UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
ENGENHARIA DE COMPUTAÇÃO

MATEUS FELIPIN DALEPIANE

Implementation of a Model Based Test Case Generator for UML State Machines

Final Report presented in partial fulfillment of the requirements for the degree of Computer Engineer

Prof. Dr. Érika Fernandes Cota
Advisor

Prof. Dr. Patrick Heckeler
Coadvisor

Porto Alegre, December 2014

“YOLO.”

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

API	*Application Program Interface*
AT	*All Transitions*
ATC	*All Transition Coverage*
ATP	*All Trantision Pairs*
BSM	*Behavioral State Machine*
CC	*Combinatorial Coverage*
CUT	*Component Under Test*
EA	*Enterprise Architect*
EFSM	*Extended Finite State Machine*
FP	*Full Predicate*
GA	*Genetic Algorithms*
GDB	*GNU Debugger*
MBD	*Model Based Design*
PC	*Predicate Coverage*
PSM	*Protocol State Machine*
PTC	*Pairwise Transition Coverage*
SAT	*boolean SATisfiability*
SQL	*Structured Query Language*
TDG	*Test Data Generator*
TT	*Transition Trees*
UML	*Unified Modeling Language*

XMI *XML Metadata Interchange*

XML *eXtensible Markup Language*

LIST OF FIGURES

ABSTRACT

This work proposes the integration of a model-based test case generation framework for UML state machines into a commercial modeling tool. The framework was proposed by Heckeler et al. in 2013, and generates test cases for robustness testing of UML state machines modeled within Enterprise Architect.

Heckeler's work uses exported model files from Enterprise Architect (EA) to access the modeled state machine information. The test case generation is composed of three main tasks: abstract test case generation, executable test case generation and test cases execution control.

The generation of the abstract test cases, called path lists, is responsible for extracting all the necessary information directly from the model, generate a new representation of the transition system that complies with the boolean satisfiability solver used to resolve transition guards, and run queries on this transition system to generate all abstract test cases in order to satisfy the target coverage. These abstract test cases are lists of transitions that should be executed in the state machine.

The second task is responsible for transforming the path lists into executable test cases. In order to generate robustness tests, this task also extends the path lists with calls to all undefined transitions, using informations from the model. The executable test cases are based on CppUnit test suite. Besides generating the test cases this task also generate configuration scripts to control Gnu Debugger (GDB) during test execution. These scripts include commands for starting and stopping the recording needed for reverse transition execution, and are used to read the state encoding variables during runtime.

The third task runs the executable test cases, uses GDB to extract the values necessary for verification and to accelerate the test case execution using GDB reverse execution. When an error is detected, this task outputs a trace that shows the triggers call order to facilitate the error reproduction.

This work proposes an integrated solution within Enterprise Architect as a plugin, using EA Application Program Interface, in order to improve the end-user experience. As an initial integration work, it focuses on the integration of the first task, the abstract test

case generation.

In order to achieve this initial integration, this work proposes a C# state machine model to store all data extracted from EA, presented as a class diagram, and a software architecture to be used within the plugin, that can easily be extended to include both new test case generation techniques and test case coverages into it.

The integrated test case generator validation is done using the same two state machines used in the framework proposal by Heckeler, by comparing the path list output from the previously implementation with the output from the EA integrated solution. The first state machine is a hash table implementation that comprises 3 states and 11 transitions. The second state machine is a traffic light controller implementation that comprises 11 states and 19 transitions.

Keywords: UML, state machine, test case generation, Enterprise Architect.

RESUMO

Este trabalho propõe a integração de um *framework* de geração de casos de teste baseado em modelos de máquinas de estados UML com uma ferramenta comercial de modelagem. O *framework* foi proposto por Heckeler et al. em 2013, e gera casos de teste para teste de robustês de máquinas de estados UML modeladas utilizando o Enterprise Architect.

O trabalho do Heckeler utiliza arquivos exportados do Enterprise Architect (EA) para acessar as informações da máquina de estados modelada. A geração de casos de teste é composta por três passos: geração de casos de teste abstratos, geração de casos de teste executáveis e controle da execução dos casos de teste.

A geração dos casos de teste abstratos, chamados de *path lists*, é responsável por extrair todas as informações necessárias diretamente do modelo, gerar uma nova representação do sistema de transição conforme a representação do solucionador de satisfazibilidade booliana utilizado para resolver guardas de transição, e executar consultas sobre este sistema de transição para gerar todos os casos de teste abstratos a fim de satisfazer a cobertura desejada. Esses casos de teste abstratos são listas de transições que devem ser executadas na máquina de estados.

O segundo passo é responsável por transformar as *path lists* em casos de teste executáveis. A fim de gerar testes de robustez, este passo também estende as *path lists* com chamadas para todas as transições não definidas em cada estado, usando informações do modelo. Os casos de teste executáveis são baseados em suíte de teste CppUnit. Além de gerar os casos de teste, este passo também gera scripts de configuração para controlar o Gnu Debugger (GDB) durante a execução do teste. Esses scripts incluem comandos para iniciar e parar a gravação necessário para a execução reversa de transição, e são usados para ler as variáveis de codificação dos estados durante a execução.

O terceiro passo executa os casos de teste executáveis, usa o GDB para extrair os valores necessários para a verificação, e acelera a execução dos casos de teste utilizando o recurso de execução reversa do GDB. Quando um erro é detectado, este passo emite um *trace* que mostra as transições que foram chamadas, para permitir a reprodução posterior

do erro.

Este trabalho propõe uma solução integrada dentro do Enterprise Architect como um plugin, utilizando a interface de programação do EA, a fim de melhorar a usabilidade para o usuário final. Como um trabalho de integração inicial, este trabalho foca na integração do primeiro passo, a geração de casos de teste abstratos.

Para realizar essa integração inicial, este trabalho propõe um modelo em C# para máquinas de estados para armazenar todos os dados extraídos de EA, apresentado em um diagrama de classes, e uma arquitetura de software para ser utilizada na implementação do plugin, que pode ser facilmente estendida para incluir tanto novos técnicas de geração de casos de teste quanto coberturas do caso de teste. A validação do gerador de casos de teste integrado é feita utilizando as mesmas duas máquinas de estados utilizadas por Heckeler na proposta do *framework*, comparando-se as *path lists* geradas pela implementação anterior com as geradas pela solução integrada com ao EA. A primeira máquina de estados é a implementação de uma tabela *hash* composta por 3 estados e 11 transições. A segunda máquina de estado é a implementação de um controlador de semáforo que compreende 11 estados e 19 transições.

Keywords: UML, state machine, test case generation, Enterprise Architect.

1 INTRODUCTION

As the computer systems evolve, they become ever smaller, faster and more reliable, allowing them to substitute their hardware counterparts. Every year more software systems are embedded everywhere around us, from home appliances to cars, trains, and airplanes.

Testing is indispensable to assure software quality and reliability, therefore more than 50% of software developing resources are spent on testing (SABHARWAL; SIBAL; SHARMA, 2011). Even with this enormous amount of effort to ensure the software quality, some bugs are still able to find their way to the final products.

Not every embedded system has the same requirements concerning safety. If a home appliance suddenly stops working due to a software error it will not compromise the safety of anyone. An error on an airplane navigation system, otherwise, may cause a crash and injure all its occupants. For this reason critical systems must be certified.

To certify a software, the whole development process must follow certain guidelines, carefully reviewing all the manual steps of the development. Model Based Development (MBD) is a development process that can improve the certification process, reducing costs and schedule (MARCIL, 2011).

A great advantage of MBD is that many modeling tools provide automated code generation, and automated test case generation. Nevertheless, sometimes the automated tools do not fit completely into the company work flow. In these cases an extra integration tool is necessary.

This work proposes a software tool to integrate a MBD editor and a testing framework, to automate state machine testing. The tool is implemented as an extension for Enterprise Architect, a commercial MBD editor used in the industry, and integrates it with a testing framework proposed by Heckeler et al. in 2013 (HECKELER et al., 2013).

The integration was proposed during my exchange program in the Eberhard Karls University of Tübingen in Germany, in 2013, while I was working with the creators of the framework on another project related with UML state machines and Enterprise Archiect.

