Department of Computer and Information Science

Final Thesis

SMT Aided Test Case Generation For Constrained Feature Models

by

Paul Borek

LIU-IDA/LITH-EX-A–14/053–SE

2014-12-15

Linköping University
INSTITUTE OF TECHNOLOGY

Linköping University
SE-581 83 Linköping, Sweden

Linköping University
581 83 Linköping

University of Linköping
Department of Computer and Information Science

Final Thesis

SMT Aided Test Case Generation For Constrained Feature Models

by

Paul Borek

LIU-IDA/LITH-EX-A–14/053–SE

2014-12-15

Supervisors: Ahmed Rezine (Linköping University),
Johan Moe (Ericsson)

Examiner: Kristian Sandahl (Linköping University)

Abstract

With the development of highly configurable and large software, a new challenge has to be addressed, when it comes to software testing. While traditional testing approaches might still apply and succeed in achieving a better quality of service, the high degree of customizable parts of such a system implies the mentioned testing activities on different configurations. If a formal notion is used to express the allowed configurations of a system, one might think of generating such configurations in an automated fashion. However, if there are constraints involved, traditional model-based test-case generation might cause problems to achieve a desired coherency. An idea is, to use those constraints to generate test-cases and to achieve coherency at the same time. SAT modulo theories has been an emerging field in current theoretical computer science and developed decision procedures to treat various theoretical fragments in a specific manner. The goal of this thesis is, to look at a translation mechanism from an expression language for constraints into SAT modulo theories and involve this technique into a test-case generation process. Furthermore, the balance between the generation of coherent test-cases as well as the problem-specific purposes of such test-cases is investigated.

Acknowledgments

There are several people who contributed to the success of this project.

First of all my supervisor at Ericsson AB, Johan Moe who laid the foundations of the thesis and guided the project in the firm.

Also, the “Roffe” team in Ericsson AB, which contributed in several social as well as topic-related aspects. Primarily Bengt Carlsson, who gave all the needed technical information and advice.

Special thanks goes to Patric Wernqvist, project manager of Ericsson AB, who gave me a deeper insight of the project context.

Finally, my LiU supervisor, Ahmed Rezine, at Linköping University who had the theoretical foundation and access to this topic and my examiner, Kristian Sandahl, who helped me with literature search and gave advice in the working progress.

Last but not least my family and friends who supported me during the time at Ericsson AB and while writing the thesis.

Table of Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The increasing amount of software needed on the market led to high demands in terms of both reliability as well as functionality of the end-product. Especially sophisticated ways of testing software are a desired method to ensure the quality of service of the resulting software. Challenges arise if software gets highly configurable. Describing the possible configuration of a software is fundamental in early stages of the project, since those descriptions are then used in different areas of software development. In the mobile and fixed networking business, Ericsson AB uses the so-called *Ericsson Common Information Model (ECIM)* to describe various software parts and their interaction. This model does not only illustrate, how the software can be configured by the user, but moreover provides a guideline for the actual development of the software and describes the system as a whole. The requirements of testing activities on such systems change, since traditional testing approaches are not enough to test software under different configurations. The reason for that is, that one test-case might pass on one configuration while it fails on another one [9]. Further, the software part which allows the user to enter the configuration, needs to be tested in order to meet the requirements given by the ECIM model as well as to be fail-safe.

A further challenge arises if the formerly mentioned ECIM involves *constraints*. Constraints are rules which describe allowed or illegal combinations of software components in one configuration. Syntax and semantics of the constraints depend on the constraint language and also on the demands of the end-products, but always affects the interaction between the running software components. However, those constraints makes it harder to generate configurations, i.e., test-cases in an automated fashion.

1.1 Project Context

The thesis project was elaborated and developed at Ericsson AB Linköping in the context of a bigger software project, which is currently developed in Linköping, Stockholm and Anyang (South Korea). Approximately 200 developers are involved at the site in Linköping.

The model is the theoretical foundation of the software and used by all the teams involved in this project. It was developed by a specific modeling group located in Stockholm. The model is extensible to several needs for the current product and will also be the foundation of future projects.

1.2 Problem Description

The foundations of the problem are configuration descriptions which are following a UML-like syntax. They describe the allowed configurations of a running system, and are also needed for other aspects in the project development. The configuration follows a tree structure, where nodes are classes and each class contains attributes. The model itself is based on an interface level for the resulting configuration, whereas the commands to insert the configuration into the system have to respect this description. Parent-child relations define the structure and contain cardinalities between the different nodes to express the dimensions of the resulting configuration. Further, dependencies are involved, which validate the entered configuration.

In the previously mentioned project, the configuration, i.e., one instance of such a model, can be inserted by the user. After this process, the software behaves in a certain way. Having a set of configurations implies having test-cases for the *configuration manager*, the software part which accepts the commands to configure the software.

Recalling the original definition[1] of a test-case, implies the crucial task of a test-case in the software development process:

> A test-case is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly.

[1] `http://softwaretestingfundamentals.com/test-case/`

> The process of developing test-cases can also help find problems in the requirements or design of an application.

In this thesis, test-cases ensure the quality of service of the configuration manager. So far, this activity has been performed sparsely in a hard coded automatism, which is time-consuming and makes it hard to cover all variations.

From a theoretical point of view it is nearly impossible to achieve satisfying results of configurations by hand. Especially the number of variables, the structure how different classes can be instantiated as well as the constraints imply a high degree of variety of configurations. Other related requirements include the semantic meaning, which determine the quality or the value of a test-case and are also fundamental in a test-case generation process.

This led to the need of an approach to generate configurations (test-cases) in an automated fashion. The generated test-cases have two purposes:

1. To test the *configuration manager*, i.e., the software part which receives the instructions to configure the system.

2. To have multiple *scenarios* for other software-testing activities.

Both purposes have somewhat different requirements. While testing the configuration manager also involves marginal values for attributes and especially negative test-cases, for scenarios used by other testing activities only valid, i.e., positive test-cases are interesting.

We can already identify different crucial properties which are necessary in order to derive a satisfying set of test-cases from a ECIM:

- **Constraint coherency:** test-cases should always be coherent with respect to the existing constraints of the model.

- **Structural coherency:** test-cases should always follow the structure given by the ECIM. By *structure* we mean parent-child relations as well as cardinalities. This property defines both the number of test-cases as well as the quality of each test-case and the quality of the test-suite as a whole, respectively.

- **Attribute value coherency:** test-cases should follow a desired distribution of attribute values. In other words, the values of the attributes should also follow certain combinatorial properties.

- **User-specified coherency:** test-cases should fulfill the needs of the tester. This requirement is crucial. It defines the quality of both the test-case as well as the created set of test-cases.

Important to note is, that we are rather interested in a small set of coherent test-cases instead of a big number of test-cases with no or low assurance about coherency.

1.3 Goal and Scope of this Thesis

As seen before, testing configurations by hand is a cumbersome task. Thus, the primary goal of this thesis is, to use the ECIM models to generate configurations in an automated fashion.

To achieve this, the functionality of the mentioned models is explained first on a small example. After a thorough study of the example and the definition of the different features, they will be linked to the existing scientific context.

Another important contribution of this work is the study of the used constraints. Therefore, the previously mentioned example will be extended by the constraints and the used *constraint language* will be illustrated. After that, the discussion about the constraints will be linked to the theoretical model, which will be later used in the practical work.

The final theoretical considerations are to create test-cases from the previously introduced concepts. Thus, the sample model will be used to explain how the process works.

Finally, the practical part of this work is the implementation of the gathered approaches in terms of an executable tool.
Both the theoretical as well as the practical work should contribute to answer the following three questions:

1. How can we use a ECIM model to derive configurations of a software product in an automated fashion? Does the ECIM model reveal enough information to achieve this task?
2. How can we achieve both the coherency of the configurations as well as appropriate values in terms of usability for software-testing?

3. Which impact do the mentioned goals have on the performance of the test-case generation process?

1.4 Structure of the Document

As mentioned before, the overall report structure is inspired by an example of an ECIM model. This example will be explained thoroughly in Chapter 2. Then, dependencies which are preventing certain configurations will be explained on the same example in Chapter 3. The process how to generate such configurations from the example model in an automated fashion will be introduced in Chapter 4, which will be continued on the actual implementation of the system in Chapter 5. After a short evaluation of the work in Chapter 6 as well as a following discussion in Chapter 7, the work will be concluded in Chapter 8.

Chapter 2

The Ericsson Common Information Model (ECIM)

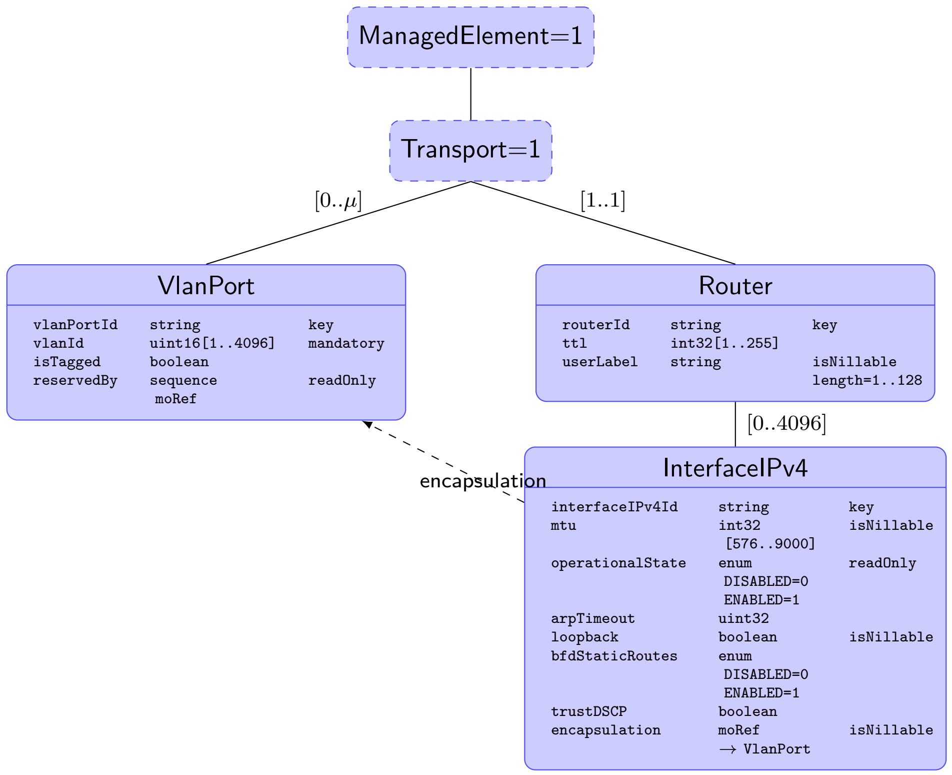
The purpose of the ECIM is primarily the support of operations and maintenance on a managed object. Such a managed object is a conceptual view of a resource such as a network component, a host system or an application. Thus, it does not only mark the boundaries of the allowed descriptions on the running software, but moreover the specification for different components of the software.

This means, that development teams use the models to implement the system and to have a static specification of the system. However, since the main purpose of this work is the automated generation of configurations, we will use them exclusively for this purpose. Configurations are usually inserted into the *configuration manager*, the part of the software which accepts the configuration commands and writes the verified configuration to the disk. This is usually done by the operator, i.e., user at the customer.

The following sections of this chapter will introduce the concepts of the ECIM specification on a small example and refer to the overall functionality, which is taken from an internal document, namely the ECIM meta-model specification.

2.1 Example Model

Figure 2.1 contains a minimal example of an ECIM. It consists of nodes and solid edges, which together form a tree. The nodes are called *classes* and contain *attributes*, whereas the solid edges are called *parent-child relations*. We will introduce these concepts in the sequel.



ManagedElement=1
Transport=1
[0..μ]
[1..1]
VlanPort
vlanPortId string key
vlanId uint16[1..4096] mandatory
isTagged boolean
reservedBy sequence moRef readOnly
Router
routerId string key
ttl int32[1..255]
userLabel string isNillable length=1..128
[0..4096]
encapsulation
InterfaceIPv4
interfaceIPv4Id string key
mtu int32 [576..9000] isNillable
operationalState enum DISABLED=0 ENABLED=1 readOnly
arpTimeout uint32
loopback boolean isNillable
bfdStaticRoutes enum DISABLED=0 ENABLED=1
trustDSCP boolean
encapsulation moRef → VlanPort isNillable

Figure 2.1: Example of a ECIM model.

Classes, parent-child relations and the used data-types for the attributes are usually embedded into a *managed information model (MIM)*, which technically acts as a name-space and is used to differentiate between fragments of the overall model. It further allows to have one class-name in several contexts. In this example we assume that all concepts belong to the same MIM for simplicity. The models are provided in *extensible markup language (XML)*, where one XML-file usually contains one MIM.

2.2 Concepts

We will now explain the most important concepts by referring to the example in Figure 2.1.

2.2.1 Classes

The main concept of an ECIM model is a *managed object class (MOC)* or simply *class*. Each class is represented by a node in Figure 2.1. The term "class" was intentionally chosen, because it is an interface-level description of the resulting configuration object, i.e., similar to object oriented programming, a resulting configuration from this model contains instances of the defined classes. An instance is called *Managed Object (MO)* in the meta-model specification. We will use the terms MO and instance interchangeably.

The classes ManagedElement and Transport have a dashed border in the example model. This means that the classes are `systemCreated`. In other words, the system takes care of the creation of the instances and excludes the user from inserting, modifying or deleting such instances.

Classes are connected by solid edges, called parent-child relations. Besides defining the structure of the resulting relations, they also specify *containment relations*. For example the classes Router and InterfaceIPv4 in Figure 2.1 are connected by a parent-child relation, which means that a instance of Router contains a "specific amount" of InterfaceIPv4-instances.

2.2.2 Cardinalities

The expression "specific amount" leads to the concept of *cardinalities*. Cardinalities are similar concepts to the ones in UML class diagrams or ER diagrams and are used to define the *dimensions* of the resulting configurations. They are defined as intervals on the parent-child relations of the example model. They define the minimal and maximal number of child-instances a parent-instance can possibly have.
In a resulting configuration of the example in Figure 2.1, an instance of Router can contain between 0 and 4096 instances of InterfaceIPv4. This works analogously for all the other cardinalities.

There are situations where cardinalities are un-specified as between Transport and VlanPort in the example. The cardinality $[0..\mu]$ indicates, that each instance of Transport can have between 0 and μ instances of VlanPort. Note that μ is a placeholder for a chosen value. To get such a value for μ, we either have to consider the constraints or the user has to provide it as a separate information.

