in the testing process, which is common for this type of system, it would suggest that the test suite size is not practically relevant in our context. Finally, we compared the coverage of the manual test suite for SES-DAQ with the ten test suites automatically generated to respond to RQ3.

7.6.5.2 Results

The bottom section of Table 7.1 shows the coverage results for the test suite written manually by SES software engineers (see line *Manual S1+S2*). Column *# Tests* shows the number of test cases of the two test suites.

When using the approach proposed in this chapter, the number of test cases (103.1, on average) is larger than in manual testing (32), and is determined by the testing strategy. In our context, the complete automatically generated test suite can be run in under 31 minutes, and, therefore, executing the test suite can easily be accommodated on a daily (or even more frequent) basis. Test execution is therefore a negligible cost factor and we will focus on test design. This is likely to be the case for many data processing systems because they are built to process megabytes of data in few seconds.

Executing this test suite can easily be accommodated on a daily (or even more frequent) basis. Therefore, the additional execution time required by the automatically generated test suite has little practical impact.

Table 7.2. Coverage of instructions/branches of SES-DAQ.

Test suite	Bytecode Coverage Avg / Min / Max	Branch Coverage Avg / Min / Max
Auto S1+S2	23,432.1 (72.2%) / 23,273 (71.7%) / 23,529 (72.5%)	978.7 (51.3%) / 957 (50.2%) / 987 (51.8%)
Manual S1+S2	23,046 (71.0%)	950 (49.8%)

Note: In total, SES-DAQ has 32,469 bytecode instructions and 1,907 branches. Auto, automatically generated test cases according to our methodology; Manual, test cases written manually by SES; S1, Sentinel-1; S2, Sentinel-2.

What matters most for our evaluation is thus whether the automated approach proposed in this chapter covers as many or more Sentinel-2 specific instructions and branches than the manual test cases, written by experts. The results show that the *Manual S1+S2* test suite covers 8 (72.7%) of the branches related to the new data requirements and 68 (71.6%) instructions, while the automated approach covers 9 (81.8%) branches and 74 (77.9%) instructions. Automated testing therefore performs slightly better than manual testing in terms of coverage. Although small, the difference in the number of instructions/branches covered is of practical importance for increasing confidence in the reliability of the system—uncovered branches may trigger critical faults (e.g. runtime exceptions), while the software is running in the field.

As an additional note, we report in Table 7.2 the overall number of instructions (see column *Bytecode Coverage*) and branches (see column *Branch Coverage*) covered by the automated and manual test suites. Results show that the proposed approach covers more bytecode instructions overall; comparing *Auto S1+S2* with *Manual S1+S2* we observe that, on average, 386.1 additional bytecode instructions are executed in the case of the test suites generated with the proposed approach. The test suites generated according to our technique take approximately 21 minutes more to execute compared with manual testing; in practice, 21 minutes are negligible.

To conclude, our model-based, automated approach fares slightly better than manual testing, as performed by experts, in terms of instruction and branch coverage corresponding to new requirements. It also achieves better coverage overall, when considering all requirements. In terms of cost, though this is very context dependent, in a situation like the one at SES, where data processing systems incur frequent changes and new versions must be produced, relying on the availability of experts is not always possible or easy. Our model-based approach is therefore a valuable alternative.

7.6.6 Threats to validity

Internal threats

To limit the threats to the internal validity of the empirical evaluation—that is, a faulty implementation of our toolset that may lead to erroneous results—we carefully inspected a subset of the generated test inputs to look for the presence of errors: data values of the original field data not preserved in the updated model instance, a shape of the updated model instance that did not coincide with the shape of the original model instance, or values that did not satisfy data constraints. To further validate all the generated test inputs, we performed two additional validation activities: (1) we relied upon the Eclipse UML2 library [Eclipse Foundation, 2016] to automatically verify that all the generated test inputs satisfied the OCL constraints of the SES-DAQ data model, and (2) we checked that SES-DAQ error handling code was not covered when SES-DAQ was executed to process the valid test inputs generated with the proposed approach.

External threats

Threats to the external validity regard the generalisability of results. The algorithm *IterativelySolve* may show different performance when executed to generate test inputs for other case studies. Factors that affect the performance of *IterativelySolve* are the size of the test inputs to generate and the characteristics of the data model (i.e. number of classes, attributes, associations, and OCL constraints). We have run experiments considering an industrial and complex case study system as benchmark for our evaluation. We have shown that the data model is complex: it contains 82 classes, 322 attributes, 56 associations, and 52 OCL constraints. Working with a nontrivial system that is already in use, gives some confidence that the scalability results can generalise to many of the industrial data processing systems on the market. Furthermore, we studied the effect of input size on the algorithm performance, dealing with test inputs that correspond to model instances containing up to 24,861 class instances and 28,827 association instances, which gave us confidence about the general scalability of the algorithm with respect to test input size.

Results on the effectiveness of the proposed approach may not generalise as well. We have shown that the technique allows for the generation of test inputs such that most of the code implementing new data requirements is executed in the presence of a nontrivial data model. In systems where repeated handcrafting of test inputs is significantly less expensive than modelling, the benefits provided by our technique might be less evident, even if the generated test cases are more effective in covering new data requirements. However, in the general case of complex data processing systems, we believe that the assumption of modelling costs being less expensive than manual testing holds.

7.7 Conclusion

In this chapter, we presented an approach to automatically generate test inputs for testing new data requirements of data processing systems. More specifically, we deal with changes that regard the structure of the input data accepted by a data processing system, or the constraints that regulate the content of the different data fields.

When test inputs coincide with complex data structures containing thousands of data items related with each other by multiple constraints, which is often the case when dealing with data processing systems, traditional approaches based on constraint solving cannot be applied because of scalability issues.

The proposed technique makes use of existing field data, a data model describing the original structure and content of the input data, and an updated data model reflecting the modifications to the structure and the content of the input data. The data model is a class diagram capturing the structure of the inputs, with constraints among classes and attributes. *The proposed approach overcomes the limitations of traditional approaches thanks to the integration of model slicing with constraint solving.*

The proposed approach generates test inputs by augmenting and adapting existing field data that matches the original data requirements. The reuse of existing field data reduces the amount of data values that need to be generated by a constraint solver. In particular, the approach uses constraint solving to generate only the data items introduced by the new data requirements, or to regenerate data items that break new or modified data constraints. As the underlying constraint solver, we rely upon

the Alloy Analyzer. The proposed technique also integrates a new slicing algorithm that allows for the incremental invoking of the constraint solver to generate portions of the new test input. The algorithm guarantees the consistency of the generated test input, which results from the composition of the data belonging to the different slices.

We validated the scalability and effectiveness of the proposed approach using an industrial case study, a satellite data acquisition system working with the European Space Agency Sentinel series of satellites [ESA, 2016]. In our study, we considered a version of the system that had been modified to accept new packet types associated with new missions. The empirical study shows that the proposed approach scales in the presence of complex data structures. In particular, the study shows that the input generation algorithm based on model slicing presented in this chapter can produce, in a reasonable amount of time, test input data that is over ten times larger in size than the data that can be generated with constraint solving only. To evaluate the effectiveness of the proposed approach, we generated test inputs to stress software robustness and we measured the code covered when executing the generated test inputs against the software. Robustness testing is critical for testing the case study system considered. The results show that the generated test inputs cover most of the source code instructions of the updated software written to implement new data requirements. The generated test inputs also achieve more code coverage than the test cases implemented by experienced software engineers, thus highlighting the benefits of the proposed approach.

Chapter 8

Related Work

This chapter provides an overview of existing work related to the approaches researched and developed for this dissertation.

We initially overview existing approaches and their suitability to solving the challenges of our data processing system context in the following complementary areas: test models (Section 8.1) and test oracle automation (Section 8.2). We also review existing work that addresses modelling DAQ systems (Section 8.3). There is a large body of work dedicated to the automation of software test case generation; a recent survey by Anand et al. provides an overview [Anand et al., 2013]. We will focus on the related work that is most relevant to the data generation approaches presented in this dissertation. Related work dealing with the automatic generation of faulty input data contained within complex data structures focuses mostly on model-based (Section 8.4) and mutation-based testing (Section 8.5) approaches. A few other approaches are also discussed (Section 8.6). Search-based software testing approaches (discussed in Section 8.7) dealing with the automatic generation of faulty input data for system level testing are still in their infancy. Section 8.8 provides an overview of approaches as they relate to the testing of new data requirements. Finally, Section 8.9 examines other related research in the area of model-based slicing.

8.1 Test models

MBT has been studied extensively and applied to several fields using different techniques and approaches [Dias Neto et al., 2007, Mussa et al., 2009]. The number of publications that discuss MBT is very large making it hard to review all of them adequately in this dissertation. Therefore, we focus on the taxonomy defined by Utting et al. [Utting et al., 2012].

Utting et al. [Utting et al., 2012] defined a six-dimension taxonomy of MBT in terms of the test model, test generation technique and test execution method. The dimensions related to the test model are: the *scope* of the model (e.g. input-only, input/output), the *characteristics* of the model (e.g. timed, deterministic) and the *modelling paradigm* (e.g. pre/post, functional, data-flow). The dimensions related to the test generation technique are the *test selection criteria* (e.g. data or requirements coverage) and the used *technology* (e.g. model checking or constraint solving). Finally, the test execution can be either *online*, where tests are executed as they are generated and the output is monitored or *offline*, where tests are generated first and then executed on the system.