In the integrated solution the state machine information is retrieved from the model

through an API, and an output XML file is generated containing the state machine description, as well as a set of test cases. This XML file is used by the testing framework to generate executable test cases and validate the state machine implementation at runtime.

2 MODEL BASED TEST CASE GENERATION

This chapter covers the necessary background for the plugin implementation. Software testing is presented, followed by model based development. Then the Unified Modeling Language is presented, with a focus on state machine representations. Afterwards, some works on test case generation from UML state machines are discussed, and finally the framework used in the plugin is presented and discussed in detail.

2.1 Software Testing

Software testing is an indispensable technique applied during software systems development, since it detects errors during development that are then fixed before the software goes into production. Therefore testing may consume more than 50% of software development resources (SABHARWAL; SIBAL; SHARMA, 2011). Software testing may be done manually or automatically. While the setup and development of automatic tests adds an extra cost, it allows the software to be continually tested, decreases the risk of human induced errors during testing, and speeds up the detection of regressions, where regressions are new bugs accidentally introduced in existing areas of a system after changes or enhancements (LIU; CHEN; JIANG, 2013).

The testing process consists of three tasks: test case generation, test case execution, and test case evaluation. Test case generation consists of defining the data input for the system, the state of the system which this data should input, and the expected output obtained from the system (AGGARWAL; SABHARWAL, 2012).

Since test case generation is usually the most difficult task, it is important to find ways to automate test case generation for software systems (AGGARWAL; SABHARWAL, 2012). One of the ways to help this task is adopting a development process that supports abstracts representations of the software specificationsin a standardized way, like Model Based Development, which will be detailed on the next section.

2.2 Model Based Development

In the traditional software development process, the requirements are defined using natural language, manually translated into high level design, and finally manually coded into the final application. Afterwards, the validation and verification are made on the final product, both manually and using manually defined automated tests. The translation from high level requirements into code often requires many iterations between the design teams and the implementation team, due to unclear requirements and design documentation problems (KELEMENOVá et al., 2013). It is also important to notice that the coding process includes the risk of random human induced errors, and that since most of the development process is manual, it is labor intensive process, making any changes in the design very costly and time consuming (MARCIL, 2011).

The Model Based Design (MBD) is an alternative to the traditional software development approach. In this approach, the model itself works as the system specification (MARCIL, 2011). In the beginning of the development process, the model captures the high level requirements. This model is then refined through iteration and simulation, creating abstract representations of software aspects like modules behavior and relationships. The model simulation provides engineers with tools to experiment concepts in a higher abstraction level, by involving implementation specifics as late as possible. Also, by constantly simulating the model it is easier to ensure that the software requirements are met (MARCIL, 2011).

In software that requires a safety certification, the use of MBD has been associated with a potential effort saving of up to 55% (MARCIL, 2011). This does not necessarily reflect on cutting the project lenght of time by half, but it allows further refinements to the final product, resulting in a safer, more reliable and more robust final solution. Most of the effort saved in the project is associated with two automated steps in MBD: automated code generation, and automated test case generation (MARCIL, 2011).

The automated code generation reduces the risk of human induced errors during the code generation phase. Also, by being able to generate thousands of lines of code in the click of a button, system engineers are able to experiment with different concepts and architectures without spending too much time with implementation details. Even though automated code generation tools are available, it is not uncommon for the code to be manually generated from the models, for reasons as the used of software frameworks unsupported by the modeling tool. In these cases it is specially interesting to be able to automate test case generation, as this may catch human errors during coding. Also, it is important to notice that in safety-critical systems the dominant portion of the software costs are verification costs (BHATT et al., 2005), so automating test case generation also offers a competitive advantage.

Model Based Designs need to be described in a modeling language. There are many tools available for MBD, as well as many modeling languages (UML, 2014), (SIMULINK, 2014), (SCADE, 2014). A language that has taken a leading position in this area is the Unified Modeling Language (UML), that will be detailed on the following section.

2.3 Unified Modeling Language

The Unified Modeling Language is a standard set of modeling diagrams designed for specification, visualization and documentation of software systems. The standard version UML 2.0 will be used in this work, that comprises thirteen standard diagram types, divided into three categories: structure diagrams, behavior diagrams and interaction diagrams. The structure diagrams include the class diagram, object diagram, component diagram, composite structure diagram, package diagram, and deployment diagram. The behavior diagrams include the use case diagram, activity diagram, and state machine diagram. The interaction diagram include the sequence diagram, communication diagram, timing diagram, and interaction diagram. This work focuses primarily on state machine diagrams.

The UML defines two kinds of state machines, referred as Behavioral State Machines (BSM) and protocol state machines (PSM). Behavioral state machines are used to specify the behavior of model elements (e.g., class instances), while protocol state machines are used to show which operations of a model element may be called depending on its current state (OMG, UML 2.2 SUPERSTRUCTURE, 2009).

To study the test case generation from UML state machine diagrams, this work focuses both on behavioral and on protocol state machines (distinction between them will be made whenever pertinent). This topic was already explored on the literature under different aspects, and these related work will be explored on the ensuing section.

2.4 Related Work

Many studies (HECKELER et al., 2013), (SABHARWAL; SIBAL; SHARMA, 2011), (MOHANTY; ACHARYA; MOHAPATRA, 2011) have been performed focusing on different aspects of test case generation based on UML state machines, such as prioritization approach, test coverage criterion and test case generation technique. Aggarwal et al. proposes (AGGARWAL; SABHARWAL, 2012) a classification of UML state machine test case generation studies in three types, regarding their focus: path prioritization, test coverage and test case generation technique used.

2.4.1 Path Prioritization Techniques

Path prioritization approaches try to use different prioritization methods the of test cases during test case generation. These approaches aim to analyse the test cases and identify test cases that are more relevant than others, to execute them first.

Sabharwal et al. proposed (SABHARWAL; SIBAL; SHARMA, 2011) an approach that uses information flow, a stack based technique, and Genetic Algorithms (GA) to identify and prioritize critical path clusters. The state machine is transformed to a State Dependency Graph, then the stack based technique is used to assign a weight to each node. The weight is used to calculate the complexity of each node. Afterwards the GA is used to find paths and prioritize those with the higher complexity. However, this approach is not suitable for complex scenarios.

Mohanty et al. proposed (MOHANTY; ACHARYA; MOHAPATRA, 2011) a technique that prioritizes test cases to perform regression testing for component based systems. The objective of this work was to select a set of test cases that test the affected components from a large test suite to detect faults as soon as possible. The tests are selected to have a maximum number of state changes and a maximum number of database access made during the transitions.

Weissleder studied (WEISSLEDER, 2011) test goal prioritization using goals like Far Elements First/Last, High Condition Branching First/Last, Many Conditions First/Last, High Positive Assignment Ratio First/Last.

2.4.2 Coverage Criteria

On the coverage criteria aspect the studies focus on how to generate tests to ensure that a given coverage is achieved, i.e. all transitions present in the state machine must be executed at least once, or all logic predicates in transition guards must be evaluated true and false at least once.

Offutt et al. described techniques (OFFUTT; ABDURAZIK, 1999) for generating tests that cover one of the following criteria: transition predicates, transitions, pairs of transitions, and sequences of transitions. The tests are specified in terms of simple algebraic predicates.

Ferreira et al. presented (FERREIRA; FARIA; PAIVA, 2010) MoCAT, a coverage analysis tool, where the achieved coverage is visually represented. The tool simulates the execution of the UML state machine and use colors to represent the transitions covered. This tool uses action and guard conditions written in C#. Also, only transitions and state coverage are supported at the time of writing.

Briand et al. investigated (BRIAND; LABICHE; WANG, 2004) the cost effectiveness of tests from four different coverage criteria: All Transitions (AT), All Transition Pairs

(ATP), all paths in Transition Trees (TT), and Full Predicate (FP). The authors observed that AT is usually not sufficient to ensure an adequate level of fault detection, while ATP is, with a high increase in cost. TT does not prove to be cost effective. An FP has the same effectiveness of ATP, although it is more expensive.

2.4.3 Test Case Generation Technique

On the technique aspect the studies focus on which technique or algorithm is used for test case generation.