2.2.3 Attributes

Attributes can be defined at a class and consist of a *name*, a *data-type* and a set of optional *properties*. In the example model in Figure 2.1, attributes are defined in the lower part of the nodes. The first column contains the name, the second one the data-type and the third one the possible properties. The name should be self explanatory. Data-types are comparable to the ones used in object-oriented programming. They will be explained in a moment. Optional properties further define characteristics regarding the value of an attribute. Examples of properties are:

- `readOnly`: this property makes the value of such an attribute not editable by the user but is set by the system instead. The user has no possibility to insert, change or delete a value of such an attribute. In the example of Figure 2.1, attribute `operationalState` on the InterfaceIPv4-class is `readOnly`.

- `isNillable`: this property makes it possible for the value to be `null`, a distinct value regardless of the used data-type. Attribute `userLabel` on Router as well as `encapsulation` and `mtu` on InterfaceIPv4 in the example are attributes, whose value can be set to `null`.

- `mandatory`: this property is the opposite of `isNillable` and makes it impossible to set a value of an attribute to `null`. The attribute `vlanId` on VlanPort in the example can not be set to `null`.

- `key`: this property has to exist on exactly *one* attribute in each class, the *key-attribute*. Each class in the example has one `key` attribute.

The data-types define the value of the attribute and are well-known from several programming languages. Some of them can be further specified with properties, which further refine the data-type:

- `boolean`: The data-type used in propositional logic containing `true` or `false`.

- `int`: As in other programming languages, it is an unconstrained integer. It cannot be greater than 64 bits, but it is possible to constraint it further by the use of the `range`-property containing `min` and `max` options. Since some of the smaller `int`-types are used more frequently, the current implementation provides the following sub-types: `uint8`, `uint16`, `uint32`, `uint64`, `int16`, `int32` and `int64`.

- `string`: A character sequence with a possible `length` property to constrain it in its length and the `validValues` property to constrain it with a regular expression. Regular expressions are defined by the *POSIX extended regular expression (ERE)* standard. This validation feature allows the representation of IP and MAC addresses, dates, QoS sequences and many more.

- `moRef`: A string-attribute, referring to another class. It acts similar to a pointer in C++. This data-type has a special meaning for *bi-directional associations* as we will describe in Section 2.2.5.

These base-types can be seen as the building-blocks for an attribute. However, there are situations where the need for a richer type arises. In these cases, we have a reference in the attribute to one of the following data-types:

- `enum`: an enumeration-type, i.e., consists of a finite number of *members*. Each member contains a *name* and an integer *value*.

- `struct`: can be seen as an inner class. Similar to a `struct` in the C programming language it contains a list of struct-members, each one of the type `enum`, `string`, `boolean`, `int` or `moRef`. If the `isExclusive` property is given, this data-type implements the semantics of a `union` in the C programming language, i.e., only one of its member can be set at instantiation time.

2.2.4 Instance uniqueness

A key attribute is used to uniquely identify an instance in a set of instances of the same class. The current implementation uses `string` for all key-attributes in each class. The name of such attribute can be arbitrarily chosen.

Even though the key attribute is needed to identify an instance, the *fully specified path name* starting from ManagedElement is uniquely determining an instance in an overall configuration (see Section 2.3 for an example).

2.2.5 Bi-directional Associations

There is also the possibility for instances to refer to other instances possibly located in a different branch of the tree. This is achieved with *bi-directional associations* involving two attributes. Both attributes need to have the `moRef` data-type.

The two end-points (i.e., instances) of such an association are called *server* and *client*, whereas both instances contain references of each other. However, the reference in the client is `readOnly`, so the client is only passively containing the referring instances, whereas the server contains a modifiable attribute of type `moRef`. The value of such an attribute is the fully specified path of the client. This implies that the client has to be inserted before the server.

The bi-directional association in the example in Figure 2.1 between InterfaceIPv4 and VlanPort consists of the attribute `encapsulation` on the InterfaceIPv4, which is the server and points to the `reservedBy` attribute of VlanPort.

2.3 Configurations

A configuration of a ECIM model is the same concept as an instance of a class. Starting from the example model in Figure 2.1, one can derive a large number of configurations depending on the business needs and the specific usage of the software. The configuration in Figure 2.2 shows a minimal example of a configuration. The value of the key-attribute is explicitly concatenated to the name, e.g., Router=1.

This example contains only of one user-inserted instance, namely Router=1 (or fully specified ManagedElement=1, Transport=1, Router=1). The other two instances are `systemCreated`. The inserted instance, further contains the only attribute `ttl=50`. In order to write this configuration to memory, the operator has to type specific commands into the configuration manager in a top-down approach, i.e., first insert ManagedElement=1, then Trans-

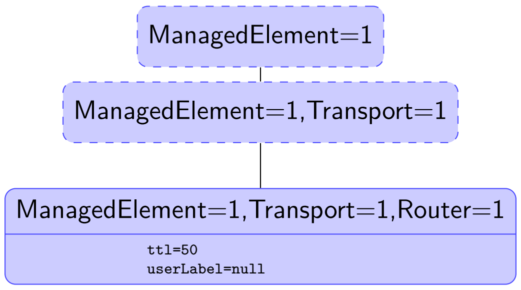

ManagedElement=1
ManagedElement=1,Transport=1
ManagedElement=1,Transport=1,Router=1
ttl=50
userLabel=null

Figure 2.2: Minimal Configuration of the model in Figure 2.1.

port=1 and then Router=1. The configuration manager always navigates into the previously created instance. Further, the `ttl` attribute on the last instance will be assigned while being inside the Router=1 instance.

The next example in Figure 2.3 shows another possibility how to configure the system using the model in Figure 2.1. Note that this time we omitted the fully specified path in each node, since it is easily derivable from the structure.

Here we can see, how a bi-directional relation on a configuration works. The InterfaceIPv4=1 node contains the path to the client instance, whereas the client keeps a `readOnly` list of all the servers, i.e., the fully specified path (as a string) of each referring server. The client's `reservedBy` attribute is only there to show where the *bi*-direcional adjective comes from. It will not be inserted by the user.

Another similar configuration in Figure 2.4 shows how it works with multiple servers pointing to the same client.

In more complex configurations like the one in Figure 2.5, it is more obvious why to have the fully specified path as a unique identifier for each instance. The example is *not* a configuration according to the example model, but to show how uniqueness in the model is guaranteed. The example model would need to change and to contain Transport not as `isSystemCreated` and have a cardinality of [2..2].

If we would only rely on the key-attribute, we would have an ambiguity problem. The configuration is not respecting the model in Figure 2.1, but is used to demonstrate the configuration possibilities.

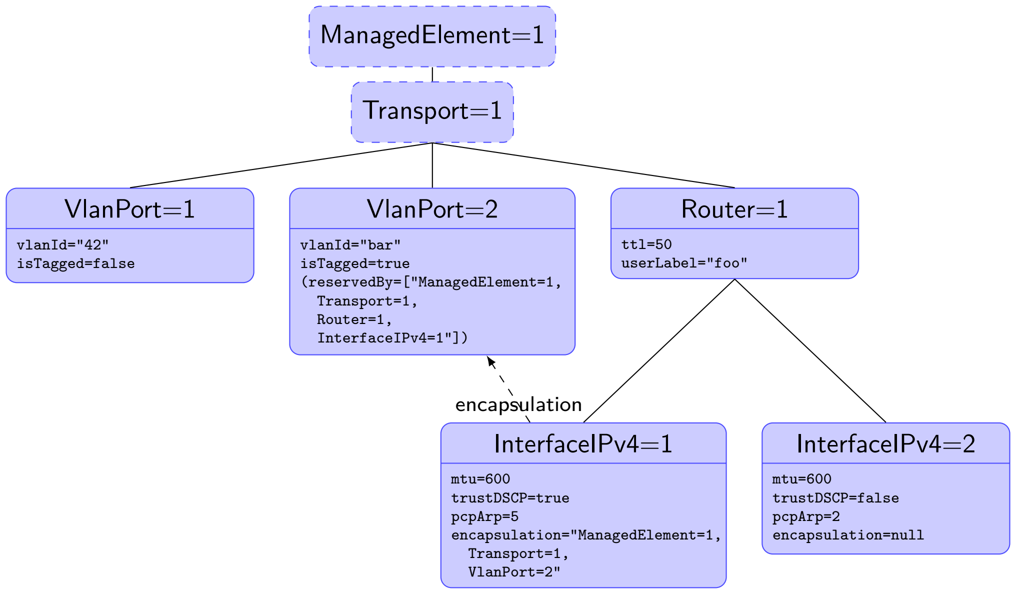
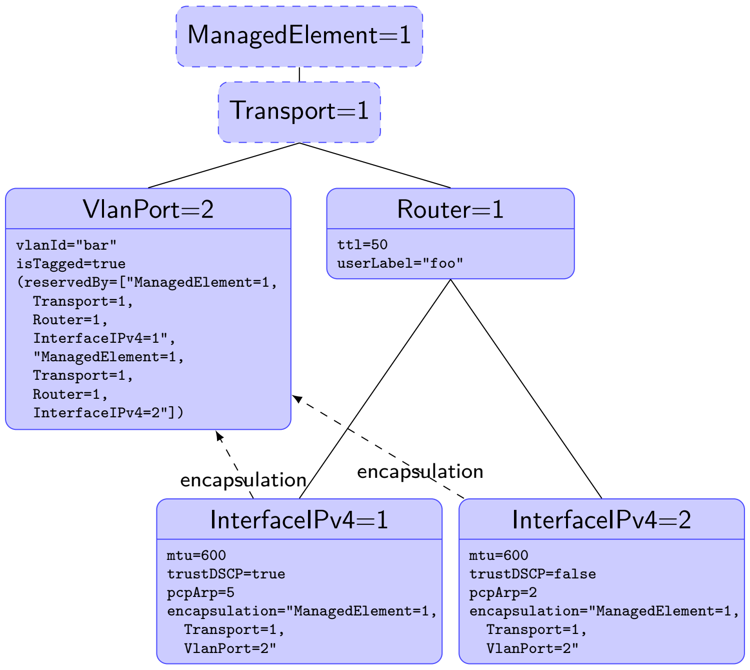


Figure 2.3: Configuration of the model in Figure 2.1.

2.4 Scientific Context

As we have seen in Chapter 1, highly configurable software is desired, but difficult to model and derive test-cases. Ericsson AB developed ECIM models to model the configuration possibilities.
Besides that, also other commercial as well as open source solutions needed this modeling approach. Thus, researchers focused on *Feature Models*, a compact representation of all products in a *software product line (SPL)*. A *configuration* is a set of selected *features* of such a feature model, which respects the structure of the feature model. The original definition of feature models was given by Kang et. al. [18], where the overall *method* of feature identification was presented. However, in the ECIM domain the features or classes are already identified and modeled. The formal definition of a feature model is the following [18, 10]:

Definition 2.1. A **feature** is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system. A **feature model** is a **feature diagram** with additional information such as descriptions, binding times, priorities and others. A **feature diagram** is a structural organization of a set of features. It is usually represented as a tree with the root representing a concept (e.g. software system) and its descendant nodes are features. A **configuration** defines one possible product of a feature



ManagedElement=1
Transport=1
VlanPort=2
vlanId="bar"
isTagged=true
(reservedBy=["ManagedElement=1, Transport=1, Router=1, InterfaceIPv4=1", "ManagedElement=1, Transport=1, Router=1, InterfaceIPv4=2"])
Router=1
ttl=50
userLabel="foo"
encapsulation
encapsulation
InterfaceIPv4=1
mtu=600
trustDSCP=true
pcpArp=5
encapsulation="ManagedElement=1, Transport=1, VlanPort=2"
InterfaceIPv4=2
mtu=600
trustDSCP=false
pcpArp=2
encapsulation="ManagedElement=1, Transport=1, VlanPort=2"

Figure 2.4: Configuration of the model in Figure 2.1 with multiple bi-directional associations.

diagram. A configuration of a feature diagram is comparable to an object of a class in object-oriented programming. A **staged configuration** is the process successively specializing a feature diagram followed by the derivation of a configuration from the most specialized feature diagram in the sequence.

Extensions of the original definition of feature models include cardinalities as well as attributes [6, 5, 11]. These extensions are essential to model ECIM in terms of feature models. Figure 2.6 shows an example of a feature model from the context of a mobile phone product line. Other examples covering both software systems as well as industrial product lines are present in current research and literature.

Important to notice is, that in our example the inner nodes of the tree are not only abstract concepts, but are actually used and need to be instantiated as any other class.

The cardinalities $\langle 1-1 \rangle$ and $\langle 0-2 \rangle$ in the example are *group cardinalities* which are applied on a group of features. They are not used in the ECIM specification.

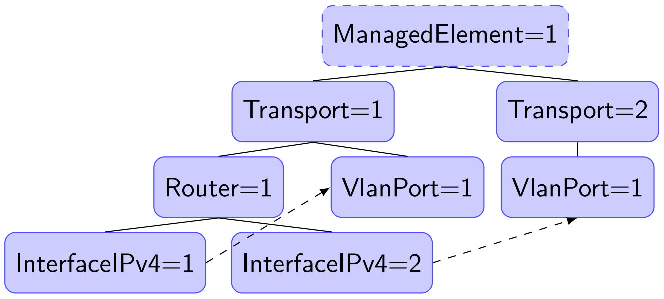


Figure 2.5: Configuration to demonstrate uniqueness of instances.

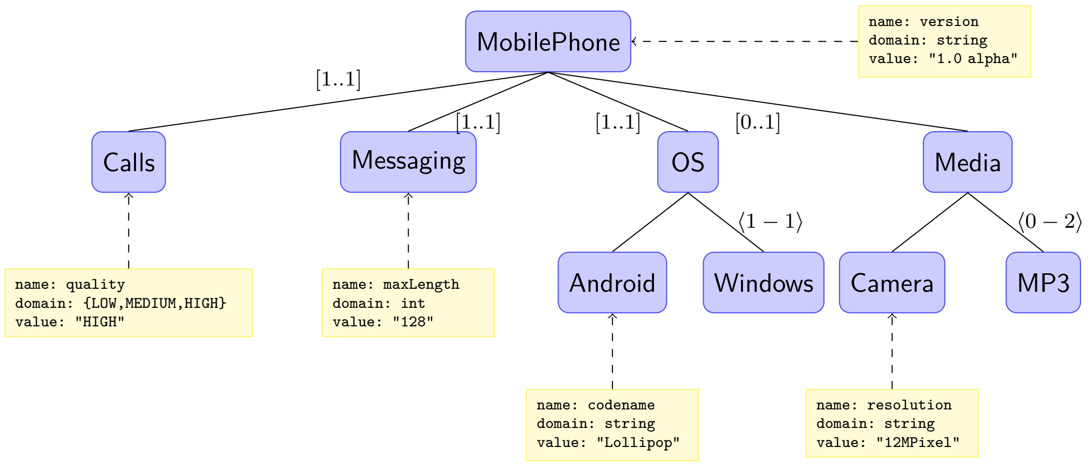


Figure 2.6: Example of the *mobile phone* feature model.