The main dimension in the taxonomy is the scope of modelling; whether the model specifies only the *inputs*, which are considered the environment of the SUT or also specifies the expected *input/output* behaviour. Utting et al. state that modelling the environment is not sufficient to produce strong oracles. *Input/output* models can provide better automated test oracles by capturing the environment as well as part of the system's intended behaviour. These models are thus able to predict the expected outputs for each input and check it against the output produced by the SUT. Test oracle automation, which is essential in any MBT approach, will be addressed in detail in the context of DAQ systems in Section 8.2.

According to Utting et al., current MBT techniques focus on *modelling the behaviour of the system* using various modelling paradigms: state-based (or pre/post) [Jaffuel and Legeard, 2006], transition-based [Tretmans, 2008], history-based [Veanes et al., 2008], functional [Gaudel and Le Gall, 2008], operational [Moonen et al., 1997], stochastic [Walton and Poore, 2000] or data-flow notations [Marre and Blanc, 2005]. None of the existing modelling techniques was adequate for testing DAQs, and data processing systems in general, because they did not provide support for the case where the system complexity lies in the input structure and constraints—and the input data has complex mappings to highly structured output data. To address the challenges in our context, we needed a model that not only specified the inputs but also specifies the output and expected *input/output* behaviour. However, this expected behaviour in DAQ systems is expressed in the form of mappings between input and output elements, while the system itself is a black-box. Therefore, we cannot apply any of the current modelling paradigms, such as state machines or data-flow models. Due to these specific characteristics of DAQ systems and the complexity of the structure of the input and output, we adapted UML/OCL models to formalise the DAQ system input, configuration and output data and the mappings between them.

Dalal et al. [Dalal et al., 1999] developed a data model to generate tests for unit testing. Their data models specify valid and invalid values for input fields and capture the relationship between these fields through a set of constraints. The goal of creating these models is automated combinatorial testing; in their approach, test oracles are created manually for each generated test case. In contrast, we created a modelling methodology (using class diagrams and constraints) that automates test validation, oracles, and test input generation for systems with complex inputs and constraints.

Our modelling approach can be partially classified according to Utting et al.'s taxonomy as follows: The test models can be categorised as *input/output* and *deterministic*.

8.2 Test Oracle Automation

Xie et Memon [Xie and Memon, 2007] define a test oracle as a combination of *oracle information* (the expected output) and the *oracle procedure* (a process that compares the *oracle information* to the actual system output). In a traditionally manually written test, typically the *oracle procedure* expects a given concrete output (e.g. for a given input x, a given output y is expected). Using more advanced techniques, a given expected output might reflect a mapping between system attributes. In our case, we make use of calls to OCL invariants as our *oracle procedure* that makes use of input class attribute assignments mapped (using implications) to output class attribute assignments (i.e. the *oracle information* in our case) to validate the system output.

Automating the test oracle is important for reducing the testing cost. Shahamiri et al. [Shahamiri et al., 2009] in their comparative study identify two main challenges in automating the oracle: defining the expected output and linking each expected output to the relevant inputs. They also give an overview of six existing techniques in research for test oracle automation. Some of these techniques rely on the output from previous versions of the system to automate the test oracle for a new version [Last et al., 2004, Manolache and Kourie, 2001, Vanmali et al., 2002]. Three techniques automate the oracle without relying on previous versions: Input/output analysis, state-based test oracles, and decision tables.

Schroeder et al. [Schroeder et al., 2002] automate an oracle by observing and analysing a small set of inputs and their outputs and then generalising the input/output relationships to the whole test suite. Although this approach can be useful in automating the oracle and reducing the oracle cost, if the system is faulty, which is expected since the goal is to test the system, the derived observations about the output will also be incorrect.

Memon et al. [Memon et al., 2000, Xie and Memon, 2007] used state invariants to automate a test oracle for Graphical User Interfaces (GUIs). They formalised the GUI by a set of states, where each state has a possible set of events with preconditions and actions that lead to other states. Actions are computed from test cases and define the expected state from the GUI's model, which is then automatically compared to the actual state. Similarly, Lamancha et al. [Lamancha et al., 2013] used UML state machines to automate the test oracle. Mouchawrab et al. [Mouchawrab et al., 2011] found that increasing the precision of oracles and not relying only on state invariants greatly increased the effectiveness in fault detection in state-based approaches. Although state based approaches can be effective in automating the oracle, they can not be applied in our context because the functionalities we want to test in DAQ systems are not stateful and cannot, therefore, be modelled using state machines.

Di Lucca et al. [Di Lucca et al., 2002] automate the oracle by combining UML class diagrams that define the structural representation of the system, use cases that describe the behaviour of the system and decision tables with constraints on input variables and expected results. Vishal et al. [Vishal et al., 2012] also used decision tables to model the input and the constraints between the input data and their outputs. Decision tables could theoretically be used to automate the oracle of DAQ systems. However, the complexity of the input and output structures and the mappings between them would lead to very large decision tables which are hard to create, inspect and maintain.

8.3 Modelling Data Acquisition Systems

Several previous studies addressed modelling DAQ systems. However, these studies mainly focused on simulation and design. Baccigalupi [Baccigalupi et al., 1993] created an error model to simulate transmission errors, such as input/output delays and noise. Plesnyaev and Pazderin [Plesnyaev and Pazderin, 2003] proposed a mathematical model to improve the accuracy of calculating energy loss in DAQ systems. Booth et al. [Booth et al., 1992] developed a simulation model of DAQ systems to help developers evaluate different architectures. Oquendo [Oquendo, 2004] formalised and refined complex software systems, including DAQ systems, to help design a concrete architecture.

The applications of these approaches are concerned with simulation or design of the system and do not address automated testing. Validation of test cases and automated oracles are, therefore, not

addressed by these approaches.

8.4 Model-Based Testing

Most model-based testing techniques target the generation of valid data structures to be used in unit testing [Boyapati et al., 2002, Senni and Fioravanti, 2012, Khurshid and Marinov, 2004] that are typically much simpler than the input files needed for testing data processing systems. TestEra, for example, generates complex input data from Alloy specifications that describe the data structure and the relations between the fields of the structure [Khurshid and Marinov, 2004]. The technique presented in this dissertation instead targets the generation of large, complex, and invalid system input data for system testing.

The generation of invalid test inputs is mainly addressed by model-based techniques that focus on security testing. These approaches, known as *threat modelling techniques*, rely on models used to capture the characteristics of typical malicious (and invalid) inputs that should be properly handled by the SUT. Models like attack trees [Morais et al., 2011], UML state machines [Hussein and Zulkernine, 2006], and transition nets [Xu et al., 2012], are used to generate sequences of illegal actions, which are not relevant for testing data processing systems where the complexity of the testing process lies in the definition of the input data and mappings to complex, structured outputs. The main limitation of these approaches is that they cannot generate test inputs from scratch to test new data requirements.

8.5 Mutation-Based Testing

Specification-based mutation testing is a mutation testing approach that uses mutation operators to seed faults into specification models (e.g. a state machine) to generate specification mutants [Jia and Harman, 2011]. Approaches that use specification mutants to generate test inputs often deal with inputs that are less complex than those processed by data processing systems. Approaches that generate mutated statecharts can only generate inputs that are sequences of system operations [Schlick et al., 2011], while approaches that mutate class diagrams have only been used to test model transformation systems in which the state diagram itself is the input [Mottu et al., 2006]. Mutated XML Schema have been used to generate invalid XML data structures [Xu et al., 2005].

Data mutation approaches use mutation operators to generate new test inputs from existing ones (i.e. by altering valid field data) [Shan and Zhu, 2009, Bertolino et al., 2014, De Jonge and Visser, 2012]. The main limitation of these approaches is that they rely upon mutation operators that are specific to the inputs of the SUT and cannot be reused across different projects; in addition, they cannot generate test inputs from scratch to test new data requirements. The approach presented in this dissertation relies instead upon generic mutation operators that are tailored to the specific fault model of the SUT with the adoption of a data model annotated with UML stereotypes and OCL queries. This has the advantage of providing a more generic solution while enabling its tailoring to a domain-specific fault model.

Shan et al., report about the effectiveness of the application of ad hoc mutation operators on the inputs of existing test cases to identify faults [Shan and Zhu, 2009]. Existing test cases are executed on the mutated inputs; test cases that do not fail are manually inspected by the software developers

because they may indicate the presence of a fault. The adoption of ad hoc mutation operators limits the applicability of the approach to specific projects; furthermore, software engineers must manually verify test results by manually inspecting the error messages of the SUT. With our approach, the test oracle is automatically generated as part of the modelling effort, albeit at a greater modelling cost.

De Jonge et al. test syntax error recovery systems by means of mutation operators for Abstract Syntax Trees (ASTs) [De Jonge and Visser, 2012]. AST operators can be applied to generate faulty inputs from files that can be parsed according to an abstract syntax tree. Although the use of ASTs gives a wider applicability to the approach, it lacks methods for controlling the generation of trivial erroneous inputs, which may affect the effectiveness of such an approach in the presence of nontrivial data structures. Furthermore, the approach from de Jonge et al. includes a trivial oracle strategy that works only for data correction systems; in fact, they look for failures by comparing the output of the SUT with the unmutated AST. The technique presented in this dissertation instead can be applied to a wider range of software systems thanks to the adoption of OCL constraints as oracles.