Doungsa-Ard et al. proposed (DOUNGSA-ARD et al., 2007) an approach using Genetic Algorithms. Initially a tool is used to generate sequences of triggers, these are used by another tool to execute the transitions, while a third tool records the transitions triggered and creates the transition path. The fitness is calculated based on the transition coverage, the chromosome is a sequence of triggers and two point crossover and random mutation are used. However, this approach suffers from the infeasible transition problem and the looping problem.

Shirole et al. also used GA (SHIROLE; SUTHAR; KUMAR, 2011) for automatic generation of test paths. In this approach the UML state machine is transformed into an extended finite state machine (EFSM), then the EFSM is transformed into an extended control flow graph. Based on this graph GA is used to generate test cases that satisfy a specified coverage. However, the authors considered only simple state machines.

Chevalley et al. proposed (CHEVALLEY; THEVENOD-FOSSE, 2001) a technique that uses statistical testing for test case generation, considering transition coverage as the criterion. The program uses two modules, a generator that is responsible for input data generation, and a collector that collects transition coverage measures. The measures are fed back into the generator in order to identify the least triggered transitions and generate further test if necessary.

Offutt et al. developed (OFFUTT; ABDURAZIK, 1999) an approach that generates test cases from UML state machine diagrams without transformation. The tool developed is integrated with the Rational Rose tool. Full predicate and transition pair coverage are achieved, however it does not achieve transition path coverage and generates many redundant test cases. Also transition guards are not handled.

This work is developed using a study that focus on the coverage criteria. The framework adopted in this project is detailed in the next section.

2.5 Framework

In 2013, during an exchange program in the Eberhard Karls University of Tübingen in Germany, I worked with Heckeler and Eichelberger on a project related to UML state

machines and Enterprise Architect. During this work I became aware of their framework for test case generation from UML state machines (HECKELER et al., 2013), as well as their interest in integrating it with EA.

This framework's main focus is on generating test cases for robustness testing, a kind of test that seeks to ensure the modules will behave as expected even outside their design domain. Another characteristic of this framework is that it presents mechanisms for test execution acceleration, using GNU debugger (GDB) reverse execution feature. Besides, GDB is also used during verification to compare the value of state encoding variable with the state expected, thus eliminating the need for code instrumentation.

Software modules are developed and tested within an application domain. Afterwards they are reused in other projects that may have an application domain different from the original ones. In these cases the modules might be subject to inputs that were not considered during their development and testing, and behave unexpectedly. The robustness tests raise the assurance level that the modules will behave as expected even outside their original domain.

The framework checks the module robustness adding calls to undefined transitions within a state. In these cases the state machine should remain in the same state it was before the undefined transition call, without any change to its internal variables. If a change is detected after one of these transitions an error is detected.

The framework uses a boolean satisfiability (SAT) solver to resolve paths where transition guards depend on the value of internal variales of the state machine, that depend on previous transition effects. The SAT solver used is Z3 (Z3, 2014), a theorem prover developed by Microsoft Research. Z3 is used through its Python interface, therefore a Python representation of the target state machine is generated in the form of a transition system.

The framework uses EA extensible markup language (XML) metadata interchange (XMI) files. Once exported from EA, these files contain all information regarding the target state machine model. This information is described in the next subsection.

2.5.1 State Machine Model

The state machine model elements used to generate the test cases are shown in Figure 2.1. The model must have the following elements: a UML state chart that describes the internal behavior of the Component Under Test (CUT) and a UML class that defines the public interface and the internal variables of the CUT. In order to provide realistic data input for tests the following elements must also be present in the model: a Test Data Generator (TDG) UML class that is used as parameter data for triggers, user-defined tags that provide code snippets for controlling the TDG inside test cases, and Z3 invariants

used to control the value ranges used during test data generation.

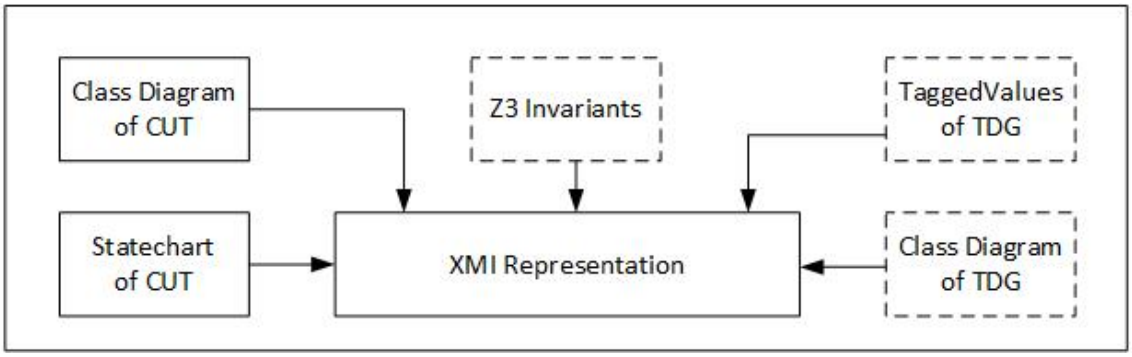

Figure 2.1: Model elements

The statechart is either a BSM or a PSM. This distinction is important because the test execution control task is able to accelerate the test cases of a BSM. This acceleration results from the fact that BSM transitions usually do not execute much computational operation, and may be easily reversed, as opposed to PSM transitions. In the PSM execution only invalid or blocking transitions are reversed.

Figure 2.2 shows a visual representation of a Statechart it consists of an initial transition, two states and a transition between them. The transition elements are also shown. The trigger is the event responsible for triggering the transition. Once the trigger is called, the guard expression is evaluated, if true the transition is executed otherwise the state machine remains in the same state. The effect is an operation that is executed once the transition is executed.

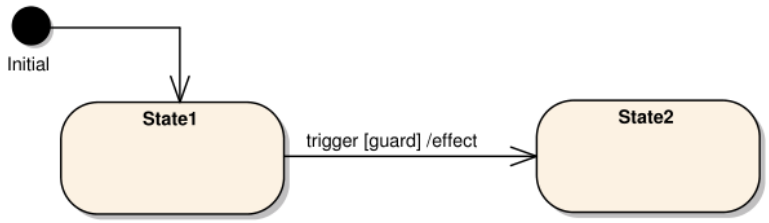

Figure 2.2: Visual representation of a state machine

2.5.2 Framework Workflow

Figure 2.3 presents the workflow of the framework with its three main tasks: abstract test case generation, executable test case generation, and test case execution and evaluation.

The generation of the abstract test cases, called path lists, is responsible for extracting all the necessary information directly from the model, generate a proper representation for

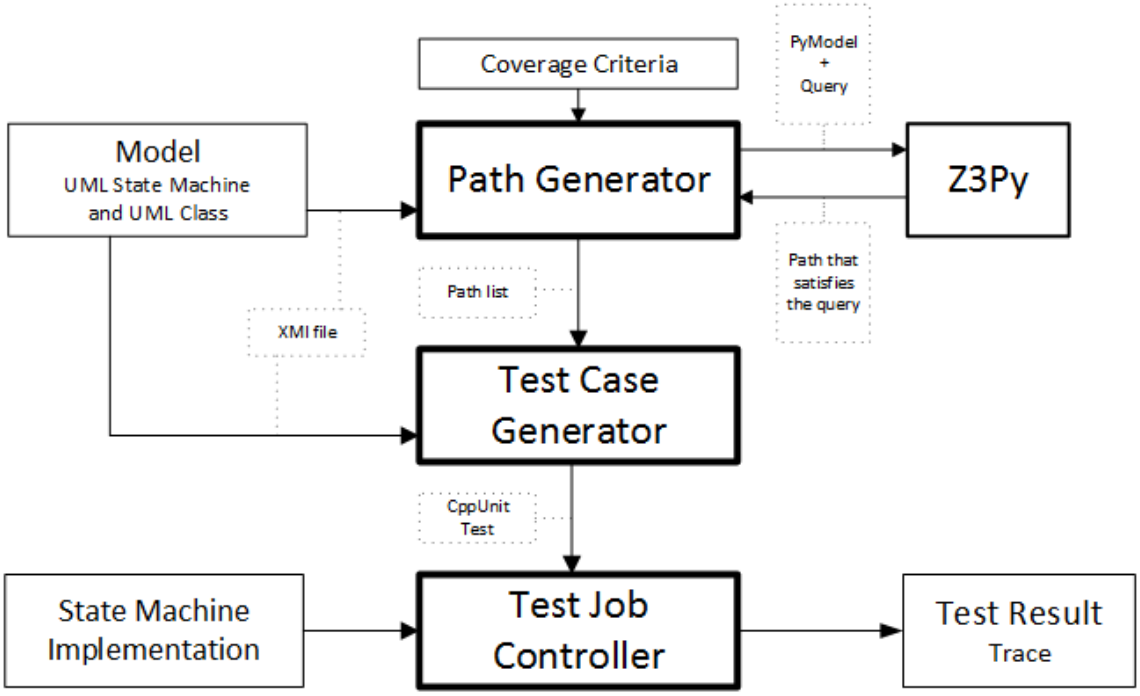

Figure 2.3: Testing framework workflow

the Z3 transition system and run queries on this transition system to generate all abstract test cases in order to satisfy the target coverage. These abstract test cases are lists of transitions that should be executed in the state machine.

