A Domain-Specific Language for Traceability in Modeling

Anisur Rahman

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

Master of Applied Science
in Electrical and Computer Engineering

Under the auspices of the
Ottawa-Carleton Institute for Electrical and Computer Engineering

uOttawa

University of Ottawa
Ottawa, Ontario, Canada
July 2013

Abstract

Requirements are a key aspect of software development. Requirements are also related with other software artefacts including designs, test cases and documentation. These artefacts are often captured with specialized *models*. However, many tools lack support for *traceability* relationships between requirements artefacts and model artefacts, leading to analysis issues. To establish traceability between models and other types of requirements artefacts, this thesis proposes a new *Domain-Specific Language* (DSL) for describing the concepts of a modeling language that would be intended to be traced using a *Requirements Management System* (RMS), with tool support handling the evolution of models and of their traceability links.

In the first part of this thesis, the syntax and metamodel of the Model Traceability DSL (MT-DSL) are defined, together with an editor implemented using Xtext. In the second part of the thesis, a library of import and maintenance functions is generated automatically (using Xtend) from model traceability descriptions written using MT-DSL. The target language for this library is the *DOORS eXtension Language* (DXL), the scripting language of a leading commercial RMS with traceability support, namely IBM Rational DOORS.

The implementation has been tested successfully for importing and evolution scenarios with two different modeling languages (User Requirements Notation and Finite State Machines).

This work hence contributes a reliable mechanism to define and support traceability between requirements and models.

Acknowledgment

I would like to express my deepest gratitude to my supervisor, Dr. Daniel Amyot, for his supervision, guidance, advice, and comments. His suggestions, detailed comments, and patient review allowed me to complete and implement this thesis.

I would like to express my gratitude to my parents for their endless support and love. I also like to thank my brothers and sisters for their good wishes.

Finally, I would like to dedicate this thesis to my wife Shaheen, my daughter Afrin and my son Asif. I would also like to specially thank my wife for her encouragement and for helping me manage the time for the thesis.

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Definition
AoURN	Aspect-oriented User Requirements Notation
API	Application Programming Interface
AST	Abstract Syntax Tree
DOM	Document Object Model
DOORS	Distributed Object-Oriented Requirements System
DSL	Domain-Specific Language
DXL	DOORS eXtension Language
EMF	Eclipse Modeling Framework
EVL	Epsilon Validation Language
FSM	Finite State Machine
GMF	Graphical Modeling Framework
GPL	General Purpose Language
GRL	Goal-oriented Requirement Language
GUI	Graphical User Interface
ID	Identifier
IDE	Integrated Development Environment
jUCMNav	Java Use Case Map Navigator
JVM	Java Virtual Machine
MDE	Model Driven Engineering
MT-DSL	Model Traceability Domain Specific Language
MWE	Modeling Workflow Engine
nsURI	namespace Uniform Resource Identifier
OCL	Object Constraint Language
OMG	Object Management Group
OSGi	Open Services Gateway initiative
ReqIF	Requirement Interchange Format
RMS	Requirements Management System
RS	Requirements Specification
SDK	Software Development Kit
TML	Traceability Metamodeling Language
UCM	Use Case Map
UML	Unified Modeling Language
URN	User Requirements Notation
VTML	Visual Trace Modeling Language

1 Introduction

This thesis explores the creation and management of traceability relations between models in modeling tools and other software artefacts using a Domain-Specific Language (DSL). Modeling tools usually do not offer support for external traceability creation and management. Traceability support for models described in modeling tools can be achieved by exporting the models to a Requirements Management System (RMS), which then offers traceability support. In our new approach, models or subsets of models with traceability requirements are characterized using a Model Traceability DSL (MT-DSL). Models described with MT-DSL can be imported in an RMS using a library of functions that is automatically generated from the MT-DSL description itself. Traceability links between the imported model and other artefacts can then be maintained inside the Requirements Management System, even as the model and related artefacts change over time.

1.1 Context and Motivation

Modern software development approaches often involve many types of artefacts, including requirements, designs, test cases, and documentation. Some artefacts are captured with structured text (e.g., in English), others with specialized languages or models. There is a need to document various traceability relationships amongst these artefacts. This is even more important in a change management context, as modifications to an artefact might cause ripple effects on many other artefacts linked directly or indirectly to it. For example, a change to a requirement might lead to changes to a scenario model, and then to the design and corresponding test cases. Similarly, a change to a model might also have an impact on linked requirements.

In modern software engineering, models are widely used artefacts to describe abstract representations of various aspects of software. This is especially important in the requirements and design phases. The Unified Modeling Language (UML), standardized

by the Object Management Group (OMG), is widely used to describe software designs [38]. The User Requirement Notation (URN) is also a well-known modeling language to graphically describe user requirements [1][3][25]. To facilitate the creation, analysis and management of models, numerous commercial and open source tools are available. Some limited version of *internal* traceability is often supported by such tools. For example, jUCMNav [28][33], a URN modeling tool, provides limited traceability support between the elements of one URN model. However, modeling tools usually do not provide any traceability support to/from artefacts *external* to the models they handle. For instance, jUCMNav does not have mechanisms to link URN elements from one model to elements from another URN model, or to elements of a UML model. Yet, traceability of models to artefacts that come before models or after them in a software development process is important. Gotel and Finkelstein argued that the majority of poor requirements problems are due to inadequate specification traceability [16]. Traceability across modeling artefacts and tools is usually achieved through the use of external tools, for example with a Requirements Management Systems (RMS).

An RMS is a software application that provides support for managing and analyzing evolving requirements. An RMS also provides traceability, change management, and impact analysis. Requirements and models can be exported from modeling tools and imported in an RMS, and thus the RMS enables traceability between artefacts from multiple sources and representations. IBM Rational DOORS is one of the world-leading RMS tools [15][18]. DOORS provides features that are needed to capture, track and manage requirements and other types of (modeling) objects. An RMS often provides application programming interfaces (APIs) for extending its capabilities, for customization, and for integration with other tools. For example, DOORS provides the DOORS eXtension Language (DXL) [21], a C-like scripting language developed for manipulating DOORS objects.

DXL can be used as an interface to link DOORS with modeling tools. For example, Jiang [27][41], Roy [42][43], and Ghanavati [11][12][13][14] have incrementally developed a mechanism to import into DOORS URN models created with jUCMNav. jUCMNav exports a URN model as a DXL script that invokes a predefined DXL library of object creation and evolution functions. When executed, these functions create

DOORS objects corresponding to URN model elements, and these objects can then be linked to/from other DOORS objects, whatever their sources.

However, this DXL library is specific to a subset of URN and to jUCMNav. It is not easy to maintain, and does not support other modeling languages (e.g., UML) or modeling tools.

1.2 Problem Statement

There are several problems that prevent the simple manipulation of models and traceability links in an RMS. Different subsets of a modeling language can be targeted for import and traceability, different modeling languages should be easily supported, and the creation of a library of functions in the RMS should be automated. The main problem statement in this thesis is hence: *Can we characterize formally the input modeling language and the traceability relationships of interest such that we can automate the import of models in an RMS?*

1.3 Objectives

The main objectives of this thesis are the following:

1) Find a generic way of expressing the aspects of a modeling language that are interesting to track, and define it as a tool-supported Domain-Specific Language;
2) Provide an interface that can be used by modeling tools to export their models to an RMS, and define it as a library of functions that can be invoked from a scripting language;
3) Implement this for a leading RMS, namely DOORS and its DXL scripting language.

1.4 Research Methodology

The research method employed for this thesis is based on *Design and Creation* [44]. This is a common research method used for solving problems in Information Technology (IT) products, where the end result is a computer-based system. The steps in this research method are awareness, suggestion, development, evaluation and conclusion:

1.4.1 Awareness

This is the recognition of the problem. Based on Sections 1.1 and 1.2, we can summarize it as follows:

- Traceability is important in software engineering.
- Models are common software engineering artefacts, but traceability between models, or between models and other types of artefacts, usually requires the use of third-party tools such as Requirements Management Systems.
- There exists previous work related to jUCMNav regarding the import of URN models in IBM DOORS, but this is not generalizable to other subsets of the language or to other languages without having different DXL libraries, each of which requiring much effort to write and maintain.

1.4.2 Suggestion

This is the proposal of a solution based on the issues identified from awareness. We have surveyed existing works and different approaches taken to provide traceability in modeling. We have analyzed the gaps of different solutions to suggest a more affordable solution for modeling tools.

While different approaches are proposed on how modeling tools can provide and maintain traceability, there is no existing solution proposing how modeling tools can quickly provide and maintain traceability support avoiding significant development. We have analyzed the possibility minimizing the development effort by exploiting the API provided by RMS tools. We have identified the necessity of automating the library generation for a tool to use the RMS API.

As the metamodels of different modeling languages are different, the models supported by different modeling tools are also different. At the beginning, we started by experimenting with an approach based on configurations used in a properties file. The objective was to support the wide variation in models in modeling tools using configurations. This approach quickly became bulky, and our resulting suggestion is to create of a Model Traceability Domain-Specific Language (MT-DSL) for describing the model artefacts and links required to be tracked in an RMS.

1.4.3 Development

We have developed the following items to implement the suggestion:

- Definition of MT-DSL (with Xtext), together with the implementation of an advanced language editor;
- Definition of transformations from MT-DSL to a target RMS scripting language (DOORS DXL);
- Implementation of the transformation (with Xtend and Java);

The development of MT-DSL is described in Chapter 3 whereas the development of the transformation to DXL is described in Chapter 4.

1.4.4 Evaluation

We have validated the practicality and usefulness of the language and transformation using two major test cases in Chapter 5.

These test cases target two different modeling languages, namely Finite State Machines and the User Requirements Notation. For each modeling language, the MT-DSL description is provided (demonstrating that several languages and their concepts can be supported), together with a DXL library automatically generated (to validate the transformation). The MT-DSL editor was used to create these descriptions.

To test each library, a model is created with this DSL and then imported into DOORS for validation. Then, to further test advanced features of the generated libraries, the models are modified (i.e., with the addition, deletion, and update of several model elements), and reimported into DOORS with new versions of modified objects and preservation of existing traceability links for non-deleted objects. Links between an FSM model and a URN model are also created, before evolving the models and assessing the impact on traceability links involving elements that are modified or deleted.

Note that the subset of the User Requirements Notation that was selected corresponds to the existing subset used in jUCMNav, for a more direct comparison with existing work.

1.4.5 Conclusion

With the ‘Evaluation’ of the ‘Development’ using the complex test cases, we have concluded that the approach described in the ‘Suggestion’ is feasible, practical and useful. We have a cost-effective and maintainable solution for tackling the model traceability issues identified in the ‘Awareness’ part. We have also replicated the essential functionalities of Jiang [27][41] and Roy [42][43] using the DSL and transformation implemented in this thesis. Additionally, it will now be simple for jUCMNav to support additional traceability models without requiring any manual DXL development. Other modeling tools and languages can also now benefit from this new technology.

1.5 Contributions

The context describing where this thesis fit is shown in the following two figures. Figure 1 shows that MT-DSL is meant to be used manually (but with support from an Eclipse-based editor) to capture the parts of a modeling language that are of interest for model traceability. The MT-DSL description of this modeling language is used to generate a library of DXL functions. Figure 2 shows that a model in some modeling tool can be converted automatically to a DXL script, which when run invokes the functions of the DXL library and imports (or re-imports) the model to the DOORS RMS.

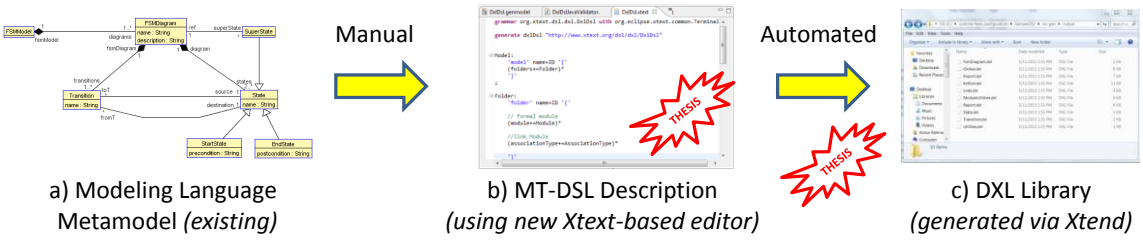

Figure 1 Generation of a DXL library for the MT-DSL description of a traceability-oriented view of a modeling language

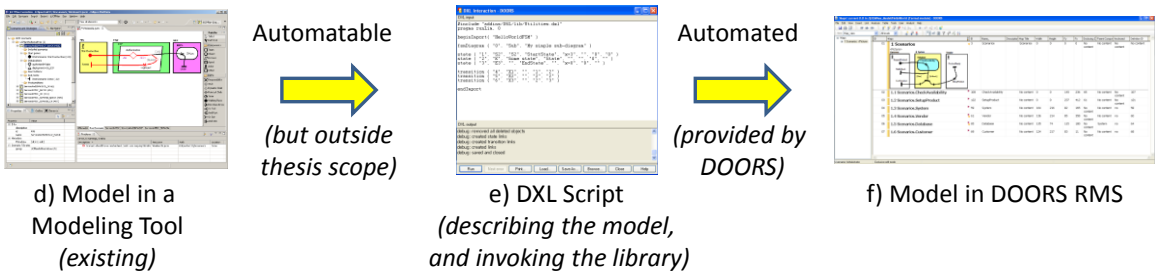

Figure 2 Use of the DXL library by DXL scripts generated by modeling tools for importing and re-importing models in the DOORS RMS

This thesis offers two important contributions. The first one is the Model Traceability Domain-Specific Language (used in Figure 1*b*) for describing traceability to be tracked in modeling languages, independently of the target Requirements Management System. MT-DSL is supported by a sophisticated editor implemented as an Eclipse plug-in, which provides content assistance, syntax highlighting and error highlighting with quick fixes.

The second contribution is a tool-supported transformation targeting the generation of libraries for a specific (but very popular) Requirements Management System. In this thesis, DXL code is generated for a traceability model described with MT-DSL (from *b* to *c* in Figure 1) so that the models created with a modeling tool (including their diagrams) can be seamlessly imported into DOORS (via a simple export from the modeling tool that transforms the model to a DXL script).

1.6 Thesis Outline

This thesis is organized as follows. In Chapter 2, we introduce the tools and technologies relevant to this thesis, including DOORS, DXL, Xtext, and Xtend. This chapter also introduces and discusses related work in the traceability domain.

Chapter 3 describes the new Xtext-based Model Traceability Domain-Specific Language used to describe the models to be traced. This chapter also describes the Eclipse-based editor supporting the DSL.

Chapter 4 defines the traceability library generation process and describes how we have automated the DXL code generation for models using Xtend.

Chapter 5 focuses on validating this work. MT-DSL is used to describe models for two different modeling languages, to generate DXL libraries, and to test its import and evolution features in relation with DOORS. This corresponds to going from *e* to *f* in Figure 2 (more than once). Threats to the validity of this work are also discussed.

In Chapter 6, the main contributions are recalled, together with a brief comparison with related work. This chapter also describes future work items that can enhance the features that have been described in this thesis.

2 Background and Related Work

This chapter provides background information about the tools and technologies that are used for this thesis such as traceability management, Requirements Management Systems (with a focus on DOORS), and Eclipse-based Domain-Specific Language (DSL) technologies, including Xtext and Xtend. This chapter also describes related work in traceability DSLs.

2.1 Traceability

Traceability provides a logical connection between artifacts of the software development process [15][30]. Traceability provides the ability to follow a specific item at the input of a phase of a software lifecycle to a specific item at the output of that phase [15]. One of the most common definitions of traceability, found in [22], is:

> *The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example the degree to which the requirements and design of a given software component match.*

Traceability is important for quality-oriented software development. For requirements management, traceability provides the ability to link product requirements to corresponding design, code and test artefacts. In change management, traceability provides important information about the possible consequences of a changing requirement, for instance through impact analysis [30]. Traceability is also important for project management in order to have a better understanding of the project's progress.

2.1.1 Traceability Maintenance

Despite being widely accepted as beneficial, the costs associated with traceability can be high and due to complexity in traceability management caused by the heavy and error-prone manual effort required, the return on investment remains debatable [29]. Unless imposed by regulations (e.g., for aviation and military applications), traceability is rarely used throughout all development stages because of the high number of artifacts involved, and because of the maintenance effort required each time a change occurs. Requirements Management Systems (RMS) usually provide support for traceability maintenance for the artifacts that are selected to be managed by developers.

Mäder and Gotel [30] provide an exhaustive and mature description of traceability maintenance as described in Figure 3.

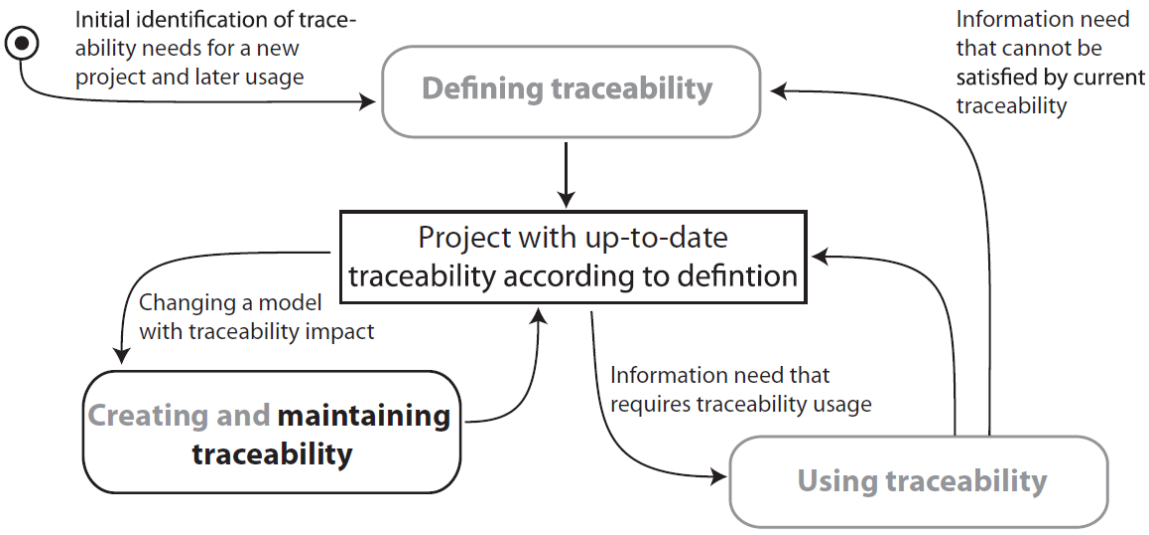

Figure 3 Traceability life cycle (from Mäder and Gotel [30])

Mäder and Gotel describe a semi-automated way of maintaining traceability by capturing flows of events in a prototype modeling tool [30]. This approach depends on Event Based Traceability concepts described by Cleland-Huang et al. [4]. Mäder and Cleland-Huang also introduced an expressive Visual Trace Modeling Language (VTML) [29]. VTML simplifies traceability queries and avoids the redundant use of internal data structure compared to other technologies such as XQuery.

Mirakhorli et al. worked on developing a Tactic-Centric Approach for automating the traceability of quality concerns [32]. In this approach a tactic classifier identifies all classes related to a given tactic and then establishes tactic-level traceability through mapping those classes to the relevant tactic. Then, a more finely-tuned classifier is used in conjunction with lightweight structural analysis to identify the subset of classes that play clearly-defined roles in the tactic. This approach works with architectural tactics built upon the concept of tactic traceability information models.

2.1.2 Models and Traceability

In the early phases of software development, several categories of requirements are often expressed in abstract terms using models. Modeling is also used to analyse natural language requirements, and to design solutions prior to implementation. Traceability of these model artefacts is important to properly ensure the coverage of requirements by the design and associated implementation, and to assess the impact of a change to the model on the associated requirements, or vice versa. However, existing modeling tools are still lacking good support for traceability maintenance for the model artifacts that are managed outside the scope of an RMS.

In terms of concepts from Figure 3, this thesis focuses on *defining* traceability (with a DSL) and on *creating and maintaining* traceability of *models* automatically in a requirements database. In contrast, Mäder and Gotel [30] focus on a semi-automated way of *maintaining* traceability, VTML [29] focuses on *using* traceability (through visual queries), and the work of Mirakhorli et al. [32] focuses on *creating* traceability links for software *quality* aspects.

2.2 Requirements Management Systems (RMS)

Requirements Management Systems (RMS) are software applications that provide support for managing and analyzing evolving requirements. An RMS also supports traceability, change management, and impact analysis. Some Requirements Management Systems enable extensions and automation with scripting languages. The International Council on Systems Engineering (INCOSE) maintains an interesting database of RMS, with comparative survey results based on multiple criteria [23].

This section focuses on IBM Rational DOORS [18][19][20], one of the most popular RMS. DOORS provides extensibility by supporting the scripting language DXL (DOORS eXtension Language) [21].

We have chosen DOORS as the target RMS tool for this thesis, given its previously demonstrated capability to support the import of models. The DOORS API has been used to import traceability models from modeling tools [19]. DOORS is already integrated with jUCMNav, and URN models from jUCMNav can be exported to DOORS [14] via DXL scripting. We will explore the possibility of automating the DXL code generation for importing models from modeling tools for the purpose of this thesis [21]. We will also explore the possibility of essentially replicating the existing DXL library for jUCMNav by generating the library automatically without manual DXL development.

2.2.1 IBM Rational DOORS

IBM Rational DOORS, one of the leading requirements management tools [23], provides various features needed to capture, track and manage requirements [18]. DOORS uses a client-server architecture and a revision control system to manage text objects, diagrams and documents specifying requirements [20][43]. Requirements can be directly entered into DOORS, using its familiar word processor style interface. DOORS also allows importing requirements from many file formats, including plain text, Rich Text Format (RTF) and Microsoft Office applications (Word, Excel and PowerPoint).

DOORS uses a very specific structure to store requirements. Requirements data is stored in *databases*. Users can create hierarchical structures of *folders* and *projects* in a database, with different access control privileges. The requirements information in DOORS is stored in *modules*. In DOORS projects, *formal modules* contain the requirements objects including text and images. Object characteristics are stored using *attributes* in a formal module. Attributes allow additional information to be associated with each requirement object [20], for instance its priority or version. DOORS comes with many pre-defined attributes, but users can customize their own.

A relationship between two objects in DOORS is established using a *link*. Links are stored in *link modules* [19]. A link connects a source object to a target object and can

be followed in either direction. Link modules also store additional object characteristics using attributes, including the link type.

Figure 4 shows a sample formal module containing different objects describing a URN model. The object attributes are displayed with columns whereas the presence of incoming/outgoing traceability links is indicated with small yellow/red triangles. Objects can be textual or graphical.