Different from the other approaches, Bertolino et al. do not use fault models to generate new faulty inputs, but use fault-models to alter the configuration files that specify which inputs an authorisation system can accept [Bertolino et al., 2014]. The fault model in this case is defined as a set of simple mutation operators specific for the grammar of the configuration files, and is used to generate new configurations that should reject inputs accepted by the original configuration. The technique presented in this dissertation does not target configuration files, but inputs. Studying the effect of errors introduced in configuration files is part of our future work.

Like our approach presented in Chapter 5, the approaches based on data mutation focus on a single objective (e.g. covering all the possible data faults of a fault model) but do not allow for covering the multiple objectives needed to perform effective robustness testing. In Section 6.4 of Chapter 6, we presented the following four search objectives for obtaining effective robustness tests: (1) including input data that covers all the classes of our data-model, (2) including data faults such that all the possible faults of our fault model have been covered, (3) covering all the clauses of the input/output constraints, and (4) maximising code coverage. The answer to *RQ1* in Section 6.7 shows that the search-based approach presented in this dissertation performs better than approaches that focus on the coverage of a single objective.

8.6 Other Test Generation Approaches

Other existing approaches use CFGs to generate both valid and invalid input data structures, but the existing approaches do not model the complex relationships among data fields [Hoffman et al., 2009, Zelenov and Zelenova, 2006]. An example of a relationship that cannot be modelled with CFGs is the fact that, in the SES-DAQ data, the first header pointer of a VCDU should be an offset to the first packet that starts in the same VCDU.

Fuzz testing approaches rely upon random inputs [Miller et al., 1990, Miller et al., 1995, Forrester and Miller, 2000] or random permutations of valid inputs generated by means of grammar-based specifications [Xiao et al., 2003, Godefroid et al., 2008]. Similarly to grammar-based approaches, fuzz testing cannot deal with inputs with a complex data structure and constraints leading to many trivially invalid inputs—unlike the ones generated by the approach presented in this dissertation.

8.7 Search-Based Software Testing

Most of the Search-Based Software Testing (SBST) techniques have primarily focused on unit testing [McMinn, 2004] or the testing of non-functional properties [Afzal et al., 2009]. Work on search-based robustness testing performed at the system level focuses either on the identification of performance issues [Briand et al., 2006], which are out of the scope of this dissertation, or functional faults caused by complex sequences of test inputs [Ali et al., 2012, Fu and Kone, 2014], or by input signals [Baresel et al., 2003, Wilmes and Windisch, 2010].

Ali et al. [Ali et al., 2012] exploit the information encoded into UML state machines and aspect-oriented modelling to generate test cases that stress the robustness of a software system by generating complex invocations of function calls. Similarly, Fu and Kone [Fu and Kone, 2014] use finite state machines to generate robustness test cases for protocol testing. In contrast with these techniques, we focus on systems for which it is important to generate complex data structures; thus, instead of using behavioural models such as state machines, we rely upon class diagrams and constraints.

In the context of embedded systems, metaheuristic search is used for the generation of input signals satisfying some given properties such as requirements on signal shapes [Baresel et al., 2003] or temporal constraints [Wilmes and Windisch, 2010]. Our work is complementary to these approaches since it addresses the problem of generating complex data structures by innovatively combining metaheuristic search and data mutation.

8.8 Test Approaches for Testing New Data Requirements

Most of the existing approaches that deal with the problem of testing evolving software are based on static program analysis techniques that aim to achieve high source code coverage [Cadar and Palikareva, 2014, Santelices et al., 2008, Xu et al., 2013]. These approaches do not deal with the generation of complex structured inputs; furthermore, although effective in achieving high code coverage, these approaches cannot guarantee that the generated tests cover all the system requirements.

Existing work on the generation of test cases in the presence of evolving models is related to the generation of test input sequences from evolving state machines [El-Fakih et al., 2004, Pap et al., 2007, Rapos and Dingel, 2012] and does not deal with the generation of complex data structures.

Most of the existing approaches for the generation of test inputs with complex data structures deal with the problem of generating test inputs from scratch. Bounded exhaustive testing approaches generate all the possible test inputs that match the structure of a given data model up to a given bound, and work with models specified in different formats: Java classes [Boyapati et al., 2002], constraint logic [Senni and Fioravanti, 2012], Alloy [Khurshid and Marinov, 2004], or Z specifications [Hörcher, 1995]. These techniques have been proven to be effective for testing software systems that process classical data structures like trees, but they may not scale once adopted to generate more complex structures like the ones required to test SES-DAQ.

A more efficient black-box test generation technique is UDITA [Gligoric et al., 2010]. UDITA requires that software engineers provide a set of generator methods, to build instances of the data structure, and predicates, to validate the generated instances. UDITA relies upon the Java Path Finder

model-checker [Visser et al., 2004] to generate all the instances that satisfy the given predicates. The implementation of generator methods can be quite expensive for complex data models; furthermore, UDITA cannot be adopted to reuse existing test inputs to test new data requirements.

8.9 Model-Based Slicing Approaches

In this dissertation, we introduced an algorithm that relies upon model slicing to improve the performance of constraints solving. Other approaches that rely upon model slicing to improve the performance of constraint solvers exists, but they focus mostly on the slicing of static models for satisfiability purposes (i.e. to verify whether it is possible to create instances of the class diagram without violating any constraint [Shaikh et al., 2010, Balaban and Maraee, 2013]). The generated slices typically contain a subset of the elements belonging to the static models. The approach proposed in this dissertation instead performs slicing on an instance of a class diagram; furthermore, the proposed approach does not simply verify satisfiability but also supports the incremental augmentation of incomplete model instances.

Kato [Uzuncaova and Khurshid, 2008] is a tool that incrementally builds a solution for an Alloy model by grouping the predicates in two formulas (i.e. slices). A solution for the Alloy model is generated by solving the base slice first, and then by conjoining the solution with the predicates in the other slice. This approach has also been used to speed up the generation of test cases for testing product lines [Uzuncaova et al., 2010]. Kato deals with performance issues that depend on the presence of several constraints in a same Alloy model but it does not deal with the scalability problems related to the generation of large collections of elements. The approach proposed in this dissertation is thus complementary to Kato.

Chapter 9

Tool Suite Description

A tool suite was developed to automate the approaches that we propose in this dissertation.

The models and constraints developed for this project (as described in Chapter 3) using RSA are exported in the UML 2.2 (.uml) file format; UML 2.2 files are in the XML format and are supported by various UML tools.

We developed our tools in Java using the Eclipse Modelling Tools package. The implementations described below make use of the Eclipse Modelling Framework (EMF) and the following Eclipse Modelling Development Tools: UML2 and OCL [Eclipse Foundation, 2016]. The mentioned Eclipse development tools are used to load the UML 2.2 files and support our development efforts.

Section 9.1 describes the initial implementation of the Input Validation and Oracle tool used for the empirical study of Chapter 4. Sections 9.2 and 9.3 describe the core tools that were used for the rest of the empirical studies of this dissertation.

Additional tooling related to search-based test generation (Chapter 6) and test data generation by constraints solving (Chapter 7) was implemented for the research of this dissertation.

9.1 Initial Input Validation and Oracle Tool

Fig. 9.1 illustrates the architecture of the initial tool that was created to implement the test validation and oracle approach. Unlike the tool (which replaced this one) presented in Section 9.2, this initial version of the tool used hard-coded data parsers. As hard-coded parsers require frequent updates due to frequently evolving data requirements changes, parsers that make use of modelling annotations (i.e. UML stereotypes) to drive model instantiation were later developed.

The tool takes as an input the model and constraints of the system together with the input, configuration and output files of the test case that needs to be validated. The tool then processes these inputs and generates logs that capture the results. The model and constraints can be in any format that can be imported into the EMF environment as an Ecore model.

At the end of the process, for each system instantiation (i.e. test case evaluation), *Result Logs* are generated to record each failing constraint and the execution time for each execution of the tool (i.e.

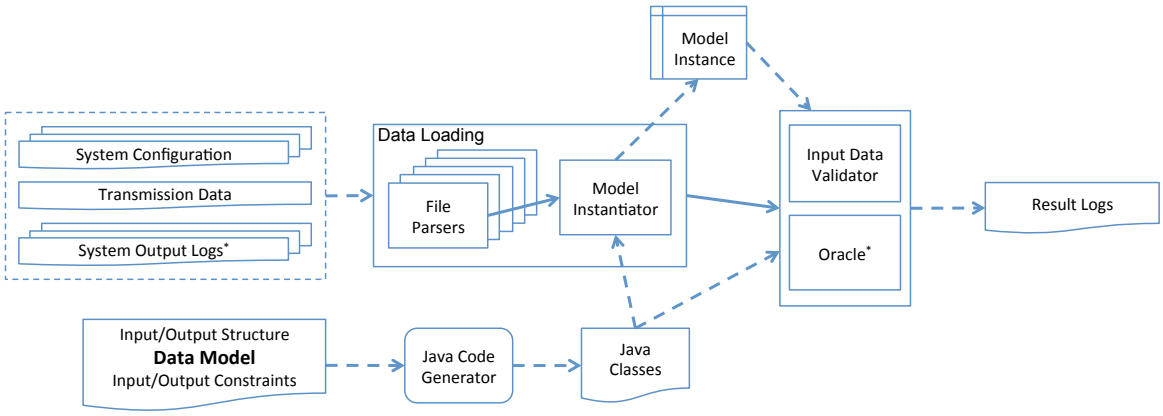


Figure 9.1. Tool architecture for the automation of test validation and oracle. Initial version using hard-coded file parsers to instantiate the *Model Instance*.
*The system functions as an Input Data Validator only, if the System Output Logs are not provided.