The Z3 transition system is a description of the state machine in Z3Py syntax, used to access Z3 Python interface. Z3 is used as a boolean satisfiability (SAT) solver to resolve paths where the transition guards depend on the value of internal variables of the state machine, that depend on previous transition effects. In these cases in order to execute a transition it might be necessary to first execute several other transitions.

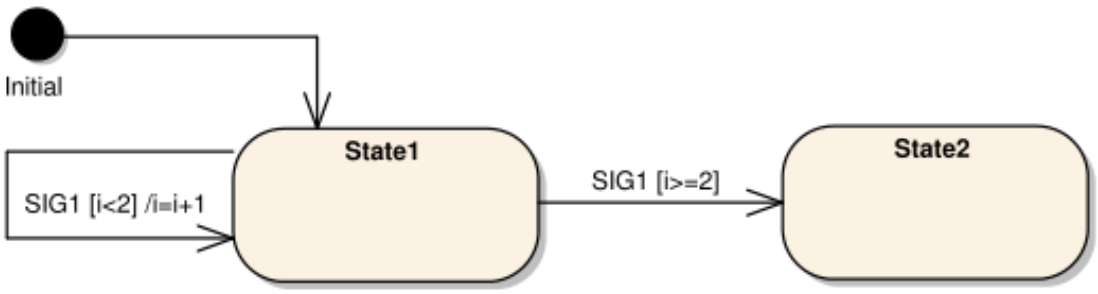

Figure 2.4: State machine example

Figure 2.4 presents an example of scenario where the SAT solver resolve a path with guards that depend on internal variables. In these state machine the initial value of i is zero, as an effect from the transition from the initial state to State1. If the objective is

to reach State2, the value of *i* must be incremented twice to satisfy the transition guard *[i>=2]*. Since it is an internal variable of the state machine the only way to increment it is executing the other transition. The SAT solver will solve this by executing the transition *SIG1[i<2]/i++* twice and proceed to execute the transition *SIG1[i>=2]*. Afterwards it will return the path list shown in Figure 2.5.

```
<path>
  <transref>
    <refid>0</refid>
    <blocked>False</blocked>
  </transref>
  <transref>
    <refid>1</refid>
    <blocked>False</blocked>
  </transref>
  <transref>
    <refid>1</refid>
    <blocked>False</blocked>
  </transref>
  <transref>
    <refid>2</refid>
    <blocked>False</blocked>
  </transref>
</path>
```

Figure 2.5: Pathlist example

The second task is responsible for transforming the path lists into executable test cases. In order to generate robustness tests, this task also extends the path lists with calls to all undefined transitions, using informations from the model. The executable test cases are based on CppUnit test suite. Besides generating the test cases this task also generate configuration scripts to control GDB during test execution. These scripts include commands for starting and stopping the recording needed for reverse transition execution, and are used to read the state encoding variables during runtime.

The third task runs the executable test cases, uses GDB to extract the values necessary for verification and to accelerate the test case execution using GDB reverse execution. When an error is detected, this task outputs a trace that shows the triggers call order to facilitate the error reproduction.

2.5.3 Coverage Criteria

Since covering all paths along the statechart would lead to an excessive and possibly infinite number of test cases, an adequate coverage criterion must be chosen. This is even more significant for robustness testing, since these also take not modeled behavior into account. For this framework approach, both structural and logical coverage criteria are used.

The structural approach focus on the state machine model structures, like states and transitions. In this framework two structural coverage criteria are used: All Transition Coverage (ATC) and Pairwise Transition Coverage (PTC). ATC requires that every transition in the state machine is executed at least once. This is extended to cover all undefined transitions outgoing from a state. PTC requires that every transition pair, a combination of an incoming transition and an outgoing transition from a state, to be executed at least once. This is extended to cover all combinations of an incoming transition of a state with all undefined outgoing transitions of the state.

The logical approach focus on transition guards, since these are logical expressions consisting of clauses connected by logical operators. This framework uses the following logical coverage techniques: Predicate Coverage (PC) and Combinatorial Coverage (CC). PC requires each predicate to be evaluated once to true and once to false. In CC the complete truth table of the predicate clauses must be tested.

The framework is able to deal with any combination of these logical and structural coverage criteria.

Next chapter will discuss the integration of this framework with Enterprise Architect, a commercial modeling tool.

3 TEST CASE GENERATOR

This chapter describes the work conducted to integrate the test case generation framework presented in Section 2.5 into the chosen modeling tool, Enterprise Architect (EA). The scope of this work is to integrate only the abstract test case generation task into the plugin. Other tasks are automatically launched once the test case generation is complete, giving the user the impression of a fully integrated environment.

The framework (see Section 2.5) accesses the model using Enterprise Architect (EA) exported XMI files, that represent an UML state machine diagram. This work proposes a C# plugin that integrates with EA using its automation Application Program Interface (API). While using the XMI file is acceptable during development and prototyping the test generation tool, the API integration is an important step towards the development of a final product, since integrating the test case generation with the EA interface makes it more user friendly.

The solution presented here is a first step towards full EA integration. In this solution the path list generator is integrated as an EA plugin while the rest of the process is still done by scripts, that still rely on the XMI file.

In order to implement this integration a C# state machine model is proposed and discussed, as well as the plugin structure. The plugin structure focuses on extensibility, making it simple to include a new test case generation technique into the plugin.

3.1 Implementation

EA has an API that enables the development of third-party plugins that run within it, and access model data from it. In this work this API is used by the test case generator to extract all necessary data from the model in order to generate test cases.

Initially the plugin structure will be presented and explained, then the state machine model structure will be presented and explained in detail.

3.1.1 Plugin Structure

Figure 3.1 shows the test generator plugin class structure.