Chapter 3

Constraints

After we have seen how we can derive configurations from a ECIM model as well as how such a model looks like, we will now focus on the used *constraints*, which are called *dependencies* in the ECIM context. Dependencies are used to express conditions which have to be valid, in order for a configuration to be accepted by the configuration manager. The current implementation of the ECIM meta-model specification expresses these constraints in Schematron.

Schematron[1] is a validation language for XML documents. It defines assertions which have to be satisfied in order to pass the test and for the document to be valid. Schematron is usually embedded into a XML file and uses XPath queries to express the constraints.

In the ECIM XML files, Schematron-rules can be defined on a class. The validation of those rules is applied to the resulting configuration.

3.1 Example Dependencies

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
 <pattern>
  <rule>
   <assert test='count(InterfaceIPv4[@loopback]) le 64'><
        value-of select="."/> Maximum number of InterfaceIPv4
        configured as loopback is 64</assert>
```

[1] http://www.schematron.com/

```
    <assert test='@ttl eq 64'><value-of select="."/> Attribute
         Time-To-Live (ttl) must be equal to 64</assert>
    <assert test='are-distinct-values(./InterfaceIPv4/
        @encapsulation)'><value-of select="."/> The
        InterfaceIPv4's encapsulation must be unique.</assert>
  </rule>
 </pattern>
</schema>
```

Listing 3.1: Dependencies on the class Router of the Example in Figure 2.1.

The example script in Listing 3.1 gives the fully specified Schematron validation on the class Router. The `<assert>`-tag is the most important one, where `test` is the XML-attribute containing the XPath expression; usually one expression for each assertion. `<value-of>` is used to define the starting point of the verification, which is the current instance "`.`". The text inside `<assert>` is the error-message. We will omit this text, since it will not be further used by our examples. However, this text is displayed at the configuration manager if the validation fails, i.e., if the `test` attribute returns `false`.

```
count(InterfaceIPv4[@loopback]) le 64
```

The `test` attribute of the first assertion is used to verify if the number of InterfaceIPv4 instances (in general, *not* under the current Router-instance), which have the `loopback` attribute set, is smaller or equal (`le`) than 64.

```
@ttl eq 64
```

The second assertion is trivial and ensures that the `ttl` attribute is always set to 64.

```
are-distinct-values(./InterfaceIPv4/@encapsulation)
```

The last assertion is used to validate, if all `encapsulation` attributes of all InterfaceIPv4 instances under the current Router instance are distinct, i.e., have different values.

InterfaceIPv4 contains assertions too, which can be seen in Listing 3.2

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
 <pattern>
  <rule>
   <assert test='(@encapsulation and not(@loopback)) or (
       @loopback and not(@encapsulation))'><value-of select=
       "."/>...</assert>
   <assert test='not(@loopback and (@bfdStaticRoutes eq 1))'>
       <value-of select="."/>...</assert>
  </rule>
 </pattern>
</schema>
```

Listing 3.2: Dependencies on the class InterfaceIPv4 of the Example in Figure 2.1.

3.2 XPath Expression Set

Beside these trivial assertions other, more complex expressions are possible, where XPath is the core of the validation language. The current implementation used in ECIM uses a reduced set of XPath expressions in order to validate the resulting configurations. An XPath expression can be of the form:

- Literals, i.e., integers and strings.
- Unary and binary *boolean* operators: $\neg$, $\wedge$, $\vee$, $\otimes$. The arguments to those operators are again an XPath expression.
- Binary *comparison* operators: $<$, $\leq$, $>$, $\geq$, $=$, $\neq$. Each operator takes two XPath expressions as arguments.
- Binary *arithmetic* operations: $+$, $-$, $*$, $\div$, mod. Each operation takes two XPath expressions as arguments.
- *Path expressions* are used to navigate inside the configuration-tree and can be seen as a list. Each element denotes one move and can have one of the following form:
  - . refers to the current instance.
  - .. refers to the parent instance.
  - `<name>` refers to a specific class name. This returns *all* instances of this name.

    - `@<name>` refers to an attribute name.

  For example, let `./../A/B/@c` be defined on a class `X`. Then it informally denotes: For each instance of `X` take the parent instance, select all child instances of type `A` then for all child instances of type `A` select all child instances of type `B`. For all `B` instances, select the value of attribute `c`.

- *Filter expressions* can be seen as predicates for path expressions, where a condition to select a specific instance is added. Consider the slightly modified example of before: `./../A/B[@c = 1]`. In this case we select only those `B` instances whose `c` attribute equals `1`.

- *Function calls* allow the use of special functions. Only a small subset is used from the original XPath range of functions. Each function takes at least one argument. `xpath`-arguments are other arbitrary Xpath expressions. Those functions include:

    - `count(xpath)`: simply returns the number of occurrences for which `xpath` holds.
    - `are-distinct-values(xpath)`: asserts if all values inside the evaluated `xpath` have a distinct value.
    - `matches(string, pattern)`: checks if the given `string`-attribute matches a regular expression `pattern`.
    - `exists(xpath)`: is mainly used for `moRef`s, i.e., if the client exists the `xpath` is pointing to. If `expr` is not a `moRef` this function checks if `xpath` is not `null`.
    - `contains(xpath, string)`: checks if a given `string` is inside an expression `xpath`.
    - `string-length(string)`: returns the length of a given `string`.

We will come back to the example once we have to generate test-cases automatically. The power of these dependencies allows the developers of ECIM to define a big variety of conditions which have to be fulfilled in order for the configuration to be valid.

3.3 Scientific Context

In order to capture the explained functionality of the used XPath expressions, we will introduce *constraint satisfaction problems (CSP)*. Then we will proceed to *satisfiabiliy modulo theories (SMT)*, a generalized successor of the famous *SAT problem*.

3.3.1 Constraint Satisfaction Problem (CSP)

The purpose of a CSP is, to determine the satisfiability of constraints. The following definition should clarify the purpose further:

Definition 3.1. [26] A **constraint satisfaction problem** (CSP) is defined by a set of **variables** $X_1, X_2, \ldots, X_n$, and a set of **constraints** $C_1, C_2, \ldots, C_m$. Each variable X_i has a nonempty **domain** D_i of possible **values**. Each **constraint** C_i involves some subset of the variables and specifies the allowable combinations of values for that subset. A **state** of the problem is defined by an **assignment** of values to some or all of the variables $\{X_i = v_i, X_j = v_j, \ldots\}$. An assignment is called **satisfiable** if it does not violate any constraint. A **complete** assignment covers every mentioned variable. A **solution** to a CSP is a complete, satisfiable assignment.

A various number of problems can be mapped to a CSP [26], including 8 queens, graph coloring and other famous puzzles as well as real world applications including artificial intelligence or resource allocation in operating systems. Further, variations of this definition include finding of the whole possible set of assignments for a given problem, the number of solutions or a maximal or minimal solution for a given problem. No matter how the original definition is changed, CSPs remain NP-hard [26] in general.

The most prominent CSP is the *satisfiability (SAT) problem of propositional logic*. It involves only variables in the domain $\{\texttt{true}, \texttt{false}\}$, i.e., propositional variables and determines their satisfiability. *SAT modulo theories (SMT)* goes one step further: it uses *many-sorted first order logic (FOL)* to allow a richer logic for describing problems. Furthermore, it constraints the interpretation of some symbols by the use of *background theories* [13]. For theoretical details of FOL and various background theories we refer to the literature, where Mendonça et al. [13] give a good overview. Later, we will refer to this paper to select suitable theories for our context.

To formulate SMT problems, the SMT-LIB-initiative has been founded in 2003 [2]. Its aim is to facilitate research and development in SMT. One of the biggest achievements of this initiative is the development of the SMT-LIB input language [2] which is used by several solvers. In general, the syntax is borrowed from Common LISP and provides a big variety of constructs to support different logics, theories and instructions for the solver. A comprehensive explanation of the SMT-LIB syntax can be found in [2].

[2]`http://smt-lib.org/`

Chapter 4

Methodology

This chapter will lead to the overall process of the test-case generation. It uses the examples from the previous chapters dealing with the ECIM models (Chapter 2) and the constraints (Chapter 3).

Before merging both examples from the previous chapters into the test-case generation, we first will study general considerations as well as the subdivision of the problem. Then the process is illustrated on an example.

4.1 General Considerations

The overall purpose is to create instances in an automated fashion. Thus, the best way would be, to generate instances from the model without any user interaction. Since one important requirement is the coherency with respect to the dependencies, we need them in a central position of the overall test-case generation process. In order to do that, we need to perform a translation from the current format of the model (XML) to the SMT readable input format. Since most of the SMT solver use the standardized SMT-LIB input language, we will use it as well. Details about the implementation and the used SMT solver are presented in Chapter 5.

No matter how the overall procedure will look like, we need to encode the concepts *class*, *attribute* and *instance*, because all of them can appear in XPath expressions of the dependencies. In the SMT domain, the theory of *uninterpreted (i.e., free) functions* is subject of current research and a lot of effort has been put into improving the decision procedures in terms of efficiency. Thus, we will start by explaining the translation of the formerly mentioned concepts.

4.2 SMT Translation of General Concepts

Data Types

We will start to translate the basic building blocks of an attribute, which are the data-types. We observe that we need a `null` value for each attribute regardless the type. Thus, we need to model a type, which is exclusively `null` or `t`, where the latter one can be of any type. A declarative data-type including *type parameters* is the desired technique. This concept is used in many functional programming languages. The desired data-type (e.g., in Haskell) would look like

```
data Attribute t = Null | Value t
```

In the SMT-LIB domain, we have the same possibility. We can model this distinct type in a similar fashion. Before showing the exact syntax of this data-type in the SMT-LIB input language, we will examine the different data-types further:

- `boolean` and `int` can be directly mapped to a SMT solver, since `Int` and `Bool` are the most fundamental data-types and supported by almost every SMT solver.
- `string` can not be directly mapped, because almost all SMT implementations lack of a built-in theory for strings. Current research is focused on this aspect and examine the theory as well as the corresponding decision procedures.

  An idea would be, to develop an own theory built upon other theories. However, since the development of such a string-theory is a rather time-consuming and error-prone task, we will leave this task for a future work and will now explain workarounds for `string`s. They can be subdivided into 3 groups:

  1. **Key attributes:** Each key attribute of a class is represented as a string. The string can be arbitrarily chosen, with the only requirement, that it remains unique for the same parent-instance.

Thus, integers would be appropriate for this purpose. After the SMT-solver runs, we will retrieve the same value, but we then can replace it with actual string values. This can be done with a simple mapping by the help of a hash-table or similar.

2. **Regular expressions:** Such attributes are further constrained with a `validValues` property and are problematic, because we cannot use the SMT solver to find values for them because of the lack of a theory for strings and especially regular expressions. However, it is also possible to use an external list of strings and then use the indexes of those strings which are valid. Thus, we evaluate the valid strings of the present list in a previous step and yield a list of integers, representing the indexes of those strings which are valid. For example, consider the following list of strings:

   ```
   ['1.2.3.4/32', '5.6.7.8/32', '127.0.0.1/16',
   '192.168.0.14/32']
   ```

   Applying the regular expression `'.+/32'` to each element in it and returning its corresponding index in the list only if it matches the regular expression, would yield the integer list `[0, 1, 3]`. Thus, for those strings in the list, which fulfill the regular expressions, we gather the index and get a new integer-list containing the indexes of valid values.

3. **Free strings:** Since the entered value is not constrained, we can generate them randomly or by using the same method as seen before. Further, it is highly unlikely that such an attribute will be further constrained by dependencies. For the case, that such a string-attribute is further constrained with a `length` attribute, one could think about using an integer for it. Otherwise a constant integer value would be enough.

- As we have seen in Sections 2.2.4 and 2.2.5, the `moRef` data-type is like a pointer to one instance in the overall configuration. Thus, we need to have a globally unique identification for each instance. Since we decided to use integers in the SMT-LIB encoding, we require those integers to be *globally unique*. As a result, `moRef` attributes will also be encoded as integers.

- `enum`s have distinct integer-values in its members and thus, they can be used directly as integers in SMT-LIB.

- `struct`s can be seen as an own `class`. Each member follows the rules defined so far.

We have seen that all data-types can be mapped to integers. Also for the `boolean` type this is possible, if we follow the usual transformation of `true` and `false` to 1 and 0.

Thus, the algebraic data-type seen before can be translated directly to a corresponding SMT-LIB type

```
(declare-datatypes () ((Attribute null
                         (mk_Attribute
                          (value Int)
                         )
                       ))
)
```

SMT solvers are aware of algebraic data-types, by using the theory of *uninterpreted functions*. This theory is primarily used to allow the definition of functions.

For further restrictions of data-types, e.g., `uint8`, we would need to constraint the used data-type further. In a purely object-oriented fashion one would consider to create a sub-type of the `Int` type. However, this is not possible at the moment. `Int` is a sort, i.e., a pre-defined type. With a sort usually comes a whole theory, since the decision procedures need to know the properties of the abstract, newly introduced sort. Thus, creating a `uint8` sort, we would need to repeat the theory of integers and add the allowed range, in this case $-2^7 \dots 2^7 - 1$. Some SMT solvers allow the definition of new sorts, but they are an arbitrary concept with no notion about relations and allowed operations on them. In those cases, the decision procedure is used to find a universe for this sort. This mechanism is not exactly what we need here.

Classes

To model a whole class of the model, we can use the same approach as before. We use a declarative data-type, but this time without the exclusive `null` value.

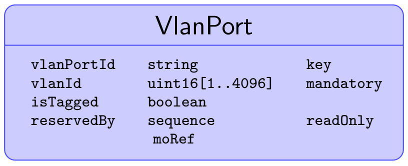

VlanPort
vlanPortId string key
vlanId uint16[1..4096] mandatory
isTagged boolean
reservedBy sequence moRef readOnly

Figure 4.1: VlanPort as a single class from the example model in Figure 2.1. We further assume that this class is nested into the `vlanPort`-MIM (name-space).