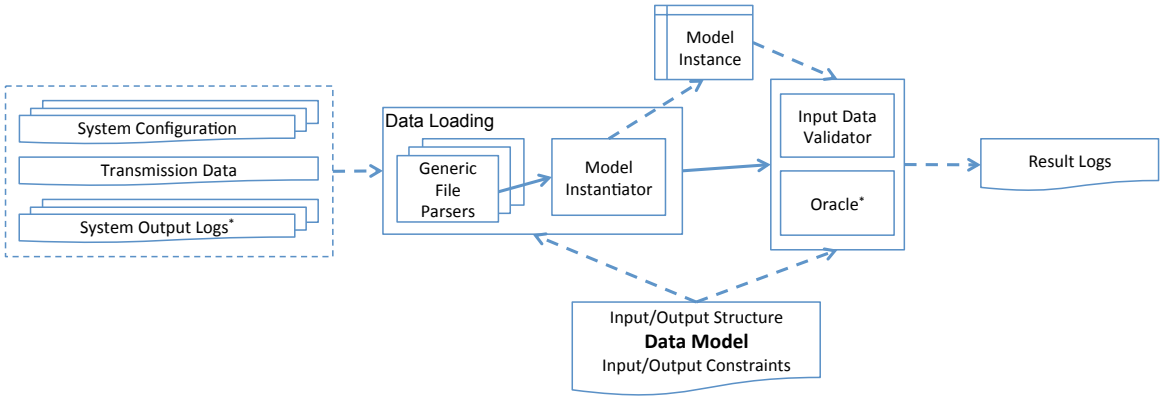


Figure 9.2. Tool architecture for the automation of test validation and oracles.
*The system functions as an Input Data Validator only, if the System Output Logs are not provided.

the times for model instantiation, test validation and oracle are recorded).

For this implementation, IBM RSA was used to create the UML class diagrams and the OCL constraints. The model and constraints were exported from RSA in the UML 2.2 (.uml) file format. These UML 2.2 files could then be imported into the EMF environment as an Ecore model. The *Java Code Generator* is a component in the EMF that takes as input the Ecore model and produces *Java Classes* that correspond to the model. The *Java Classes* directly support model instantiation and the OCL calls.

9.2 Input Validation and Oracle

Figure 9.2 shows the tool developed to validate system input data and provide a test oracle. The *Input Data Validator* only requires the *System Configuration* and *Transmission Data* as input. The *Oracle* additionally requires the *System Output Logs*. The following subsections describe the functionalities developed to implement the input validation and oracle tool.

9.2.1 Data loading

The data loading activity consists of reading concrete system data (i.e. input, output, and configuration data) and loading it into memory in the form of a *Model Instance*— that is, a collection of object instances conforming with the *Data Model* produced by the software engineers using our modelling methodology.

Although the specific format of the system data depends on the SUT, we believe that generic parsers can be used to load data into memory, thus freeing software engineers from the burden of writing a parser (or multiple parsers, given that different file types might be involved) for loading the data. For instance, XStream [XStream, 2016] is a well-known framework that loads XML data and provides an object oriented Application Programming Interface (API) to access and modify the data. In case the SUT uses a nonstandard data format, software engineers can take advantage of the data loading APIs of the SUT itself to ease the implementation of ad hoc parsers.

For this project, we developed generic file parsers that can automatically parse and instantiate the data associated with SES-DAQ. The *«FileInfo»* stereotype (described in Section 3.4) is used to indicate the type of file to be parsed (and hence, the particular generic file parser to use). For each of the system files, the data loader is guided by its associated class diagram file representation, annotated with parser-specific stereotypes. The following file types are supported for the SES-DAQ implementation: satellite bytestream files, XML files, and text files.

For example, to parse SES-DAQ bytestream data, the data loader is driven by the stereotypes introduced in Section 3.4. Note that in the case of our satellite bytestream parser, *System Configuration* information is required as it is used by the parser to provide information (as explained in Section 3.4, using OCL queries, specified by the *«GeneralisedAttribute»* and *«ConditionalAttribute»* stereotypes) while performing the data model instantiation related to the input transmission data.

9.2.2 Input validation

The input validation step consists of checking if the input for a given system is correct.

Given the instantiated input data, the input OCL constraints are verified by the tool. These constraints specify content rules that a valid system input must comply to. Detected failures are reported to the software engineer in the *Result Logs* for further inspection.

9.2.3 Oracle validation

The oracle validation step consists of checking if the output generated for a given system input is correct.

Given the instantiated output and input data, the tool checks if the input/output OCL constraints are satisfied. These constraints specify the expected output in the case of an invalid input. A violated constraint indicates that an erroneous input is not properly managed by the SUT, and thus corresponds to a test failure. Such failures are reported to the software engineer in the *Result Logs* for further inspection.

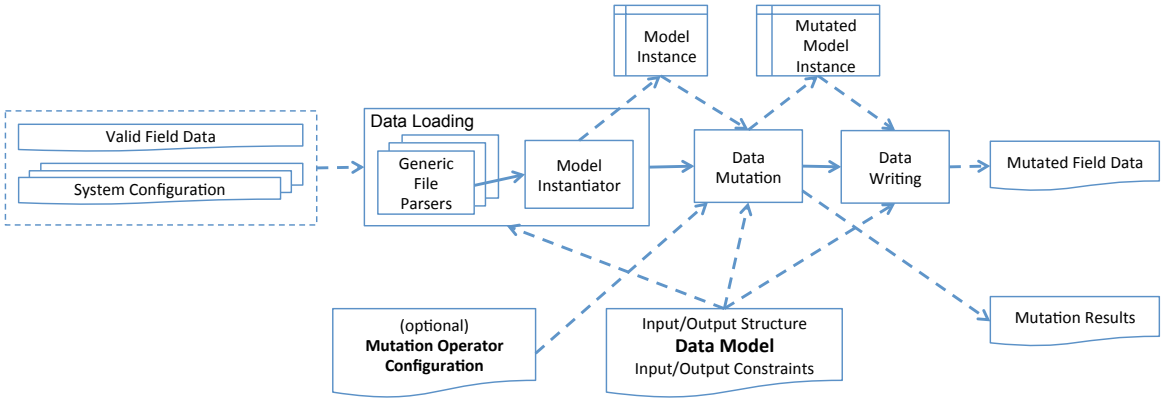


Figure 9.3. Tool architecture for the automation of test input generation.

9.3 Test Input Generation

Figure 9.3 shows the tool developed to generate test inputs via the mutation of valid field data. The following subsections describe the functionalities developed to implement the tool.

9.3.1 Data loading

The same data loading functionality developed for the *Input Validation and Oracle* tool of Section 9.2 is used. In this case, the tool requires the *System Configuration* files and *Valid Field Data* (unlike the input validation and oracle tool, the inputted transmission data must contain only valid data).

Sometimes the input field data may be too large to fit in memory or we might simply wish to consider only limited chunks of a field data file. For this reason, our parser enables software engineers to specify both the amount of data to load (e.g. 50 CADUs) and the location of the data to load. Software engineers can specify two possible locations: *beginning*, which indicates that the data is loaded from the beginning of the stream, or *random*, which indicates that the parser randomly chooses a position in the stream, then identifies the first synchronisation block that follows that position and starts loading the data. Section 2.1 provides an overview of the format of the transmission data.

9.3.2 Data mutation

Given the *Model Instance*, data mutation applies mutation operators according to the fault model (captured within the *Data Model*). The tool supports single or multiple data mutations.

With no parameters, the mutation proceeds as follows: a mutation operator is randomly selected and then applied to one of the elements within the *Model Instance* to which it can be applied (also selected randomly).

Optionally, the *Mutation Operator Configuration* can be used as follows:

(1) It can specify the mutation operator to be executed against the *Model Instance*—in this case, one of the elements to which the mutation operator can be applied is selected randomly. If there are no elements mutatable by the operator, an error is returned.
(2) It can specify the mutation operator and the type of element (i.e. the specific class instance type or attribute type) to target—one of the specified elements to which the mutation operator can be

applied is selected randomly. If there are no elements of the specified type that are mutatable by the operator, an error is returned.

(3) It can provide a list of $\langle target, operator \rangle$ pairs exactly specifying an ordered list of mutation operators to use and the exact model elements (i.e. the specific class instances or attributes) to mutate. If the operations cannot be completed, an error is returned.

(4) It can provide a list as specified in (3), followed by a final list item that matches the function of list item (1) or list item (2). If the operations cannot be completed, an error is returned.

The *Data mutation* step returns *Mutation Results* detailing (1) an ordered list summarising the successful application of mutation operator(s) and their target(s) within the *Model Instance* or (2) in the event of an error, information pertaining to the attempted operation(s).

9.3.3 Data writing

The generation of concrete *Mutated Field Data* from the *Mutated Model Instance* can be handled by using the same solution adopted for data loading. XStream APIs, for example, handle both the loading and saving of information from and to XML files [XStream, 2016]. For the case of SES-DAQ, we implemented a data writing tool that implements the same logic as the parser (i.e. using the same stereotypes introduced in Section 3.4) but for writing data instead of reading it.