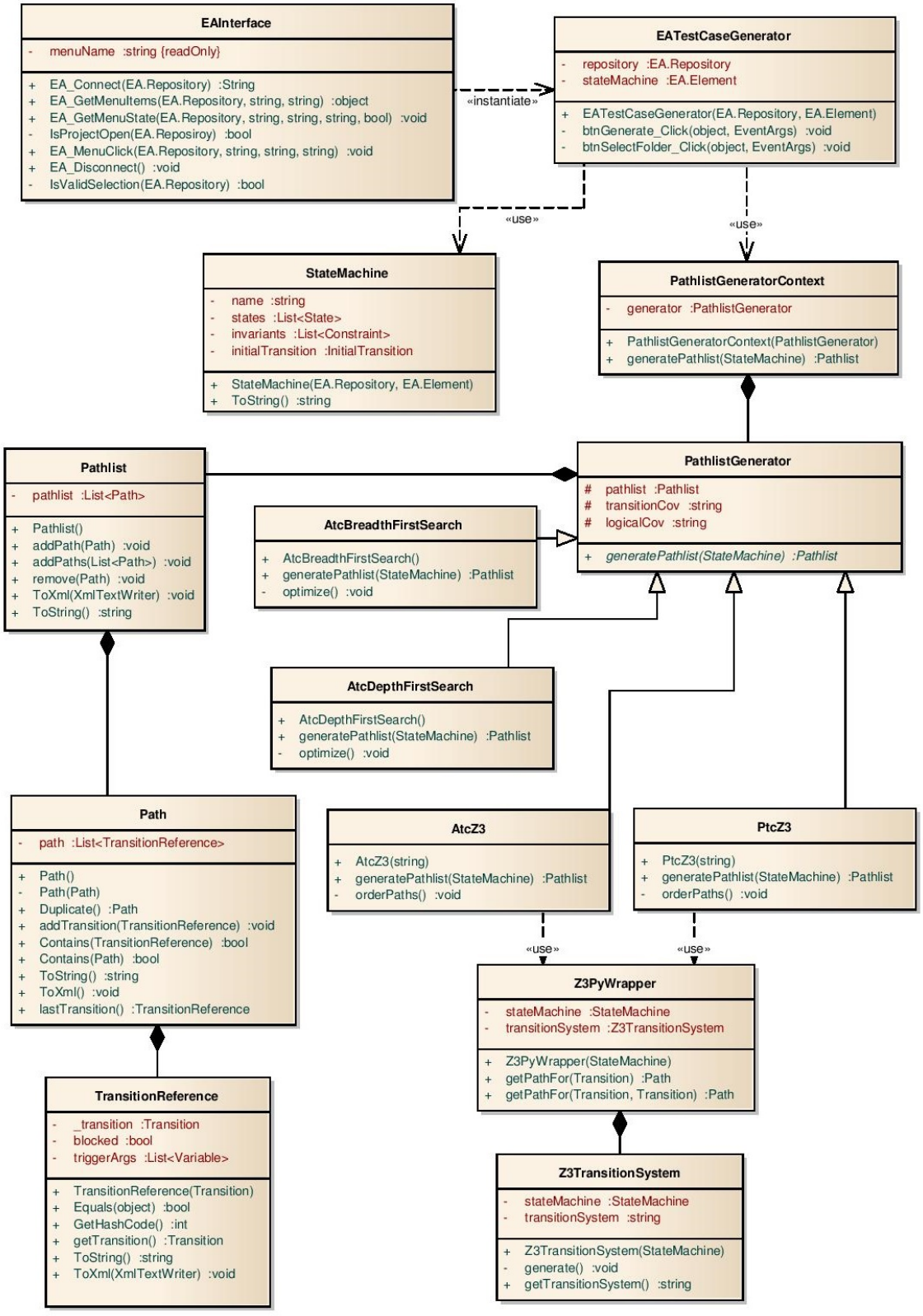

Figure 3.1: Test case generator plugin structure

EAInterface is the base interface class for EA plugins, through this class EA is able to check for the plugin presence, as well as start it from inside its interface. Section 3.4 will explain how the plugin is accessed within EA interface. Once the user starts the plugin, this class checks if it was called from a StateChart element, if this is true the EATestCaseGenerator is started, otherwise the user gets and error indication that it should be called from a StateChart.

The EATestCaseGenerator is the main plugin class. This class contains the methods that handle the user interaction with the plugin, therefore it is the class that instantiates the StateMachine model and the PathlistGenerator according to the options selected by the user. The constructor takes as parameters the EA repository and the EA element that represents the target state machine. The EA repository contains methods to retrieve any element within the EA model, this is useful for retrieving elements using their EA unique identifier, for example.

The StateMachine class is the main model class that represents the target state machine for which test cases will be generated. This class will be explained in details in Section 3.1.2.

The PathlistGenerator uses the strategy design pattern (GAMMA et al., 1995). This pattern allows the path list generation technique to be selected at runtime and improve the extensibility of the plugin, since it is possible to include a new path list generation technique simply by creating a new class that extends PathlistGenerator and including an option at EATestCaseGenerator to use it.

The Pathlist class represents one path list. It stores a list of paths and contains methods to add and remove paths from this list. It also contains a method used to generate a XML representation of the path list, used to generate the plugin output.

The Path class represents one path within the state machine. A path is a list of transitions, that correspond to an execution within the state machine. This class has methods to create a duplicate of an already existing path, add transitions to a path, and check whether a path contains a certain transition or another path within it. It also contain a method to generate a XML representation of a Path, that is used in the generation of the Pathlist XML.

The TransitionReference class contains information of a transition within a Path. Besides a reference to the Transition, this class also stores whether this transition is blocked or not and the list of arguments used in this transition triggers. This information is not used during pathlist generation, but are already included in the design to support future integration of the executable test case generator (see Section 2.5).

Both AtcBreadthFirstSearch and AtcDepthFirstSearch classes implement ATC. They use search algorithms to travel through the state machine transitions and generate a path

list with the minimum number of paths to cover all transitions from the state machine. However, these path list generation techniques do not take guard evaluation into account and cannot be directly used with the executable test case generator. They are implemented as examples of possible extensions to the path list generation plugin.

AtcZ3 and PtcZ3 classes are responsible for the generation of pathlists with ATC and PTC, respectively, using Z3. The constructor takes the logical coverage configuration as parameter, since is possible to generate ATC/PTC path lists with either PC or CC logic coverage. These classes use the Z3PyWrapper to generate the Z3Py transition system and query paths from it.

The Z3PyWrapper class uses the Z3TransitionSystem class to generate the Z3Py transition system representation of the state machine, and a has method to query this transition system for path that reach a transition or path that reach a transition pair. It translates the target transition, or transition-pair, into a valid Z3Py query, runs it in a Python interpreter, retrieve the output and generate a Path object from it.

The Z3TransitionSystem class is responsible for generating a Z3Py transition system representation from the target StateMahine model. This representation is in the form of a string (see Section 3.2).

3.1.2 Data structure

In order to integrate the framework into a plugin it is necessary to model all the elements of the state machines within the plugin. Figure 3.2 shows the class structure to model the target state machine.

The StateMachine is the main class of the model. This class is responsible for the instantiation and management of all other elements of the state machine, as well as the instantiation of state, trigger and transition factories. The constructor takes as parameters the EA repository and the EA element that corresponds to the target state machine. Since EA transitions only reference the triggers by name, the StateMachine constructor iterates through all elements within the EA element corresponding to the state machine, finds the trigger elements, and adds them to the trigger factory. Afterwards the constructor instantiates all states, identifies the initial state, instantiates the global constraints of the state machine, the initial values of internal variables, and the initial transition.

In order to keep compatibility with the models previously used by the framework, the global constraints, the initial values and the internal variable names are read from the note field in the initial transition, as shown in Figure 3.3. These parameters are separated by a line of dashes.