One important characteristic of DOORS is its support for *suspect links*, which indicate that the object at the other end of a link (be it source or destination) was modified. Such indications help users focus their attention on modifications that can impact them.

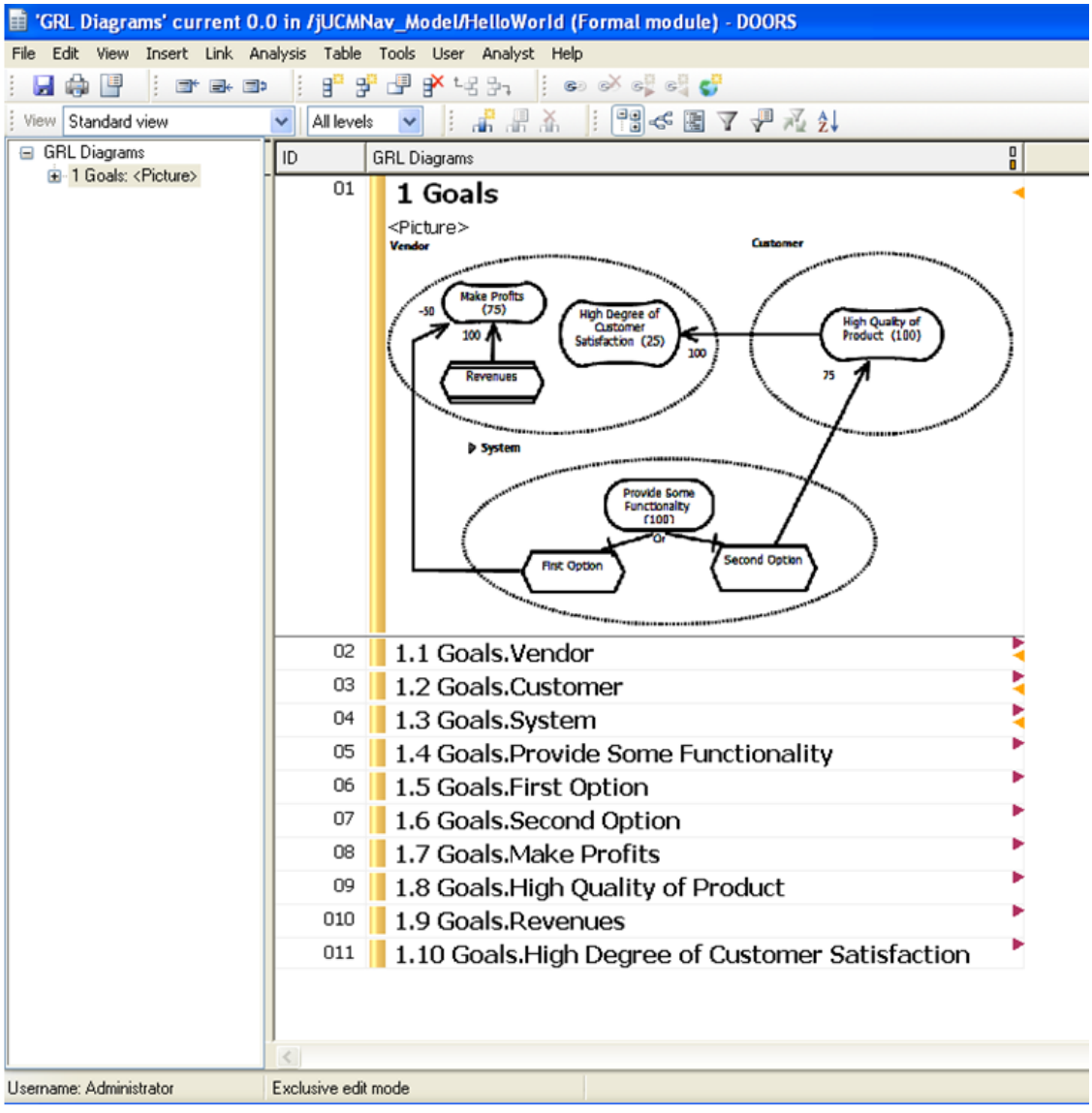

Figure 4 Formal module of a URN model in IBM Rational DOORS

2.2.2 DOORS eXtension Language – DXL

DOORS provides application programming interfaces (APIs) for capability extensions, customization, automation, and linking to other tools. The main interface is the DOORS eXtension Language (DXL) [19][43]. DXL is a scripting language that can be used to manipulate objects and links, and to connect DOORS with external tools such as configuration management databases [21]. DXL also allows the creation of reusable functions.

DXL applications provide end users with a seamless extension to the DOORS Graphical User Interface (GUI) [21]. DXL users can develop custom DXL scripts to automate changes to DOORS objects and their attributes that are commonly done manually using the GUI. DXL takes its syntax and many of its features from C and C++ [21][43].

The code in Figure 5 is an example DXL function that uses recursive calls. The code describes a utility function that finds an ancestor object with a given ID for a given object. The code first verifies if the provided ID matches the ID of the object itself. If it does, it returns the given object. If not, it calls the method recursively using the ancestor ID, and the parent of the given object.

```
Object findAncestor( string ancestorID, Object theObject ) {
    Object ancestorObject = null

    if ( theObject."ID" "" == ancestorID )
        ancestorObject = theObject
    else
        ancestorObject = findAncestor( ancestorID, parent( theObject ) )
    return ancestorObject
}
```

Figure 5 Sample DXL code from jUCMNav's DXL library

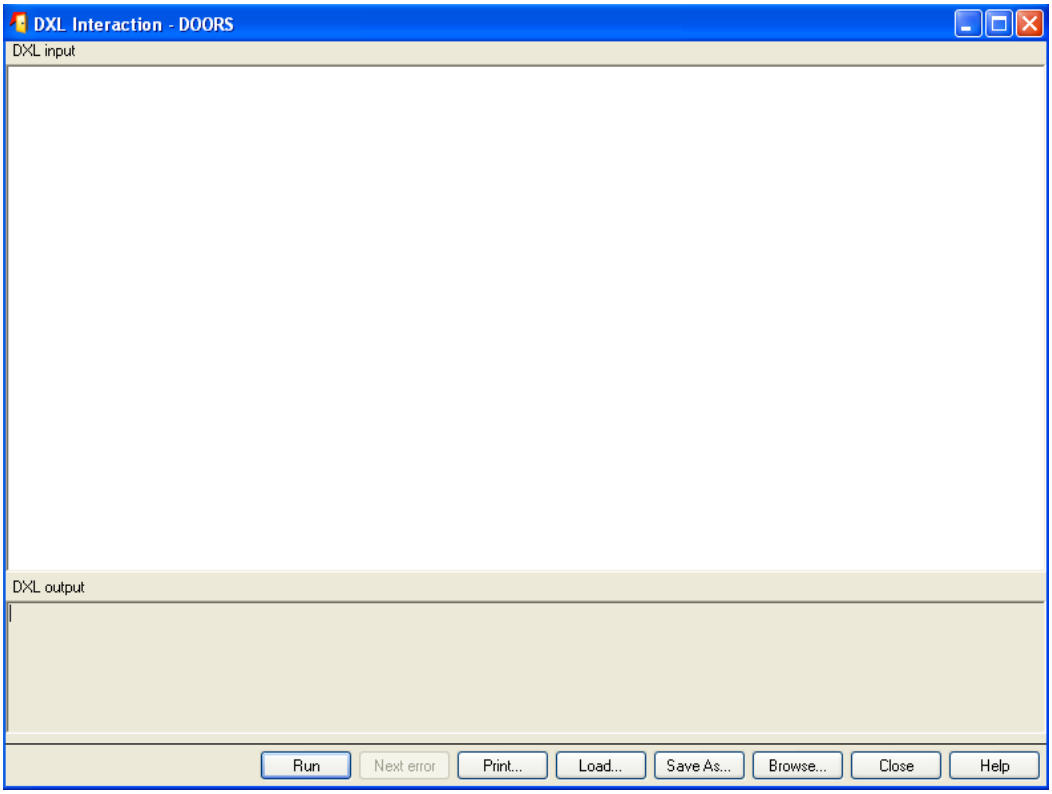

Figure 6 DXL interaction window

DOORS provide an interaction window (Figure 6) where DXL scripts can be loaded and executed. The input DXL script in the interaction window can call libraries of custom DXL functions along with built-in DXL functions. Custom DXL code executed here can be used to create objects inside DOORS and manage them externally.

One objective of this thesis is to generate DXL libraries for languages supported by various modeling tools, in an automated way. We will be generating DXL function *libraries* for the traceability models described using the DSL introduced in this thesis. In this approach, the modeling tools (e.g., jUCMNav, some UML tool, or some FSM tool) must generate the *scripts* capturing the model elements and relationships to be traced in DOORS. The scripts will simply be composed of a sequence of calls that will invoke the library functions. The generation of the scripts themselves by the modeling tools, although simple, is outside the scope of the thesis as this requires modifying these tools or creating export filters.

2.3 Domain-Specific Languages (DSLs)

Fowler defined a Domain-Specific Language (DSL) as "a computer programming language of limited expressiveness focused on a particular domain" [10]. In contrast to a General Purpose Language (GPL) such as Java [26], a DSL is a language used to describe concepts in a particular domain, which can be more or less anything [48]. GPLs on the other hand can be used to solve any computer problems, but a specific GPL might not represent the best way to solve a specific problem, and DSLs can help here.

DSLs are meant to be easy to use and substantially expressive in the domain of their application [31]. A DSL is precise and easy to understand by stakeholders as it only describes one specific domain of interest [10]. Deursen claims that a DSL provides a better solution than a GPL for a specific domain of interest and can be used in software engineering to enhance quality, flexibility and timely delivery of systems [6]. DSL also allows performing validation at the domain level [6].

Mernik argued that using a DSL increases productivity and reduces maintenance cost by offering substantial gain in expressiveness [31]. DSLs can make a complicated block of code more concise and easier to understand, and thus improves productivity. Fowler argued that the most common source of project failure is related to the communication with the customers and users of that software [10]. A DSL helps improving this communication by providing a precise language to deal with the particular domain of interest.

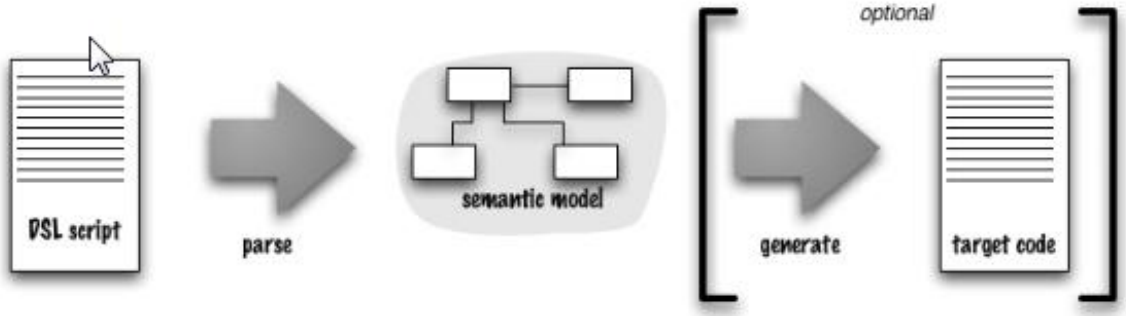

Figure 7 The overall architecture of DSL processing (from Fowler [10])

Fowler provided a broad structure of DSL implementations [10], described in Figure 7. He describes a DSL as a thin layer over the semantic model where all semantic behaviour of a model is captured. The DSL is to populate that model through parsing steps. The se-

mantic model is described here as a subset of any domain model. Domain models are used by many software systems to represent the core concepts and behaviour of the system. Even though not all parts of the domain model are best handled by DSLs, often a significant portion of such model can be expressed with a DSL [10]. The semantic model can represent any data structure including object models, and can be manipulated similar to any other normal object model. As described in Figure 7, a DSL is of often associated with optional code generation for a target platform.

Two common drawbacks of DSL include their proliferation (which implies that developers need to know many more languages) and the availability of good tool support such as editors, parsers, compilers, and debuggers. Recent Eclipse-based technologies now available help address this last drawback by supporting the construction of editors and parsers (Xtext) and of transformation tools for code generation (Xtend). These tools will be explored further in the next two sections.

2.4 Xtext

In Chapter 3, we will use Xtext to implement the new DSL. Xtext is an Eclipse-based framework for programming language development and for DSLs [47]. Xtext is a sophisticated framework to help implementing a DSL with proper Integrated Development Environment (IDE) support [48].

Xtext was originally developed under the openArchitectureWare (oAW) project in 2006 [39]. Xtext is now part of the Eclipse Modeling Framework (EMF) project [8]. The last version of Xtext under oAW is 4.3.1. The oAW website currently has documentation for older versions and documents [39]. Releases of new versions of Xtext are now done as part of the annual Eclipse release.

Xtext not only facilitates building a DSL, it can support the construction of an Eclipse-based editor for the language. Xtext covers most aspects of language features including language infrastructure, parser, and compiler or interpreter. Xtext also allows customizing these features according to individual needs.

2.4.1 Xtext-Based Editor Features

Xtext-based editors include many language usability features such syntax coloring, content assistance, validation and quick fixes:

Syntax Coloring: Xtext facilitates developing an Eclipse-based editor with out-of-the-box support for syntax coloring as shown in Figure 8. Xtext supports customizing the highlighting as needed for users. Xtext syntax coloring is based on the lexical structure and the semantic data [47].

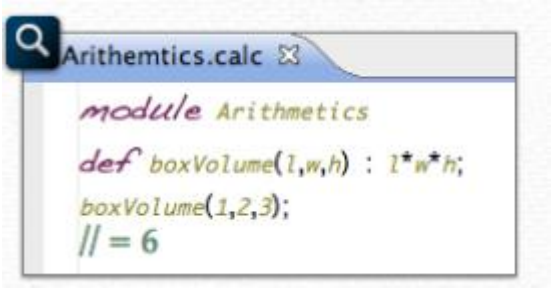

Figure 8 Xtext syntax coloring (from Xtext [47])

Content Assistance: An Xtext editor also implements content assistance, the feature implemented by most programming editors nowadays (Figure 9). Xtext can help propose valid code completions at any point in the editor. The content assistance helps the users cope with the syntactical details of the language [47].

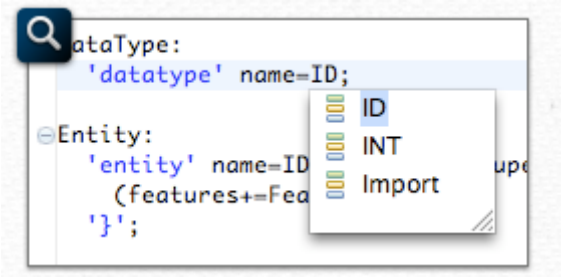

Figure 9 Xtext content assistance (from Xtext [47])

Validation and Quick Fixes: Xtext-based editors provide model validation with outstanding support for analyzing the static model of a DSL program. With custom quick fixes (Figure 10), the validation errors can be fixed with quick key strokes [48].

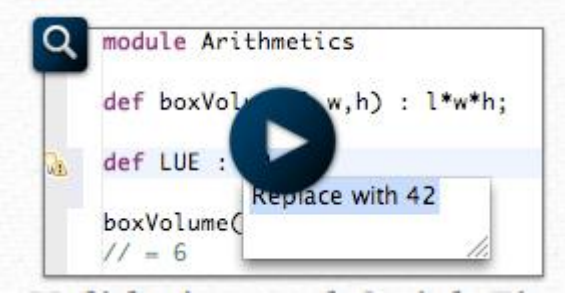

Figure 10 Xtext validation and quick fixes (from Xtext [47])

2.4.2 Grammar Mixing

Xtext supports reusing existing grammars [48]. Xtext provides a default grammar (org.eclipse.xtext.common.Terminals), and any Xtext grammar that is created using the Xtext wizards uses this grammar by default. This default grammar provides a common set of terminal rules for grammars that are generated using Xtext, and defines reasonable defaults for hidden terminals to be used in the grammar.

```
grammar org.xtext.dsl.dxl.DxlDsl
with org.eclipse.xtext.common.Terminals
```

In the above grammar description the language called `org.xtext.dsl.dxl.DxlDsl` reuses the existing default grammar org.eclipse.xtext.common.Terminals. To reuse an existing grammar, the grammar files needs to be in the class path of the inheriting language. It is also possible to inherit a grammar description from a different plug-in where a plug-in dependency needs to be added in the plug-in's MANIFEST.MF file [48].

2.4.3 EMF Model

Xtext uses EMF models to create in-memory object graphs while processing text. The in-memory object graph is called Abstract Syntax Tree (AST), see for instance Figure 11. Depending on the community, this concept is also called Document Object Model (DOM), model or semantic model [48].

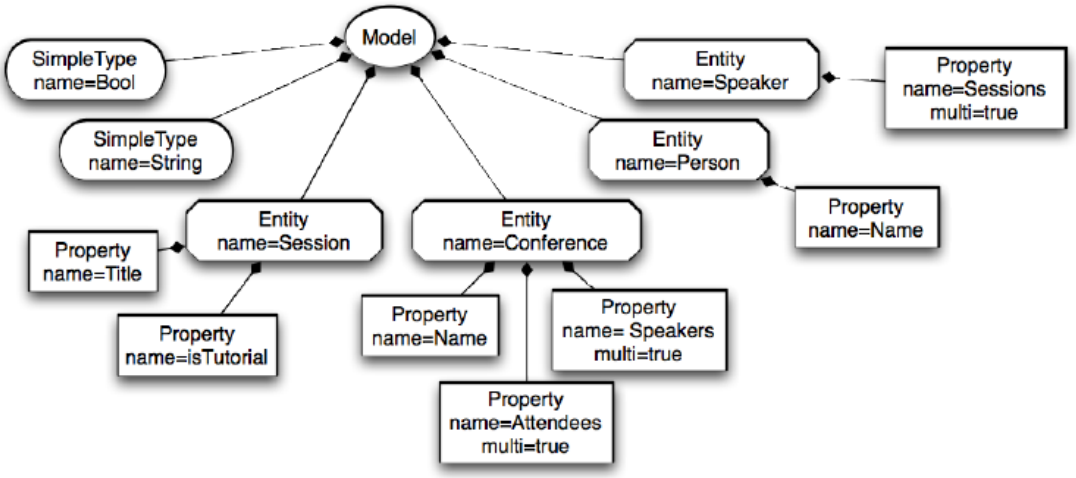

Figure 11 Sample AST (from Xtext [48])

The EMF Ecore metamodel contains an EPackage consisting of important conceptual building blocks such EClasses, EDataType and EEnums, as shown in Figure 12 [48].

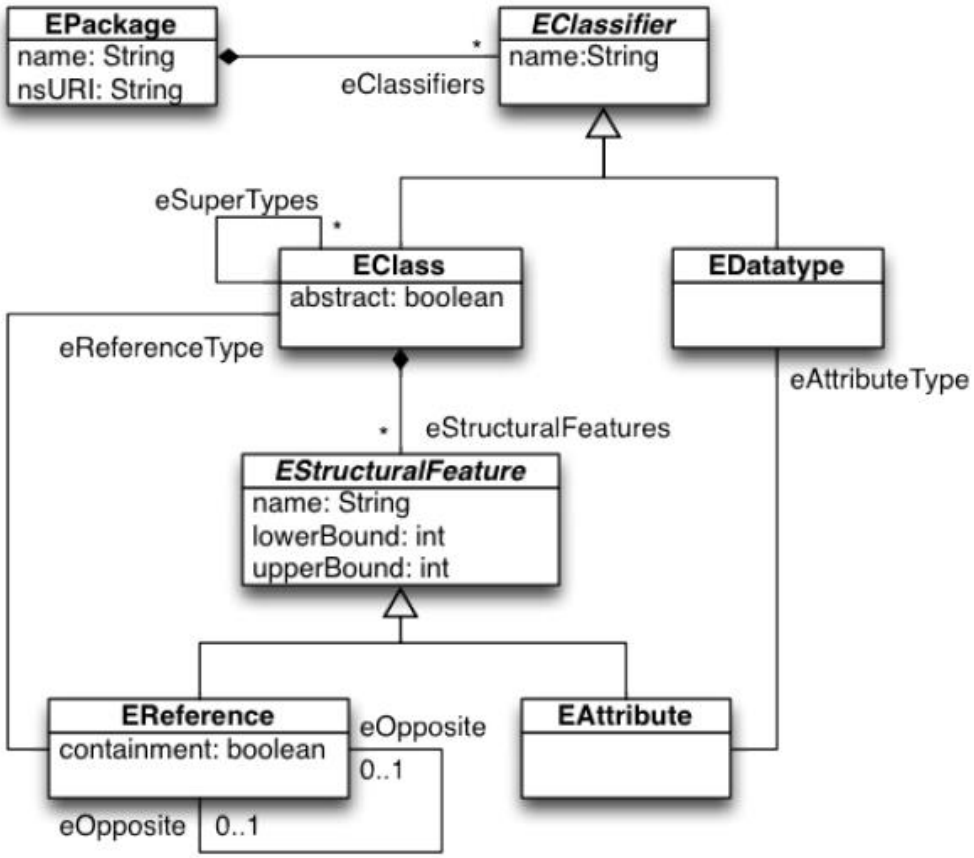

Figure 12 Ecore concepts (from Xtext [48])

Xtext provides an implementation of Ecore resources that encapsulates the parser that converts text to an EMF model and the sterilizer working in the opposite direction [48]. An Xtext model thus looks like any other Ecore-based model from the outside and makes seamless integration with other EMF-based tools. Figure 13 describes XtextResource, which is the EMF resource implementation of Xtext models.