To visualize this, consider Figure 4.1, with the single class VlanPort, this time nested into the MIM (name-space) `vlanPort`. This class will be encoded as follows:

```
(declare-data-types () ((vlanPort__VlanPort
                        (mk_VlanPort
                         (vlanPort Attribute)
                         (vlanId Attribute)
                     (isTagged Attribute)
                    )
                  ))
)
```

Denote, that we do not need to have the attribute `reservedBy` on the class, since it is a `readOnly` attribute. We can also see, that the MIM name takes part of the overall name to avoid name clashes for the same class name in other MIMs.

Instances

Finally, the encoding for *instances* is rather simple, by using a constant function. Since each constant needs to be uniquely accessible by the SMT solver, we agreed on the following name schema (assuming that a key 1 was already assigned on this instance):

```
(declare-fun vlanPort__VlanPort___1 () (vlanPort__VlanPort))
```

More general, we agreed on the name-schema

```
<mim-name>__<class-name>___<key>
```

4.3 General Approach

As mentioned before, we will model the dependencies as well as the used classes and instances with the use of the SMT-LIB input language. Once we have done this, we will run the SMT solver and check for satisfiability. If the overall encoding is satisfiable, we can retrieve an *assignment* from the SMT solver. The assignment contains values for the different attributes of those instances involved in at least one dependency. That means, we get a partial test-case from the SMT solver's assignment.

However, there are still 3 remaining problems to solve:

1. The overall structure of the configuration, i.e., parent-child relations between the classes as well as the number of instances for each instance (cardinalities).
2. The encoding of the used constraint language of the dependencies (in our case the used XPath expression-set).
3. Attributes, which are not covered by the dependencies (unconstrained attributes).

4.4 Structural Characteristics

As denoted before, the structural characteristics involve any hierarchical relations as well as cardinalities. We decided to achieve a desirable "structural coverage". That means, that the resulting set of configurations are able to explore a desirable amount of selected cardinality-values from the initial cardinality-range. This should involve also combinations between selected cardinality values.

Czarnecki et. al. [11] define *staged configuration* as the subsequent *specializations* of a feature model in each stage. This definition can be also found in Section 2.4.

They define *specialization steps* and categorize them into 6 groups. We will follow the *feature cloning* approach.

Definition 4.1. Feature Cloning: Given a cardinality of the form $[a..b]$ with $a < b$. Then we can select a $m \in [a..b]$, and re-define the existing cardinality to $[0..(b - m)]$.

The article does not explain the exact implementation of the cloning step. Moreover the selection of m is not specified further. To achieve a desired "structural coverage", we need to select m in a specific manner in each step of the process. We decided to define the staged configuration further and express the stages in a different form.

Definition 4.2. Staged Configuration using Feature Cloning: Let $\mathcal{I} = [a..b]$ be a cardinality, s denote the number of stages (test-cases) and $1 \leq i \leq s$. Further, let $m_i \in \mathcal{I}$ be a cardinality of the test-case i. Then for all $1 \leq j < j' \leq s$ we require $m_{j'} = m_j + x$. We call x the *skip-size*.

m_i is the selected value of the cardinality. It gives the number of cloned instances. x indicates which parts of the cardinality will be skipped. It can be seen, that x could vary for each stage as well as for each cardinality. For our purpose we assume a constant x. A possible staged configuration would choose $m_1 = a, m_2 = a + x, m_3 = a + 2x, \ldots, m_s = b$.

Example 4.1. Assume a cardinality [0..10]. Then the test-cases in Table 4.1 are selected using a staged configuration with a skip-size. We can observe,

skip-size	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}	m_{11}
1	0	1	2	3	4	5	6	7	8	9	10
2	0	2	4	6	8	10					
5	0	5	10								
10	0	10									

Table 4.1: Example of a staged configuration on the cardinality [0..10] with different skip-sizes.

that $x = 1$ always leads to 100% coverage, since we will produce test-cases for all possible cardinality values. $x = (b - a)$ yields the lowest amount of test-cases involving only two border-cases a and b.

We will later see how the implementation of this concept is done.

4.5 Dependencies

As explained in Section 3, we will use a subset of XPath expressions and thus we need to translate them into SMT-LIB assertions.

We will now recall the syntax of the XPath expressions by explaining them in terms of SMT-LIB assertions and instructions.

Integer and String-literals

Especially integers need a special treatment once they occur in any constraint. The integer can be in the context of a function or of an attribute value. We need to treat both cases according to the context. For example, if we encounter an integer literal in a dependency like

```
count(a) = 10
```

The right hand side of the equation can be directly taken as an integer. Functions like `count` always return a plain integer. If we have a dependency like

```
a = 10
```

we need to model 10 as `(mk_Attribute 10)`, i.e., we need to tell the SMT solver explicitly that we are using 10 in context of an attribute value.

Operators

All boolean and arithmetic *operators* are directly mappable to the SMT-LIB input language. The comprehensive list of those mappings can be found in Appendix C. Since most of them are available also as operators in the SMT-LIB input language, the transformation is rather straight forward.

Functions and Path Expressions

Most of the functions can be expressed not solely by the SMT-LIB input language, but in combination with a preprocessing step which gathers information to represent them as SMT-LIB assertions.

This applies for example to `count(xpath)`. Here we need to process the argument `xpath` first, before encoding the actual `count`-function. If the argument is a path expression terminating with a class we simply count the occurrences of instances of the class and replace the expression with the corresponding integer, which we know in advance due to the previously performed staged configuration.

If we have a path expression with an attribute `a` at the end, we count the occurrences of attributes which are not `null`, i.e.

```
(+ (ite (= (a <instance1>) null) 0 1)
   (ite (= (a <instance2>) null) 0 1)
   ...
)
```

In most of the cases we have a filtered expression as an argument. In this case, we can compute the `count` of it with the following SMT-LIB expression:

```
(+ (ite <filter applied on instance1> 1 0)
   (ite <filter applied on instance2> 1 0)
   ...
)
```

A special role has `matches(pattern, a)`. In this case, since `a` is a string, we require `a` to have an attached string-list. Then, `matches` will be applied in advance, i.e., the indexes of the string-list which match the `pattern` will be the result-set, for which we can create regular assertions. `contains` works in the same way.

Path Expressions

Path expressions represent the structural dependencies of a test-case and thus, cannot be represented in SMT-LIB input language. Gathering values of attributes (`@a`) or instances (`A`) can only be done in advance and have to be represented into the context of the using function or filter.

4.6 Unconstrained Attributes

Unconstrained attributes do not appear in any constraint and thus, the values found by the SMT solver are less interesting for us since it can take any value. One possibility to achieve different values is, `(distinct ...)`, which ensures difference in values of instances and also attribute values. However, the SMT solver can not achieve a certain goal in terms of a combinatorial coverage. We will now introduce a concept which tries to achieve this.

4.6.1 Combinatorial Testing

The flexibility of a ECIM model was explained thoroughly. However, with flexibility comes complexity in terms of possible configurations. This problem is called *software configuration space explosion* [25]. A popular approach to this issue is called *combinatorial testing*, which computes a *covering array*, i.e., a small set of configurations, such that all possible *t-way* combinations of settings appear in at least one configuration. t is usually a given *interaction strength* [25]. If we set $t = 2$ (pair-wise testing), the resulting array needs to contain all combinations for all pairs of input values. This approach

Table 4.2: Resulting test-cases of 4 boolean variables.

a	**b**	**c**	**d**
`true`	`true`	`true`	`true`
`false`	`false`	`false`	`true`
`false`	`true`	`false`	`false`
`true`	`false`	`true`	`false`
`true`	`false`	`false`	`false`
`false`	`false`	`true`	`false`

already reduces the number of test-cases tremendously for low t. The negligible remaining problem is, that an error, which raises due to a combination of $t + 1$ values, can not always be found.

Example 4.2. Assume 4 boolean variables a, b, c and d in a configurable software. Computing all possible combinations yields $2^4 = 16$ combinations. However, we can reduce the amount of test-cases by looking at all *pair-combinations* of variables. In other words, we compute *2-way* combinations. Consider the result in Table 4.2. We have 6 test-cases which reduces the combinatorial explosion problem.

The drawback of this example is, that, we could not cover an error occurring with $a = \texttt{true}, b = \texttt{true}, c = \texttt{false}$ or even an error involving all four variables which is not present in the table.

While this example is only for demonstration purposes, another example with 10 boolean input variables would result in 27 test-cases using a pair-wise approach instead of $2^{10} = 1,024$ test-cases achieving full coverage.

The problem, to compute an optimal covering set, i.e., a set of configurations, such that all possible *t-way* combinations appear in at least one configuration is NP-hard [19]. However, in the recent years there were many approaches published, by using various strategies from various topics. Lei et. al. [21] uses a so-called *In-Parameter-Order* strategy, where the idea is, to start with a simple and small test set and then achieve the result by growing horizontally and vertically. For details about this technique we refer to [21]. Other approaches use greedy algorithms, hill climbing algorithms or heuristics so speed up existing procedures.

For our case it would be a desired mean to include such an approach. However, we can only apply this approach to unconstrained integers and we need to partition the input space of certain types, like integers.

4.7 Final Considerations

To sum up the previous Sections, one can see that the parsing of the constraints need to be done rather early. Since the structural considerations are currently using a greedy approach, several information of the constraints could even instruct the SMT solver by finding a coherent structure.

After that, the structure has to be explored. To do that, we start with the smallest possible product and apply the staged configuration approach of Czarnecki et. al. [11]. The skip-factor in Definition 4.2 will be user-defined and fixed for each cardinality.

The main part is the actual test-case generation. It is obvious, that some of the produced configurations can not satisfy the constraints due to structural incompatibilities. This is, where the SMT solver can help: it either finds a satisfiable attribute-assignment or reports unsatisfiability. In the latter case, we have to delete the test-case from our resulting test-set.

The resulting order of the different tasks of the test case generator leads to the following steps, which have to be performed for each test-case:

1. Run the t-wise algorithm to assign values to the unconstrained attributes.

2. The previously prepared structure by the staged configuration is decorated with `key` attributes to distinguish the instances in the following SMT solving phase.

3. SMT-LIB code has to be produced in the following order:

   (a) Attribute and class data-type declarations.

   (b) Instances declaration including assertions regarding the data-types as well as assertions about bi-directional associations.

   (c) SMT assertions according to the dependencies.

   (d) Trailing instructions to check for satisfiability and, if so, to retrieve a model.

4. SMT solver run.

   (a) In case of unsatisfiability, delete test-case from the result set.

   (b) In case of satisfiability

      i. translate result into real world values.

ii. report the real values in the final format to the user.

Steps 3 and 4 are applied to each abstract test-case.

4.7.1 Theoretical Considerations

An important aspect when trying to apply theoretical results to industry, is to make a complexity analysis. Since most of the theories in SMT solvers are NP-hard, decidable or even undecidable, such an analysis can give reasons to change the encoding or even the overall approach. By evaluating the current achievements, one can observe that most of the theories become at least decidable by cutting of quantifiers. This is an important observation and will be considered in our case. We will now look at the different facets in our encoding and investigate the complexity. Later, in Chapter 6 we will examine these parts further in terms of the used SMT solver in practice.

In our scenario we use algebraic data-types for the attribute type and the classes, uninterpreted functions for the instances as well as boolean logic and integer arithmetic for the constraints. All other aspects will be outsourced to the surrounding implementation.

Uninterpreted Functions

Uninterpreted functions are an important façet in most of the usages of a SMT solver. This applies also to our scenario. Most decision procedures implement a so-called *congruence closure algorithm.* Those algorithms are the focus of a long-lasting research. Already Nelson and Oppen, two important researchers in the SMT domain, worked on the combination of different theories but also on congruence closure algorithms [23]. Their results were based on the work of Downey et. al. [14].

Determining the complexity of the congruence closure algorithm has been proven to be difficult. The current achievement is an average complexity of $\mathcal{O}(n \log n)$ but it is unknown whether this is optimal or not [24].

Inductive Data-types

A *inductive data-type* [3] such as used in our scenario uses *constructors*, and possible *testers* and *selectors.* The Σ-signature of inductive data-types associates a function symbol with each constructor and selector and one predicate with each tester [3]. Oppen provided an algorithm to decide over inductive data-types with one constructor in polynomial time. The general problem of inductive data-types remains NP hard, but since data-types were shown to be handy in practice, reasonably efficient algorithms exist.

Arithmetic

The general arithmetic can be deduced from *presburger arithmetic*, a calculus with the only symbols $\{\mathbb{N}, +, -, \leq\}$. Ground satisfiability of presburger arithmetic over the real numbers is decidable in polynomial time, whether using ground formulas over $\mathbb{R}$ is NP-complete [3].

An interesting property comes with the use of *difference logic*: This logic requires each atom to be $a - b \;\oplus\; t$ with a, b to be uninterpreted constants, $\oplus \in \{=, \leq\}$ and t to be an integer. The quantifier free satisfiability problem of difference logic is $\mathcal{O}(n^2)$. The obvious extension of multiplication, complicates the overall complexity discussion: even conjunctions of integer-based ground formulas become undecidable [3].

4.8 Example

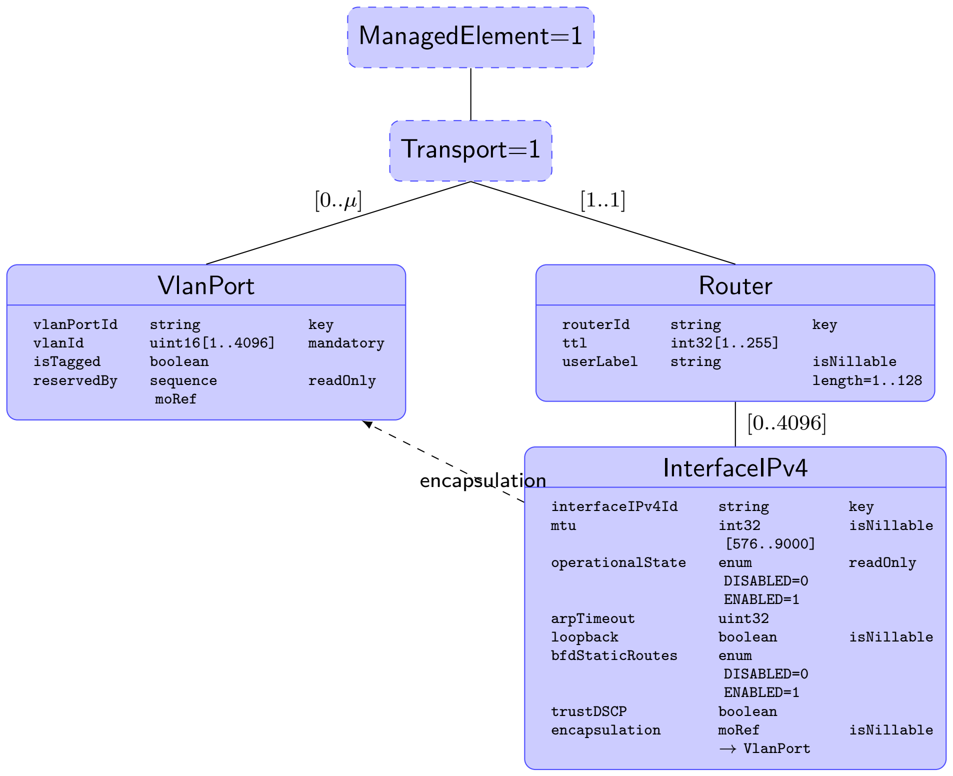
We will continue with the examples of the model and the dependencies of the previous chapters and partially show how test-cases are generated. First, recall the example of Chapter 2 in Figure 4.2. We will use it together with the dependencies on Router and InterfaceIPv4.