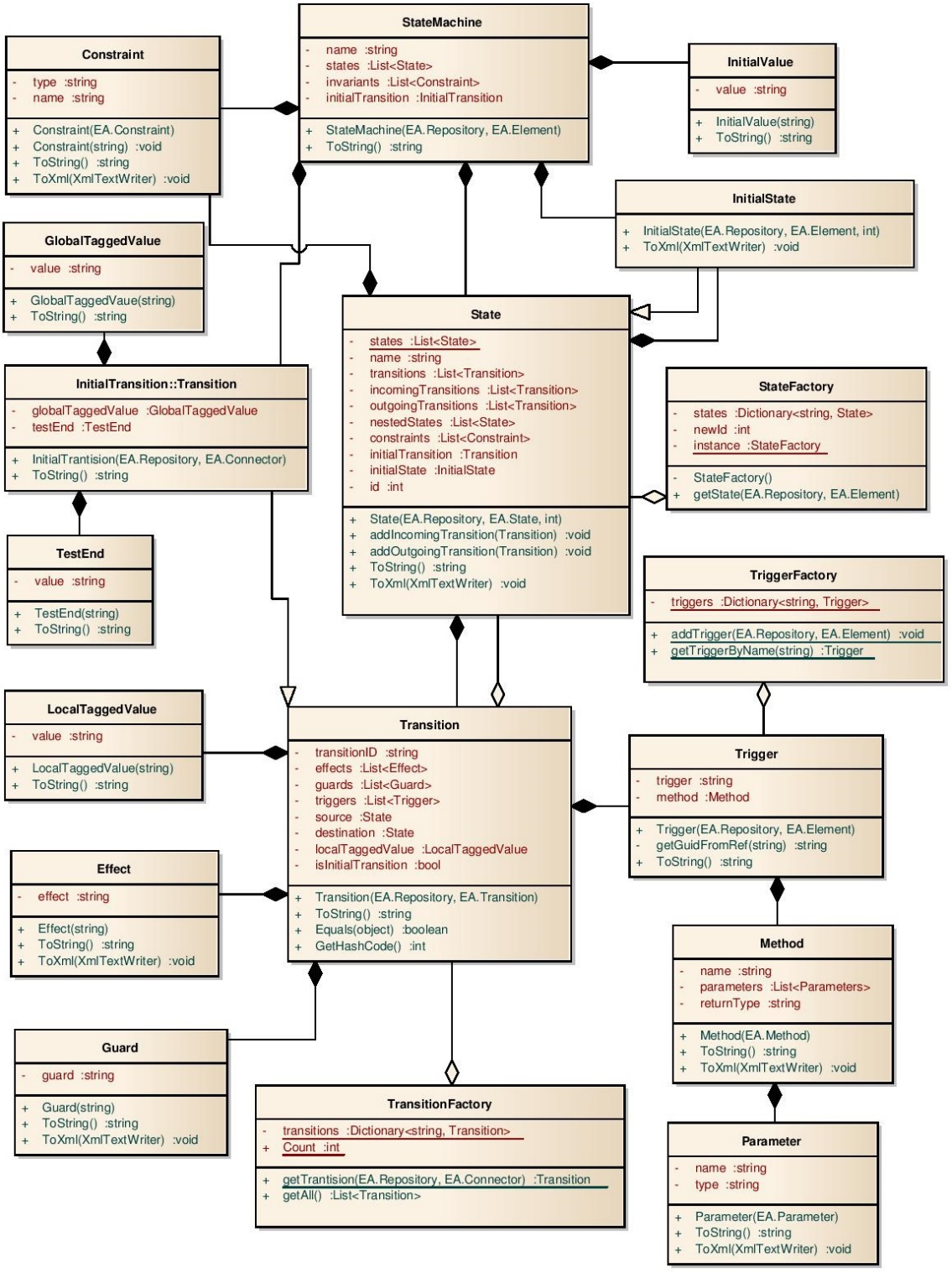

Figure 3.2: Class diagram of state machine model

The StateMachine InitialTransition may contain tagged values. These are Global-TaggedValues and TestEnd objects, they consist of C++ commands used to control de TDG. During the executable test case generation these commands are included in the test case. In the implementation of the path list generator these tagged values are not used,

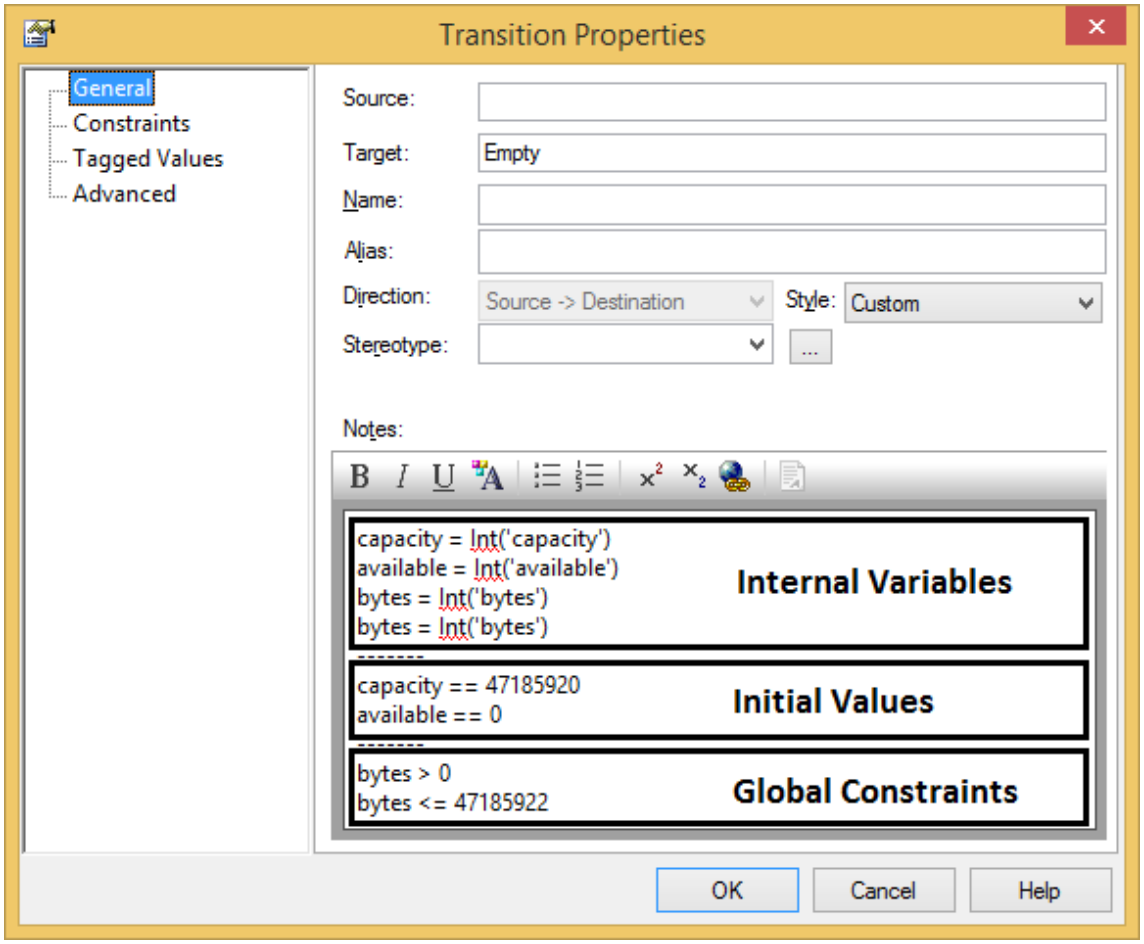

Figure 3.3: Internal Variables, Inital Values and Global Constraints specification

nevertheless they are already included in the plugin model to facilitate future extensions.

The StateFactory class is a singleton (GAMMA et al., 1995) responsible for managing state objects. The management of states is important to ensure that multiple references to a single EA State in the EA model will correspond to multiple references to a single State object in the plugin model. This class is able to find State objects based on their EA unique id. Besides, this class is also responsible for giving a unique integer id for each state that is used afterwards in the Z3 transition system representation.

The State class has a one-to-one correspondence with states in the state machine model. It contains the state constraints, the state transitions (both incoming and outgoing), and the states nested within the state. In the case where nested states are present, the initial nested state and the initial transition from it are also stored.

The TransitionFactory class is also a singleton, responsible for managing transition objects. It works in an analogous way to the StateFactory, mapping a unique EA identifier to a Transition object, and giving a unique identifier for each transition that is later used in the Z3 transition system. It also provides a method to retrieve a list of all Transitions of the state machine, that is used during the Z3 transition system generation.

The Transition class also has a one-to-one correspondence with transitions in the state machine diagram. It contains two references to States, one to the state from which the transition is executed, called source state, and another to the state reached once the transition is executed, called destination state. It also contains a list of transition triggers, that represent all events capable of triggering this transition, a list of guards, that represent all guard expressions that must be evaluated before the transition is executed, and a list of effects, that are actions to be executed once the transition is executed. It also may contain a local tagged value, that is used to control the TDG. This tagged value is also not used by the path list generator, but is included in the model for future use.

The TriggerFactory class is also a singleton, responsible for managing trigger objects. This class maps the trigger name to a Trigger object. The trigger name is used instead of the unique EA identifier because the EA transition object does not store a reference to the trigger object, it stores only the trigger name. This results in the limitation that the trigger name must be unique within the state machine. It would be possible to overcome this limitation using Structured Query Language (SQL) queries in the EA repository to retrieve the transition trigger object directly, but a restriction was imposed on the use of SQL queries.

The restriction on the use of SQL queries on the EA repository comes from the fact that EA models may be stored on a central EA server, and at the time of implementation there was not enough information about the impact of an SQL queries in this configuration.

The Trigger class represents EA trigger elements, each trigger has a name and might be of one of two kinds: signal or call. Signal triggers are associated to events received by the state machine, and are usually used in BSM implementations. In contrast, call triggers are associated to a method call from a class, and are usually used in PSM implementations. A Trigger object corresponding to a signal trigger will contain only the trigger name. Meanwhile, Trigger objects corresponding to a call trigger will contain a reference to the corresponding Method.