Chapter 10

Conclusions and Future Work

This chapter is organised as follow. Section 10.1 summarises the contributions of this dissertation. Section 10.2 discusses potential future work.

10.1 Summary

In this dissertation, we addressed the challenges of testing data processing systems. In data processing systems, inputs are complex and the current practice is that software test engineers have to manually write test cases; they have to carefully handcraft input data, while ensuring compliance with the multiple constraints between data fields to prevent the generation of trivially invalid inputs. Assessing test results often means analysing complex output and log data. Software engineers typically write scripts to evaluate whether the generated outputs for a given input are correct (i.e. the test oracles are written manually); such an approach requires a high development effort per test case and should system specifications change, the scripts must be reassessed to ensure they are still valid.

We presented four approaches to enable the automated testing of data processing systems using model-driven approaches. Each of the approaches addresses distinct and important problems related to testing. These techniques are supported by the data models generated according to our modelling methodology introduced in Chapter 3. Our modelling methodology uses UML class diagrams and OCL constraints as a notation; we also created a profile to define stereotypes to support data parsing and test input generation. The results of our empirical evaluation showed that the application of the modelling methodology is scalable as the size of the model and constraints was manageable on a real data processing DAQ system.

The first approach is a technique for the automated validation of test inputs and oracles based on the application and tailoring of MDE technologies. The results of our empirical evaluation showed that the approach is scalable as the input and oracle validation process executed within reasonable times on real transmission files. For a transmission file of 2 GB, it took less than one hour to apply input data and oracle validation. In terms of scalability, the relationship between execution time and input file size is linear, suggesting that much larger files can be handled in the future using our approach.

The second approach is a model-based technique that automatically generates faulty test inputs

for the purpose of robustness testing, by relying upon generic mutation operators that alter data collected in the field. An empirical evaluation performed with a data processing system, shows that our automated approach achieves slightly better instruction coverage than the manual testing taking place in practice, based on domain expertise.

The third approach is an evolutionary algorithm to automate the robustness testing of data processing systems through optimised test suites. The empirical results obtained by applying our search-based testing approach to test an industrial data processing system show that it outperforms the previous approaches based on fault coverage and random generation: higher coverage is achieved with smaller test suites. Furthermore, we show that fitness functions based on models alone (i.e. not requiring test case execution) can achieve good coverage results, while significantly reducing test suite size. This is of practical importance as test generation is much quicker and often more practical when no test execution is required.

Finally, the fourth approach is an automated, model-based approach that reuses field data to generate test inputs that fit new data requirements for the purpose of testing data processing systems. The empirical study shows that the proposed approach scales in the presence of complex data structures. In particular, the study shows that the input generation algorithm based on model slicing and constraint solving can produce, in a reasonable amount of time, test input data that is over ten times larger in size than the data that can be generated with constraint solving only. To evaluate the effectiveness of the proposed approach, we generated test inputs to stress software robustness and we measured the code covered when executing the generated test inputs against the software. The results show that the generated test inputs cover most of the source code instructions of the updated software written to implement new data requirements. The generated test inputs also achieve more code coverage than the test cases implemented by experienced software engineers.

10.2 Future Work

Future work should prove the generalisability of the proposed techniques to other contexts. In particular, we identified three topics of interest.

The modelling methodology and related approaches should be applied to other case study systems. One threat to the external validity of the empirical results of this dissertation is that we only consider the application of our approaches against the SES-DAQ system. Future studies need to be made considering other data processing systems in order to better assess the scalability, applicability, and effectiveness of the approaches. Additionally, when seeking to generate optimised test suites we will consider the use of a multi-objective optimisation algorithm; for example, in the case that we want to maximise code coverage while keeping the test suite size at a minimum.

When developing data processing systems, external data needs to be loaded by the system for analysis. This functionality is often achieved by using custom, manually written data parsers. This work is complex and labour intensive and the related software implementations require frequent refactoring due to frequent changes in data requirements. Although there are many examples of tools supporting text-based parsing (e.g. [XStream, 2016]) and others supporting binary parsing, we believe that the bytestream parser (that uses our data model annotated with our custom UML profile) developed for this project is one of the first generic model-driven parser approaches targeting binary files. The com-

plexity of applying the bytestream parser in different contexts should be studied as part of our future work.

The effect of errors introduced in configuration files should be studied. The work of this dissertation considered only mutations to the input of a data processing system (e.g. in this dissertation, we considered the mutation of the satellite bytestream data received by the SES-DAQ). Data processing systems also accept configuration information used to define how such systems process the input data. Studying the effect of errors introduced in configuration files is part of our future work.

List of Papers

Published papers included in this dissertation:

- Daniel Di Nardo, Nadia Alshahwan, Lionel C. Briand, Elizabeta Fourneret, Tomislav Nakić-Alfirević, and Vincent Masquelier. "Model based test validation and oracles for data acquisition systems." In *Proceedings of the 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pp. 540-550. IEEE, 2013.
- Daniel Di Nardo, Fabrizio Pastore, and Lionel Briand. "Generating complex and faulty test data through model-based mutation analysis." In *Proceedings of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1-10. IEEE, 2015.
- Daniel Di Nardo, Fabrizio Pastore, Andrea Arcuri, and Lionel Briand. "Evolutionary robustness testing of data processing systems using models and data mutation." In *Proceedings of the 2015 IEEE/ACM 30th International Conference on Automated Software Engineering (ASE)*, pp. 126-137. IEEE, 2015.

Unpublished papers included in this dissertation:

- Daniel Di Nardo, Fabrizio Pastore, and Lionel Briand. "Augmenting field data for testing systems subject to incremental requirements changes." Submitted for publication, 2016.

Bibliography

[Afzal et al., 2009] Afzal, W., Torkar, R., and Feldt, R. (2009). A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957 – 976.

[Ali et al., 2010] Ali, S., Briand, L., Hemmati, H., and Panesar-Walawege, R. (2010). A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, 36(6):742–762.

[Ali et al., 2012] Ali, S., Briand, L. C., and Hemmati, H. (2012). Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems. *Software and Systems Modeling*, 11(4):633–670.

[Ali et al., 2013] Ali, S., Zohaib Iqbal, M., Arcuri, A., and Briand, L. C. (2013). Generating test data from OCL constraints with search techniques. *IEEE Transactions on Software Engineering*, 39(10):1376–1402.

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

[Anand et al., 2013] Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., and McMinn, P. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001.

[Anastasakis et al., 2007] Anastasakis, K., Bordbar, B., Georg, G., and Ray, I. (2007). UML2Alloy: A challenging model transformation. In *Model Driven Engineering Languages and Systems*, pages 436–450. Springer, Berlin, Heidelberg.

[Arcuri and Briand, 2014] Arcuri, A. and Briand, L. (2014). A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250.

[Atos, 2016] Atos (2016). Bull welcome - bullx B500 blade system. http://www.bull.com/bullx-b500-range. Accessed Mar. 12, 2016.

[Baccigalupi et al., 1993] Baccigalupi, A., DaPonte, P., and D'Apuzzo, M. (1993). An improved error model of data acquisition systems. In *Proceedings of the Confrence on Instrumentation and Measurement Technology Conference*, pages 781–787.

[Balaban and Maraee, 2013] Balaban, M. and Maraee, A. (2013). Finite satisfiability of UML class diagrams with constrained class hierarchy. *ACM Transactions on Software Engineering and Methodology*, 22(3):24:1–24:42.

[Baresel et al., 2003] Baresel, A., Pohlheim, H., and Sadeghipour, S. (2003). Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In *Genetic and Evolutionary Computation — GECCO 2003*, volume 2724 of *Lecture Notes in Computer Science*, pages 2428–2441. Springer Berlin Heidelberg.

[Bertolino et al., 2014] Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E., Martinelli, F., and Mori, P. (2014). Testing of PolPA-based usage control systems. *Software Quality Journal*, 22(2):241–271.

[Booth et al., 1992] Booth, A., Botlo, M., Dorenbosch, J., Kapoor, V., Milner, E. C., Wang, C., and Wang, E. M. (1992). DAQSIM: a data acquisition system simulation tool. In *Proceedings of the Conference of Nuclear Science Symposium and Medical Imaging Conference*, volume 1, pages 485–487.

[Boyapati et al., 2002] Boyapati, C., Khurshid, S., and Marinov, D. (2002). Korat: Automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133, New York, NY, USA. ACM.

[Briand et al., 2006] Briand, L. C., Labiche, Y., and Shousha, M. (2006). Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines*, 7(2):145–170.

[Cabot et al., 2008] Cabot, J., Clarisó, R., and Riera, D. (2008). Verification of UML/OCL class diagrams using constraint programming. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshops (ICSTW).*, pages 73–80, Los Alamitos, CA. IEEE.

[Cadar and Palikareva, 2014] Cadar, C. and Palikareva, H. (2014). Shadow symbolic execution for better testing of evolving software. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 432–435, New York, NY, USA. ACM.

[CCSDS, 2003] CCSDS (2003). Space packet protocol, CCSDS 133.0-B-1. http://public.ccsds.org/publications/archive/133x0b1c2.pdf.

[CCSDS, 2006] CCSDS (2006). AOS space data link protocol, CCSDS 732.0-B-2. http://public.ccsds.org/publications/archive/732x0b2c1s.pdf.