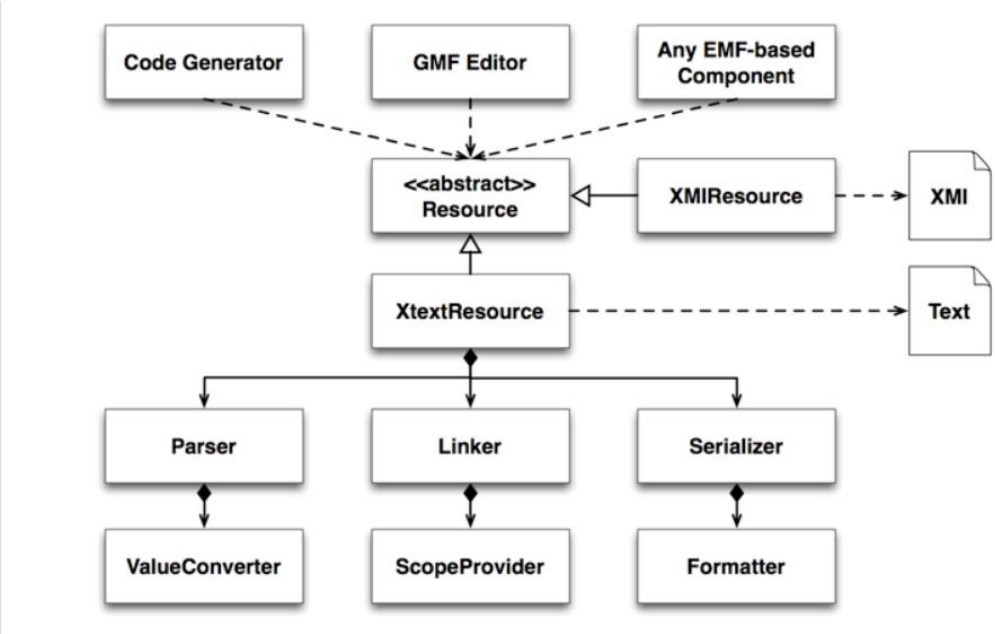

Figure 13 Xtext EMF resource implementation (from Xtext [48])

Xtext can infer the Ecore model automatically from the DSL syntax, or an existing Ecore model can be imported. The `generate` declaration in the grammar is used to advice the Xtext framework to infer the EPackage.

```
generate dxlDsl "http://www.xtext.org/dsl/dxl/DxlDsl"
```

The statement above advises the framework to generate an EPackage with the name 'dxlDsl' and with the *namespace Uniform Resource Identifier* (nsURI) 'http://www.xtext.org/dsl/dxl/DxlDsl". The nsURI is also used to import existing EPackage [48].

2.5 Xtend

In Chapter 4, we will use Xtend to automate the DXL library generation based on the DSL model. Xtend is a programming language that compiles into idiomatic Java source code [26][45]. Although Xtend has Java-based syntax and semantics, it has improvements on many features compared to Java [46].

Xtend has been implemented on top of Xtext and removes unnecessary syntactical noise. Xtend provides good default visibility by removing semicolons and empty parentheses. Xtend resembles Java's type system with no compromises in order to avoid any interoperability issue.

The code written in Xtend looks very similar to Java. As seen in Figure 14, there are no empty parentheses at the end of the first line of the sample code. The conventional semicolons at the end of the statements are also absent.

```
class HelloWorld {
  def static void main(String[] args) {
    println("Hello World")
  }
}
```

Figure 14 Sample Xtend code (from the Xtend User Guide [46])

The Xtend code shown in Figure 14 is seamlessly converted to the Java code shown in Figure 15.

```
// Generated Java Source Code
import org.eclipse.xtext.xbase.lib.InputOutput;

public class HelloWorld {
  public static void main(final String[] args) {
    InputOutput.<String>println("Hello World");
  }
}
```

Figure 15 Xtend code converted to Java (from the Xtend User Guide [46])

This “hello world” program does not necessarily unveil any strength of Xtend. Xtend has however many features useful for handling languages and editors, including: lambda expressions, improved operator overloading, powerful type-based switch expressions, multiple dispatch, template expressions with intelligent white space handling, and shorthands for accessing and defining getters and setters.

2.6 Traceability DSLs

With the advent of DSLs, it is now common for more than one modeling language to be used simultaneously for a same Model Driven Engineering (MDE) process. An MDE process typically involves different models of potentially overlapping views of a system. Thus the traceability relations between these different models also need to be described using a separate model where DSLs can be used. Drivalos et al. argued that traceability information is a first-class MDE artefact and should be maintained in a different form of model with a semantically-rich traceability metamodel and inter-model constraints [7].

2.6.1 Traceability Metamodeling Language (TML)

Drivalos et al. propose the Traceability Metamodeling Language (TML), a DSL for describing traceability metamodels at a high level of abstraction [7]. They argue that TML enables the construction and maintenance of traceability metamodels and accompanying constraints with reduced effort. TML is a modeling language and its metamodel is described using ECore as shown in Figure 16. The following are the main concepts of TML as described in [7]:

- Trace: Acts as the root of a TML model. A Trace defines a name and contains a number of TraceLinks and Contexts.
- TraceLink: Represents a traceability link between a number of elements. It contains TraceLinkEnds and associates to a number of Contexts through which it can capture custom information.
- TraceLinkEnd: Represents an end of a traceability link, which defines the type of elements it can links to.

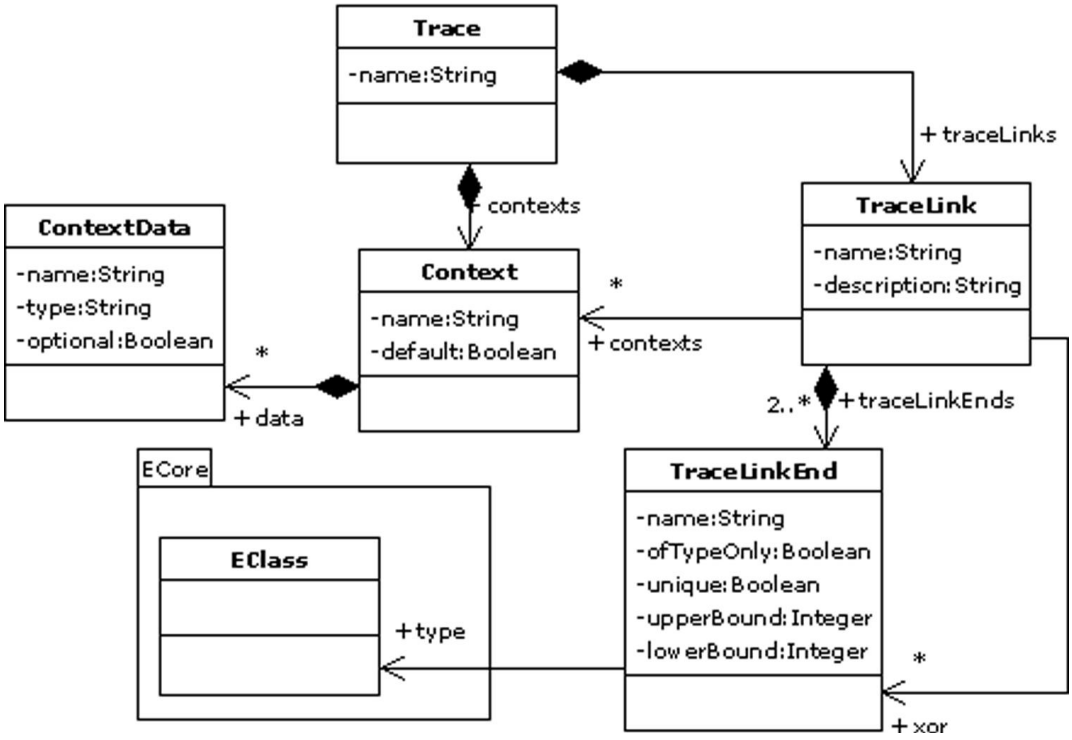

Figure 16 The Traceability Metamodeling Language (from Drivalos et al. [7])

- Context: The Context enables traceability metamodel designers to attach custom information about traceability links. Each context defines a number of ContextData.
- ContextData: ContextData captures additional information of primitive type.

TML models are automatically transformed to ECore metamodels and the accompanying constraints using a translational process. The constraints are expressed in the Epsilon Validation Language (EVL) [7][9], which is an extension of OCL [37].

Note that TML is not used to import models in a Requirements Management System such as DOORS. TML is closer to 3rd-party traceability, with one tool for model, one tool for requirements management, and a third tool for traceability. The traceability is generic, but the part between a model and other types of requirements would be handled outside of the RMS, which can negatively impact usability and utility (e.g., one would want to reuse existing DOORS functionalities as much as possible).

2.6.2 DSL for Requirement Interchange Format (ReqIF/RIF)

Graf et al. presented an Eclipse-based implementation and DSL of the Requirement Interchange Format (ReqIF) based requirement editing platform [17]. RIF is an emerging standard for requirements interchange, which is driven by the German automotive industry.

```
1  grammar de.itemis.xtext.Oclnl with org.eclipse.xtext.common.Terminals
2
3  generate oclnl "http://www.itemis.de/xtext/Oclnl"
4
5  Ocl:
6  (statement+=Statement)*;
7
8  Statement:
9  (subject1=ID) (logicalRule+=LogicalRule)* (predicate=Predicate) |
10 (subject1=ID) "SHALL" (nllPredicate+=ID)+;
11
12 LogicalRule:
13 (logicalop=LogicalOp) (subject2=ID);
14
15 enum LogicalOp:
16  and|or|xor;
17
18 Predicate:
19 (condition=('is equal to' | 'is greater than' | 'is less than' |
20     'is not equal to' | 'is enabled' | 'is disabled')) (value=STRING)?;
```

Figure 17 DSL definition for the RIF format (from Graf et al. [17])

Figure 17 shows the DSL definition for RIF using Xtext. Based on this DSL, a traceability framework (shown in Figure 18) was implemented.

Graf et al. describe that the general concept of traceability in their VERDE project led to the decision of implementing a traceability solution independent of the types of artifacts involved [17]. They argue that the solution described in Figure 18 can be a general solution for the traceability of EMF-based elements.

As described in Figure 18, the core of the solution is a mapping table with three elements that are source elements, target elements and additional information like descriptions. In the proposed general solution, the elements are identified by a data structure called “Tracepoint” where the inner structure of the tracepoint will depend on the metamodel being used.

Graf et al. describe that their technical solution is often based on the import of requirement into the target models of other tools. They see this as an important drawback as not all models support extensions.

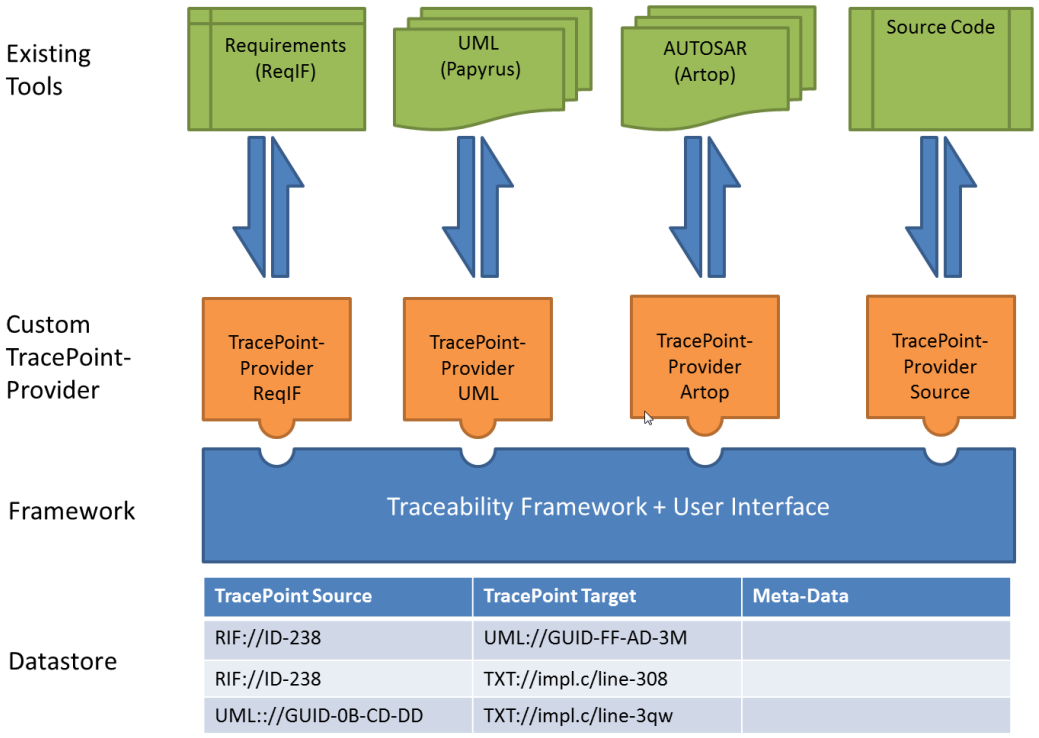

Figure 18 Traceability framework proposed for the RIF format (from Graf et al. [17])

2.6.3 Support for URN Modeling and Traceability

The User Requirements Notation (URN) is a standardized modeling language that integrates goals (with the Goal-oriented Requirement Language view, GRL) and scenarios (with the Use Case Map view, UCM) [1][3][25].

Traceability support exists inside the language (e.g., *URN links* enable connecting any two URN model elements), but also outside the language using an external RMS, namely DOORS [34][35]. Existing support is provided by jUCMNav (a modeling tool supporting URN) with an existing and hard-coded DXL library [21] initially developed by Jiang for UCMNav [41], an earlier version of jUCMNav [27][28]. This corresponds to a manually created library in part *c* of Figure 1. This thesis *automates* the DXL library creation using Xtend based on the model traceability DSL described in Chapter 3. The DXL library we generate has the same structure and functionalities for manipulating model objects (e.g., addition and deletion) as those found in the DXL library available for jUCMNav. For example, if we describe the same subset of URN for which jUCMNav

currently provides traceability support using our DSL, the generated DXL library will be similar and will provide the same overall functionality.

Figure 19 describes the metamodel capturing the Use Case Map modeling elements of interest and their relationships (without attributes to keep the figure simple) used by Jiang for exporting models to DOORS. The corresponding DXL library from Jiang is for a subset of the UCM view, but including the results of scenario executions.

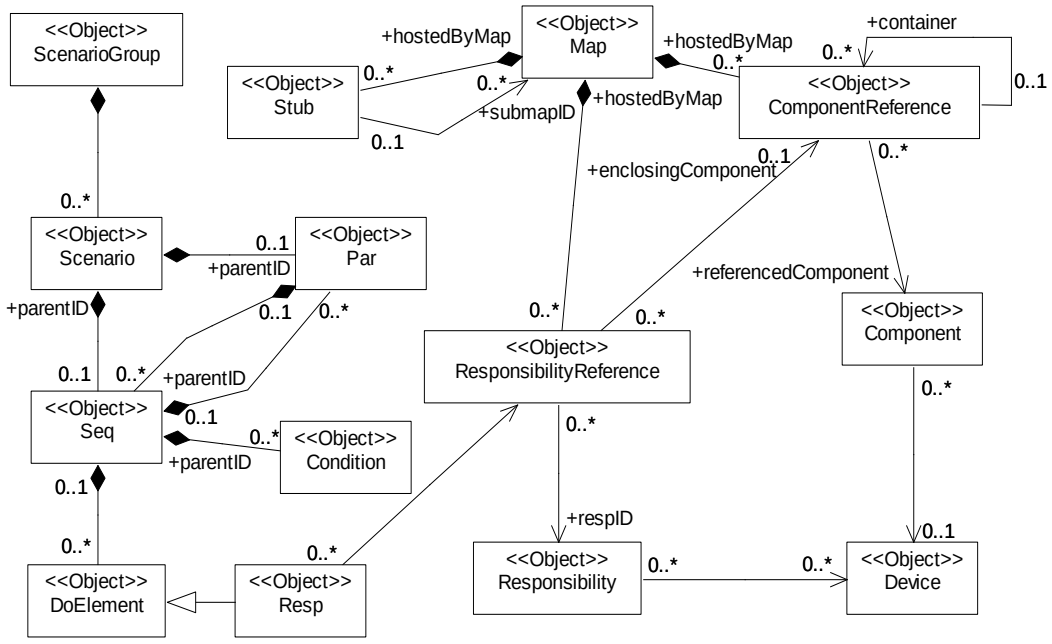

Figure 19 Jiang's UCM metamodel for export to DOORS (from [27])

Roy extended Jiang's library to support GRL goal models in addition to UCM scenario models [43]. The metamodel for this new view, shown in Figure 20, supports basic goal modeling concepts (intentional elements, actors, and links). Again, their attributes are not shown here for simplicity. This part also distinguishes between definitions of the elements and references to the element definitions local to the diagram. This is useful to support multiple diagrams and for having multiple references sharing the same definition. The DXL export mechanism was ported to jUCMNav, demonstrating that many different tools can generate DXL scripts targeting the same library.

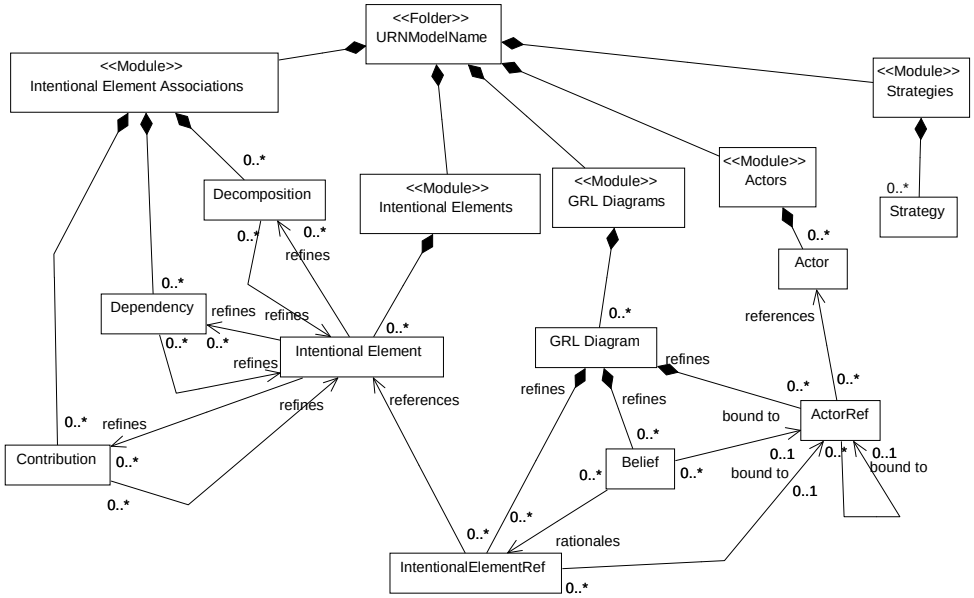

Figure 20 Roy's additional GRL metamodel for export to DOORS (from [43])

Ghanavati extended Roy's library to support a *URN profile* [2] for the modeling of laws and regulations [11][12]. The library is used by jUCMNav to support URN-based compliance with institutional policies, government regulation, and applicable legislation. One particularity of her work is that her DXL library supports new types of links between two parts of a URN model (one for the organization model, one for the legislation model, see Figure 21). The other particularity is that Ghanavati provided additional DXL functions to automatically infer, via transitivity, new traceability links based on pairs of existing and consecutive links.

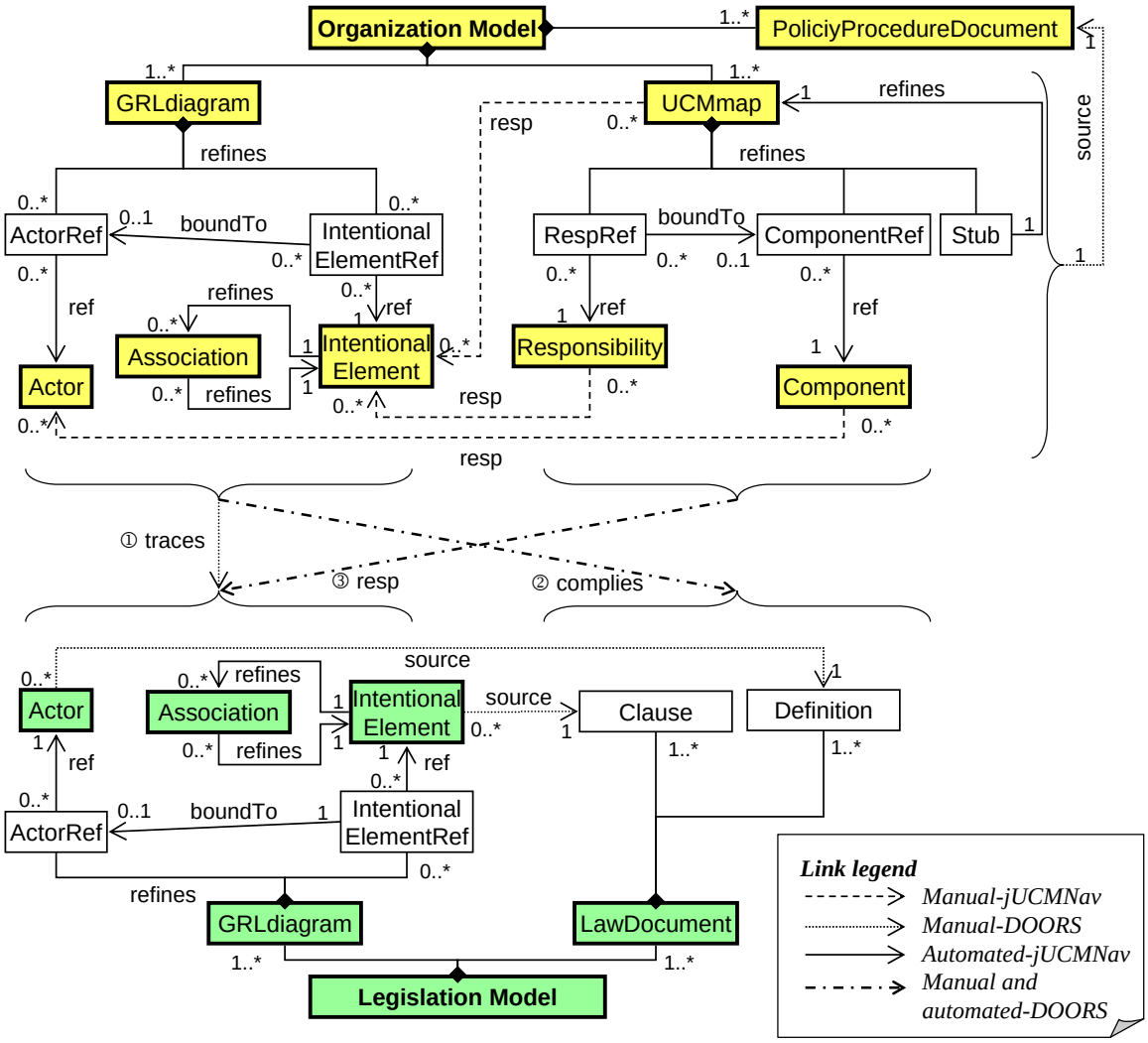

Figure 21 Ghanavati's extended metamodel for supporting legal compliance in DOORS (from [11])

2.7 Summary

This chapter introduces the technology and tools used in this thesis. Section 2.1 first presented important traceability concepts. In Section 2.2, IBM Rational DOORS and DXL have been introduced. This section also discussed sample DXL code and how DXL can be used to enhance DOORS. After an introduction to DSLs in Section 2.3, Section 2.4 described the Xtext technology for creating editors and other tools, and Section 2.5 introduced the Xtend Java-like language used for manipulating and transforming models. Section 2.6 described existing DSL-related work that has been proposed for creating and

maintaining traceability, including the approaches of Jiang, Roy and Ghanavati for UCMNav/jUCMNav and based on DOORS/DXL. It was noted that TML and similar approaches to 2rd-party traceability can lead to usability challenges for RMS-based analysis.