The Method object stores the method name, a list of Parameters and the method return type. Each Parameter has a name and an associated type.

3.2 Transition System

As explained in Section 2.5.2, the Z3 transition system is used to represent the state machine within Z3Py and enable the plugin to query for paths that contain a certain transition or transition pair. In order to explain the transition system structure, an example of state machine is shown in Figure 3.4.

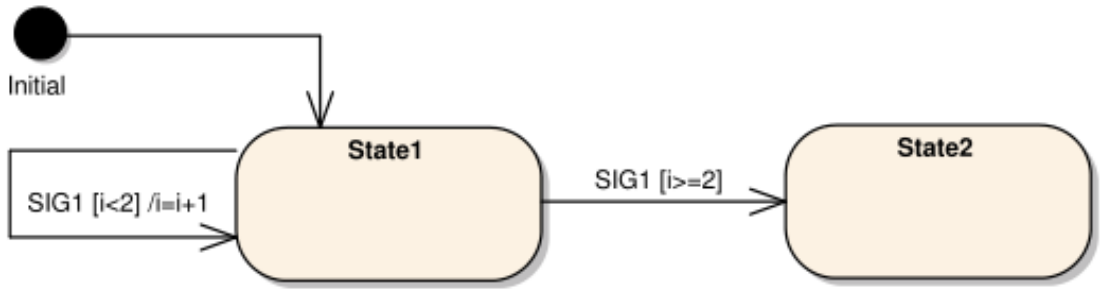

Figure 3.4: State machine example

Each transition in Z3Py transition system consists of two lists, the first is a list of constraints related to the transition and the second is a list of effects caused by this transition. The transition constraints hold the following information: source state, global invariants (when present) and transition guards (when present). The state effects hold the following information: target state, transition identifier, temporary variables from trigger parameters (when present) and transition effects. Figure 3.5 shows the Z3Py transition system generated from the state machine from Figure 3.4.

```
1  t0={"constraint":And(s==Initial),
2  "action":[State1, IntVal(0), i]}
3  t1={"constraint":And(s==State1, i<2),
4  "action":[State1, IntVal(1), i+1]}
5  t2={"constraint":And(s==State1, i>=2),
6  "action":[State2, IntVal(2), i]}
```

Figure 3.5: Transition system example

Once the Z3 transition system is generated the plugin is ready to query for paths. Figure 3.6 shows an example of a query for a path that executed transition t2 from previous example. The plugin will generate as many of these queries as there are transitions or transition pairs in the state machine. Once all paths are generated the path list is ordered, according to the state machine type. This distinction is necessary because non-blocking transitions in PSM are not reversed, therefore each PSM path can have only one non-blocking transition. BSM path may have more than one non-blocking transition per path (see Section 2.5.1).

```
1  ptr.query(Tran==2)
```

Figure 3.6: Transition system query example

3.3 Output

Since the plugin currently only integrates the path list generation, the output is a XML file containing a path list. This XML file also contains a representation of the state machine, that may be used by the executable test case generator in place of the EA XMI model file, to generate the test cases including calls for undefined transitions. The output for file for state machine from Figure 3.4 is shown in Figure 3.7.

```
<transitionsystem>
  <initialstate>
    <id>1</id>
    <name>Initial</name>
  </initialstate>
  <state>
    <id>0</id>
    <name>State1</name>
  </state>
  <state>
    <id>2</id>
    <name>State2</name>
  </state>
  <transition>
    <id>0</id>
    <name />
    <source stateref="1" />
    <target stateref="0" />
  </transition>
  <transition>
    <id>1</id>
    <name />
    <trigger name="SIG1" />
    <guard>i&lt;2</guard>
    <effect>i+1</effect>
    <source stateref="0" />
    <target stateref="0" />
  </transition>
  <transition>
    <id>2</id>
    <name />
    <trigger name="SIG1" />
    <guard>i&gt;=2</guard>
    <source stateref="0" />
    <target stateref="2" />
  </transition>
</transitionsystem>
<pathlist coverage="ATC" predicates="PC">
  <path>
    <transref>
      <refid>0</refid>
      <blocked>False</blocked>
    </transref>
    <transref>
      <refid>1</refid>
      <blocked>False</blocked>
    </transref>
    <transref>
      <refid>1</refid>
      <blocked>False</blocked>
    </transref>
    <transref>
      <refid>2</refid>
      <blocked>False</blocked>
    </transref>
  </path>
</pathlist>
```

Figure 3.7: Output XML example

3.4 User Interface

The plugin access is through EA interface. The user should have a project loaded with a StateChart diagram in it. The user may launch the plugin by right-clicking the StateChart diagram within the project browser, selecting Extensions, EA TestCaseGenerator, as shown in Figure 3.8.

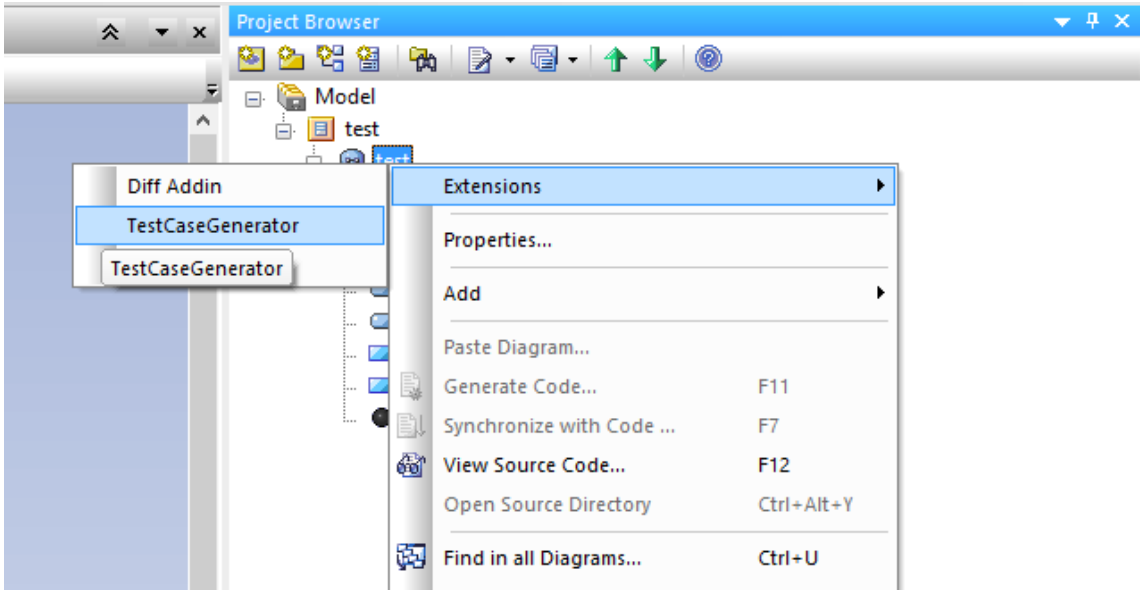

Figure 3.8: Plugin access within Enterprise Architect

The plugin user interface is composed of two tabs, as shown in Figure 3.9. Generation tab allows the user to select the output folder where the path list generated will be stored and start the generation process. Algorithm tab allows the user to select which technique will be used in the path list generation. Once Z3 is selected as the technique, the Z3 specific configurations are unlocked.