[CCSDS, 2011] CCSDS (2011). TM synchronization and channel coding, CCSDS 131.0-B-2. http://public.ccsds.org/publications/archive/131x0b2ec1.pdf.

[CCSDS, 2016] CCSDS (2016). The Consultative Committee for Space Data Systems (CCSDS). http://www.ccsds.org. Accessed Mar. 11, 2016.

[Cornett, 2016] Cornett, S. (2016). Minimum acceptable code coverage. http://www.bullseye.com/minimum.html. Accessed Mar. 11, 2016.

[Dalal et al., 1999] Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., and Horowitz, B. M. (1999). Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294, New York, NY, USA. ACM.

[De Jonge and Visser, 2012] De Jonge, M. and Visser, E. (2012). Automated evaluation of syntax error recovery. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 322–325, New York, NY, USA. ACM.

[Di Lucca et al., 2002] Di Lucca, G. A., Fasolino, A. R., Faralli, F., and De Carlini, U. (2002). Testing web applications. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, ICSM '02, pages 310–319, Washington, DC, USA. IEEE Computer Society.

[Di Nardo et al., 2013] Di Nardo, D., Alshahwan, N., Briand, L. C., Fourneret, E., Nakić-Alfirević, T., and Masquelier, V. (2013). Model based test validation and oracles for data acquisition systems. In *Proceedings of the 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE'13)*, pages 540–550, Los Alamitos, CA. IEEE.

[Di Nardo et al., 2015a] Di Nardo, D., Pastore, F., Arcuri, A., and Briand, L. (2015a). Evolutionary Robustness Testing of Data Processing Systems Using Models and Data Mutation. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*, pages 126–137, Los Alamitos, CA. IEEE.

[Di Nardo et al., 2015b] Di Nardo, D., Pastore, F., and Briand, L. (2015b). Generating complex and faulty test data through model-based mutation analysis. In *8th International Conference on Software Testing, Verification and Validation (ICST'15)*, Los Alamitos, CA. IEEE Computer Society.

[Di Nardo et al., 2016] Di Nardo, D., Pastore, F., and Briand, L. (2016). Augmenting field data for testing systems subject to incremental requirements changes. Submitted for publication.

[Dias Neto et al., 2007] Dias Neto, A. C., Subramanyan, R., Vieira, M., and Travassos, G. H. (2007). A survey on model-based testing approaches: A systematic review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech '07): held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 31–36.

[Eclipse Foundation, 2016] Eclipse Foundation (2016). Eclipse model development tools (mdt).

[El-Fakih et al., 2004] El-Fakih, K., Yevtushenko, N., and v. Bochmann, G. (2004). Fsm-based incremental conformance testing methods. *Software Engineering, IEEE Transactions on*, 30(7):425–436.

[ESA, 2016] ESA (2016). Overview / Copernicus / Observing the Earth / Our Activities / ESA. http://www.esa.int/Our_Activities/Observing_the_Earth/Copernicus/Overview4. Accessed Mar. 11, 2016.

[FlightRadar24, 2016] FlightRadar24 (2016). Flightradar24.com - live flight tracker! http://www.flightradar24.com. Accessed on Mar. 22, 2016.

[Forrester and Miller, 2000] Forrester, J. E. and Miller, B. P. (2000). An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th USENIX Windows System Symposium*, pages 59–68, Berkeley, CA, USA. Seattle, USENIX Association.

[Fraser and Arcuri, 2012] Fraser, G. and Arcuri, A. (2012). The seed is strong: Seeding strategies in search-based software testing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 121–130.

[Fu and Kone, 2014] Fu, Y. and Kone, O. (2014). Security and robustness by protocol testing. *Systems Journal*, 8(3):699–707.

[Gaudel and Le Gall, 2008] Gaudel, M.-C. and Le Gall, P. (2008). Testing data types implementations from algebraic specifications. In *Formal methods and testing*, pages 209–239. Springer-Verlag.

[Gligoric et al., 2010] Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., and Marinov, D. (2010). Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 225–234. ACM.

[Godefroid et al., 2008] Godefroid, P., Kiezun, A., and Levin, M. Y. (2008). Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA. ACM.

[Groovy Community, 2016] Groovy Community (2016). The Groovy programming language. http://www.groovy-lang.org. Accessed Mar. 15, 2016.

[Hoffman et al., 2009] Hoffman, D., Wang, H.-Y., Chang, M., and Ly-Gagnon, D. (2009). Grammar based testing of HTML injection vulnerabilities in RSS feeds. In *Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, TAIC-PART '09, pages 105–110, Washington, DC, USA. IEEE Computer Society.

[Hörcher, 1995] Hörcher, H.-M. (1995). Improving software tests using z specifications. In *Proceedings of the 9th International Conference of Z Usres on The Z Formal Specification Notation*, ZUM '95, pages 152–166, London, UK, UK. Springer-Verlag.

[Hussein and Zulkernine, 2006] Hussein, M. and Zulkernine, M. (2006). UMLintr: A UML profile for specifying intrusions. In *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, ECBS '06, pages 279–288, Washington, DC, USA. IEEE Computer Society.

[Iqbal et al., 2015] Iqbal, M., Arcuri, A., and Briand, L. (2015). Environment modeling and simulation for automated testing of soft real-time embedded software. *Software and Systems Modeling*, 14(1):483–524.

[Iqbal et al., 2012] Iqbal, M. Z., Ali, S., Yue, T., and Briand, L. (2012). Experiences of applying UML/MARTE on three industrial projects. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS '12)*, pages 642–658.

[ISO/IEEE, 2010] ISO/IEEE (2010). Systems and software engineering – vocabulary.

[Jackson, 2002] Jackson, D. (2002). Alloy: A lightweight object modelling notation. *ACM Transactions of Software Engineering and Methodology*, 11(2):256–290.

[Jackson, 2015] Jackson, D. (2015). Alloy analyzer website. http://alloy.mit.edu/.

[Jaffuel and Legeard, 2006] Jaffuel, E. and Legeard, B. (2006). LEIRIOS test generator: automated test generation from B models. In *Proceedings of the 7th Tnternational Conference on Formal Specification and Development in B*, B'07, pages 277–280.

[Jia and Harman, 2011] Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678.

[Khurshid and Marinov, 2004] Khurshid, S. and Marinov, D. (2004). TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering*, 11(4):403–434.

[Knowles and Corne, 1999] Knowles, J. and Corne, D. (1999). The Pareto archived evolution strategy: A new baseline algorithm for Pareto multiobjective optimisation. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99*. IEEE Computer Society.

[Lamancha et al., 2013] Lamancha, B. P., Polo, M., Caivano, D., Piattini, M., and Visaggio, G. (2013). Automated generation of test oracles using a model-driven approach. *Information and Software Technology*, 55(2):301–319.

[Last et al., 2004] Last, M., Friedman, M., and Kandel, A. (2004). Using data mining for automated software testing. *International Journal of Software Engineering and Knowledge Engineering*, 14(4):369–393.

[Le Berre and Parrain, 2010] Le Berre, D. and Parrain, A. (2010). The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64.

[Leuschel et al., 2011] Leuschel, M., Falampin, J., Fritz, F., and Plagge, D. (2011). Automated property verification for large scale B models with ProB. *Formal Aspects of Computing*, 23(6):683–709.

[Li et al., 2013] Li, N., Meng, X., Offutt, J., and Deng, L. (2013). Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report). In *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE'13)*, pages 380–389, Pasadena, CA. IEEE, IEEE.

[Luke, 2013] Luke, S. (2013). *Essentials of Metaheuristics*. Lulu, second edition. Available at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[Manolache and Kourie, 2001] Manolache, L. I. and Kourie, D. G. (2001). Software testing using model programs. *Software Practical Experience*, 31(13):1211–1236.

[Marre and Blanc, 2005] Marre, B. and Blanc, B. (2005). Test selection strategies for Lustre descriptions in GATeL. *Theory on Computure Science Electronic Notes*, 111:93–111.

[Maven, Apache, 2016] Maven, Apache (2016). Apache Maven project. http://maven.apache.org. Accessed Mar. 15, 2016.

[McCarthy, 2001] McCarthy, K. (2001). Dow Jones average collapses to 0.20. http://www.theregister.co.uk/2001/03/19/dow_jones_average_collapses. Accessed Mar. 11, 2016.

[McMinn, 2004] McMinn, P. (2004). Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156.

[Memon et al., 2000] Memon, A. M., Pollack, M. E., and Soffa, M. L. (2000). Automated test oracles for GUIs. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: 21st century applications (SIGSOFT/FSE '00)*, pages 30–39.

[Mens and Tourwé, 2004] Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

[Miller et al., 1990] Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44.

[Miller et al., 1995] Miller, B. P., Koski, D., Lee, C. P., Maganty, V., Murthy, R., Natarajan, A., and Steidl, J. (1995). Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin-Madison, Computer Sciences Department.

[Moonen et al., 1997] Moonen, J. R., Romijn, J. M., Sies, O., Springintveld, J. G., Feijs, L. G., and Koymans, R. L. (1997). A two-level approach to automated conformance testing of VHDL designs. Technical report, Moonen Philips Research Laboratories, Amsterdam, Netherlands.

[Morais et al., 2011] Morais, A., Cavalli, A., and Martins, E. (2011). A model-based attack injection approach for security validation. In *Proceedings of the 4th International Conference on Security of Information and Networks*, SIN '11, pages 103–110, New York, NY, USA. ACM.