The next chapter defines and illustrates our new Model Traceability DSL, the first main contribution of this thesis, and exploits Xtext to build an Eclipse-based editor for this language.

3 Model Traceability Domain-Specific Language

In this chapter, we define the new Model Traceability Domain-Specific Language (MT-DSL) introduced in this thesis. DSLs are used in software engineering to describe a problem or solution technique specific to a particular domain. Creating a specific language for a problem domain is useful as this allows describing the requirements and solutions more explicitly for the specific domain. MT-DSL is to describe a view of a modeling language metamodel that is of interest for traceability in requirements management (part *b* in Figure 1). In addition to the definition of MT-DSL, this chapter describes the Xtext implementation of an Eclipse-based editor to support the new DSL.

3.1 The MT-DSL Metamodel

In order to capture the structure of models, one needs a description language that includes concepts often used to specify modeling languages themselves (e.g., classes, attributes, associations, data types, and some kind of packaging facility). Figure 22 gives an overview of the abstract MT-DSL metamodel for describing the views of modeling languages that need traceability. Its concepts are analogous to what is found typically in a meta-metamodel such as OMG's Meta Object Facility [36], Eclipse's Ecore [8] (Figure 12), and the ITU-T Z.111 standard [24].

The DSL uses 'Model' as a top-level element (and there is only one instance of Model per language/notation for which we want to generate a library). The Model element contains folders. A 'Folder' may contain any number of modules. A 'Module' can contain any number of 'Classes', where classes typically capture the language concepts one wants to trace in an RMS. 'Classes' may contained typed 'Attributes' and may be linked via typed 'Associations'.

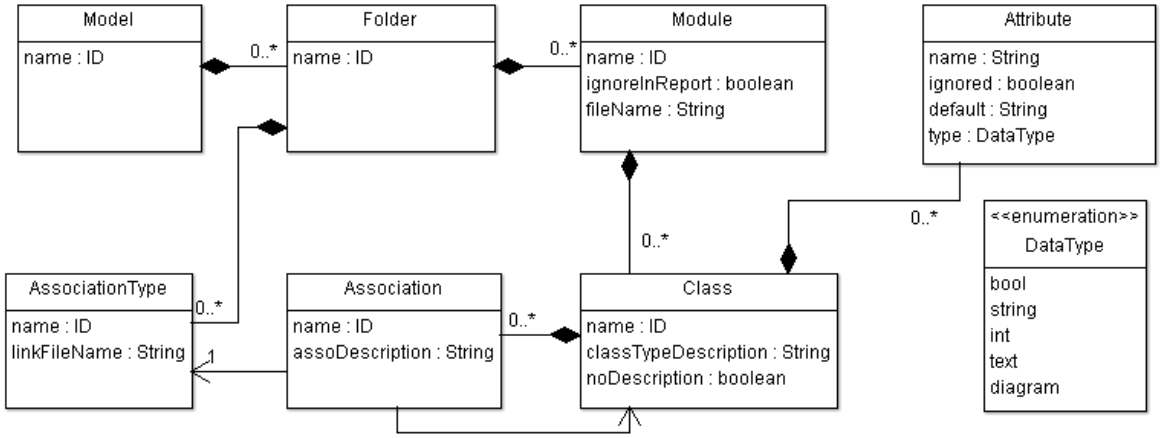

Figure 22 Metamodel for MT-DSL

3.1.1 Model

'Model' contains zero or more 'Folder' elements. There is only one model per MT-DSL description. The model has the following attribute:

- **name:** Describes the name of the desired subset of the target modeling language or notation.

3.1.2 Folder

'Folder' is a structuring element that contains zero or more 'Module' elements. A 'Folder' also declares the zero or more 'AssociationType' elements that can be used within the scope of the 'Folder'. A 'Folder'contains the following attribute:

- **name:** Used as the identifier of the folder within the 'Model'. This attribute is also used to describe a name that will be used to represent a folder or equivalent structure in the RMS (e.g., in DOORS).

3.1.3 Module

A 'Module' contains a number of 'Class' elements. A module essentially groups classes with similar attributes. A 'Module' has the following attributes:

- **name:** Used as the identifier within the scope of the parent 'Folder'.

- **ignoreInReport:** Provides an option to override the default behaviour on whether this module will be part of the generated report or not. By default all modules are part of the report unless overridden using this attributes.
- **fileName**: Provides an option to specify a filename for this module to be used by the RMS.

3.1.4 Class

A ‘Class’ contains ‘Association’ and ‘Attributes’ elements. The class might have zero or more ‘Association’ elements used to describe associations with other classes. Similarly, the class might have any number of ‘Attribute’ elements, each describing one attribute of the class. The ‘Class’ has the following attributes:

- **name:** Used as the identifier within the scope of the ‘Module’.
- **classTypeDescription**: The default behaviour for a ‘Class’ is to determine the ‘Type’ where necessary based on the class name. This attribute provides an option to provide the ‘Type’ description for a traceability model by overriding the default behaviour.
- **noDescription:** Used to override the default behaviour of having a description attribute for every class in the RMS.

3.1.5 Attribute

An ‘Attribute’ is to represent an attribute of a ‘Class’. The ‘Attribute’ element has the following attributes:

- **name**: Captures the name of the ‘Attribute’.
- **type**: This is the data type that the ‘Attribute’ represents. This can be one of the supported data types in the language that the ‘DataType’ enumeration represents**.**
- **ignored**: Used to indicate that when this ‘Attribute’ of the ‘Class’ is modified, the ‘Class’ should not be flagged as modified for generating change reports. For example, if the X-Y coordinates of a modeling element are stored in the RMS database but we do not care to track changes to the co-

ordinates over time (and flag them as an important modification in the RMS), then this attribute can be used.

- **default**: Provides the 'Attribute' name to be used in as heading in the RMS. By default this can be determined from the 'name' of the attribute, but this default can be overridden here.

3.1.6 AssociationType

This element is used to declare available types for 'Association' that can be used within the scope of a folder. The following are the attributes for 'AssociationType':

- **name:** Used as the identifier within the scope of the parent 'Folder'.
- **linkFileName:** Specifies a filename for the linked module to be used in column header in the RMS.

3.1.7 Association

An 'Association' is contained by a source class, and also has an associated target class. An association has an 'AssociationType'. The 'Association' is a very important concept as it captures traceability links to be tracked between modeling elements. The following are the attributes for 'Association':

- **name:** Used as the identifier within the scope of the parent 'Class'.
- **assoDescription:** This attributes describes the association.

Note that unlike most meta-metamodels, the concept of association *multiplicities* is not included here, as it was deemed unnecessary in a traceability context. Generic many-to-many relationships are assumed to be modeled through the associations here.

3.1.8 DataType

This enumeration is used to provide the list of data types that are supported for the 'Attribute' elements of a 'Class'. The following is a list of supported types:

- **boolean:** The value can be either 'true' or 'false'.
- **string:** Represent the conventional strings used in many traditional programming languages.

- **int:** Represents integers.
- **text:** Represents a long text (many sentences).
- **diagram:** Represents a diagram.

3.2 DSL Generation

For the purpose of this thesis, we have implemented with Xtext a DSL for the MT-DSL metamodel described Figure 22. A textual syntax is used here instead of a visual notation to enable *faster input/creation* of DSL descriptions and *advanced editing features* such as text completion, at the cost of not being able to visualize the result of a file created with MT-DSL.

3.2.1 Generating DSLs using Xtext

In this thesis, we have chosen *Xtext*, a language development framework, as the preferred technology to implement support for MT-DSL [47]. As described in Section 2.4, Xtext provides many features needed to implement a language with a powerful Eclipse-based sophisticated editor.

Xtext is implemented as an Eclipse plug-in [48]. A new Xtext project is created using an Eclipse wizard, as shown in Figure 23. Our implementation of MT-DSL is using DxlDsl as a project and extension name. The Xtext editor then opens on Eclipse with a sample DSL in the editor (Figure 24).

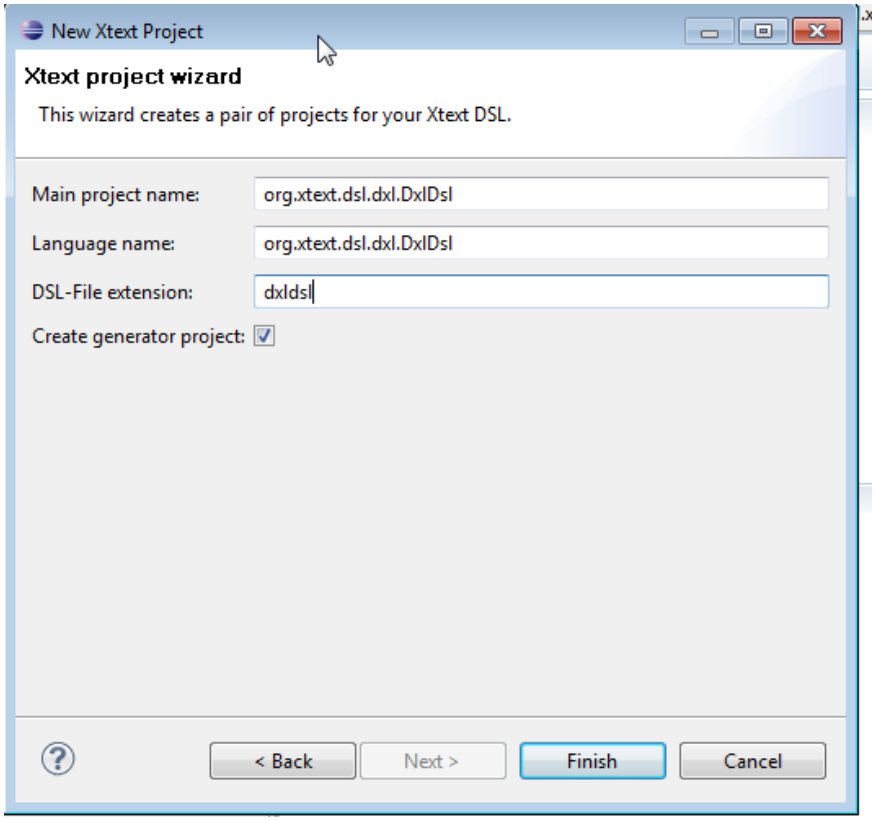

Figure 23 Eclipse's new Xtext project wizard

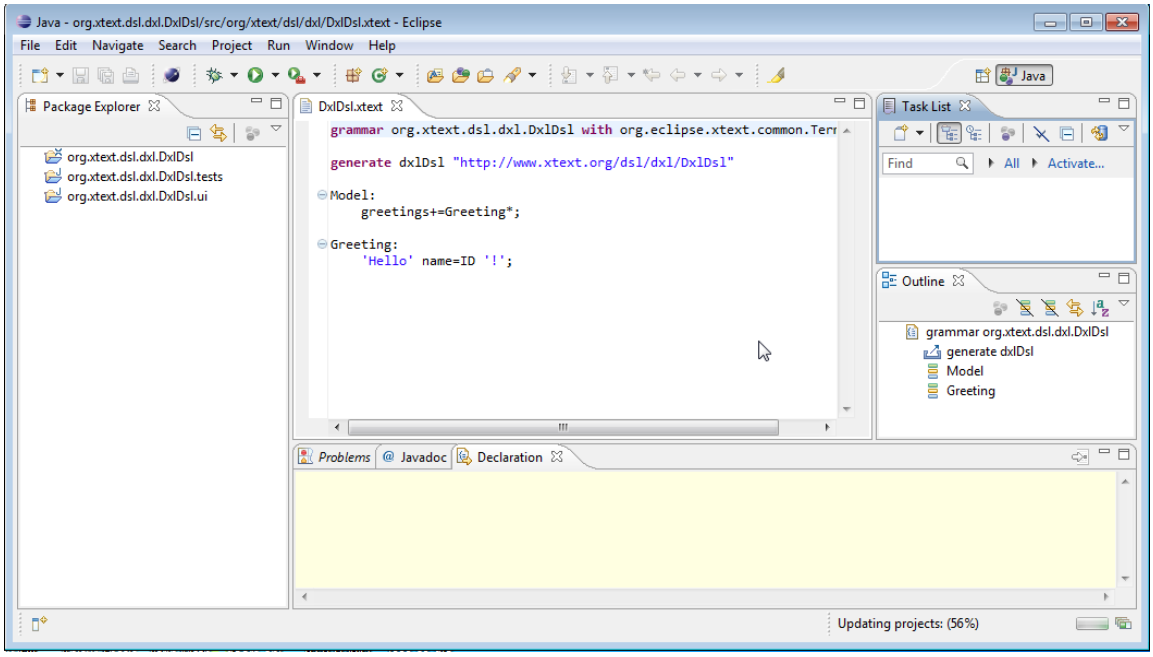

Figure 24 Xtext editor on Eclipse for a new project

The Xtext project wizard creates three projects in the Eclipse workspace, as seen in Figure 24. They are explained in Table 1, and their implementation is available in Appendix C:

Table 1 Projects generated by the Xtext wizard

Projects	**Project descriptions**
org.xtext.dsl.dxl.DxlDsl	This is the main Xtext project, which contains the grammar definition and all other runtime component (parser, lexer, linker, validation, etc.).
org.xtext.dsl.dxl.DxlDsl.tests	Unit test cases for the DSL project.
org.xtext.dsl.dxl.DxlDsl.gui	The Eclipse editor and any other work bench related functionality.

The example content on the editor for the newly created project is modified with the grammar definition for the DSL (e.g., that of MT-DSL from Appendix A, to be discussed in Section 3.3), as shown in Figure 25.

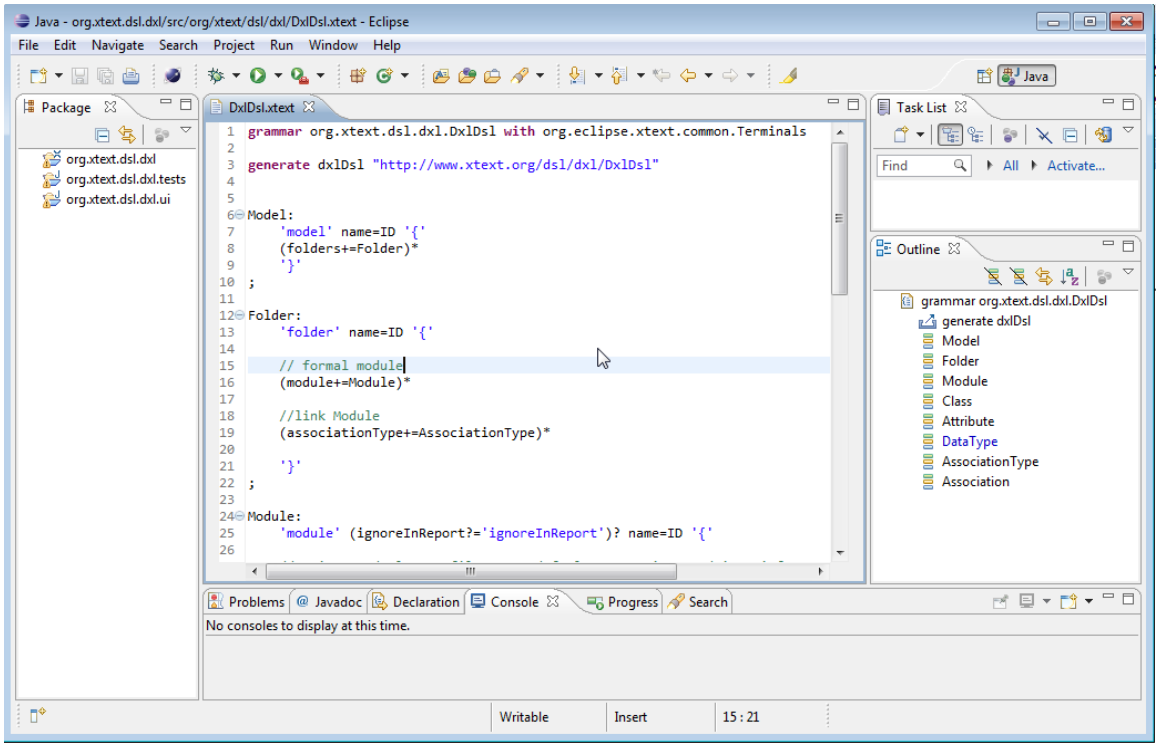

Figure 25 Xtext editor with the DSL grammar language

After updating the editor contents with the grammar language for the DSL, the corresponding language infrastructure for the new language needs to be generated [48] (Figure 26).

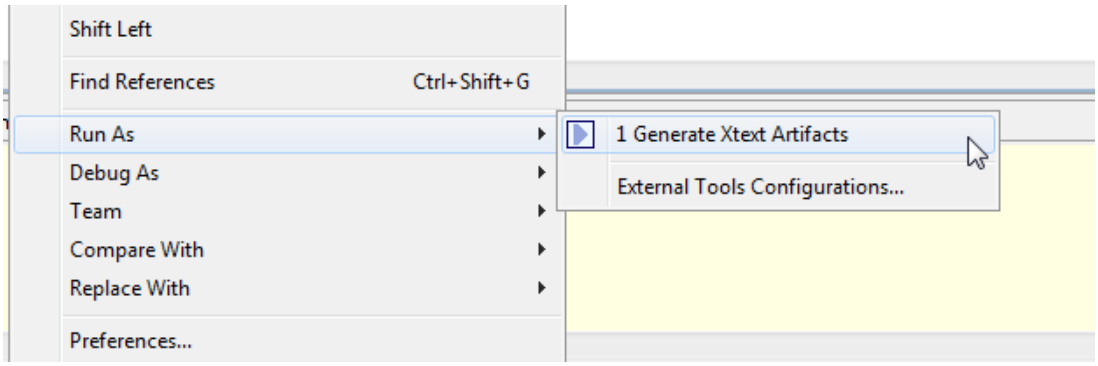

Figure 26 Generating Xtext artifacts/infrastructure for a DSL

After the generation of artifacts, feedback is provided on the Eclipse console as shown in Figure 27.

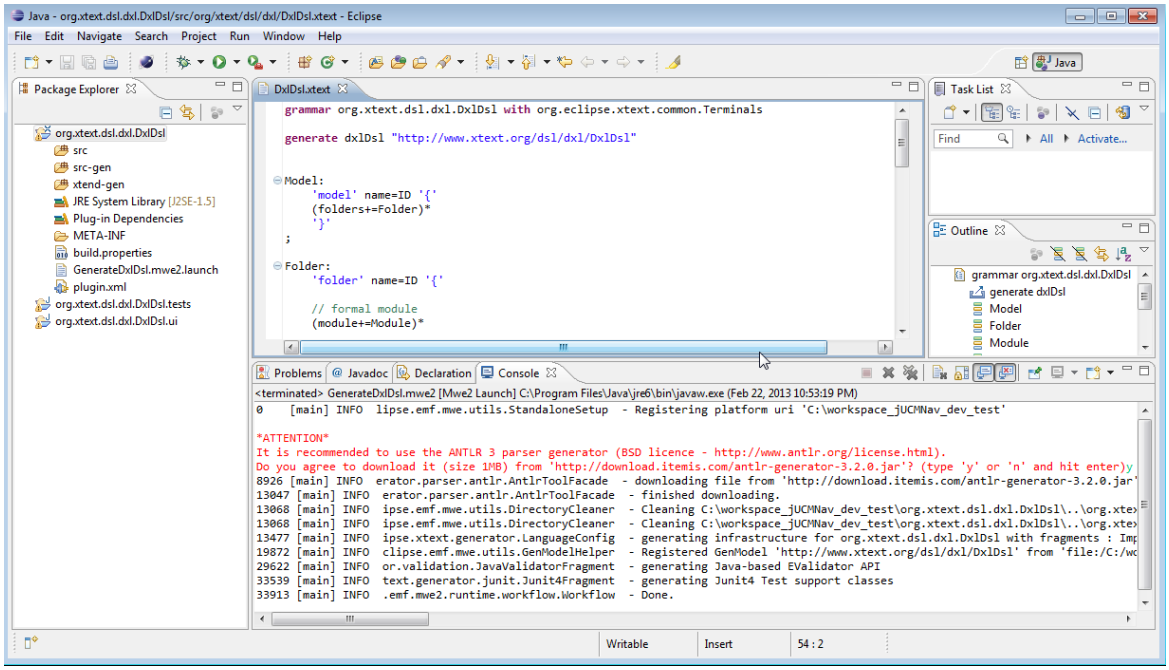

Figure 27 Feedback from Xtext artifact generation

Once this process is completed, new files are generated in the project, which now has a runnable language infrastructure with a powerful default Eclipse editor.

3.2.2 How Xtext Works

Xtext provides a set of domain-specific language and modern API to properly describe different aspects of a DSL [48]. Based on the provided information, Xtext provides a fully implemented language running on a Java Virtual Machine (JVM).

Complier Components

The compiler components generated from the grammar of the language include:

- Parser
- Type safe Abstract Syntax Tree (AST)
- Serializer and code formatter
- Scoping framework and linking
- Compiler checks and static analysis, i.e., validation
- Code generator.

The compiler components described above are independent of Eclipse or OSGi (Open Services Gateway initiative). Therefore, the compiler components can be used with any supported Java environment.

Runtime Components

The runtime components are based on the Eclipse Modeling Framework (EMF) [8][48]. Xtext is thus integrated with EMF and other related Eclipse frameworks, for example the Graphical Modeling Framework (GMF).

The runtime components of the language include a parser, a lexer, a linker, and validation functions.

IDE Support

Xtext comes with a sophisticated Eclipse-based Integrated Development Environment (IDE) support. It provides great default functionalities for the DSL editor.

Xtext Configuration

Xtext comes with a decent default implementation. Xtext also supports customization, providing APIs for common customization spots [48]. Xtext uses Google Guice[1], a light-

[1] http://code.google.com/p/google-guice/

weight dependency injection framework for the language and IDE infrastructure. An external and central module is used to configure the dependency injection component.

3.3 Grammar Language

This section describes the language definition implemented with Xtext for the purpose of this thesis. The abstract metamodel for the MT-DSL language is described in Figure 22. The complete definition, which uses Xtext to provide a textual concrete syntax, is described in Appendix A and will be discussed construct by construct in this section. Each construct actually covers one concept from the MT-DSL metamodel seen in Section 3.1

Figure 28 shows the Outline view for the grammar language in Eclipse, which displays the different constructs of the grammar language.