1. Abstract Test Case

First, we want to create *abstract* test-cases, i.e., only containing the number of instances as well as the order of them, which is implicitly given by the structure. This allows us having a starting point for the SMT solver which is then called to find values for the attributes or report unsatisfiability.

We first have to define the unknown cardinality $\mu = 10$. Further, we will use a constant skip-factor $x = 2$. The resulting set of abstract test-cases (without system created instances) is:

- Router(1)
- Router(1), VlanPort(2)
- Router(1), VlanPort(2), InterfaceIPv4(2)
- Router(1), VlanPort(2), InterfaceIPv4(4)



ManagedElement=1
Transport=1
[0..μ]
[1..1]
VlanPort
vlanPortId string key
vlanId uint16[1..4096] mandatory
isTagged boolean
reservedBy sequence moRef readOnly
Router
routerId string key
ttl int32[1..255]
userLabel string isNillable length=1..128
[0..4096]
InterfaceIPv4
interfaceIPv4Id string key
mtu int32 [576..9000] isNillable
operationalState enum DISABLED=0 ENABLED=1 readOnly
arpTimeout uint32
loopback boolean isNillable
bfdStaticRoutes enum DISABLED=0 ENABLED=1
trustDSCP boolean
encapsulation moRef → VlanPort isNillable
encapsulation

Figure 4.2: Example of a ECIM model.

- Router(1), VlanPort(2), InterfaceIPv4(6)
- Router(1), VlanPort(2), InterfaceIPv4(8)
- ...
- Router(1), VlanPort(2), InterfaceIPv4(4096)
- Router(1), VlanPort(4)
- Router(1), VlanPort(4), InterfaceIPv4(2)
- ...
- Router(1), VlanPort(4), InterfaceIPv4(4096)
- ...
- Router(1), VlanPort(10), InterfaceIPv4(4096)

We will illustrate the functionality of the SMT solving part on the resulting abstract test-case:

1*ManagedElement, 1*Transport, 1*Router, 2*VlanPort, 2*InterfaceIPv4

Another thing we have to take into consideration is the order of instances. This is another opportunity for the SMT solver to help with. We can see abstract instances as integer values and translate the needed sequential order of the instances into an adequate mathematical relation: a *total order*. In other words, we translate each parent-child relation and each bi-directional association into a $a < b$, where a and b are abstract instances. Important to notice is that the client of a bi-directional association needs to be inserted before the server, since the latter one needs to know the full path of the client.

For our purpose we would have the following fragment of SMT-LIB code:

```
(declare-const Router Int)
(declare-const VlanPort Int)
(declare-const InterfaceIPv4 Int)

(assert (< Router VlanPort))
(assert (< Router InterfaceIPv4))
```

```
(assert (< VlanPort InterfaceIPv4))
(check-sat)
(get-model)
```

The same idea applies as before: if the SMT solver is able to find a solution, i.e., if the constraints are satisfiable, we have an explicit order of the instances. This is obviously the case in this example, e.g., with `Router=0`, `VlanPort=1`, `InterfaceIPv4=2` we satisfy the assertions.

For the following steps, we need to review the dependencies on both classes Router (Listing 3.1) and InterfaceIPv4 (Listing 3.2).

2. Unconstrained Attributes

As discussed before, we are interested for those attributes which are not part of a dependency. At the same time we will exclude key-attributes, since they are assumed to be assigned in advance and globally unique, which for our test-case means: VlanPort=1, VlanPort=2, Router=3, InterfaceIPv4=4 and InterfaceIPv4=5. The rest of the attributes we have to take into account in the combinatorial approach are:

- VlanPort.`vlanId` ([1..4096])
- VlanPort.`isTagged` (`boolean`)
- Router.`userLabel` (`string`, `isNillable`)
- InterfaceIPv4.`mtu` ([576..9000], `isNillable`)
- InterfaceIPv4.`arpTimeout` (`uint32`)
- InterfaceIPv4.`trustDSCP` (`boolean`)

Further to the data-types we assume a partitioning of the input values where it is possible. Thus, for `vlanId` we encode the input-range into 3 equidistant partitions For the string attribute `userLabel` we assume to have a prepared string-list:

- "Ericsson SSR 8020"
- "Ericsson SSR 8004"
- "Ericsson SmartEdge 100"

- "Ericsson SmartEdge 1200"

If we would like to run the combinatorial approach, one might notice that in some scenarios we do not have enough values. For $t = 2$ we would only have 22 value combinations for all test-cases, which is obviously not enough to supply the rest of the test-cases. However, in that case we could increase t and continue. We can continue until $t = n$ where n is the number of unconstrained attributes, which means, that we test all combinations.

Thus, the start parameter for the computation of the covering array are the one in Table 4.3.

attribute	**input parameter**
VlanPort.`vlanId`	$[1..1366], [1367..2731], [2732..4096]$
VlanPort.`isTagged`	`true`, `false`
Router.`userLabel`	`null`, $[0, 1, 2, 3]$
InterfaceIPv4.`mtu`	`null`, $[576..3384], [3385..6192], [6193..9000]$
InterfaceIPv4.`arpTimeout`	3 partitions from `uint32`
InterfaceIPv4.`trustDSCP`	`true`, `false`

Table 4.3: The input parameters for the computation of the covering array. The partitions of `uint32` were ignored due to the big numbers in the ranges.

After executing the algorithm to compute the covering array we get 22 different test-cases. This was done by the same algorithm which will be used in the implementation in Chapter 5. The resulting values are:

- `vlanId`: [1..1366]
- `isTagged`: `false`
- `userLabel`: `null`
- `mtu`: `null`
- `arpTimeout`: [0..1431655765]
- `trustDSCP`: `false`

A random value from the chosen intervals of the integer-attributes will be selected in step 4, the post-processing step.

3. SMT solver

After getting values for the unconstrained attributes, we can start the transformation into SMT-LIB as discussed in Section 4.5. The overall SMT-LIB code of the presented example can be seen in Appendix B.
The definition of the attribute-type and the 3 used classes are the following:

```
(declare-datatypes () ((Attribute null
                    (mk_Attribute
                     (value Int)
                    )
                  ))
)

;VlanPort
(declare-datatypes () ((VlanPort
                        (mk_VlanPort
                         (vlanPortId Attribute)
                        )
                      ))
)

;Router
(declare-datatypes () ((Router
                        (mk_Router
                         (routerId Attribute)
                         (ttl Attribute)
                        )
                      ))
)

;InterfaceIPv4
(declare-datatypes () ((InterfaceIPv4
                        (mk_InterfaceIPv4
                         (interfaceIPv4Id Attribute)
                         (encapsulation Attribute)
                         (loopback Attribute)
                         (bfdStaticRoutes Attribute)
                        )
                      ))
)
```

We can see that the key attributes are included, since they are needed for eventual bi-directional associations. For example VlanPort has no constrained attributes, but the key-attribute needs to be included for the bi-directional association from InterfaceIPv4.

The next encoding involves the instances as well as the data-type restrictions of the attributes, which we will show on one instance of each class.

```
(declare-const VlanPort___1 VlanPort)
(assert (= (vlanPortId VlanPort___1)
           (mk_Attribute 1)
        )
)
...
(declare-const Router___3 Router)
(assert (= (routerId Router___3)
           (mk_Attribute 3)
        )
)
(assert (and (<= 1 (value (ttl Router___3)))
             (<= (value (ttl Router___3)) 255)
        )
)

(declare-const InterfaceIPv4___4 (InterfaceIPv4))
(assert (= (interfaceIPv4Id InterfaceIPv4___4)
           (mk_Attribute 4)
        )
)
(assert (or (= (encapsulation InterfaceIPv4___4) null)
            (and (<= 1
                     (value (encapsulation InterfaceIPv4___4))
                 )
                 (<= (value (encapsulation InterfaceIPv4___4))
                     2)
            )
        )
)
```

```
(assert (or (= (loopback InterfaceIPv4___4) null)
            (= (loopback InterfaceIPv4___4)
               (mk_Attribute 0)
            )
            (= (loopback InterfaceIPv4___4)
               (mk_Attribute 1)
            )
        )
)
(assert (or (= (bfdStaticRoutes InterfaceIPv4___4)
               (mk_Attribute 0)
            )
            (= (bfdStaticRoutes InterfaceIPv4___4)
               (mk_Attribute 1)
            )
        )
)
...
```

Now all instances are set up and we need the final assertions: the dependencies. We use the transformation approach illustrated before:

```
(assert (<= (+ (ite (= (loopback InterfaceIPv4___4) null) 0 1)
               (ite (= (loopback InterfaceIPv4___5) null) 0 1)
            )
        64))
(assert (= (value (ttl Router___3)) 64))
(assert (distinct (encapsulation InterfaceIPv4___4)
                  (encapsulation InterfaceIPv4___5)
        ))

(assert (or (and (not (= (encapsulation InterfaceIPv4___4)
                         null))
                 (not (= (loopback InterfaceIPv4___4)
                         (mk_Attribute 1)))
            )
            (and (= (loopback InterfaceIPv4___4)
                    (mk_Attribute 1))
                 (not (= (encapsulation InterfaceIPv4___4)
                         null))
            )
         )
)
(assert (not (and (= (loopback InterfaceIPv4___4)
                     (mk_Attribute 1))
                  (= (bfdStaticRoutes InterfaceIPv4___4)
                     (mk_Attribute 1))
             )
        )
)

(assert (or (and (not (= (encapsulation InterfaceIPv4___5)
                         null))
                 (not (= (loopback InterfaceIPv4___5)
                         (mk_Attribute 1)))
            )
            (and (= (loopback InterfaceIPv4___5)
                    (mk_Attribute 1))
                 (not (= (encapsulation InterfaceIPv4___5)
                         null))
            )
         )
)
(assert (not (and (= (loopback InterfaceIPv4___5)
                     (mk_Attribute 1))
                  (= (bfdStaticRoutes InterfaceIPv4___5)
                     (mk_Attribute 1))
             )
        )
)
```

The final instructions for the SMT solver are those to check for satisfiability as well as retrieving the model. The full input script as well as the output of the SMT solver can be found in Appendix B.

4. Post-Processing

This step is needed to translate the SMT solver output into real values. In this step, all encoding aspects of data-types are translated into real enum-values, strings, booleans as well as selecting random values of the chosen ranges from the combinatorial algorithm. Finally, real syntax for the configuration manager is produced, which is trivial and the syntax will not be discussed here.

Chapter 5

Implementation

This chapter will discuss the practical work based on Chapter 4 and will further describe technical details.

5.1 Goal and Approach

The goal of the tool is, to generate positive test-cases, i.e., configurations which should always pass while configuring the underlying system. The name of the tool is *coteca* (<u>co</u>nfiguration <u>te</u>st-<u>ca</u>se).

These test-cases are based on the theoretical model explained in Chapter 2. Following the approach of the previous chapter, we can arrange the tool in way, that the different stages in the generation process are each encapsulated in various modules or classes.

First, the different parts of this comprehensive work were identified, starting from the input format to the desired output format.

To cover all needs to generate tests, the model specification sometimes lacks of further information, but also simplified assumptions are needed to achieve the goal properly. Examples for these assumptions and missing model details are regular expressions, missing limits of cardinalities, the path to the SMT solver and other user-defined sizes. These facts have to be provided by the user. To have the possibility to save these annotations, we did not require them to be passed as command line arguments, but to be stored in a so-called *annotation file*, which will be provided as an XML file. It will be explained in Section 5.3.

5.2 Work-flow and Architecture

To describe the work-flow of the final tool, enumerating the different responsibilities is a good practice. After that, these responsibilities will become steps in the work-flow in terms of classes or modules. The final thought goes more into a practical direction and explains the resulting architecture.

The following steps are necessary to generate test-cases:

1. The **XML-Parser** parses the XML files. Internally calls the parser for the *annotation file* (see Section 5.3).

2. The **Dependency-Handler** is responsible for parsing, refining and storing the given dependencies in Schematron (XPath). By refining, we mean already the partial translation into SMT-LIB format. The complete SMT-LIB code for finding a solution for a set of assertions will be completed once we know the structural properties.

3. The **Structure-Handler** resolves the structural aspects of the model-files, i.e., deals mostly with parent-child relations and the cardinalities, as we have seen in Section 4.4. The staged configuration will be implemented here.

4. The **Test Case Generator** prepares the input file for the SMT solver. For each test-case, one SMT run is performed. It also translates the reported model by the SMT solver into corresponding ECIM configurations.

5. **Test Case Writer** is needed to write the test cases into a machine-readable format. In our case it is a sequence of configuration-commands which are understandable by the configuration manager.

Before we explain these modules in context of a general architecture as well as the detailed technical work of each of these steps in the next Section, we will sketch out some general comments:

- If we are following a classical automated test-case generation process, we would need to include an intermediate step between the SMT aided Test Case Generation (step 4) and the Test Case Writer (step 5), which translates the intermediate format of the test cases into a more general, non-executable format (such as XML), which is then used by the Test Case Writer. Due to time limitations, we decided to implement the specific version only.

- The reason, to separate the Dependency Handler from the XML Parser has also to do with responsibilities. Further, the results of the Dependency Handler could be used in all the preceding steps, so a thorough knowledge about the domain is needed, which would have been simply too much for the XML Parser. Further, for the sake of an extensible architecture, the dependency handler could handle dependencies in a variety of formats (currently only Schematron).

The previously presented steps are following a logical order and look almost like the stages of a compiler. This order and the separation of responsibilities leads to the use of a *behavioral design pattern* [16], which we will use in our architecture.