[Mottu et al., 2006] Mottu, J.-M., Baudry, B., and Le Traon, Y. (2006). Mutation analysis testing for model transformations. In *Proceedings of the 2nd European Conference on Model Driven Architecture: Foundations and Applications (ECMDA-FA'06)*, pages 376–390, Berlin, Heidelberg. Springer-Verlag.

[Mouchawrab et al., 2011] Mouchawrab, S., Briand, L. C., Labiche, Y., and Di Penta, M. (2011). Assessing, comparing, and combining state machine-based testing and structural testing: a series of experiments. *Transactions on Software Engineering (TSE)*, 37(2):161–187.

[Mountainminds, 2006] Mountainminds (2006). EclEmma - Java Code Coverage for Eclipse.

[Mussa et al., 2009] Mussa, M., Ouchani, S., Al Sammane, W., and Hamou-Lhadj, A. (2009). A survey of model-driven testing techniques. In *Proceedings of the 9th International Conference on Quality Software (QSIC '09)*, pages 167–172.

[NASA, 2016] NASA (2016). NASA - Direct Readout Laboratory - RT-STPS. https://directreadout.sci.gsfc.nasa.gov/?id=dspContent&cid=69. Accessed Mar. 11, 2016.

[OMG, 2015] OMG (2015). The object management group, the object constraint language. http://www.omg.org/spec/OCL/.

[Oquendo, 2004] Oquendo, F. (2004). Formally refining software architectures with π-ARL: a case study. *SIGSOFT Software Engineering Notes*, 29(5):1–26.

[Pap et al., 2007] Pap, Z., Subramaniam, M., Kovács, G., and Németh, G. Á. (2007). A bounded incremental test generation algorithm for finite state machines. In Petrenko, A., Veanes, M., Tretmans, J., and Grieskamp, W., editors, *Testing of Software and Communicating Systems*, volume 4581 of *Lecture Notes in Computer Science*, pages 244–259. Springer, Berlin, Heidelberg.

[Parks and Miller, 1998] Parks, G. T. and Miller, I. (1998). Selective breeding in a multiobjective genetic algorithm. In *Parallel Problem Solving From Nature—PPSN V*. Springer Berlin Heidelberg.

[Plesnyaev and Pazderin, 2003] Plesnyaev, E. and Pazderin, A. (2003). Data acquisition system faults detection. In *Proceedings of the IEEE Conference on Control Applications*, volume 2, pages 1390–1394.

[Pretschner et al., 2013] Pretschner, A., Holling, D., Eschbach, R., and Gemmar, M. (2013). A generic fault model for quality assurance. In *Model-Driven Engineering Languages and Systems*, pages 87–103. Springer.

[Rapos and Dingel, 2012] Rapos, E. J. and Dingel, J. (2012). Incremental test case generation for UML-RT models using symbolic execution. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 962–963, Los Alamitos, CA, USA. IEEE Computer Society.

[Sabetzadeh et al., 2011] Sabetzadeh, M., Nejati, S., Briand, L., and Mills, A.-H. E. (2011). Using SysML for modeling of safety-critical software-hardware interfaces: Guidelines and industry experience. In *Proceedings of the IEEE 13th International Symposium on High-Assurance Systems Engineering*, HASE '11, pages 193–201.

[Santelices et al., 2008] Santelices, R., Chittimalli, P. K., Apiwattanapong, T., Orso, A., and Harrold, M. J. (2008). Test-suite augmentation for evolving software. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE'08, pages 218–227, Washington, DC, USA. IEEE Computer Society.

[Schlick et al., 2011] Schlick, R., Herzner, W., and Jöbstl, E. (2011). Fault-based generation of test cases from UML-models: Approach and some experiences. In *Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'11, pages 270–283, Berlin, Heidelberg. Springer-Verlag.

[Schroeder et al., 2002] Schroeder, P. J., Faherty, P., and Korel, B. (2002). Generating expected results for automated black-box testing. In *Proceedings of the 17th IEEE International Conference in Automated Software Engineering (ASE '02)*, pages 139–148.

[Schroeder, 2010] Schroeder, S. (2010). Word lens: Where translation and augmented reality meet. http://mashable.com/2010/12/17/word-lens/.

[Senni and Fioravanti, 2012] Senni, V. and Fioravanti, F. (2012). Generation of test data structures using constraint logic programming. In *Proceedings of the 6th International Conference on Tests and Proofs*, TAP'12, pages 115–131, Berlin, Heidelberg. Springer-Verlag.

[SES, 2016] SES (2016). SES - Global Satellite Services Provider - Your Satellite Company. http://www.ses.com. Accessed Mar. 11, 2016.

[Shahamiri et al., 2009] Shahamiri, S. R., Kadir, W. M. N. W., and Mohd-Hashim., S. Z. (2009). A comparative study on automated software test oracle methods. In *Proceedings of the 4th International Conference on Software Engineering Advances (ICSEA '09)*, pages 140–145.

[Shaikh et al., 2010] Shaikh, A., Clarisó, R., Wiil, U. K., and Memon, N. (2010). Verification-driven slicing of UML/OCL models. In *Proceedings of the 2010 IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 185–194, New York, NY, USA. ACM.

[Shan and Zhu, 2009] Shan, L. and Zhu, H. (2009). Generating structurally complex test cases by data mutation. *The Computer Journal*, 52(5):571–588.

[The Eclipse Foundation, 2013] The Eclipse Foundation (2013). Ecore tools. http://www.eclipse.org/modeling/emft/?project=ecoretools.

[Tretmans, 2008] Tretmans, J. (2008). Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer-Verlag.

[Utting et al., 2012] Utting, M., Pretschner, A., and Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability (STVR)*, 22(5):297–312.

[Uzuncaova and Khurshid, 2008] Uzuncaova, E. and Khurshid, S. (2008). Constraint prioritization for efficient analysis of declarative models. In Cuellar, J., Maibaum, T., and Sere, K., editors, *FM 2008: Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 310–325. Springer-Verlag, Berlin Heidelberg.

[Uzuncaova et al., 2010] Uzuncaova, E., Khurshid, S., and Batory, D. (2010). Incremental test generation for software product lines. *IEEE Transactions on Software Engineering*, 36(3):309–322.

[Vanmali et al., 2002] Vanmali, M., Last, M., and Kandel, A. (2002). Using a neural network in the software testing process. *International Journal of Intelligent Systems*, 17(1):45–62.

[Varrette et al., 2014] Varrette, S., Bouvry, P., Cartiaux, H., and Georgatos, F. (2014). Management of an academic hpc cluster: The ul experience. In *Proceedings of the 2014 International Conference on High Performance Computing & Simulation (HPCS)*, pages 959–967, Bologna, Italy. IEEE.

[Veanes et al., 2008] Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., and Nachmanson, L. (2008). Model-based testing of object-oriented reactive systems with SpecExplorer. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer-Verlag.

[Vishal et al., 2012] Vishal, V., Kovacioglu, M., Kherazi, R., and Mousavi, M. R. (2012). Integrating model-based and constraint-based testing using SpecExplorer. In *Proceedings of the 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW '12)*, pages 219–224.

[Visser et al., 2004] Visser, W., Păsăreanu, C. S., and Khurshid, S. (2004). Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'04, pages 97–107, New York, NY, USA. ACM.

[Walton and Poore, 2000] Walton, G. H. and Poore, J. H. (2000). Generating transition probabilities to support model-based software testing. *Software: Practice and Experience*, 30(10):1095–1106.

[Wicker and Bhargava, 1999] Wicker, S. B. and Bhargava, V. K., editors (1999). *Reed-Solomon Codes and Their Applications*. John Wiley & Sons, Inc., New York, NY, USA.

[Wilmes and Windisch, 2010] Wilmes, B. and Windisch, A. (2010). Considering signal constraints in search-based testing of continuous systems. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 202–211.

[Wohlin et al., 2000] Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., and Wesslen, A. (2000). *The Experimentation in Software Engineering – An Introduction*. Kluwer.

[Xiao et al., 2003] Xiao, S., Deng, L., Li, S., and Wang, X. (2003). Integrated TCP/IP protocol software testing for vulnerability detection. In *2003 International Conference on Computer Networks and Mobile Computing (ICCNMC'03).*, pages 311–319, Washington, DC. IEEE, IEEE Computer Society.

[Xie and Memon, 2007] Xie, Q. and Memon, A. M. (2007). Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions in Software Engineering Methodology (TOSEM)*, 16(1).

[XStream, 2016] XStream (2016). XStream - About XStream. http://x-stream.github.io/. Accessed Mar. 11, 2016.

[Xu et al., 2012] Xu, D., Tu, M., Sanford, M., Thomas, L., Woodraska, D., and Xu, W. (2012). Automated security test generation with formal threat models. *IEEE Transactions of Dependable and Secure Computing*, 9(4):526–539.

[Xu et al., 2005] Xu, W., Offutt, J., and Luo, J. (2005). Testing web services by XML perturbation. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 266–275.

[Xu et al., 2013] Xu, Z., Cohen, M. B., Motycka, W., and Rothermel, G. (2013). Continuous test suite augmentation in software product lines. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 52–61, New York, NY, USA. ACM.