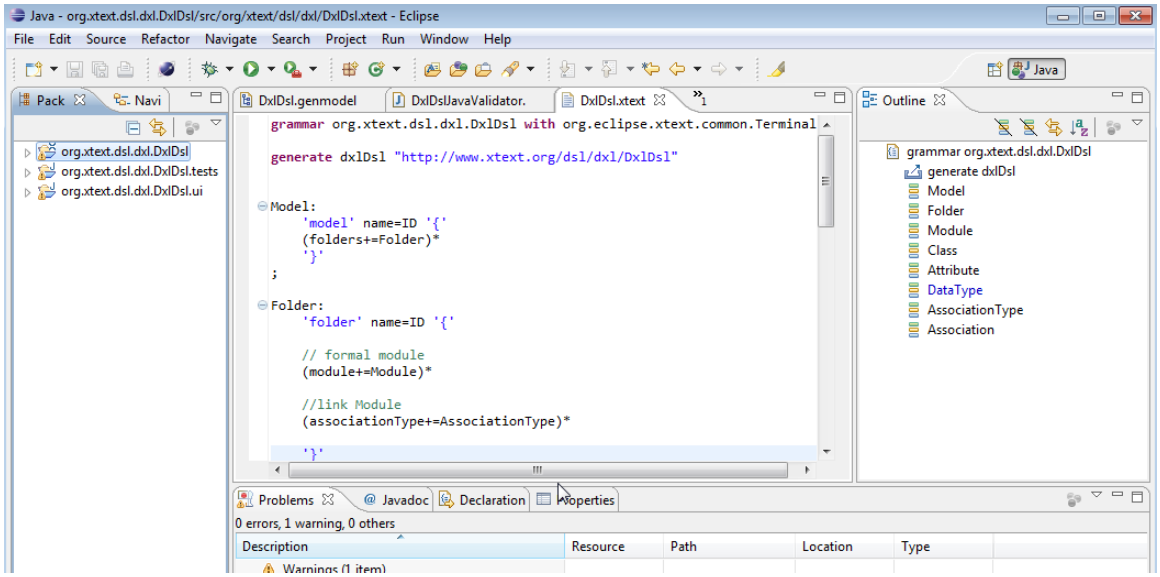

Figure 28 Outline view for the grammar language

The meaning of the different grammar constructs are described below:

1.
```
grammar org.xtext.dsl.dxl.DxlDsl with
        org.eclipse.xtext.common.Terminals
```

Each Xtext language description starts with a language declaration [48]. The header of the language defines some property of the language. The first line above is the name of the language. As seen from the name structure, Xtext follows Java's class path mecha-

nism to name a language. The file name needs to correspond to the language name with '.xtext' as an extension. The file name for our MT-DSL is "DxlDsl.xtext", and the file must be placed in a package 'org.xtext.dsl.dxl.DxlDsl' in the project's class path.

The second line here is the relationship of the language with another language. An Xtext grammar can be declared to reuse another existing grammar [48]. This procedure is called 'grammar mixing'. Common terminals predefined in Xtext are hence reused.

2
```
generate dxlDsl "http://www.xtext.org/dsl/dxl/DxlDsl"
```

Xtext creates Ecore-based in-memory object graphs while processing text. The Ecore model contains an EPackage consisting of EClasses, EDataTpye and EEnums. Xtext can infer the Ecore model from the DSL grammar or an existing Ecore model can be imported. The 'generate' declaration in the grammar is used to advice the Xtext framework to *infer* the EPackage, here with the name 'dxlDsl' and with the nsURI 'http://www.xtext.org/dsl/dxl/DxlDsl'. The nsURI is used to import existing EPackages.

3
```
Model:
	'model' name=ID '{'
		(folders+=Folder)*
	'}'
;
```

The first rule in the Xtext grammar is always used as the entry or the start rule [47]. The rule model starts with the keyword 'model', followed by an identifier, which will be parsed by a rule *ID*, described in the grammar org.eclipse.xtext.common.Terminals and previously imported here. The rule *ID* parses a single word, which is used as an identifier. The value return by the call *ID* is assigned to a feature (=) called *name*. The above rule also describes that the Model will contain an arbitrary number (*) of *Folders* and these Folders will be added (+=) to a feature called *folders*.

4

```
Folder:
	'folder' name=ID '{'

		// formal modules
		(module+=Module)*

		// link modules
		(associationType+=AssociationType)*
	'}'
;
```

The rule *Folder* starts with the keyword ‘folder’ followed by a mandatory name. The name will be unique within the scope of the *Model*, and is to be used as the identifier for the *Folder*. A *Folder* can contain an arbitrary number (*) of *Module* that will be added (+=) to a feature called *module*. A *Folder* also contains an arbitrary number (*) of *AssociationType* that will be added (+=) to a feature called *associationType*.

5

```
Module:
	'module' (ignoreInReport?='ignoreInReport')? name=ID '{'

	// Option to declare a file name: default convention used
	// 'Maps' for map, 'Devices' for 'device', etc.
		('fileName' fileName=STRING)?
		(classes+=Class)*
	'}'
;
```

The rule *Module* starts with the keyword ‘module’, followed by an optional keyword ‘ignoreInReport’. This keyword is parenthesized and added to a feature called *ignoreInReport*. The cardinal (?) is used here to indicate that the feature *ignoreInReport* is optional. The assignment operator (?=) implies that the feature *ignoreInReport* is of type *Boolean* and will be false if the keyword is absent. This rule also describes a mandatory identifier that will be added to the feature called *name*. The name will be unique within the scope of a *Folder*. The rule also describes an optional (?) and parenthesized ‘fileName’ clause, which will be added to the feature called *filename*. This is to override the default naming convention used for file name for this module to be used in the RMS. Finally, a module can have an arbitrary number of *Class* in a feature *classes*.

6

```
Class:
      'class' (noDescription?='noDescription')? name=ID
          ('shows as' classTypeDescription=STRING)? '{'
             (attributes+=Attribute)*
             (associations+=Association)*
      '}'
;
```

The rule *Class* starts with the keyword ‘class’, followed by an optional keyword ‘noDescription’. As seen before, the keyword represent an optional, Boolean value. The feature *noDescription* is followed by the identifier of the *Class*, which is added to the feature called *name*. The feature *name* is followed by an optional clause starting with the keyword ‘shows as’. This clause provides an opportunity for describing a class type, which is added to the feature called *classTypeDescription*, if provided. Otherwise, a default description is generated from the name. The rule *Class* can also have an arbitrary number of *Attribute*, added to the feature *attributes*. Similarly, *Class* can have an arbitrary number of *Association* added to *associations*.

7

```
Attribute:
      (ignored?='ignored')? type=DataType name=STRING
          ('shows as' default=STRING)?
;
```

The rule *Attribute* starts with an optional Boolean clause with keyword ‘ignored’, added to the feature *ignored*. Each *Attribute* has one mandatory *DataType* added to the feature called *type*, and a mandatory *name*. The rule Attribute ends with an optional ‘shows as’ clause, followed by a string added to the feature called *default*, which provides the option to describe the attribute using a different label rather than using the attribute name. Note that attributes for name, identifier and description are implicit in the DSL and do not need to be mentioned explicitly.

8

```
DataType:
      'bool' | 'string' | 'int' | 'text' | 'diagram'
      // Note: not more than 1 diagram attribute per class
;
```

The rule *DataType* describes the supported data types: bool, string, int, text, and diagram.

9

```
AssociationType:
	'associationType' name=ID linkFileName=STRING
;
```

The rule *AssociationType* starts with the keyword ‘associationType’ followed by an identifier *name*, unique within the scope of the parent *Folder*. The *name* is followed by the feature *linkFileName*, use to describe a label for the association type.

10

```
Association:
	'association' name=ID':'assoType=[AssociationType] 'to'
	moduleType=STRING('.'classType=STRING)?
		(assoDescription=STRING)?
	// Note: the classType must be defined in the module-
	Type.
;
```

Finally, the rule *Association* starts with the keyword ‘association’ followed by a *name*. This is followed by the feature *assoType*, joined by a ‘:’ (colon), which is a cross-reference to rule *AssociationType* as indicated by the square brackets. This feature is followed by the feature *moduleType* and an optional feature *classType* separated by a ‘.’ (dot) if present. The features *classType* and *moduleType* are used to describe the identifiers of the linked *Class* and of the *Module* where the linked *Class* belongs. If the optional classType is not provided, then it is assumed by default that the name of the *Class* is the same as the name of the *Module*. The rule *Association* ends with an optional feature called *assoDescription*, used to provide a description of the association.

3.4 Generated Ecore Model and Configuration Files

Xtext produces automatically the Ecore model as a part of its artifact generation process. The Ecore model of the textual language describes the structure of the Abstract Syntax Tree (AST) of the language [47]. The generated Ecore model the grammar is shown in Figure 29 using the default EMF Ecore editor in Eclipse. It corresponds to the MT-DSL metamodel seen in Figure 22.

The Xtext generator also uses a special DSL called MWE2 (Modeling Workflow Engine[2]) to describe the object graphs in a declarative manner [47]. The Xtext generator also contributes to project sharing files such as MANIFEST.MF and plugin.xml.

The language infrastructure developed using Xtext is configured using dependency injection controlled by Google Guice, and Guice provides the appropriate configuration items when an object requiring dependency injection is instantiated.

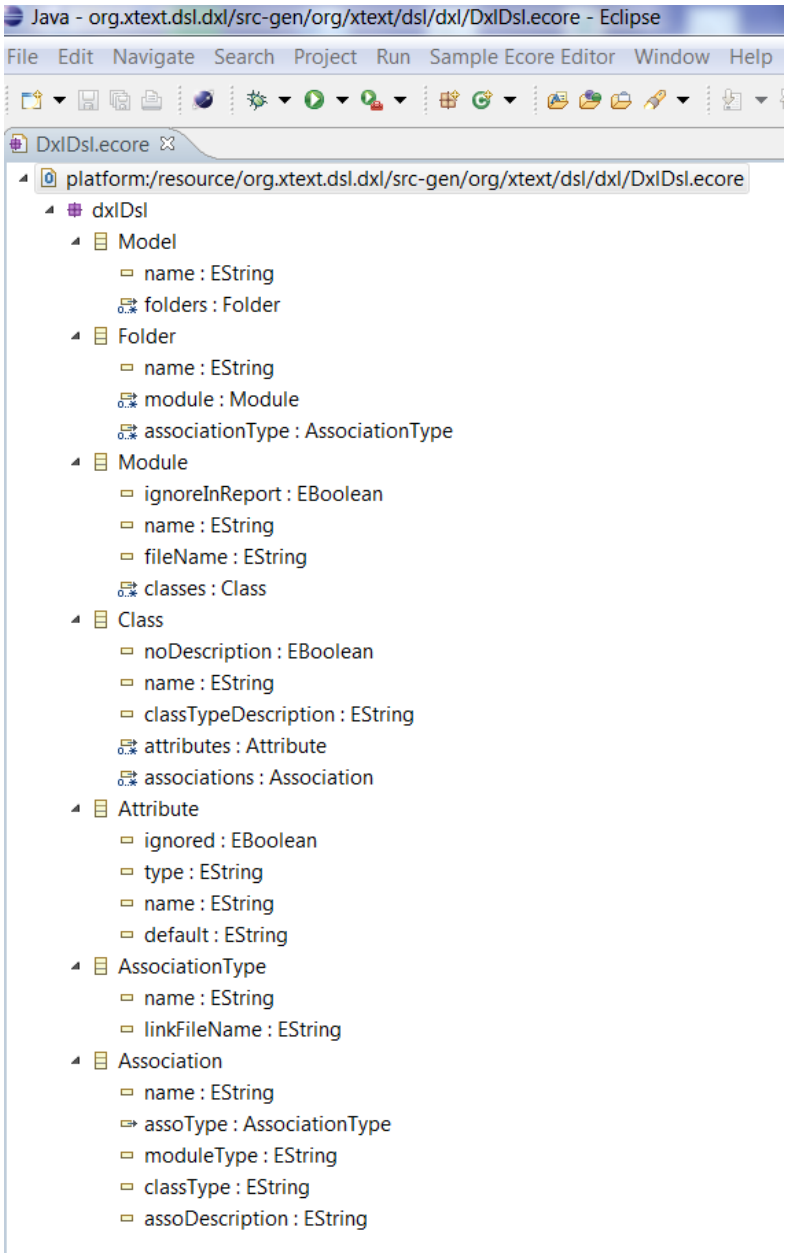

Figure 29 Generated Ecore model

[2] http://help.eclipse.org/helios/index.jsp?topic=%2Forg.eclipse.xtext.doc%2Fhelp%2FMWE2.html

3.5 MT-DSL Language Editor

The Eclipse-based editor for MT-DSL comes with powerful default editor features. However, Xtext also provides an opportunity to implement custom editor features and custom error handling applicable to the language domain. These features are implemented and illustrated in this section.

Appendix E shows how to start and run the MT-DSL editor on a new file.

3.5.1 Content Assistance and Eclipse Views

The editor provides contextual content assistance for all the rules described in Section 3.3, and this is made available by using Ctrl+Space on the keyboard. Figure 87 in Appendix E shows an empty DSL langue file opened in the editor. As described in Section 3.3, the language starts with the keyword 'model' followed by the model name. Pressing Ctrl+Space in the editor before typing anything will hence add the keyword 'model' and a space automatically. At this point, the model name can be entered (e.g., 'SampleModel'). Pressing Ctrl+Space again in this new context, followed by Enter, will add curly braces automatically, on different lines. Pressing Ctrl+Space once more, this time between the curly braces (where the cursor was put by default), will result in the editor providing the option to choose the folder block instead of typing, as shown in Figure 30.

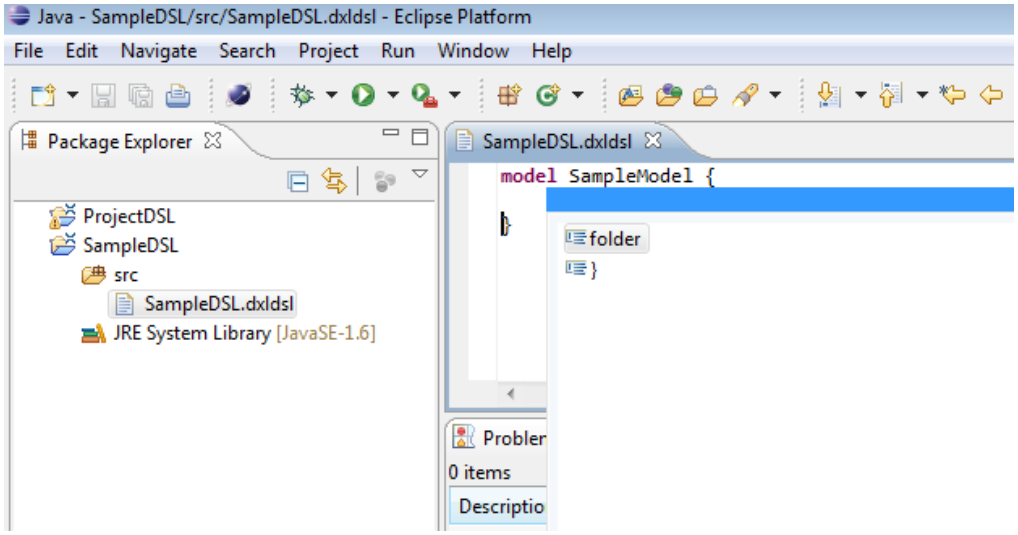

Figure 30 Content assistance for the folder block in the model traceability file

The DSL language editor is by default integrated with most common views available in Eclipse (e.g., Package explorer, Navigator, Outline, Problems, Console, etc.). Updating the text will automatically update the other views. In the previous example, by adding the folder ‘myFolder’, the resulting structure is reflected in the Outline view (see Figure 31). The interface allows navigating in different parts of the editor using the Outline view. Similarly, selecting any item in the Problems view results in the editor navigating to the relevant portion of the file.

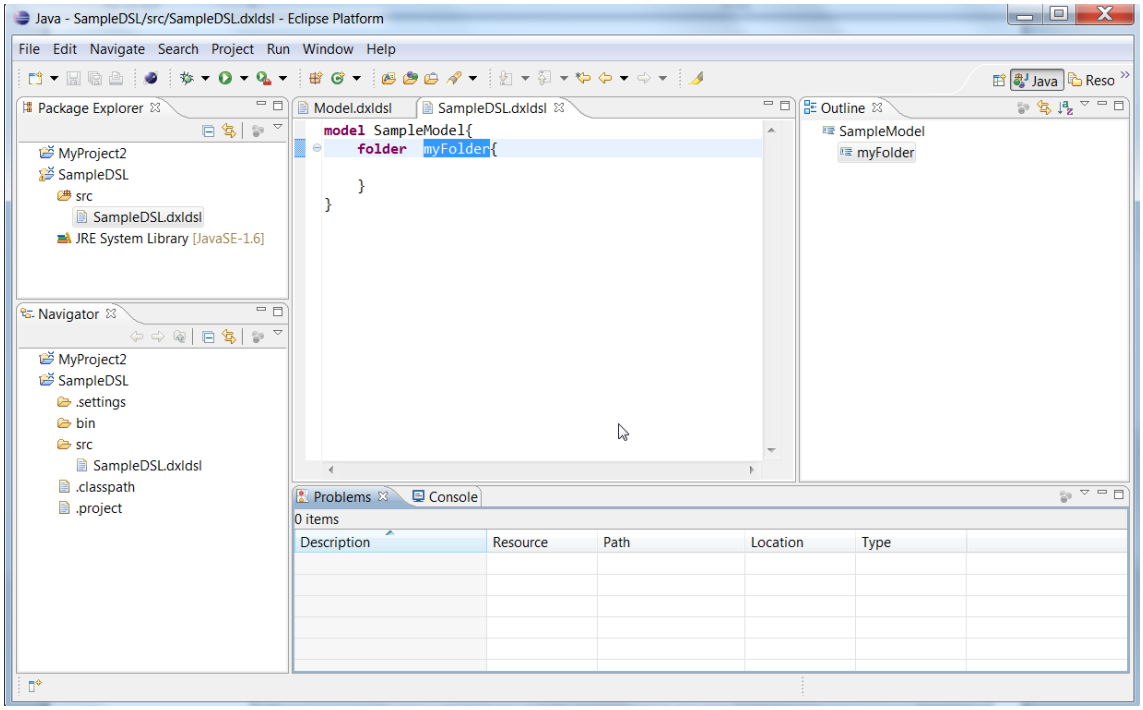

Figure 31 Integrated views for the Eclipse-based MT-DSL editor

3.5.2 Default Error Handling in the Editor

The Xtext-based editor for MT-DSL provides usable error handling. Figure 32 shows a new file created using the editor. Note that the editor marks error in different views (Package, Problems, and Editor views). In this example, the error (mismatched input ‘<EOF>’) is caused by a missing mandatory keyword (‘model’). The error message is also displayed if the mouse is hovered on the error mark at the beginning of the line in the editor or on the mark on the right side of the editor.

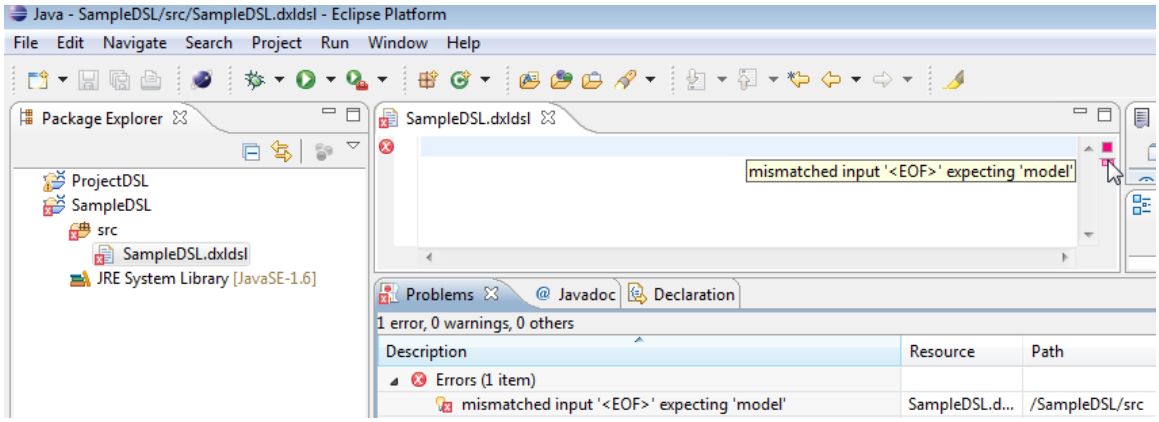

Figure 32 Editor feedback with multiple marks and with hovering

3.5.3 Custom Error Handling

Xtext provides the possibility to statically validate domain-specific constraints [48]. This is usually done by doing static analysis, and Xtext provides dedicated hooks for custom validation rules. For the purpose of this thesis, we decided to validate several aspects, including that an association references a class that exists in the described module.

The top part of Figure 33 shows the validator class DxlDslJavaValidator.java generated by Xtext in the org.xtext.dsl.dxl.validation package. This class is the default template for custom domain-specific error handling, and the validator is originally empty.