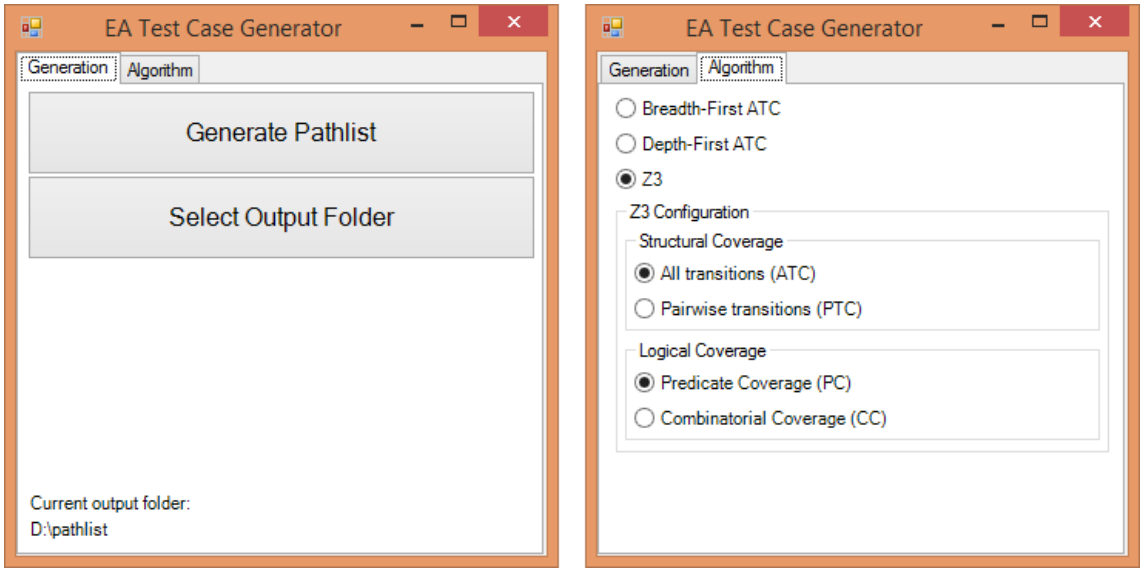

Figure 3.9: Plugin user interface

3.5 Self Installer

The plugin is deployed in the form of a DLL file. In order to facilitate the deployment of the plugin, a Windows installer is available. This installer is generated using Wix Toolset (WIX TOOLSET, 2014), and is responsible for extracting the DLL file into the correct folder and creating the necessary entries at Windows Registry, so that EA can access the plugin.

3.6 Validation

Since there is a previous implementation of the path list generator, not integrated with EA, it is used to validate the integrated version of this task. The validation was done by comparing the output path lists of the plugin with the output from the previously implemented path list generator, for the same state machines. Since the output path lists consist of XML files, this comparison was done using a text diff tool. The state machines used for the validation were the same used by Heckeler et al. in their experiments (HECKELER et al., 2013), a hash table implementation and a traffic light controller.

The hash table implementation is a state machine that comprises 3 states and 11 transitions, while the traffic light controller is a state machine composed by 11 states and 19 transitions.

4 CONCLUSION AND FUTURE WORK

This work used a combination of techniques and principles acquired during the course of Computer Engineering such as software modeling techniques and programming.

The work presented in this report is the initial step to have a fully integrated UML state machine testing solution within Enterprise Architect. At this point, the plugin serves as an interface between the user and the framework, improving the user experience while using the test generation framework.

In order to implement this plugin it was necessary to model the state machine within it, as well as to implement the path list generation techniques supported by it. In order to validate this work the output from the plugin was compared to the output of the module previously implemented.

Future work should implement the other two tasks, executable test case generation and test control, within the EA plugin to have a fully integrated state machine testing solution. Also, it would be valuable to replace the Z3Py interface with the Z3 C++ API (Z3 C++ API, 2014), this would increase the integration level between the plugin and the SAT solver. Yet another improvement in the plugin would be to study the impact of SQL queries in the EA repository and possibly use them to query for objects when necessary, like trigger references.

REFERENCES

AGGARWAL, M.; SABHARWAL, S. Test Case Generation from UML State Machine Diagram: a survey. In: COMPUTER AND COMMUNICATION TECHNOLOGY (ICCCT), 2012 THIRD INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2012. p.133–140.

BHATT, D.; HALL, B.; DAJANI-BROWN, S.; HICKMAN, S.; PAULITSCH, M. Model-based development and the implications to design assurance and certification. In: DIGITAL AVIONICS SYSTEMS CONFERENCE, 2005. DASC 2005. THE 24TH. **Proceedings...** [S.l.: s.n.], 2005. v.2, p.13 pp. Vol. 2–.

BRIAND, L.; LABICHE, Y.; WANG, Y. Using simulation to empirically investigate test coverage criteria based on statechart. In: SOFTWARE ENGINEERING, 2004. ICSE 2004. PROCEEDINGS. 26TH INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2004. p.86–95.

CHEVALLEY, P.; THEVENOD-FOSSE, P. Automated generation of statistical test cases from UML state diagrams. In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 2001. COMPSAC 2001. 25TH ANNUAL INTERNATIONAL. **Proceedings...** [S.l.: s.n.], 2001. p.205–214.

DOUNGSA-ARD, C.; DAHAL, K.; HOSSAIN, A.; SUWANNASART, T. Test Data Generation from UML State Machine Diagrams using GAs. In: SOFTWARE ENGINEERING ADVANCES, 2007. ICSEA 2007. INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2007. p.47–47.

FERREIRA, R.; FARIA, J.; PAIVA, A. Test Coverage Analysis of UML State Machines. In: SOFTWARE TESTING, VERIFICATION, AND VALIDATION WORKSHOPS (ICSTW), 2010 THIRD INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2010. p.284–289.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns**: elements

of reusable object-oriented software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

HECKELER, P.; EICHELBERGER, H.; KROPF, T.; RUF, J.; HUSTER, S.; BURG, S.; ROSENSTIEL, W.; SCHLICH, B. Accelerated Model-based Robustness Testing of State Machine Implementations. In: SPECIAL INTEREST GROUP ON APPLIED COMPUTING (SIGAPP) APPLIED COMPUTING REVIEW, New York, NY, USA. **Proceedings...** ACM, 2013. v.13, n.3, p.50–67.

KELEMENOVá, T.; KELEMEN, M.; MIKOVá, L.; MAXIM, V.; PRADA, E.; LIPTáK, T.; MENDA, F. Model Based Design and HIL Simulations. **American Journal of Mechanical Engineering**, [S.l.], v.1, n.7, p.276–281, 2013.

LIU, Z.; CHEN, Q.; JIANG, X. A Maintainability Spreadsheet-Driven Regression Test Automation Framework. In: COMPUTATIONAL SCIENCE AND ENGINEERING (CSE), 2013 IEEE 16TH INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2013. p.1181–1184.

MARCIL, L. MBD and code generation: a cost-effective way to speed up HMI certification. In: DIGITAL AVIONICS SYSTEMS CONFERENCE (DASC), 2011 IEEE/AIAA 30TH. **Proceedings...** [S.l.: s.n.], 2011. p.8B1–1–8B1–10.

MOHANTY, S.; ACHARYA, A.; MOHAPATRA, D. A model based prioritization technique for component based software retesting using UML state chart diagram. In: ELECTRONICS COMPUTER TECHNOLOGY (ICECT), 2011 3RD INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2011. v.2, p.364–368.

OFFUTT, J.; ABDURAZIK, A. Generating tests from UML specifications. In: UML-99, LNCS VOL. 1723. **Proceedings...** [S.l.: s.n.], 1999. p.416–429.

OMG, UML 2.2 Superstructure. Disponível em: <http://doc.omg.org/formal/2009-02-02.pdf >.

SABHARWAL, S.; SIBAL, R.; SHARMA, C. Applying genetic algorithm for prioritization of test case scenarios derived from UML diagrams. In: INTERNATIONAL JOURNAL OF COMPUTER SCIENCE ISSUES. **Proceedings...** [S.l.: s.n.], 2011. v.8, n.2, p.433–444.

SCADE. Disponível em: <http://www.esterel-technologies.com/products/scade-suite/ >.

SHIROLE, M.; SUTHAR, A.; KUMAR, R. Generation of improved test cases from UML state diagram using genetic algorithm. In: INDIA SOFTWARE ENGINEERING CONFERENCE. **Proceedings...** ACM, 2011. p.125–134.

SIMULINK. Disponível em: <http://www.mathworks.com/products/simulink/ >.

UML. Disponível em: <http://www.uml.org/ >.

WEISSLEDER, S. Towards Impact Analysis of Test Goal Prioritization on the Efficient Execution of Automatically Generated Test Suites Based on State Machines. In: QUALITY SOFTWARE (QSIC), 2011 11TH INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2011. p.150–155.

WIX Toolset. Disponível em: <http://wixtoolset.org/ >.

Z3 C++ API. Disponível em: <http://research.microsoft.com/en-us/um/redmond/projects/z3/group__cppapi.html >.

Z3. Disponível em: <http://z3.codeplex.com/ >.