[Yahoo!, 2016] Yahoo! (2016). Yahoo Finance - Business Finance, Stock Market, Quotes, News. http://finance.yahoo.com. Accessed on Mar. 22, 2016.

[Zelenov and Zelenova, 2006] Zelenov, S. and Zelenova, S. (2006). Automated generation of positive and negative tests for parsers. In Grieskamp, W. and Weise, C., editors, *Formal Approaches to Software Testing*, volume 3997 of *Lecture Notes in Computer Science*, pages 187–202. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Zitzler et al., 2001] Zitzler, E., Laumanns, M., and Thiele, L. (2001). SPEA2: Improving the strength Pareto evolutionary algorithm. In *Proceedings of EUROGEN 2001 - Evolutionary Methods for Design, Optimisation and Control with Applications to Industrial Problems*.

Appendix A

SES-DAQ Supplemental Data Model Information

A.1 Configuration Data

Fig A.1 shows a simplified data model of the configuration files used for the SES-DAQ system. The class *Configuration* is created as a root for the configuration content of the model. It is annotated with the *«ConfigData»* stereotype. The class *Configuration* contains two other classes (*RtStpsConfig* and *ValidApidsConfig*); each of the contained classes is a root of a corresponding modelled configuration data file. The two files modelled here are the XML Real-time Software Telemetry Processing System (RT-STPS) configuration file[1] and the text file containing the valid APIDs properties.

The RT-STPS file contains information directing the SES-DAQ on how to process the satellite transmission data. For example, it contains information on what VCIDs are valid for the VCDUs of

[1]The SES-DAQ software incorporates the National Aeronautics and Space Administration (NASA) Direct Readout Laboratory's RT-STPS software package to process the transmission data (i.e. the incoming CADU bytestream) [NASA, 2016].

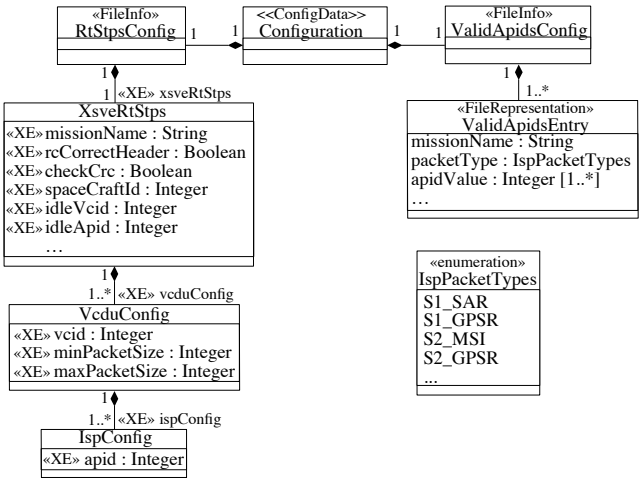


Figure A.1. Simplified model example for the configuration data in the case study system. Note: XE, XmlElement.

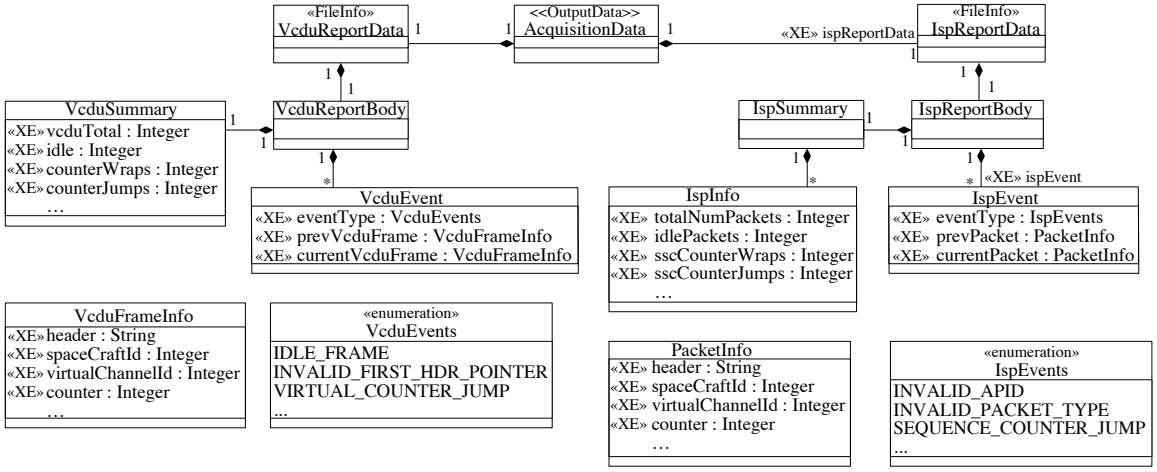


Figure A.2. Simplified model example for the output data in the case study system. Note: XE, XmlElement.

the transmission; and for each virtual channel, it lists the valid APIDs for the packets on that channel.

The valid APIDs properties file contains a list of valid APIDs for each of the packet types of the Sentinel missions. For example, for the Sentinel-1 mission, packets of the GPSR type can have an APID equal to one of 17 values.

A.2 Output Data

Fig A.2 shows a simplified data model of the output log files created by the SES-DAQ system. The class *AcquisitionData* is created as a root for the output content of the model. It is annotated with the *«OutputData»* stereotype. The class *AcquisitionData* contains two other classes (*VcduReportData* and *IspReportData*); each of the contained classes is the root of a corresponding modelled output log file. The two files modelled here are the VCDU report log and the ISP report log (both in the XML format).

The VCDU report log contains information related to the VCDUs received in a transmission; for example, it reports the number of idle VCDUs received. The log also reports on errors and events of interest occurring at the VCDU level; for example, it reports each occurrence of a *VIRTUAL_COUNTER_JUMP* (i.e. when the virtual channel frame count of a VCDU header is not one greater than the virtual channel frame count of the previously occurring VCDU header on the same virtual channel).

The ISP report log contains information related to the packets received in a transmission; for example, it reports the number of idle packets received. The log also reports on errors and events of interest occurring at the packet level; for example, it reports each occurrence of an *INVALID_APID* (i.e. when the APID value of a packet primary header does not correspond to a valid value contained in the RT-STPS configuration file—either the idle packet APID value or one of the valid active packet APID values).

```
<XSVE_RT_STPS>
    <Mission_Name>S1A</Mission_Name>
    <RS_Correct_Header>true</RS_Correct_Header>
    <Check_CRC>true</Check_CRC>
    <Spacecraft_ID>43</Spacecraft_ID>
    <Idle_VCID>63</Idle_VCID>
    <Idle_APID>2047</Idle_APID>
    ...
    <VCDU>
        <VCID>45</VCID>
        <Min_Packet_Size>7</Min_Packet_Size>
        <Max_Packet_Size>65540</Max_Packet_Size>
        <ISP>
            <APID>772</APID>
        </ISP>
    </VCDU>
    ...
</XSVE_RT_STPS>
```

Figure A.3. Example of a Real-time Software Telemetry Processing System configuration file.

```
S1.SAR.APIDs = 1052
S1.GPSR.APIDs = 769, 771, 772, 775, 777, 779, 780, 781, 785, 787, 788, 790, 791, 793, 795, 796, 797
...
```

Figure A.4. Example entries of the valid APID properties file.

A.3 Automated Parsing of XML Files

The parts of our data model that correspond to XML files exactly reflect the structure of the XML files upon which they are based. XML child elements are represented as class attributes or contained classes. XML sibling elements are represented as attributes of a same class. The attributes are assigned values based on text content of the XML element. The stereotype *«XmlElement»* is used to map the XML tag names to the attributes of the class diagram.

For example, Fig. A.3 shows the RT-STPS XML file that is modelled by the classes of Fig A.1 having the root *RtStpsConfg*. In the XML file, VCID is a child element of the VCDU element; correspondingly, the attribute *vcid* is contained within the *VcduConfig* class. In the XML file, the VCID, Min_Packet_Size, Max_Packet_Size, and ISP elements are all siblings; they are modelled as the *VcduConfig* class attributes *vcid*, *minPacketSize*, *maxPacketSize*, and *ispConfig*, respectively.

A.4 Automated Parsing of Text Files

The *«FileRepresentation»* stereotype is applied to the class that models the text file data. The *«FileRepresentation»* stereotype contains a list of row representations. Each row representation contains a regular expression corresponding to a line entry of the file; each regular expression contains capturing groups. Each row representation additionally contains mappings between the capturing groups of the regular expression and the attributes of the modelled class.

For example, Fig. A.4 shows the text file that is modelled by the class *ValidApidsConfig* of Fig A.1. The class *ValidApidsConfig* contains a set of *ValidApidsEntry* classes. In this example, one *ValidApid-*

sEntry instance is created for each row of the valid apids properties file. The class *ValidApidsEntry* is annotated with the stereotype *«FileRepresentation»* that contains the regular expression information and mapping information. The lines of the file of Fig. A.4 are represented by a single row representation (i.e. regular expression): ^(S[1-2])\.([A-Z]+)\.APIDs\s=\s([\w\s.,]+). While parsing, the regular expression is used to match the row entries of the file. The first capturing group is mapped to *missionName*, the second capturing group is mapped to *packetType*, and the third capturing group is mapped to the *apidValue* list. For example, the first *ValidApidsEntry* instance created corresponding to the first line of the file given by Fig. A.4 would contain the following attribute assignments: $missionName = S1$, $packetType = S1\_SAR$, $apidValue = \{1052\}$.