Architecture

What we need is a hierarchical module-structure as well as a pipeline-fashioned work where each module has its own responsibilities: it simply completes its task on a specific object and passes it to the next element in the "pipeline".

The *chain of responsibility* pattern is exactly following our needs: it consists of a set of *processing objects*. Each of the processing objects (or handler) has its own responsibility to handle the *command object*, an object which is passed through the chain. In other words, each processing object handles the parts of a command object it knows about and passes the rest to the next processing object in the chain.

A nice variation of the classical pattern is, that some processing nodes can act as dispatchers, i.e., by branching the chain into a tree and directing the command into a variety of directions.

In our context, this allows us to implement each step as a chain-element. The command object needs to be generic enough for all steps to produce a desired output. The final architecture looks like the one in Figure 5.1. For a comprehensive architecture, we refer to Figure A.1 in Appendix A. Note that the `go` function in the `Handler` is needed, to mask the administrative code, i.e., calling the `handle` function of the subclass, checking if the next handler is set and if so, calling `go` again. It propagates the call, while providing an independent but uniform implementation of the single responsibility. For a code example of the `go` function we refer to the Implementation Details in Section 5.5.

The module which constructs and launches the chain will be our main program. Consider Figure 5.2 on page 48 for an example of a sequence

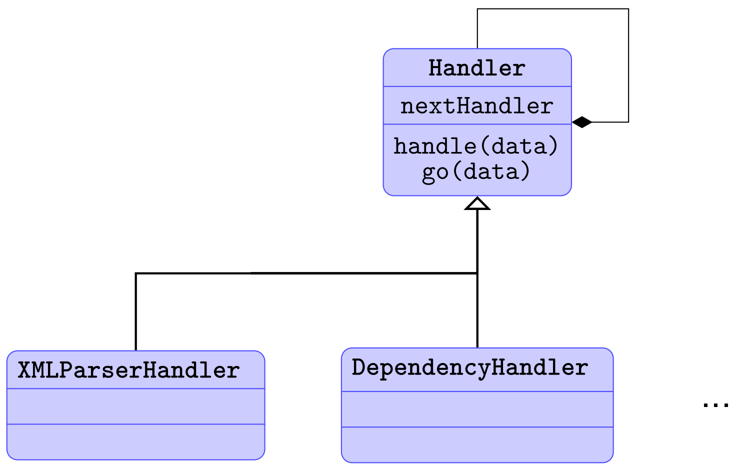


Figure 5.1: The class hierarchy of the different steps.

diagram containing 3 chain elements. The `main` thread contains a list containing the 3 involved handlers and calls `go` on the first element in this list. Inside `go`, a local function `handle` is called, which changes the arguments before calling again `go` on the next element in the chain with the modified arguments and so on. For details about the implementation, we refer to Section 5.5.

5.3 User Annotations

As we have seen, there are situations where the information taken from the MIM model is not enough to produce suitable test-cases. To provide all the needed information, we require the user to provide an *annotation* file. This file is also in XML-format and provides a way for the user to create test-cases for different scenario, i.e., hardware configurations.

Limits for cardinalities are one example of such a missing information. Most of the time, the user knows how many children there are required for a real test-case. Another problem arises, if the provided model contains cycles. In some rare scenarios it could be that a cycle inside the parent-child relations is present. To address these problems, the user can refine parent-child relations by, either providing one fixed cardinality to choose a fixed number of children, where 0 excludes the relation or by reducing it.

Another example are static parts of the tree. The user might not be interested in testing the whole `mim`-tree, but only fragments of it. Allowing

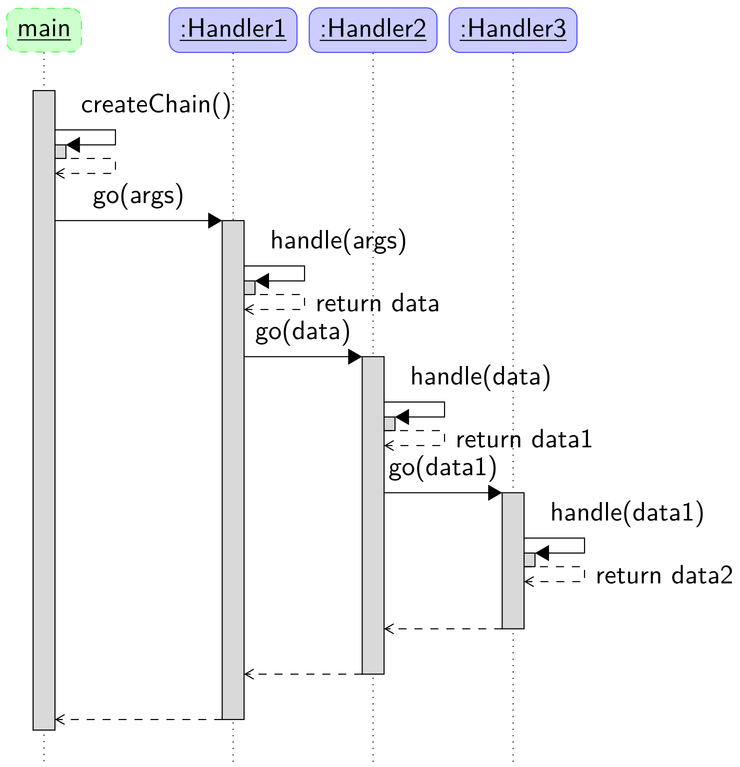


Figure 5.2: Schematic sequence diagram.

the tool to create only configurations belonging to a sub-tree, the output still needs to contain the upper nodes from the root node (`ManagedElement`) to the subtree. Further, some of the nodes are `systemCreated`, i.e., hard-coded into the system. For those nodes, we need to have fixed values. In other words, the user can specify static parts of the tree, which have to exist To test the desired fragment of a `mim`-tree.

5.4 Technical Aspects

We decided to use Python because of the dynamic nature of the language, the big variety of libraries and the good XML Parsing support. It supports object orientation and the decorator feature, which facilitates time measurement. There is also the possibility to call external processes, i.e., in our case the SMT solver.

We used the SMT solver z3 from Microsoft[1], a high-performance SMT solver which allows SMT-LIB as input language and supports a big variety of theories. Further, the authors are frequently publishing articles about theories, theorem proving and other SMT-related subjects. The solver is currently used for verification and testing purposes of several Microsoft products. It also took place in several SMT competitions and the documentation provides good support to develop applications.

The formerly discussed *chain-of-responsibility* pattern can be implemented using the object-oriented features as well as operator overloading of Python. As denoted before, we have a `go` function, which encapsulates the call to `handle` which is then overridden in the sub-classes. The overloaded `__add__` operator is used to connect a chain element to the existing one. The code of the base class `Handler` can be found in Listing D in the Appendix D.

The remaining discussion is, which object we are going to pass to the chain: the first choice was, to create an object containing all the needed information from the XML files as well as user annotations. Later, we decided to use a built-in `dict` object, which is internally implemented through hash-tables using open addressing for hash-collisions. Both indicators for fast access times and vanishingly small memory overhead on re-hashing. Thus, we are dealing with simple key-value pairs. This allows a fail-safe implementation.

A nice feature of Python comes with operator overloading. By overloading the `+` operator (`__add__`), we achieve `list`-semantics for our chain.

[1]`http://z3.codeplex.com/`

Listing 5.1: The main program.

```python
def __main__(args):
        h=ArgumentHandler()
        h+=XMLParserHandler()
        h+=DependencyHandler()
        h+=StructureHandler()
        h+=TestCaseGenerationHandler() #TestCaseWriter is
            called on each test-case internally
        h+=StatisticsHandler() #Used to gather several
            statistics

        h.go({"args":args})

        return 0
```

This allows the main program to be more flexible and maintain the single handlers as list elements. An example implementation of the main-function can be seen in Listing 5.1. Even the part of the program, which handles the command line argument from the user is a handler, where the arguments object is wrapped into a `dict`. This can be seen as the starting point of the overall procedure. After the construction of the chain, `go` is called on the first element, which internally behaves as you can see in Listing D.

5.5 Implementation Details

We will now have a look at specific details of the implementation. The command line usage of the tool is:

```
coteca.py [-h] [-s] annotations xml-files
```

`-h` is an optional argument and used to print the help.

`-s` is an optional argument to simulate the creation. This is used to verify for completeness, i.e., it checks if the given model-files are not lacking in any relations, data-types, or classes.

`annotations` is the location of the annotation-file.

`xml-files` are the provided XML files containing the MIM including all data-types, classes etc.

Besides these arguments, a log-file can be specified where information about the test-case creation run is collected.

The different Python modules from the chain are explained in the sequel, where most of them contain one or two python classes. Other Python modules not explained here are utilities, i.e., commonly used functions and exceptions, such as translation functions between the output of the SMT solver and the input language of the configuration manager.

XML Parser

As the name already suggest, the XML Parser parses the XML files containing the ECIM models. It uses the `lxml` package of Python. The procedure works as follows:

1. Put each XML file into a list of parsed files.

2. Parse annotation file by the help of `AnnotationParserHandler`. This is the starting point of the `dict` object. Several previously discussed information, such as the SMT solver command as well as command line arguments and its input file name, string-list locations, the static tree and the user defined cardinalities are stored after this step.

3. Parse classes, attributes and their data types. This also involves the properties on classes (`isSystemCreated`) and on attributes. Also the dependencies, i.e., the input of the SMT solver in a future step are stored in plain-text. The data-types are parsed as much as possible, but can only be completed at the next step.

4. Resolve missing data-types. This step is needed, because in the previous step, we could have encountered a derived data-type whose location was already processed.

5. Store relationships, i.e., parent-child relations as well as bi-directional associations. This step also creates the tree from the parent-child relations, which are available as pairs in the XML files. Afterwards, the JSON module of python is used to find any circular references. In that case an Exception is thrown.

6. Insert annotated classes. The static classes, which can be found in the annotation file are inserted directly into the `dict` object. This step has to be done at the end, since we need to ensure, that the tree is fully created.

The parser raises an Exception if a class or a data-type is missing. The classes are uniquely accessible by their containing `mim` as well as the `name`. The name of a class is therefore `<mim-name>__<class-name>`. This assures that we do not have any name-clashes, if one class with the same name appears in different `mim`s.

Dependency Handler

As mentioned before, the `DependencyHandler` is executed directly after the Parser to interpret the XPath expressions and store them into the `dict` object. It uses the `SchematronParserHandler` as well as a utility-module called `DependencyRefinement` to translate the parsed XPath expressions into a format, which is close to the SMT-LIB format.

The `SchematronParser` uses the `ply` module, a Python implementation for `lex` and `yacc`. The error-handling is basic, but indicates the line number as well as the position inside the input in case of an error in the XPath scripts. The result is again passed to the class on the `dict` object, by replacing the previously stored dependencies in plain-text.

Since the `SchematronParser` is unaware of the different classes, and knows them only by name, the `DependencyRefinement` has knowledge about the previously parsed classes, i.e., it can navigate through the tree, when hitting a `..` or a `@<attribute-name>`.

This module uses a so-called *context* for the current class, the current attribute and the current member (for structs). This is needed, because when a path expression is encountered, the current class is set to the one where the dependency was originally found. Since at this point, the `DependencyRefinement` module is unaware of instances, we simply navigate through the path with the abstract classes. The real instances are then resolved at a later point.

For the different XPath functions the module treats them differently, either by returning a *masked* part

- `#<class>.COUNT#`
- `#<class>.<attribute>.COUNT#`

or by setting a so-called *validation* on the class or attribute itself. Such a validation indicates an outsourced operation, such as `match` for regular expressions or `contains`.

StructureHandler

The StructureHandler has the task to resolve the *parent-child paths* from the tree in form of a Python-dict which is a trivial task involving a recursive function. The result is the set of distinct paths from the tree.

The next step is, to create abstract test-cases, i.e., only containing topological information. The final test-cases are created later to avoid memory-overhead. An abstract test-case only defines the set of classes for each test-case. To accomplish this, every single path is inserted as an abstract test-case.

Then it creates the combination of all paths and inserts them as abstract test-cases.

A further important task is, to decorate instances with useful information for the next step, the actual *test-case generation phase*. Such information are the previously gathered validations both on classes and on attributes from the DependencyRefinement module, the level in the current tree of the test-case as well as partitions and translation-functions to translate SMT results back to real values.

Now we can recursively *blow up* the previously created abstract test-cases. The recursion is not a performance problem, since the maximal depth is the maximal depth of the tree. This is done by using the staged configuration approach as described in Section 1.4. Instead of the *skip-size*, we use a globally defined *skip-factor*. This factor is given by the annotation file and is a percent value of how many cardinality-values are excluded from the generation process.

TestCaseGenerationHandler

This module is responsible for the execution of the final considerations in the last chapter (Section 4.7), namely:

1. Running the t-wise algorithm. This is done by the iterator implemented in the Python module named *allpairs*[2]. Other than the name suggests, it is a general version which supports different values for t. If we run out of values for a specific t, because the t-wise array is too small for the number of our instances, we simply reset the iterator and start over with $t + 1$.

[2]https://pypi.python.org/pypi/AllPairs/2.0.1

2. Distributing the key attributes. An internal mapping between the test-case number and the actual key number is done in another small `dict`.

3. Prepare the SMT code. To accomplish this, a utility module `SMTCodeGen` is used, which is aware of the exact syntax of the SMT-LIB input language.

4. Write the prepared SMT-LIB code into a file and run the SMT code.

5. Check if the result was satisfiable. If it is the case, then parse and translate the output model of the SMT solver using the SMT-LIB parser into the adequate format of the test-cases and call the `TestCaseWriter`. Again, this parser uses `ply`. If it is unsatisfiable, it simply writes the SMT code into a specific file. This mechanism is used to understand the generated code of the tool and examine it further. The same mechanism is used, if the SMT solver returns something else than `SAT`, `UNSAT` or `TIMEOUT` in case a timeout is given in the annotation file.

Before calling the SMT solver, the input file consists of the SMT encoding as seen in Section 4.8:

- The declarative datatype `attribute`, which allows values to be `null`.
- For each class, a declarative data-type with its attributes.
- The instances of each class as a constant. In SMT terms this is a function without parameters.
- Assertions regarding the instances, i.e., attribute ranges, fixed key values and distinctive bi-directional associations. It also sets `moRef` attributes to null, if the client is missing in the set of classes.
- Dependencies are inserted as assertions.
- Trailing instructions for the SMT solver. This instructs the SMT solver to search for satisfiability and retrieve a model.

This file is passed to the SMT solver together with the user-defined parameters for the SMT solver, such as time-out from the user annotations.