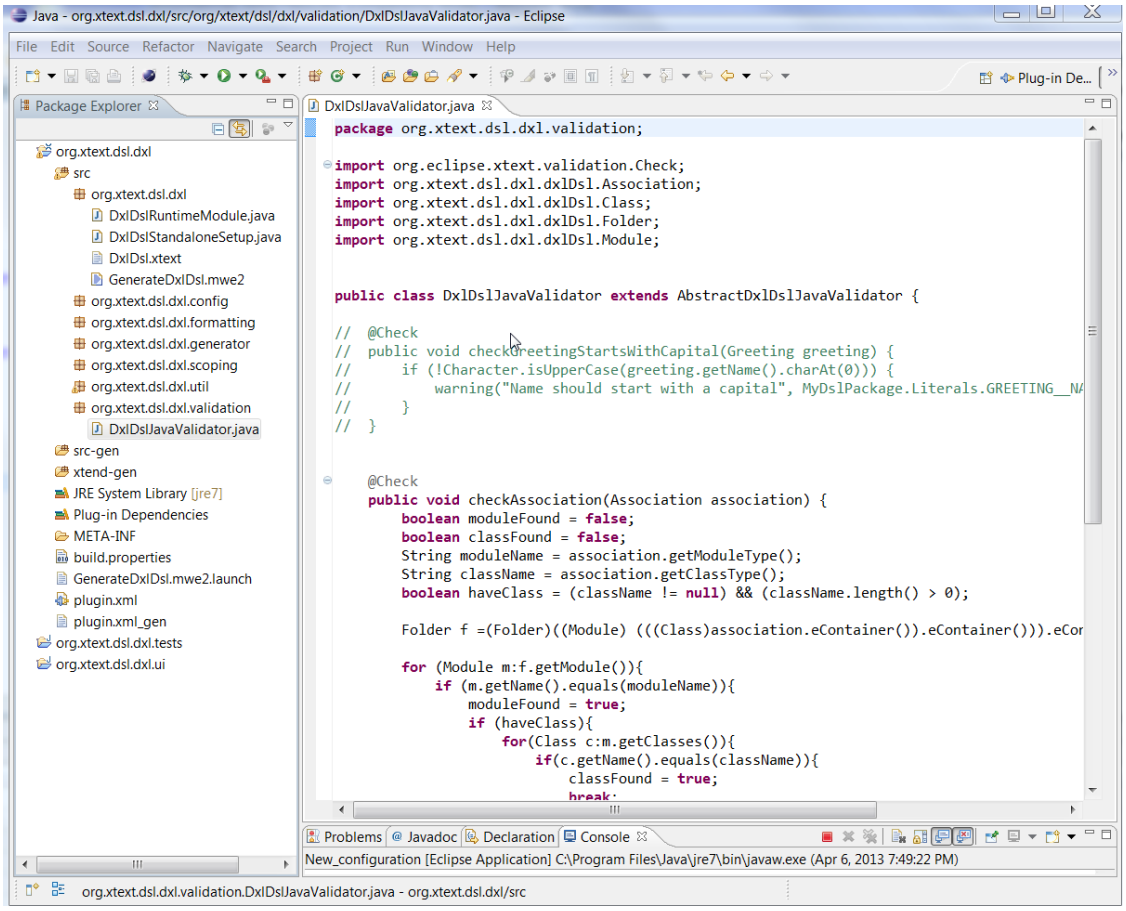

Figure 33 Custom error handling option in Xtext

For the purpose of this custom domain-specific association validation, a validation method `checkAssociation` was added, as shown in Figure 33. The method is described below:

```
@Check
public void checkAssociation(Association association) {
   boolean moduleFound = false;
   boolean classFound = false;
   String moduleName = association.getModuleType();
   String className = association.getClassType();
   boolean haveClass = (className != null) &&
                       (className.length() > 0);

   Folder f =(Folder)((Module) (((Class)association.
             eContainer()).eContainer())).eContainer();
```

```
    for (Module m:f.getModule()){
          if (m.getName().equals(moduleName)){
                moduleFound = true;
                if (haveClass){
                      for(Class c:m.getClasses()){
                            if(c.getName().equals(className)){
                                  classFound = true;
                                  break;
                            }
                      }
                }
          }
          if (classFound){
                break;
          }
    }

    if (!moduleFound){
          error("Module not found!", org.xtext.dsl.dxl.dxlDsl.
                DxlDslPackage.Literals.ASSOCIATION__MODULE_TYPE);

    }else if (haveClass && !classFound){
          error("Class not found in Module!",
                org.xtext.dsl.dxl.dxlDsl.DxlDslPackage
                      .Literals.ASSOCIATION__CLASS_TYPE);

    }
}
```

The code above navigates through all the modules in the *Folder* to verify if the module and class mentioned in the association exists. If the module does not exist or if the class mentioned in the module does not exist, an appropriate validation error is displayed as shown in Figure 34.Quick fixes can also be added to solve such issues.

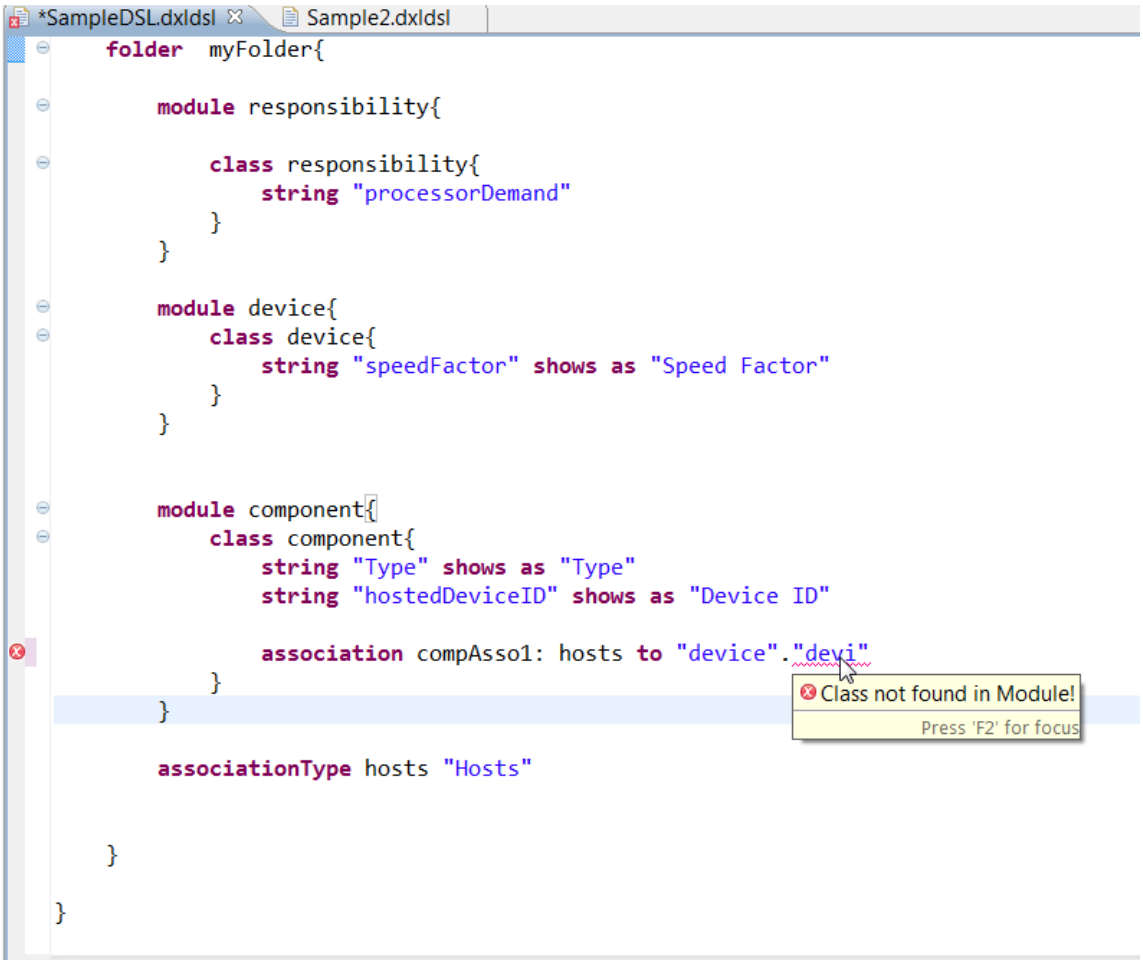

Figure 34 Custom editor feedback on error

3.5.4 Editor Preferences

The default editor behavior can be modified by updating the editor's preference page. For MT-DSL, two categories of options are available. In Figure 35, *Syntax Coloring* is selected on the left, and several related options become available, including font, color, and style for comments, keywords, numbers and String.

When the Preference page is lauched from the context menu for the editor, the window only shows preference options available to the language editor as shown in Figure 36.

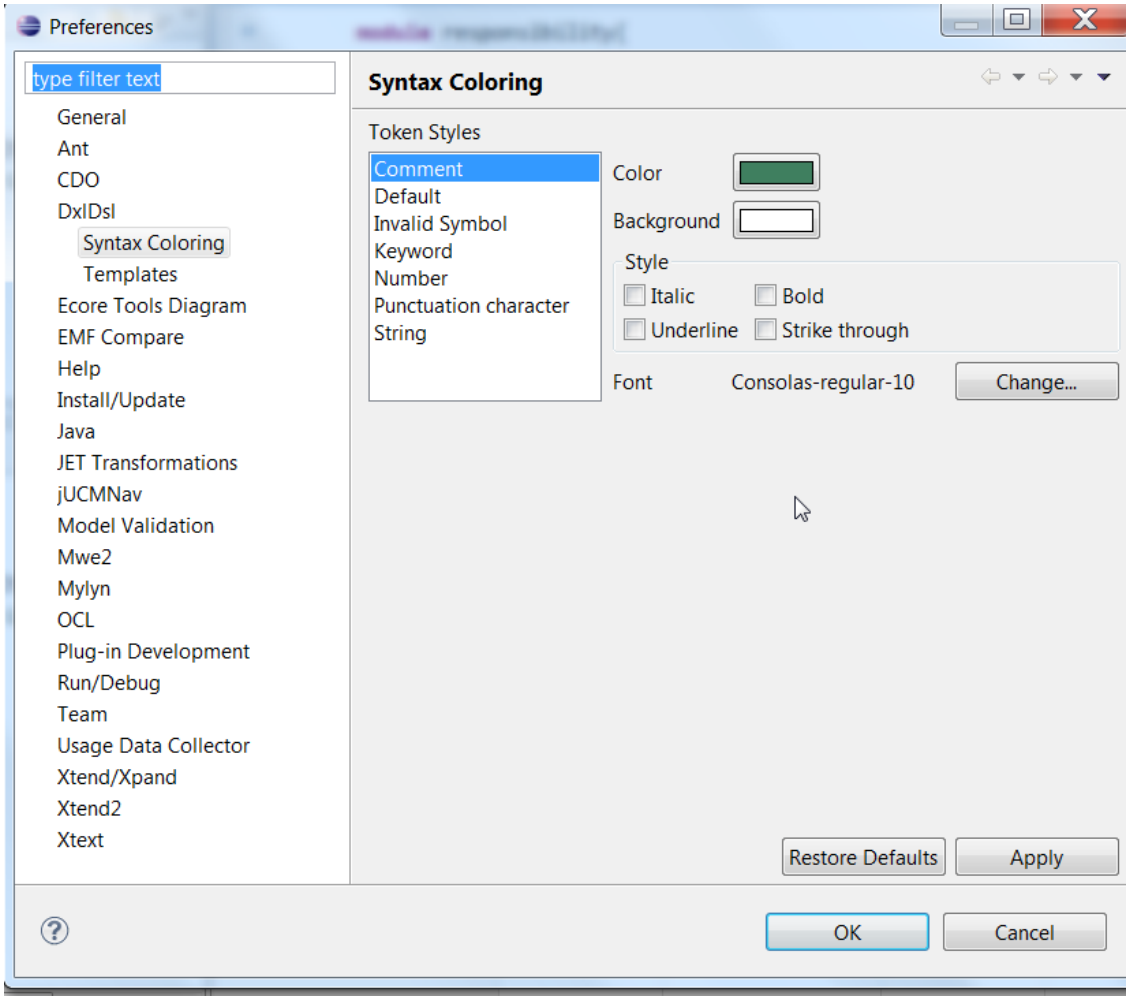

Figure 35 Editor preferences for syntax coloring

The second category of options available to the editor is *Templates* for the content assisstant. New templates can be created and managed, and existing templates can be imported (Figure 36).

The template creation window is displayed in Figure 37. A template has a name and a DSL-dependent context (all listed in the pull-down menu). Sample code can be provided for the chosen context. For example, 'Model' is selected in Figure 37. In an empty file (where the context is Model), pressing Ctrl+Space will insert the corresponding template code in the file. Such a feature can help accelerate the development of language descriptions, and help beginners understand the syntax and grammar of the language.

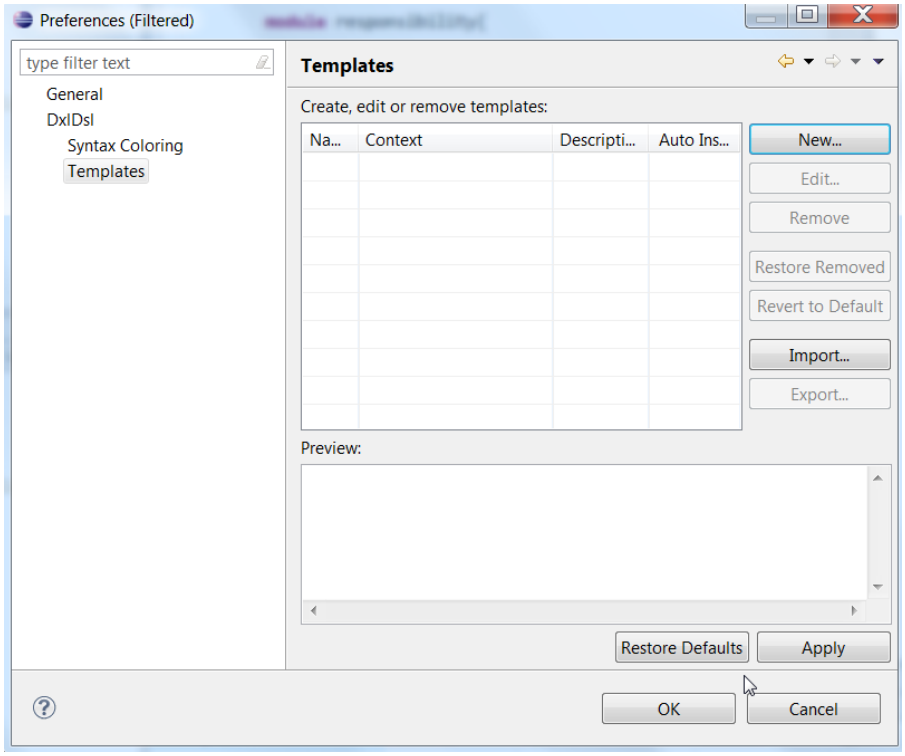

Figure 36 Editor preference for custom code templates

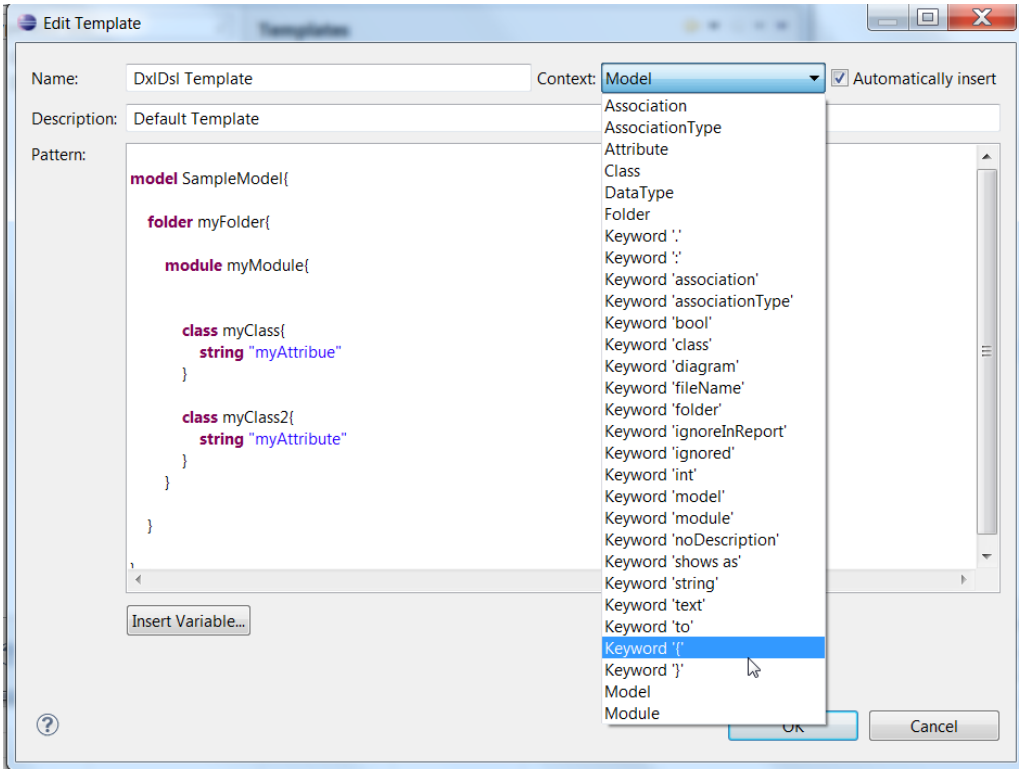

Figure 37 Code templates for different contexts

3.6 Example Modeling Language Description

A simple description of the desirable traceability management view of a modeling language is shown in Figure 38. It shows three modules, each with one class of interest. Several attributes are specified. Note that string attributes for ‘ID’, ‘Name’, and ‘Description” are predefined for classes and do not need to be declared explicitly.

The class ‘component’ also shows an example of association, called ‘compAsso1’ of type ‘Hosts’, which references class ‘device’ in module ‘device’. This means that we are interested in tracking in an RMS how components are associated to devices in this modeling language.

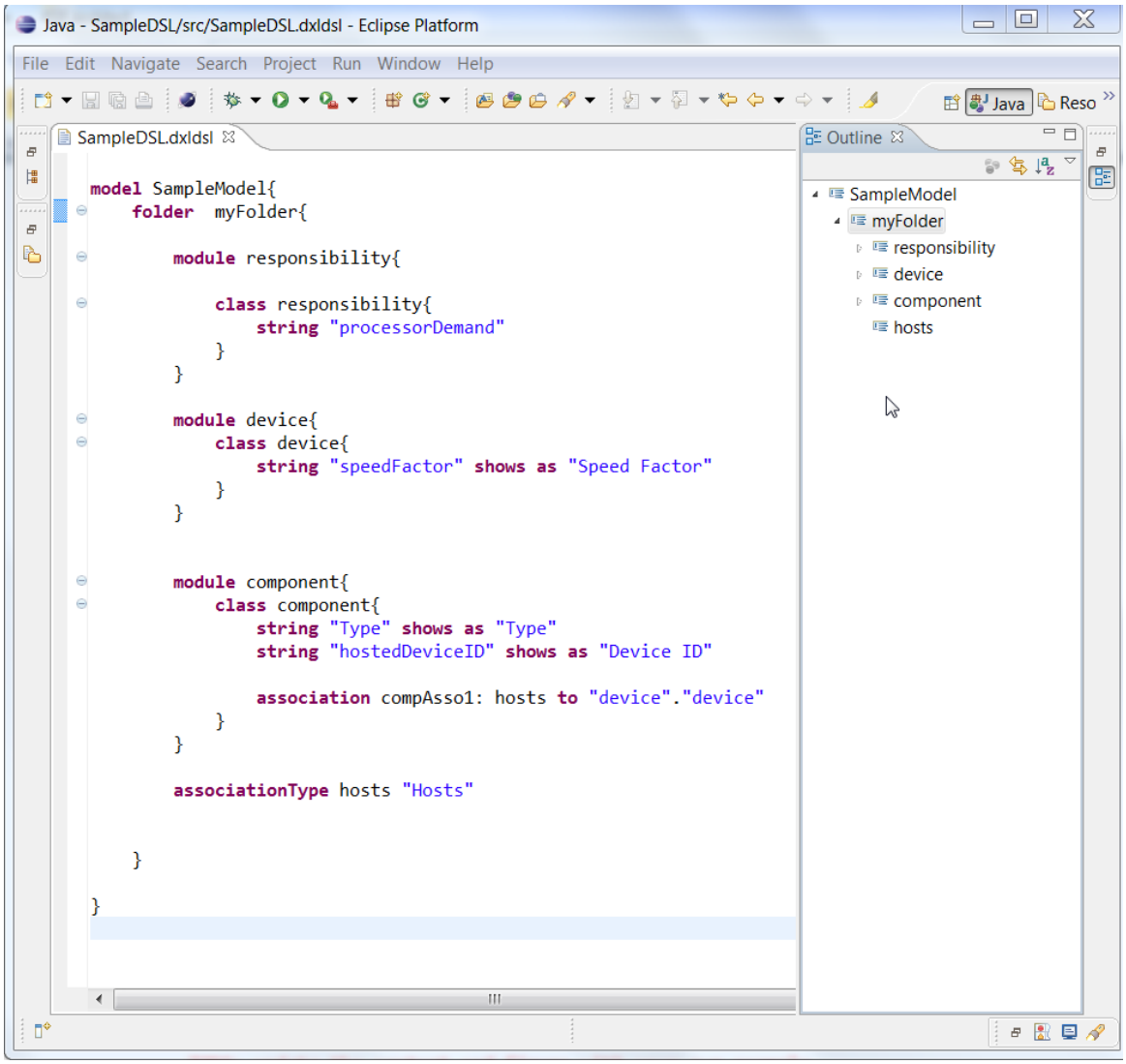

Figure 38 A simple MT-DSL modeling language description

3.7 Summary

This chapter describes a new DSL for model traceability management. Section 3.1 presented the metamodel for MT-DSL, whereas Section 3.2 described how a DSL language is generated using Xtext. Section 3.3 defined the grammar of MT-DSL and Section 3.4 discussed the generation of an Ecore model for the DSL Section 3.5 introduced different editor features including content assistance and error handling. This section also describes how to provide error feedback on domain-specific errors, and how to customize the editor using Preferences on the look of the editor and on templates. Section 3.6 briefly discussed a simple example language described with MT-DSL.

The next chapter will present our Xtend-based approach for generating a DXL library for a given MT-DSL description.

4 Traceability Library Generation

This chapter describes the DXL traceability library generation from MT-DSL descriptions, using Xtend (from *b* to *c* in Figure 1). The DXL library is a collection of files containing functions meant to be used in IBM Rational DOORS [18], one of the most well-known Requirements Management System. The library's functions are meant to be invoked by a DXL script representing a model and generated by a modeling tool (see part *e* of Figure 2). This will allow the integration of modeling tools with an RMS. For the purpose of this thesis, we have used the existing DOORS library for jUCMNav [14] to understand the nature of such functions, in order to automate their generation for other modeling languages/subsets/tools. Some of the existing DXL functions are also reused.

4.1 Xtend

Xtend is a programming language having its root in Java but with many improvements. Xtend compiles to Java code and executes on a JVM in a way similar to any other compiled Java code [26]. Xtend supports Java's type system and generics, and can be seamlessly used with any other Java library [45]. Xtend has many enhancements to provide more powerful computing capabilities, together with syntactic simplifications (as seen in Section 2.5).

4.1.1 Xtend Support in Xtext

Xtend is promoted as an Xtext validation and transformation language. As soon as Xtext artifacts are generated for a project, as describe in Section 3.2 and Figure 27, a *code generator stub* is placed in the runtime project of the DSL. For the purpose of this thesis, we have implemented the Xtext project org.xtext.dsl.dxl as described in Figure 23. In our implementation project, Xtext produced the generator stub DxlDslGenerator.xtend in the package org.xtext.dsl.dxl as shown in Figure 39. Further detail on the content of the generated template Xtend file and the implementation will be discussed in Section 4.2.