As mentioned earlier, the next module, namely the `TestCaseWriter` is called after *every* successful SMT run. This allows the user to interrupt the overall test-case generation process at anytime and still retrieving the currently generated test-cases.

TestCaseWriter

Gathers the test-cases in machine-readable code from the `dict` object and transforms them into a sequence of configuration commands. This code is then printed to the screen, written to a file or both. These information are also coming from the annotation file and are stored inside the `dict` object.

Chapter 6

Evaluation

After we have seen the implementation of the tool, we will describe how the results were evaluated. This will be done on a practical evaluation in Section 6.3. In the integration evaluation we will shortly describe how the tool can be integrated in practice in terms of the build chain. First of all, we will terminate the theoretical evaluation of Section 4.7.1.

6.1 Final Comments on the Theoretical Evaluation

As already stated in Section 4.7.1, we will now complete the work by examining the complexity of the different SMT facets on the used SMT solver z3.

There are no current information available on the reasoning system behind z3 in terms of *uninterpreted functions*. However, it can be assumed that the authors decided to implement a congruence closure algorithms, due to several publications and slides on the web. They further mention an improvement of the original algorithm [12], called Ackermann-reduction, which makes congruence closure polynomially solvable.

According to de Moura et. al. [8], the way z3 handles algebraic data types is a reduction to uninterpreted function symbols.

The *linear arithmetic* decision procedure used in z3 is largely based on the one used in Yices[1], another SMT solver. More precisely, the procedure

[1]`http://yices.csl.sri.com/`

is based on a simplex algorithm [15], which is based on a tableau-based solution. Simplex has exponential worst-case complexity.

6.2 Metrics

To understand the differences between the selected models for the evaluation in the next section, we need to introduce metrics in order to compare the properties of a selected model to another one.

AECR

Mendonça et. al. [22, p. 2] introduce the *Extra Constraint Representativeness (ECR)* on a feature to be the ratio between the number of classes involved in a constraint and the overall number of classes. Since we are dealing with more complex constraints involving attributes, we will use a slightly modified ratio, namely *Attributed Extra Constraint Representativeness (AECR)*, as follows:

$$AECR_{\mathcal{FM}} = \frac{\sum\limits_{\mathcal{C} \in \mathcal{FM}} d_{\mathcal{C}}}{\sum\limits_{\mathcal{C} \in \mathcal{FM}} a_{\mathcal{C}}}$$

where $d_{\mathcal{C}}$ is the number of attributes in a class $\mathcal{C}$ dependent on at least one constraint and $a_{\mathcal{C}}$ is the number of attributes of a class $\mathcal{C}$. Another name for this metric would be *constraint density.* We have now a metric to determine how much a feature model is covered by dependencies.

Computing AECR on the example model seen in Figures 2.1 and 4.2 would yield $9/15 = 0.6$, which means that 60 % of the model is covered by dependencies.

6.3 Practical Evaluation

For the practical evaluation, we looked at two different models. To gather information about the model, the test-cases and running times, we appended a `StatisticsHandler` to the end of our chain. Some of the produced configuration sequences were then executed on the final system. We will have a look at the models and the results in the next section.

Metric	**M1**	**M2**
# classes	7	11
# attributes	42	28
max. tree depth	5	6
dependencies	7	11
max. cardinality	10	4096
AECR	0.5222	0.6667

Table 6.1: Specification of the test-models *M1* and *M2*.

6.3.1 Model description

As we can see in Table 6.1, we named the two models **M1** and **M2**. The two models are similar in the number of classes and attributes. However, since the cardinalities are bigger at **M2**, we can also deduce a bigger number of configurations from **M2**. The AECR are similar for both cases.

We decided to take these two models to demonstrate, how the tool and the SMT solver can handle large cardinalities, i.e., instances per test-case in terms of execution-time. The skip-factor was chosen individually for each test-case. We can already see that selecting `ALL` for **M2** would give us an infeasible amount of test-cases which is impossible to generate as well as to test. We will respect these facts in the next subsection.

6.3.2 Practical Evaluation Results

The practical evaluation was performed on the test-models from the previous section. The summary of the results can be seen in Table 6.2. For a comprehensive list, i.e., the results for each test-case per test run, we refer to Appendix E. Due to the large amount of test-cases of $\mathbf{T3_{M2}}$, the comprehensive results were omitted in this case. The test-cases vary in the skip-factor and thus, in the resulting test-cases. The maximal number of instances depends on the skip-factor and the cardinalities. This is the reason for the high number in test-case $\mathbf{T1_{M1}}$. Surprisingly, all test-cases verified by the SMT solver resulted in satisfying ones. The time-out for $\mathbf{T1_{M1}}$ was set to 20 minutes and can be found somewhere between 658 and 2,872 instances. A reason for this timeout we will discuss in the discussion chapter.

Metric	**T1$_{\mathbf{M1}}$**	**T2$_{\mathbf{M2}}$**	**T3$_{\mathbf{M2}}$**
# of test-cases	15	135	769
skip-factor	0.7	0.7	ALL
max. instances	4,119	519	260
# SAT	7	135	260
# UNSAT	0	0	0
# TIMEOUT	8	0	0
timeout	20 min	20 min	20 min

Table 6.2: Summary of Evaluation.

6.4 Integration Evaluation

From a software-engineering point of view, the tool has to be used, whenever the MIM model changes. This will happen at the project initialization phase, after the requirement analysis has been performed and the system architecture has been decided. Further, it could be the case that the model changes during the development. In this case, after the change of the model-files the tool could sit right at the version control system and create test-cases, if the model files are committed to a specific repository. The drawback of this integration is, that one might have only a part of the model and not the overall tree. The previously described models (from Section 6.3) are all parts of the model. Further, we could run into performance issues when executing the tool on the comprehensive model.

Chapter 7

Discussion

We will now discuss the results of the evaluation in Chapter 6 and also general aspects of the thesis work.

In general, we can observe, that a SMT solver can be useful to find coherent test-cases. We have seen that the transformation of constraints (in our case XPath) is rather crucial, but not a performance over-head. The real performance-issue is a high upper bound of the cardinalities. We used the fragment of the biggest value, i.e., 4096. Together with the structural properties of the resulting test-cases this forms a challenging problem.

The performance issues of the SMT solver for test-cases with a high amount of instances can have many reasons. Besides the theoretical considerations, which we concluded in the previous chapter, the use of a different encoding could be considered.

7.1 SMT Encodings

As denoted in Section 3.3, there are more theories one could consider when modeling this problem. We only considered 3 theories, namely algebraic data-types, uninterpreted function symbols and linear arithmetics. However, one could think about using the array theory or bit-vectors to represent various aspects of the problem description. Since the theory of bit-vectors uses bit-blasting which is directly mapped to the SAT solver, the complexity remains the same, but current achievements developed strong heuristics to approach good running times.

There is even the possibility to engineer own theories [7] in z3 and integrate the theory into the solver. This theory could be a direct mapping for the problem description. z3 offers a variety of APIs in various languages to have programming support for this achievement. However, a deep theoretical understanding of various facets as well as verification would be needed to engineer a theory. Implementing such a solution would change the architecture of the tool tremendously, because of the strong coupling between solver and the tool. Since we spent most of the time in designing the tool and understanding the context, we omitted this feature. An idea how engineering such a theory could look like is given by Bjørner et. al. [7].

7.2 Evaluation Discussion

As we have seen, the performance of using SMT solving highly depends on the number of instances per test-case. To understand where the SMT solver starts to time-out one could further investigate the timeout-limit in Table 6.2 and discuss different encodings further. However, after spatial tests, we found out, that the number of instances is not the crucial factor which leads to an explosion in the execution times. We believe that the specific structural layout of a test-case is the reason for the SMT solver to time out or not.

Another important aspect to mention is the environment. These tests were executed in a normal Laptop PC (HP EliteBook 2.3 Ghz with 8GM Main Memory) within a virtual machine. It has to be assumed that several performance measures are tremendously better, if run on an appropriate setup. The virtual machine was equipped with 3.5 GB main memory and the used operating system was Debian 7.5 64-bit.

7.3 Impact

The impact of the presented tool can vary. On one hand it does show how to produce configurations based on the dependencies on a practical level. On the other hand, it can be seen that this approach it is not mature enough to be used in a software testing environment. The reason for that is the missing interaction with the user defined objectives. Thus, the tool does produce test-cases, but rather explores the structure in a certain way which might not be conform with the user-desired semantics. However, the architecture of the tool makes it extensible to be extended with this feature. Including this feature would imply that we could improve the reliability of both the configuration manager as well as other testing activities. As a result, we would increase the reliability of communication systems (in the context of Ericsson AB). Reliable communication has many benefits in terms of sustainability, society and environment. It improves the possibility to share knowledge, to prevent from disasters and to implement changes necessary for a sustainable society.

7.4 Future Work

There are several technical aspects when talking about future work both in terms of the tool `coteca` as well as scientific considerations:

- *A deeper analysis of the model.* The development of several metrics to distinguish models and achieve good heuristics could improve the overall generation process. This would also include another subcommand to create a summary of the model including various analyses to have a first impression of the model.

- *Support for more MIM data-types.* The current implementation accepts only the most important data-types. However, sometimes the models also contain sequences or sequences of structs. On the other hand, this would imply to find a suitable theory in the SMT domain, which is not trivial. Currently, z3 supports the notion of lists, but only in a pure algebraic fashion, i.e., no notion about the length or membership-relation is known. In other words, the list concept is not a theory, but rather an encoding of another theory, namely uninterpreted functions.

- *String and regular expression support.* Currently, the user has to provide lists for specifically important string attributes. There are solutions to explore regular expressions. HAMPI [20], Kaluza (which is build upon Kudzu [27]) or Rex [29] are tools which allow to define string constraints and search for solutions. All of them are built upon a SMT solver and use other formalisms such as automata theory to explore and find instances. Another approach to this, would be the development of a theory upon the SMT solver and search for valid strings. However, this would imply the development of a suitable string-theory and also, to develop a logic. String logics and string decision procedures are hot topics of current research.

- *Performance optimizations.* The tool certainly needs some performance optimizations. Even though a profiler was used to locate performance overheads, further algorithmic optimizations have to be considered. Also in terms of constraint transformation, some performance could be improved by shifting parts of the simplification work from SMT to the Python modules. The used memory is generally under control, but could be further improved by removing redundant data. This could be done with a data-flow analyzer.

- *Concurrency.* This could be an important performance gain. No concurrency at all is considered at the moment. However, several functions could run in parallel. For example the `DependencyHandler` and the `StructuralHandler`. As long as the structure resolving does not depend on the constraints, they can be run in parallel.

- *Structure resolving.* The current work of the `StructureHandler` is a brute-force algorithm which explores the range of a cardinality by a given user input. Grieskamp et. al. [17] present a way to combine interaction coverage and path coverage to explore most paths of parameter combinations. The overall topic of *combinatorial interaction testing* is NP-hard, but could be mapped to our scenario. Another idea is the treatment of a structure as a grammar and apply grammar derivation algorithms to it [4, 1]. These algorithms could then be changed, such that they satisfy a certain *derivation coverage* or other user-defined criterion.

- *Normalization.* Similar to relational data-bases, normalization of feature models is the goal to achieve uniqueness in representation and structure. Van Deursen et. al. [28] apply various normalization steps on a *feature diagram algebra.* These normalization steps are applied on traditional feature diagrams, but they could be extended to capture also cardinality-based feature models. The result of such a normalization would imply an easier encoding and thus, could lead to performance gains.

Except for future work from a practical point of view, there are also scientific obstacles to overcome. In general we have to distinguish between two scientific directions:

1. Problem specific development of solutions *involving* a SMT solver.
2. Development of decision procedures *for* the SMT solver.

The first direction is more about how to usefully involve a SMT solver into research, ignoring the internal details and implementation. In our case the focus was lying on the test-case generation for feature models.

One might see that the second direction involves a lot more theoretical knowledge. An example for this are all recent papers about *congruence closure algorithms* or about *string-logics* and many more.

Chapter 8

Conclusion

As we have seen, using SMT to generate test-cases from a set of constraints on a model which is the one we have seen in Section 2 is challenging and implies a deep theoretical knowledge. We have seen how highly configurable software can be modeled and how configurations can be derived to test them as well as how to use constraints to steer the test-case generation.

On the other hand, SMT solving provides a comfortable way to delegate decisions to a formal method which is specifically designed for such a task. With the benefit of a comfortable handling there comes the drawback of performance or complexity together with the encoding, which could have an astonishing impact on the running time. However, we believe that there would be some space for improvement by smartly choosing an encoding.

The tool is currently still in the stage of testing, but is already partially used on smaller models to generate configurations for the development. It can be assumed that the work will continue and the tool will be completed to a degree to cover the comprehensive functionality of the models. Once this work is done, it is likely that the tool will be included into the build-chain of the current project set-up.

Bibliography

[1] Ebrahim Bagheri, Faezeh Ensan, and Dragan Gasevic. Grammar-based Test Generation for Software Product Line Feature Models. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '12, pages 87–101, Riverton, NJ, USA, 2012. IBM Corp.

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[3] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[4] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[5] David Benavides, Antonio Ruiz Cortés, Pablo Trinidad, and Sergio Segura. A Survey on the Automated Analyses of Feature Models. In José Cristóbal Riquelme Santos and Pere Botella, editors, *XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006), Octubre 3-6, 2006, Sitges, Barcelona, Spain.*, pages 367–376, 2006.

[6] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *LNCS, ADVANCED INFORMATION SYSTEMS ENGINEERING: 17TH INTERNATIONAL CONFERENCE, CAISE 2005*, page 2005. Springer, 2005.

[7] Nikolaj Bjørner. Engineering Theories with Z3. In Konstantin Korovin, Stephan Schulz, and Eugenia Ternovska, editors, *IWIL 2012: The 9th International Workshop on the Implementation of Logics, Merida, Venezuela, March 10, 2012*, volume 22 of *EPiC Series*, pages 1–2. EasyChair, 2012.

[8] Nikolaj Bjørner and Leonardo de Moura. Tractability and Modern Satisfiability Modulo Teory Solvers. In *Handbook of Tractability.* Cambridge University Press, 2012.

[9] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In David S. Rosenblum and Sebastian G. Elbaum, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, pages 129–139. ACM, 2007.

[10] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged Configuration Using Feature models. In Robert L. Nord, editor, *Software Product Lines, Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004, Proceedings*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer, 2004.

[11] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: An Appetizer. In Marcel Vinicius Medeiros Oliveira and Jim Woodcock, editors, *SBMF*, volume 5902 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2009.