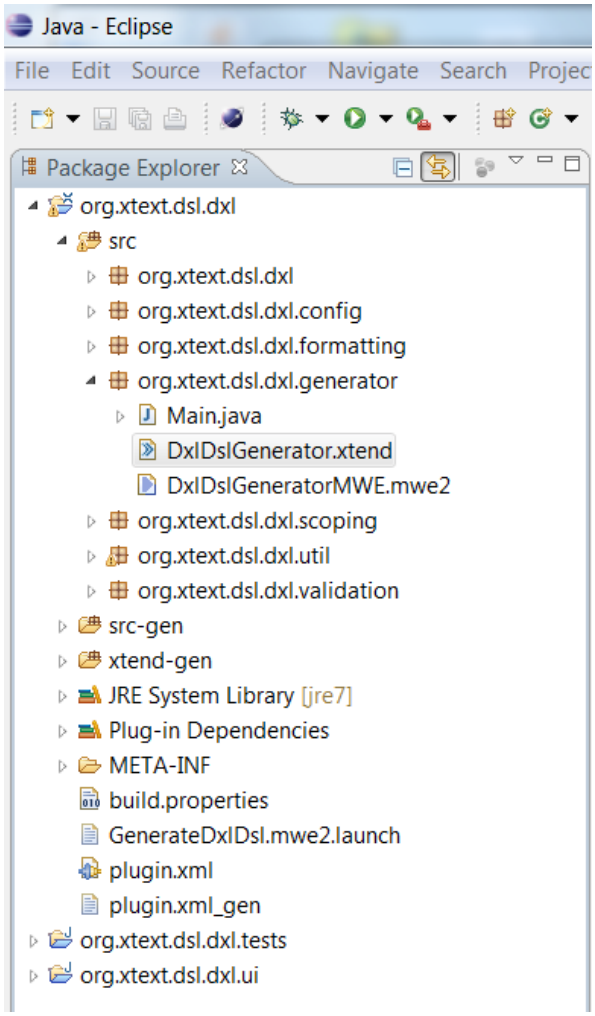

Figure 39 Generated code generator stub in the Package Explorer

4.1.2 Modeling Workflow Engine 2

As shown in Figure 39, the package also includes a generated MWE2 file called DxlDslGeneratorMWE.mwe2. Xtext 2.3 uses MWE2 as the engine for code generation. MWE2 is declarative and can be configured externally.

Workflows are typically executed in a single JVM, but they can be developed to have multiple components in multiple threads or processes. This is however not required for this thesis's implementation.

4.2 DXL Library Generation using Xtend

Xtend is used in this thesis to implement the code generator to produce the DXL library whose functions will be invoked when importing models in DOORS. This section represents an important contribution of this thesis.

4.2.1 Implementation Assumptions

This section describes the assumptions made for the purpose of implementing the DXL language generation using Xtend.

Default Class Attributes

To simplify the use of MT-DSL by engineers, it is assumed that each `Class` will contain implicitly three attributes by default. These are `name`, `id`, and `description`. For example, if a `Class` is defined in MT-DSL with no attribute, the code generator will still generate the three default attributes for that `Class` in the DXL library.

The language exceptionally allows to describe a `Class` *without* a `description` attribute. To override the default behaviour, the `Class` needs to be specified using the optional flag `noDescription` (see Section 3.1.4).

Default Report Options

By default, each `Module` is included in the DOORS report that is generated after each model import. The implementation allows overriding this default behaviour for a `Module` to be *excluded* in the report. This can be described using the optional flag `ignoreInReport` for the `Module` (see Section 3.1.3).

Any changes to an attribute of a `Class` flags the `Class` as *modified* in the generated report by default. The language allows ignoring changes to some attributes in a `Class`. This can be achieved by using the optional flag `ignored` (see Section 3.1.5).

4.2.2 General Utility Classes

This section describes the utility Java classes developed for implementing the code generator. These utility classes are included in the package org.xtext.dsl.dxl.util, available in Appendix C.

The first utility Java class is called `FileUtil` and contains a single utility method called *`getFileContents`*. This utility method reads content from any given file from the file system and returns the contents as a String. Its signature is:

```
static public String getFileContents(File file)
```

Package org.xtext.dsl.dxl.util also includes a utility class, `StringUtil`, which contains three methods for string manipulation, whose signatures are:

```
public static String getVarName(String str)
public static String getToFirstLower(String str)
public static String getToFirstUpper(String str)
```

The first method is `getVarName`, which converts any String to a format suitable to be used as a variable name in DXL, i.e., by replacing all spaces with underscores. Method `getToFirstLower` converts the first character of a string to lower case, to comply with general conventions where variable names start with a lower case letter. Similarly, method `getToFirstUpper` converts the first character of a string to upper case (for conventions related to class names).

4.2.3 Invoking DXL Generation in Xtext

Figure 39 shows that the Xtext-created generator stub in our implementation project is DxlDslGenerator.xtend, an Xtend file. As seen in Figure 40, `DxlDslGenerator` implements interface `IGenerator`, with a single method `doGenerate`. This method has two input parameters. The first is an Ecore `Resource` that provides access to the model described with the DSL. The second parameter is an `IFileSystemAccess` that provides the generator with access to the file system where the output files are to be generated.

```
/*
 * generated by Xtext
 */
package org.xtext.dsl.dxl.generator

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.IGenerator
import org.eclipse.xtext.generator.IFileSystemAccess

import static extension org.eclipse.xtext.xtend2.lib.ResourceExtensions.*
import org.xtext.dsl.dxl.*
import org.xtext.dsl.dxl.dxlDsl.*
import org.xtext.dsl.dxl.util.*
import org.eclipse.xtext.naming.IQualifiedNameProvider
import org.xtext.dsl.dxl.config.StaticContentProvider

import org.xtext.dsl.dxl.dxlDsl.Class

import com.google.inject.Inject


class DxlDslGenerator implements IGenerator {

    override void doGenerate(Resource resource, IFileSystemAccess fsa) {
```

Figure 40 Xtext generated generator stub *DxlDslGenerator*

All the implementation code developed for the purpose of DXL library generation is invoked from the `doGenerate` method and will be discussed in the following sections.

4.2.4 Initialization

The `doGenerate` method starts by calling the `initialize` method in class `UtilitiesHelper` in package `org.xtext.dsl.dxl.util`.

```
// initialization
UtilitiesHelper::initialize()
```

The `initialize` method initializes a list of files (field `fileList` in `UtilitiesHelper`) to an empty list. This is a list of DXL files, whose names and number depend on the MT-DSL description provided as input, that is being tracked during the transformation to DXL.

4.2.5 Global Variable Descriptions

In the implementation of the DXL library, a file is created to have all global configurations for the DXL code. As this implementation is based on the existing jUCMNav DXL library, we reuse the static portion of the file `Global.dxl` that would not change for any MT-DSL description. The static portion is included in a file called `GlobalStatic.dxl` and is loaded using utility class `StaticContentProvider` included in the `org.xtext.dsl.dxl.config` package.

The generation of the dynamic portion of the existing jUCMNav DXL file `Global.dxl` makes use of a helper class `GlobalVariableHelper` with a method called `createDynamicContent`. As described below, the `Global.dxl` file includes global variables for all modules and association types to be referenced in the library.

```
public static String createDynamicContent
            (org.xtext.dsl.dxl.dxlDsl.Folder f){

StringBuffer dc = new StringBuffer();
            // comments
      dc.append("\n\n");
      dc.append("// global variables\n");

      // folders
      dc.append("// folder \n");

      dc.append("Folder " +
      GeneratorHelper.getFolderVariableName(f) + " \n");
      dc.append("\n");

      // modules
      dc.append("// all  formal modules \n");
      for (Module m:f.getModule()){
            dc.append("Module " +
            GeneratorHelper.getModuleVariableName(m)
                        + " \n");
      }

      //module for links
      dc.append("\n");
      dc.append("// Link modules \n");
      for (AssociationType at: f.getAssociationType()){
            dc.append("Module " +
                  GeneratorHelper.getLinkModuleVariableName(
                        at) + " \n");
      }
```

```
        //Global String
        dc.append("\n\n") ;
        dc.append("// global constants\n") ;
        dc.append("\n\n");

        for (Module m:f.getModule()){
              dc.append("const string "
                    + GeneratorHelper.getModuleFlieVariable(m)
                    + " = \"" + getModuleFileName(m)
                    + "\"  \n") ;
        }

        for (AssociationType at: f.getAssociationType()){
              dc.append("const string "
                    +GeneratorHelper.getLinkFileVariable(at)
                    + " = \"" + at.getLinkFileName()
                    + "\"  \n") ;
        }

        return dc.toString();
  }
```

4.2.6 DXL Utilities Library

The existing DXL library for jUCMNav includes some utility functions in the file `ModuleUtilities.dxl`. No modification for this utility DXL code is required and hence it is included as is (with a few adaptations for generating the file in the desired location with desired file name described later in Section 4.3) in the generated DXL code for the MT-DSL description. The following is the portion of the `doGenerate` method in `DxlDslGenerator` that produces the code from the `ModuleUtilities.dxl`.

```
// Generating ModuleUtilities.dxl
var moduleUtilitiesFileName =
      UtilitiesHelper::MODULEUTILITIES_FILE_NAME
var moduleUtilitiesFile =  UtilitiesHelper::OUTPUT_LOCATION
           + moduleUtilitiesFileName
var moduleUtilitiesFileContent =
      StaticContentProvider::getContentFromStaticFile(
                  "ModuleUtilitiesStatic.dxl")
fsa.generateFile( moduleUtilitiesFile ,
                              moduleUtilitiesFileContent )
UtilitiesHelper::addFileToList(moduleUtilitiesFileName);
```

The method `getContentFromStaticFile` is used to copy this utility portion of the DXL code from a file called `ModuleUtilitiesStatic.dxl`.

4.2.7 DXL Library for Import Initialization in DOORS

The following is the portion of the `doGenerate` method that produces dynamic DXL code based on the MT-DSL modeling language description, which is similar to the DXL found in the `InitExit.dxl` file from the existing jUCMNav DXL library. This portion of the DXL library contains the helper DXL functions related to the *initialization* of the import process in DOORS. This generated file also contains the helper DXL functions that will *finalize* the import process. The code allows generating the file in the desired location with a desired file name, described later in Section 4.3.

```
//InitExit.dxl
var initExitFileContent = ""

for(f:resource.allContentsIterable.filter(typeof(
           org.xtext.dsl.dxl.dxlDsl.Folder))){

initExitFileContent = initExitFileContent +
           InitExitHelper::getInitExitFileContent(f)
}

var initExitFileName = UtilitiesHelper::INITEXIT_FILE_NAME
var initExitFile =  UtilitiesHelper::OUTPUT_LOCATION +
 initExitFileName

fsa.generateFile( initExitFile , initExitFileContent )
UtilitiesHelper::addFileToList(initExitFileName);
```

As seen in the code above, a helper class called `InitExitHelper` is used to generate the dynamic portion of the code based on the MT-DSL description. The content from method `getInitExitFileContent` from `InitExitHelper` is shown below. It invokes many other methods in the helper class `InitExitHelper`, each of which generates a DXL function based on the MT-DSL description. The details of these implementation methods are included in Appendix C.

```
public static String getInitExitFileContent(Folder f){
  StringBuffer content = new StringBuffer();
  String fileComents = getFileComments();
  String checkCreateFolderMethodContents =
            getCheckCreateFolderMethodContents(f);
  String saveCloseAllModulesMethodContents =
            getSaveCloseAllModulesMethodContents();
  String checkDeletedModulesMethodContents =
            getCheckDeletedModulesMethodContents(f);
  String removeModuleMethodContents =
            getRemoveModuleMethodContents();
  String removeLinkModulesMethodContents =
            getRemoveLinkModulesMethodContents(f);
  String checkCreateModuleMethodContents =
            getCheckCreateModuleMethodContents(f);
  String checkCreateModulesMethodContents =
            getCheckCreateModulesMethodContents(f);
  String checkCreateLinkModulesMethodContents =
            getCheckCreateLinkModulesMethodContents(f);
  String openModulesMethodContents =
            getOpenModulesMethodContents(f);
  String checkStartedFromModuleMethodContents =
            getCheckStartedFromModuleMethodContents(f);
  String saveCloseFinalMethodContents =
            getSaveCloseFinalMethodContents(f);

  content.append(fileComents);
  content.append(checkCreateFolderMethodContents);
  content.append(saveCloseAllModulesMethodContents);
  content.append(checkDeletedModulesMethodContents);
  content.append(removeModuleMethodContents );
  content.append(removeLinkModulesMethodContents );
  content.append(checkCreateModuleMethodContents );
  content.append(checkCreateModulesMethodContents );
  content.append(checkCreateLinkModulesMethodContents );
  content.append(openModulesMethodContents);
  content.append(checkStartedFromModuleMethodContents );
  content.append(saveCloseFinalMethodContents);
  content.toString();
}
```

4.2.8 DXL Library for Modules

The DXL library is created to support the import in DOORS of all `Modules` defined in the MT-DSL description. The following code from the `doGenerate` method shows that one DXL file is produced per `Module`. The name of the file is generated from the name of the `Module` using the method called `getLibFileNameForModule` in the helper class `UtilitiesHelper`.

```
for(m:resource.allContentsIterable.filter(typeof
                (org.xtext.dsl.dxl.dxlDsl.Module))){

            // getting the object name as file name
            var filename =  UtilitiesHelper::
                  getLibFileNameForModule(m)
            var file = UtilitiesHelper::OUTPUT_LOCATION + fileName
            fsa.generateFile(file , m.compile)
            UtilitiesHelper::addFileToList(fileName);
      }
```

Figure 41 shows the Xtend function compiles within the Xtend generator that takes a `Module` described in MT-DSL as input and generates the content for the DXL library for the corresponding DOORS module.

```
    def compile(org.xtext.dsl.dxl.dxlDsl.Module m)
    '''
/*****************************************************************************
// Author Anisur Rahman, Feb 2012
// - Automated DXL generation Based on earlier version by Gunter Mussbacher
*/

/*****************************************************************************
// Author Anisur Rahman, Feb 2012
// - imports «m.name» (updates object if it exists otherwise creates new one)
// - always returns true
// - assumptions for this function
//  - «m.name»Module exists and is ready to be used
*/
«FOR c:m.classes»

«GeneratorHelper::createFileVariableForImage(m,c)»

bool «c.name»(«GeneratorHelper::getArgumentListForAttributes(m,c)»)
{
    Object foundObject«GeneratorHelper::declareLastVariableForImage(m,c)»

    «GeneratorHelper::declareLocalVariableForIntAttribute(m,c)»
    foundObject = findObject( «c.name.toFirstLower»ID, «m.name.toFirstLower»Module )
    if ( null foundObject ) {
        «GeneratorHelper::getDefaultforObjectNotFound(m,c)»
        foundObject."ID" = «c.name.toFirstLower»ID
        «GeneratorHelper::getStrForSetNameForObjectNotFound(m,c)»
        «IF !(c.noDescription)»
        foundObject."Description_" = «c.name.toFirstLower»Description
        «ENDIF»
```

Figure 41 Xtend function compile within *`DxlDslGenerator`*

As described in this figure, the codes loops through all available classes (and their attributes) inside the `Module` and generates DXL code for each class. The details of the implementation are available in Appendix C.

One important thing to observe in the code generated for `Class`es in `Module`s is that it handles the updates of corresponding DOORS objects and attributes. For example, observe the MT-DSL description for `Class component` in `Module component` from Appendix B:

```
module component{
    class component{
        string "Type" shows as "Type"
        string "hostedDeviceID" shows as "Device ID"

        association compAsso1: hosts
                              to "device"."device" "Device ID"
    }
}
```

The commented DXL code generated for this `Module` (in file `component.dxl`) is provided in Figure 42. This code respects the same logic as the original DXL code manually produced by Jiang [27][41]. Functions are created for each `Class` in the `Module` (there is only one here) and the function signature contains all `Attribute`s, including the implicit ID, name, and description (strings). Their names are prefixed with the name of the `Class`, and spaces are removed. If no object with the same ID exists in the corresponding DOORS module, then one new DOORS object is created and its attributes (including predefined ones for this RMS, e.g., Object Heading) are set. If however an object with the same ID already exists, then only its attributes of interest (i.e., ID, name, description, and additional MT-DSL attributes not prefixed with `ignored`) are updated while keeping track of the history of changes in DOORS (otherwise, only the latest value is kept, without the history).

Such a function can then be invoked by a DXL script describing a particular model (part *e* in Figure 2), e.g., with:

```
component( "38", "Component1", "First sample component",
           "process", "Device3" )
```

```
/*****************************************************************
// Author Anisur Rahman, Feb 2012
// - imports component (updates object if it exists otherwise creates
//   a new one)
// - always returns true
// - assumptions for this function
//    - componentModule exists and is ready to be used
*/

bool component(string componentID, string componentName,
               string componentDescription, string componentType,
               string componentHostedDeviceID)
{
      Object foundObject

      foundObject = findObject( componentID, componentModule )
      if ( null foundObject ) {
            foundObject = createNewObject(componentModule )
            foundObject."ID" = componentID
            foundObject."Object Heading" = componentName
            foundObject."Name_" = componentName
            foundObject."ObjectType_" = "component"
            foundObject."Description_" = componentDescription
            foundObject."Type" =  componentType
            foundObject."Device ID" =  componentHostedDeviceID
            foundObject."New" = true
            foundObject."Deleted" = false
       } else {
            if( foundObject."Name_" "" != componentName ) {
                  foundObject."Object Heading" = componentName
                  foundObject."Name_" = componentName
            }
            if( foundObject."Description_" "" != componentDescription )
                  foundObject."Description_" = componentDescription
            if( foundObject."Type" "" !=  componentType  )
                  foundObject."Type" = componentType
            if( foundObject."Device ID" "" != componentHostedDeviceID )
                  foundObject."Device ID" = componentHostedDeviceID

            foundObject."Deleted" = false
      }

      debug("imported component " foundObject."ID" "\n",3)
      return true

}
```

Figure 42 DXL code generated for the MT-DSL module `component`

4.2.9 DXL Library to Create Reports

The next section in the `doGenerate` Xtend function generates the DXL code used to create reports in DOORS. The DOORS reports for model import provide feedback about the import operation, and on re-import operations about models that have been updated or modified since last re-import or initial import.

```
//create report

var reportContentFolders = ""
for(f:resource.allContentsIterable.filter
      (typeof(org.xtext.dsl.dxl.dxlDsl.Folder))){
   reportContentFolders = reportContentFolders +
               ReportHelper::createContentForFolder(f)
}

reportContentFolders =
     ReportHelper::createDynamicContentForReport
                (reportContentFolders)

var content =
     StaticContentProvider::getContentFromStaticFile
     ("ReportStatic.dxl")
content = content + reportContentFolders

var reportFileName =  UtilitiesHelper::REPORT_FILE_NAME
var reportFile =  UtilitiesHelper::OUTPUT_LOCATION
                     + reportFileName

fsa.generateFile(reportFile, content )

UtilitiesHelper::addFileToList(reportFileName);
```

As seen in the above code, the portion of the DXL code that does not change with the model is loaded using the utility class `StaticContentProvider`. The dynamic portion of the DXL code to generate the report that depends on the MT-DSL description is generated invoking the Java method `createDynamicContentForReport` in the helper class `ReportHelper`. The DSL allows the option to exclude any Module in the report. This can be described with the optional attribute `ignoreInReport`. The content of the utility class `ReportHelper` is attached in Appendix C.

4.2.10 DXL Library to Create Links

MT-DSL allows describing links (Associations) between `Class`es of a same or different `Module`s. DXL code is generated for allowing importing the links in DOORS link modules. The portion of the code in the generator class to create links is described below.

```
// Generating Link
var linkContent = ""
for(m:resource.allContentsIterable.filter(typeof(
            org.xtext.dsl.dxl.dxlDsl.Module))){

      if (GeneratorHelper::isModuleContainAssociation(m)){
            linkContent = linkContent + m.generateLinks
      }
}

var linkFileName =  UtilitiesHelper::LINK_FILE_NAME
var linkFile = UtilitiesHelper::OUTPUT_LOCATION +
       linkFileName
fsa.generateFile(linkFile , linkContent)
UtilitiesHelper::addFileToList(linkFileName);
```

The Xtend function `generateLinks` is invoked to generate the DXL code for links, all stored by default in file `Links.dxl`. The details of the code are included in Appendix C.

The `component` example from Appendix B contains an MT-DSL instruction that declares an association of type hosts from components to devices:

```
association compAsso1: hosts to "device"."device" "Device ID"
```

This instruction generates the DXL code found in Figure 43. Again, this code respects the same logic as the original DXL code manually produced by Jiang [27][41]. This function essentially tries to find the device with the corresponding ID and, if it exists and if the component is not logically deleted, then a link of the required type (`hosts` here) is created. Note the conciseness of the syntax for creating DOORS links (source->link->target).

```
/*****************************************************************
// Author Anisur Rahman Feb 2012
// Automated DXL generation Based on earlier version by Jean-François
// Roy, Gunter Mussbacher, Bo Jiang
// - creates links within the component module and from the component
//   module to the definition modules
// - always returns true
// - assumptions for this function
// - The other modules to create link with already exist and are ready
//   to be used
// - referencesLinkModule, refinesLinkModule, boundToLinkModule exist
//   and are ready to be used
*/

bool createcomponentLinks() {
    Object currentObject, targetObject
    string s, p
    int i, j
    bool b

    for currentObject in componentModule do {
        // skip over deleted objects which could not be removed
        // because of links
        b = currentObject."Deleted"
        if( !b ) {
            if( currentObject."ObjectType_" "" == "component" ) {
                // create link between component and device in
                // device module
                targetObject = findObject( currentObject."Device ID" "",
                                           deviceModule )

                if ( !( null targetObject ) )
                    currentObject->fileNameLinkHosts->targetObject
            }
        }
    }
    debug("created component links\n", 3)
    return true
}
```

Figure 43 DXL code generated for an MT-DSL association from `component` to `device`

4.2.11 DXL Library to Start the Import Procedure

This portion of the DXL code is responsible to start importing models in DOORS. The process starts with displaying confirmation dialogs to users and continues upon receiving user confirmation. This code is also responsible for invoking other generated functions including importing modules and links. The process also provides feedback about the import process and generates a report after the import process ends.

```
//import.dxl
var importFileContent = ""

for(f:resource.allContentsIterable.filter(typeof(
      org.xtext.dsl.dxl.dxlDsl.Folder))){

      importFileContent = importFileContent +
            ImportFileHelper::getImportFileContent(f)
}

var importFileName = UtilitiesHelper::IMPORT_FILE_NAME
var importFile =  UtilitiesHelper::OUTPUT_LOCATION
            +  importFileName

fsa.generateFile( importFile , importFileContent )
UtilitiesHelper::addFileToList(importFileName);
```

The helper Java class called `ImportFileHelper` is invoked to generate the DXL code for the import process. The Java class `ImportFileHelper` has different methods to generate code responsible for initiating, importing, and finalizing the import process (see Appendix C).