[14] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the Common Subexpression Problem. *J. ACM*, 27(4):758–771, 1980.

[15] Bruno Dutertre and Leonardo Mendonça de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 81–94, 2006.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[17] Wolfgang Grieskamp, Xiao Qu, Xiangjun Wei, Nicolas Kicillof, and Myra B. Cohen. Interaction Coverage Meets Path Coverage by SMT Constraint Solving. In Manuel Núñez, Paul Baker, and Mercedes G. Merayo, editors, *Testing of Software and Communication Systems, 21st IFIP WG 6.1 International Conference, TESTCOM 2009 and 9th International Workshop, FATES 2009, Eindhoven, The Netherlands, November 2-4, 2009. Proceedings*, volume 5826 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2009.

[18] K. C. Kang, Jaejoon Lee, and P. Donohoe. Feature-oriented product line engineering. *Software, IEEE*, 19(4):58–65, 2002.

[19] Richard M. Karp. Reducibility Among Combinatorial Problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[20] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A Solver for String Constraints, 2009.

[21] Yu Lei and Kuo-Chung Tai. In-Parameter-Order: A Test Generation Strategy for Pairwise Testing. In *HASE*, pages 254–261. IEEE Computer Society, 1998.

[22] Marcílio Mendonça, Andrzej Wasowski, Krzysztof Czarnecki, and Donald D. Cowan. Efficient compilation techniques for large scale feature models. In Yannis Smaragdakis and Jeremy G. Siek, editors, *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings*, pages 13–22. ACM, 2008.

[23] Greg Nelson and Derek C. Oppen. Fast Decision Procedures Based on Congruence Closure. *J. ACM*, 27(2):356–364, 1980.

[24] Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Challenges in Satisfiability Modulo Theories. In Franz Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2007.

[25] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Evaluating Interaction Patterns in Configurable Software Systems. Technical Report CS-TR-4940, Computer Science Department, University of Maryland, College Park, June 2009.

[26] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[27] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for JavaScript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 513–528. IEEE Computer Society, 2010.

[28] Arie van Deursen and Paul Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10:2002, 2001.

[29] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic Regular Expression Explorer. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 498–507, April 2010.

Appendix A

Figures

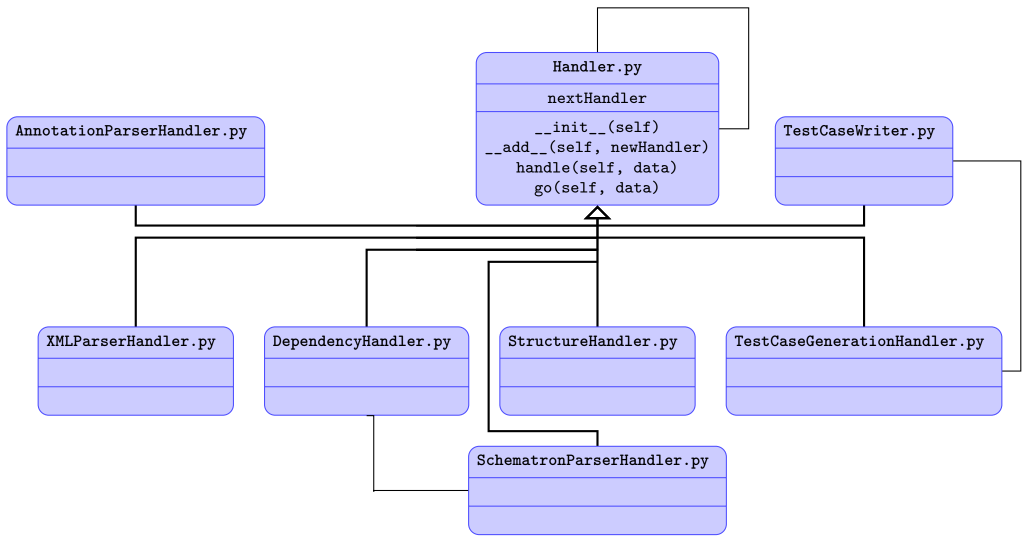


Figure A.1: Architecture of `coteca`.

Appendix B

SMT Encoding Example

B.1 Input Script

```
(declare-datatypes () ((Attribute null
                      (mk_Attribute
                       (value Int)
                      )
                    ))
)

;VlanPort
(declare-datatypes () ((VlanPort
                         (mk_VlanPort
                          (vlanPortId Attribute)
                         )
                       ))
)

;Router
(declare-datatypes () ((Router
                         (mk_Router
                          (routerId Attribute)
                          (ttl Attribute)
                         )
                       ))
)
```

```
;InterfaceIPv4
(declare-datatypes () ((InterfaceIPv4
                        (mk_InterfaceIPv4
                         (interfaceIPv4Id Attribute)
                         (encapsulation Attribute)
                         (loopback Attribute)
                         (bfdStaticRoutes Attribute)
                        )
                      ))
)

;INSTANCES
(declare-const VlanPort___1 VlanPort)
(assert (= (vlanPortId VlanPort___1)
           (mk_Attribute 1)
        )
)

(declare-const VlanPort___2 VlanPort)
(assert (= (vlanPortId VlanPort___2)
           (mk_Attribute 2)
        )
)

(declare-const Router___3 Router)
(assert (= (routerId Router___3)
           (mk_Attribute 3)
        )
)
(assert (and (<= 1 (value (ttl Router___3)))
             (<= (value (ttl Router___3)) 255)
        )
)

(declare-const InterfaceIPv4___4 (InterfaceIPv4))
(assert (= (interfaceIPv4Id InterfaceIPv4___4)
           (mk_Attribute 4)
        )
)
```

```
(assert (or (= (encapsulation InterfaceIPv4___4) null)
            (and (<= 1
                     (value (encapsulation InterfaceIPv4___4))
                 )
                 (<= (value (encapsulation InterfaceIPv4___4))
                     2)
            )
        )
)
(assert (or (= (loopback InterfaceIPv4___4) null)
            (= (loopback InterfaceIPv4___4)
               (mk_Attribute 0)
            )
            (= (loopback InterfaceIPv4___4)
               (mk_Attribute 1)
            )
        )
)
(assert (or (= (bfdStaticRoutes InterfaceIPv4___4)
               (mk_Attribute 0)
            )
            (= (bfdStaticRoutes InterfaceIPv4___4)
               (mk_Attribute 1)
            )
        )
)


(declare-const InterfaceIPv4___5 (InterfaceIPv4))
(assert (= (interfaceIPv4Id InterfaceIPv4___5)
           (mk_Attribute 5)
        )
)
(assert (or (= (encapsulation InterfaceIPv4___5) null)
            (and (<= 1
                     (value (encapsulation InterfaceIPv4___5))
                 )
                 (<= (value (encapsulation InterfaceIPv4___5))
                     2)
            )
        )
)
(assert (or (= (loopback InterfaceIPv4___5) null)
            (= (loopback InterfaceIPv4___5)
               (mk_Attribute 0)
            )
            (= (loopback InterfaceIPv4___5)
               (mk_Attribute 1)
            )
        )
)
(assert (or (= (bfdStaticRoutes InterfaceIPv4___5)
               (mk_Attribute 0)
            )
            (= (bfdStaticRoutes InterfaceIPv4___5)
               (mk_Attribute 1)
            )
        )
)

;DEPENDENCIES
(assert (<= (+ (ite (= (loopback InterfaceIPv4___4) null) 0 1)
               (ite (= (loopback InterfaceIPv4___5) null) 0 1)
            )
        64))
(assert (= (value (ttl Router___3)) 64))
(assert (distinct (encapsulation InterfaceIPv4___4)
                  (encapsulation InterfaceIPv4___5)
        ))

(assert (or (and (not (= (encapsulation InterfaceIPv4___4) null))
                 (not (= (loopback InterfaceIPv4___4) (mk_Attribute 1)))
            )
            (and (= (loopback InterfaceIPv4___4) (mk_Attribute 1))
                 (not (= (encapsulation InterfaceIPv4___4) null))
            )
         )
)
```

```
(assert (not (and (= (loopback InterfaceIPv4___4) (mk_Attribute 1))
                  (= (bfdStaticRoutes InterfaceIPv4___4)
                     (mk_Attribute 1))
             )
        )
)

(assert (or (and (not (= (encapsulation InterfaceIPv4___5) null))
                 (not (= (loopback InterfaceIPv4___5) (mk_Attribute 1)))
            )
            (and (= (loopback InterfaceIPv4___5) (mk_Attribute 1))
                 (not (= (encapsulation InterfaceIPv4___5) null))
            )
         )
)
(assert (not (and (= (loopback InterfaceIPv4___5) (mk_Attribute 1))
                  (= (bfdStaticRoutes InterfaceIPv4___5)
                     (mk_Attribute 1))
             )
        )
)

(check-sat)
(get-model)
```

B.2 SMT solver plain output

```
sat
(model
  (define-fun InterfaceIPv4___4 () InterfaceIPv4
    (mk_InterfaceIPv4 (mk_Attribute 4)
                  (mk_Attribute 1)
                  (mk_Attribute 1)
                  (mk_Attribute 0)))
  (define-fun VlanPort___1 () VlanPort
    (mk_VlanPort (mk_Attribute 1)))
  (define-fun VlanPort___2 () VlanPort
  (mk_VlanPort (mk_Attribute 2)))
(define-fun Router___3 () Router
  (mk_Router (mk_Attribute 3) null))
(define-fun InterfaceIPv4___5 () InterfaceIPv4
  (mk_InterfaceIPv4 (mk_Attribute 5)
              (mk_Attribute 2)
              (mk_Attribute 1)
              (mk_Attribute 0)))
```

Appendix C

XPath To SMT-LIB Mappings

Expression	**SMT-LIB**
$\neg a$	`(not a)`
$a \vee b$	`(or a b)`
$a \wedge b$	`(and a b)`
$a \otimes b$	`(xor a b)`
$a < b$	`(< a b)`
$a \leq b$	`(<= a b)`
$a > b$	`(> a b)`
$a \geq b$	`(>= a b)`
$a = b$	`(= a b)`
$a \neq b$	`(not (= a b))`
$a + b$	`(+ a b)`
$a - b$	`(- a b)`
$a * b$	`(* a b)`
$a \div b$	`(div a b)`
$a \mod b$	`(mod a b)`
`../A/B/@c` (path)	Will be resolved by Python.
`A[@c = 1]` (filter)	Depending on the filter, operators and functions will be

	expressed in SMT-LIB, if possible.
`count(a)`	Will be resolved by Python and inserted as an integer.

`are-distinct-values(a, b, c)`	`(distinct (value a)`
	`          (value b)`
	`          (value c))`

`matches("*.txt", a)`	see Section 4.5
`contains("txt", a)`	see Section 4.5
`exists(a)` and `a` is a `moRef`	`(= a x)` for each `x` to be the key of the pointing instance of `a`.
`exists(a)` and `a` is not a `moRef`	`(not (= a null))`
`string-length(a)`	Will be resolved by Python.

Appendix D

Code Listings

The abstract Handler class

```
## The Handler is the baseclass of the chain-elements
# in the chain of responsibility pattern.
class Handler(object):
        ## Constructor initializes the nextHandler to None
        def __init__(self):
                self.__nextHandler=None

        ## Overloaded list-append operator allows to form
        #  a chain (list) by linking together multiple
           handlers.
        def __add__(self, newHandler):
                if not isinstance(newHandler, Handler):
                        raise TypeError("Handler.__add__()
                             expects
                               Handler!")
                if self.__nextHandler:
                        self.__nextHandler+newHandler
                else:
                        self.__nextHandler=newHandler
                return self

        ## Should be called at the first chain-element.
        #  Performs the local handle of the data-object,
        #  before passing it to the next handler.
        #  Returns the changed object
        def go(args, data):
                ret=self.handle(data)
                if self.__nextHandler:
                        self.__nextHandler.go(ret)
                return ret

        ## Default implementation of the handler.
        #  Will be overridden by the different
        #  subclasses.
        def handle(self, data):
                return data
```

Appendix E

Evaluation Results

A	**B**	**C**	**D**	**E**	**F**	**G**
0	2	0.0050	2.4915	0.0829	16.6297	SAT
1	3	0.0062	2.0820	0.0666	10.6588	SAT
2	6	0.0096	1.6048	0.0737	7.6583	SAT
3	417	2.5724	6.1688	300.4867	116.8116	SAT
4	827	7.9414	9.6027	1247.3448	157.0677	TIMEOUT
5	1236	12.1816	9.8556	1244.6125	102.1718	TIMEOUT
6	9	0.0104	1.1565	0.0545	5.2324	SAT
7	215	0.7217	3.3566	18.0344	24.9899	SAT
8	420	2.6286	6.2586	258.8998	98.4923	SAT
9	625	3.7925	6.0680	663.4714	174.9422	SAT
10	12	0.0190	1.5833	0.0693	3.6453	SAT
11	150	0.6653	4.4354	6.9245	10.4080	SAT
12	287	1.4074	4.9038	36.8440	26.1792	SAT
13	423	2.1427	5.0654	212.4466	99.1499	SAT

Table E.2: Evaluation Results for test-case $\mathbf{T1_{M1}}$, where **A** is the number of the test-case, **B** is the number of instances, **C** is the time for code preparation (in seconds), **D** is the average code preparation time per instance (in milliseconds), **E** is the SMT solving time (in seconds), **F** is the average SMT solving time per instance (in seconds) and **G** is the result of the SMT solver.

A	**B**	**C**	**D**	**E**	**F**	**G**
0	2	0.0009	0.4450	0.1067	53.3280	SAT
1	79	0.0821	1.0392	0.1462	1.8508	SAT
2	80	0.0944	1.1802	0.1557	1.9460	SAT
3	81	0.0861	1.0634	0.1476	1.8222	SAT
4	156	0.1556	0.9971	0.2496	1.6002	SAT
5	157	0.1470	0.9366	0.2361	1.5035	SAT
6	158	0.1631	1.0320	0.2314	1.4648	SAT
7	233	0.2425	1.0406	0.3189	1.3686	SAT
8	234	0.1904	0.8135	0.2682	1.1460	SAT
9	235	0.2027	0.8624	0.2340	0.9956	SAT

Table E.3: Evaluation Results for test-case $\mathbf{T2_{M2}}$, where **A** is the number of the test-case, **B** is the number of instances, **C** is the time for code preparation (in seconds), **D** is the average code preparation time per instance (in milliseconds), **E** is the SMT solving time (in seconds), **F** is the average SMT solving time per instance (in milliseconds) and **G** is the result.