4.2.12 DXL Utility File with the List of Library Files

This section describes the generation of a utility DXL file that includes import statements for all other generated DXL library files. To use the generated DXL library, the DXL script describing a model to be imported (Figure 2*e*) only needs to import this utility file as this file itself contains import statements for all the generated library files.

```
// utility file - having list/import of all other files
var utilityFileName= UtilitiesHelper::UTILITY_FILE_NAME
var utilityFile =  UtilitiesHelper::OUTPUT_LOCATION
                      + utilityFileName
var utilityFilecontent =
     UtilitiesHelper::getUtilityFileContents
fsa.generateFile(utilityFile , utilityFilecontent)
```

The utility Java method `getUtilityFileContents` in class `UtilitiesHelper` is invoked to get the contents of this utility DXL file. The Java code for this class is provided in Appendix C.

4.3 Configuring the DXL Generation

In order to accommodate changing the default behaviour of the code generator in the future, the Java class `UtilitiesHelper` in package `org.xtext.dsl.dxl.util` has been used to centralize all configuration items. The following declarations enable the configuration of the code generator.

```
public static final String OUTPUT_LOCATION = "/output/" ;
public static final String DOORS_LIB_LOCATION =
                      "addins/DSL/lib/";
public static final String UTILITY_FILE_NAME =
                      "Utilities.dxl";
public static final String REPORT_FILE_NAME = "Report.dxl";
public static final String LINK_FILE_NAME = "Links.dxl";
public static final String GLOBAL_FILE_NAME = "Global.dxl";
public static final String IMPORT_FILE_NAME = "Import.dxl";
public static final String INITEXIT_FILE_NAME =
                                "InitExit.dxl";
public static final String MODULEUTILITIES_FILE_NAME =
                           "ModuleUtilities.dxl";
```

As described above, the output location where the code will be generated can be modified. The name of the library files for `Modules` are determined from their name. The names for all other DXL library files can be modified here. The configuration also allows describing the location where library files would be placed in DOORS.

4.4 Generating a DXL Library to Import jUCMNav Models in DOORS

For the purpose of the thesis, we have created a sample MT-DSL description, presented in Appendix B, which captures part of the URN traceability view currently supported in the existing DXL library for jUCMNav. This is essentially done to compare the code generated by our transformation with an existing baseline, and to enable DXL model files generated by jUCMNav to use our library. Figure 44 uses the outline view of the MT-DSL editor to summarize all currently supported modules and links in this library.

Figure 44 Outline view for jUCMNav model described in MT-DSL

Table 2 describes the generated DXL library for this MT-DSL description of a view of jUCMNav/URN, together with the size (lines of code), number of DXL functions, and description of each file.

Table 2 Generated DXL library for jUCMNav/URN models

Library File Name	**Size**	**Func**	**Description**
import.dxl	192	2	This DXL library file provides the utility method that would be invoked to start the model import process in RMS. This has the DXL function `beginImport` to start the import process.
InitExit.dxl	596	11	This library file contains all the DXL functions to initialize and finalize the import process (including GUI interactions).
ModuleUtilities.dxl	169	7	This file includes the helper DXL functions that are invoked during the model import process in DOORS.
Utilities.dxl	21	0	This file contains the list of import statements to import all other library files.
Report.dxl	199	6	This DXL library file contains generated DXL code for creating a report at the end of the import process.
Links.dxl	275	4	This file contains DXL library code for the links described in the modules.
Global.dxl	136	0	This file declares global variables used in all DXL files in the library.
Actor.dxl	41	1	DXL file for module Actor
Component.dxl	49	1	DXL file for module Component
Device.dxl	44	1	DXL file for module Device
ElementLink.dxl	99	2	DXL file for module ElementLink
GrlDiagram.dxl	277	5	DXL file for module GrlDiagram
IntentionalElement.dxl	48	1	DXL file for module IntentionalElement
Map.dxl	233	4	DXL file for module Map
Responsibility.dxl	44	1	DXL file for module Responsibility
Strategy.dxl	44	1	DXL file for module Strategy

For the single MT-DSL description in Appendix B (189 lines, 5.3 KB), a total of 16 DXL files were produced (2467 lines of commented DXL code, 96.6 KB), which contain a total of 47 DXL functions. MT-DSL descriptions are hence much more concise than the DXL equivalent (in addition to being less prone to errors given the support from the Eclipse-based editor). 7 files (in *italic* in Table 2) are always generated for every MT-DSL description, and then one additional file per MT-DSL module (in **bold** in Table 2) is also produced. The number of functions in the later module files corresponds to the number of classes that each module defines. The number of functions in `Links.dxl` corresponds to the number of modules that have `association` instructions.

4.5 Installing the Generated DXL Library

The generated DXL library needs to be installed in DOORS prior to using it while importing models (DXL scripts) from other tools. If DOORS 8.x or 9.x is installed in its default location, the generated library should be installed in the '<path to DOORS>\lib\dxl\addins' folder, as shown in Figure 45.

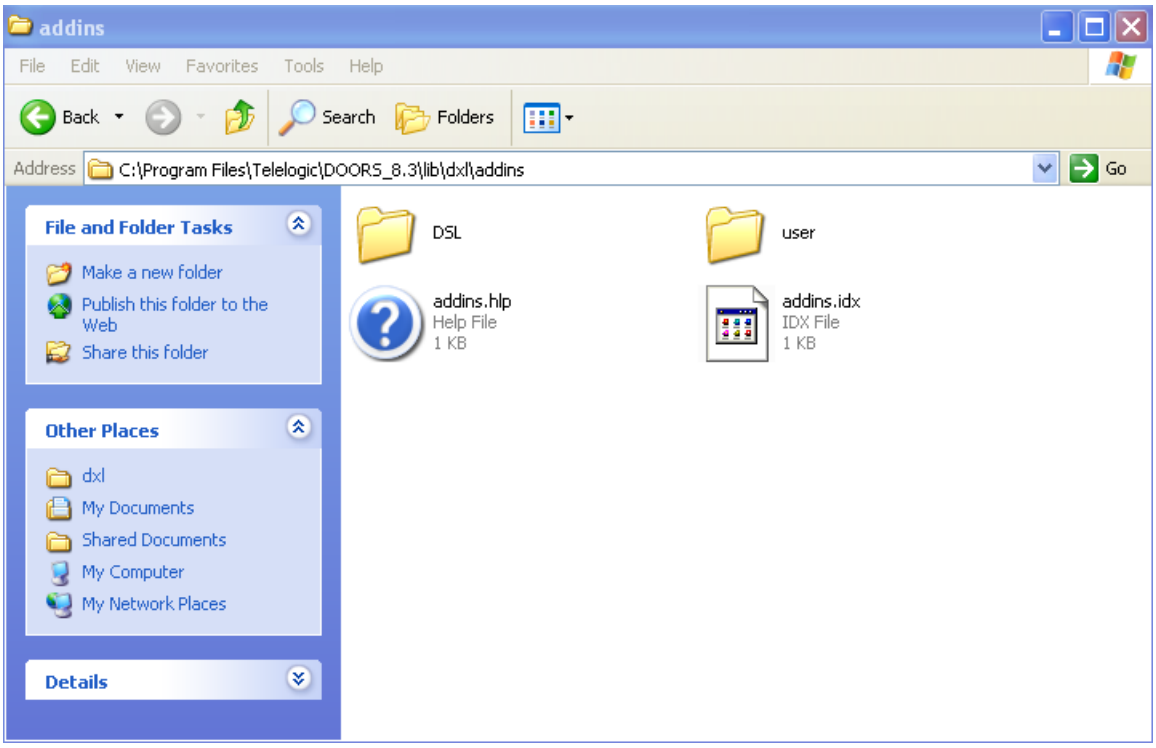

Figure 45 Installing DXL library in DOORS 8.x

However, the generated library can be its own package with its own directory structure. In this case, the library can be stored in a top folder called, for example, `DSL/lib`. The directory structure is used in the generated DXL file called `Utilities.dxl`, as shown in Figure 46. This file contains include statements for all other generated DXL files so that only this file needs to be included in DXL scripts invoking the library. The location is configurable, as explained in Section 4.3.

Utilities.dxl

```
3  /*****************************************************************************
4  // Generated version
5  // Author Anisur Rahman, Aug 2012
6  // Based on version by Gunter Mussbacher & initial version by Bo Jiang
7  */
8
9  #include "addins/DSL/lib/Global.dxl"
10 #include "addins/DSL/lib/ModuleUtilities.dxl"
11 #include "addins/DSL/lib/InitExit.dxl"
12 #include "addins/DSL/lib/Responsibility.dxl"
13 #include "addins/DSL/lib/Actor.dxl"
14 #include "addins/DSL/lib/IntentionalElement.dxl"
15 #include "addins/DSL/lib/Map.dxl"
16 #include "addins/DSL/lib/GrlDiagram.dxl"
17 #include "addins/DSL/lib/Component.dxl"
18 #include "addins/DSL/lib/Device.dxl"
19 #include "addins/DSL/lib/Elementlink.dxl"
20 #include "addins/DSL/lib/Strategy.dxl"
21 #include "addins/DSL/lib/Report.dxl"
22 #include "addins/DSL/lib/Links.dxl"
23 #include "addins/DSL/lib/Import.dxl"
24
```

Figure 46 Contents from generated DXL file 'Utilities.dxl'

4.6 Summary

This chapter provided details on the design and implementation of a DXL code generator for MT-DSL using Xtend. Section 4.1 described Xtend and its combined use with Xtext for code generation. Section 4.2 declared the assumptions made for the implementation and discussed the main functions of the code generation implementation, together with the DXL files generated, Xtend-based transformations, and illustrations based on the MT-DSL description example from Appendix B. Section 4.3 described generation items that can be configured or modified later with minimum effort. Section 4.4 discussed the files generated for the URN/jUCMNav MT-DSL description, and highlighted their purpose and the savings made by using MT-DSL, namely:

- A reduction of over 90% in the number of lines of code.
- One simple domain-specific file to manipulate (with a validating and user-friendly editor) instead of 16 files with error-prone DXL code.

Installation instructions were finally provided.

The next chapter will focus on two experiments aimed at validating the correctness of the generated libraries of two different languages by importing and re-importing modified models in DOORS, hence indirectly validating the MT-DSL language and the code generation.

5 Experiments and Validation

This chapter uses two experiments to validate the hypothesis in Section 1.2 as well as the solution proposed in this thesis. MT-DSL descriptions are provided for two different modeling languages for which DXL libraries are provided to track a specific view of the language. In addition, for each description, a model is created as a DXL script, imported in DOORS (by invoking the corresponding DXL library), modified, and re-imported to endure that consistency is maintained and that history is tracked. Section 5.1 uses a subset of a simple Finite State Machine (FSM) language, whereas Section 5.2 uses the subset of the User Requirements Notation (URN) supported by jUCMNav already discussed in Appendix B and in the previous chapter. Section 5.3 further validates the tool-supported approach against the use and evolution of links to elements external to the model. We have linked URN and FSM models (each model being composed of elements external to the other) and then modified and deleted elements with external links. These sections illustrate the steps used for the validation.

5.1 FSM Models

In this section, the thesis is partly validated by using a simple Finite State Machine (FSM) language.

5.1.1 FSM Metamodel and Tracked Subset

The FSM metamodel chosen for validation is shown in Figure 47. In a nutshell, an FSM model contain diagrams, each composed of states connected by transitions. Start and end states can be used, and super states (containing a diagram) can be used to decompose complex models hierarchically.

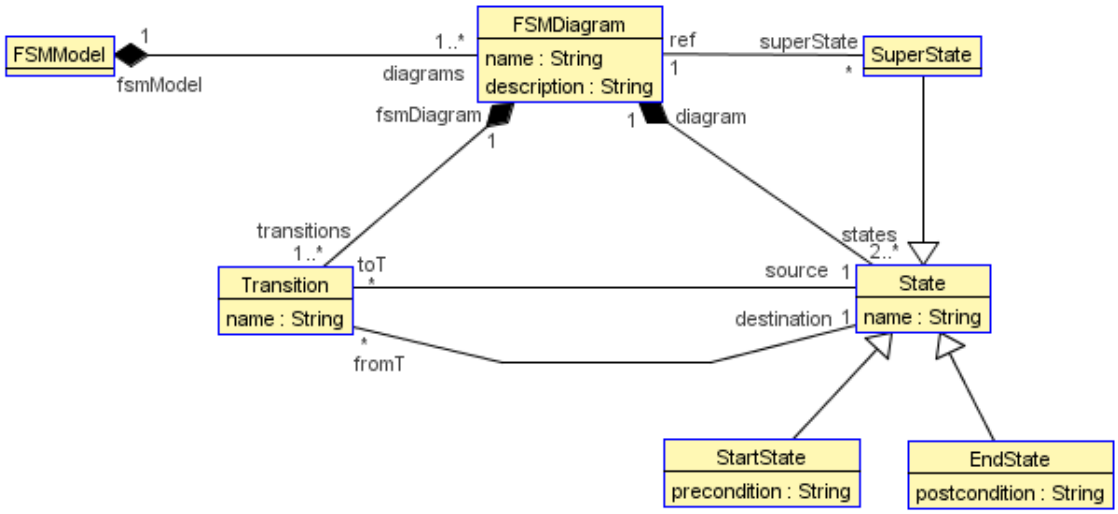

Figure 47 Finite State Machine (FSM) model

One does not have to track all modeling elements of a language in an RMS. A view can be defined, essentially as a subset of the classes, attributes and associations of a meta-model. Our selected FSM subset for the DXL library generation is described in Figure 48. The links mark with 'X' will not be tracked through model traceability in DOORS.

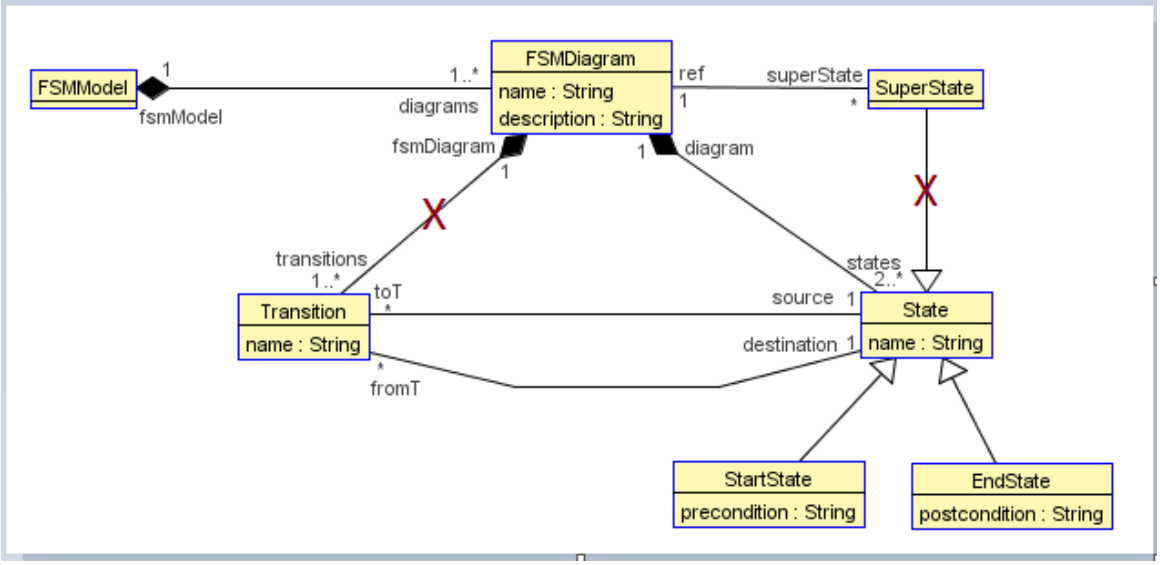

Figure 48 FSM subset of interest for traceability

5.1.2 MT-DSL Description

The FSM subset in Figure 48 is described using MT-DSL to generate the DXL library. For simplicity from a requirements management perspective, it was decided here to track only one state class, but with a type (describing which of the four classes/subclasses of the metamodel is used), and the union of their attributes and associations.

```
model fsmModel{

      folder fsmModel{

            module fsmDiagram{
                  class fsmDiagram{
                  }
            }

            module state{
                  class state{
                        string "Type"
                        string "preCondition" shows as "Precondition"
                        string "postCondition" shows as "Postcondition"
                        string "FSMparent" shows as "FSM diagram ID"
                        string "FSMincluded" shows as "Sub-FSM"

                        association diag : contains to
                             "fsmDiagram"."fsmDiagram" "FSM diagram ID"

                        association included : superStateOf to
                                  "fsmDiagram"."fsmDiagram" "Sub-FSM"
                  }
            }

            module transition{
                  class transition{
                        string "sourceID" shows as "Source ID"
                        string "destinationID" shows as
                                                "Destination ID"

                        association src_s : source to
                                        "state"."state" "Source ID"
                        association dest_s : destination to
                                      "state"."state" "Destination ID"
                  }
            }

            associationType superStateOf "Super State Of"
            associationType source "Source"
            associationType destination "Destination"
            associationType contains "Contains"

      }
}
```

5.1.3 DXL Library

Following the observations of Section 4.4, the list of files in the generated DXL library for the MT-DSL FSM description in Section 5.1.2 is shown in Figure 49.

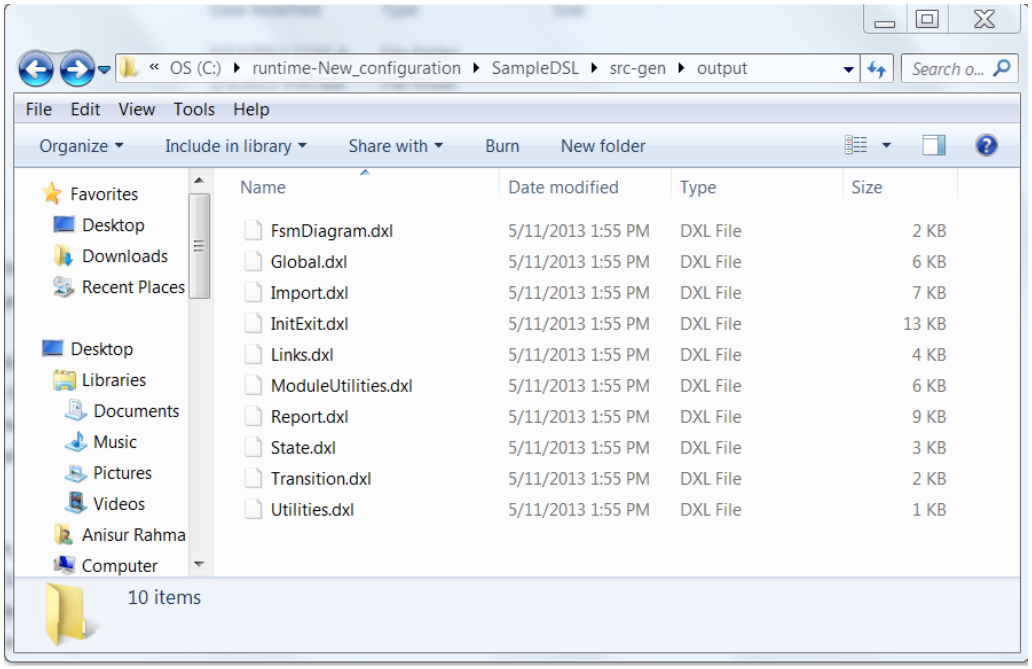

Figure 49 Generated DXL library for the chosen FSM language subset

5.1.4 Test Scenarios

We have used the following DXL script to invoke the generated library and to import the FSM model in DOORS.

```
#include "addins/DSL/lib/Utilities.dxl"
pragma runLim, 0

beginImport( "HelloWorldFSM" )

fsmDiagram ( "0", "Sub", "My simple sub-diagram" )

state ( "1", "S2", "S2", "StartState", "x=3", "", "0","0" )
state ( "2", "K", "Some state", "State", "", "", "0", "" )
state ( "3", "E3", "", "EndState", "", "x=0", "0", "" )

transition ( "4", "X1", "", "1", "2" )
transition ( "5", "X2", "", "2", "2" )
transition ( "6", "X3", "", "2", "3" )
endImport
```

This DXL script describes that the FSM model is to be imported in DOORS in a folder called 'HelloWorldFSM'. The script describes one `FSMDiagram`, 3 `State`s (of different `Type`s), and 3 `Transition`s, as explained in Table 3.

Table 3 FSM models described in the DXL script

FSM Object	ID	Name	Description	Other details
FSMDiagram	0	Sub	My simple sub-diagram	
State	1	S2	S2	Type : StartState Precondition : x=3 FSM Diagram ID : 0 Sub-FSM : 0
State	2	K	Some State	Type : State FSM Diagram ID : 0
State	3	E3		Type : EndState Postcondition: x=0 FSM Diagram ID : 0
Transition	4	X1		Source ID : 1 Destination ID : 2
Transition	5	X2		Source ID : 2 Destination ID : 2
Transition	6	X3		Source ID : 2 Destination ID : 3

5.1.5 Importing the FSM Model in DOORS

To import the FSM model in DOORS, in a selected folder of the database (e.g., FSM_Model), one needs to select 'Tools->Edit DXL…' as shown in Figure 50.

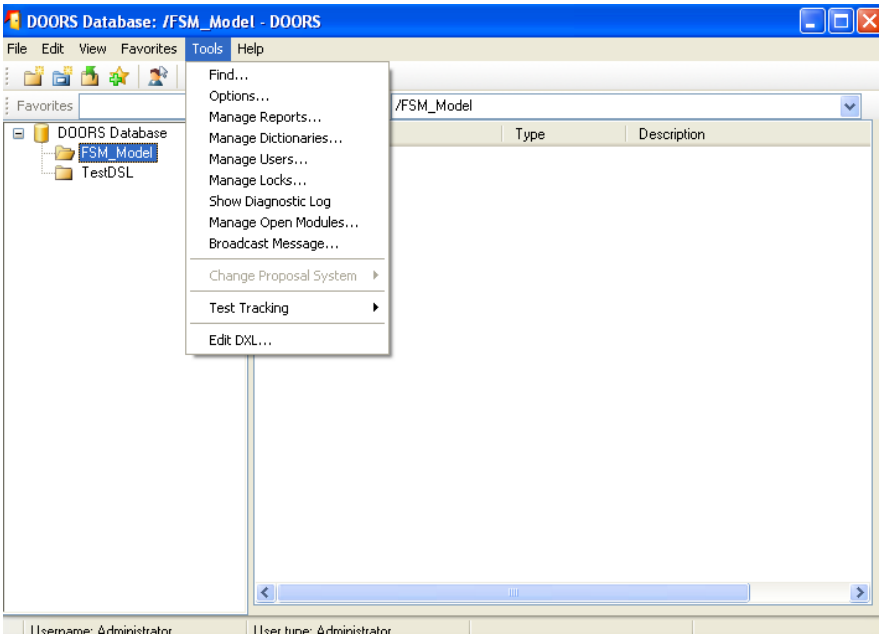

Figure 50 Tools menu in DOORS menu bar

In the DXL Interaction window, the DXL script can be loaded and then run (Figure 51).

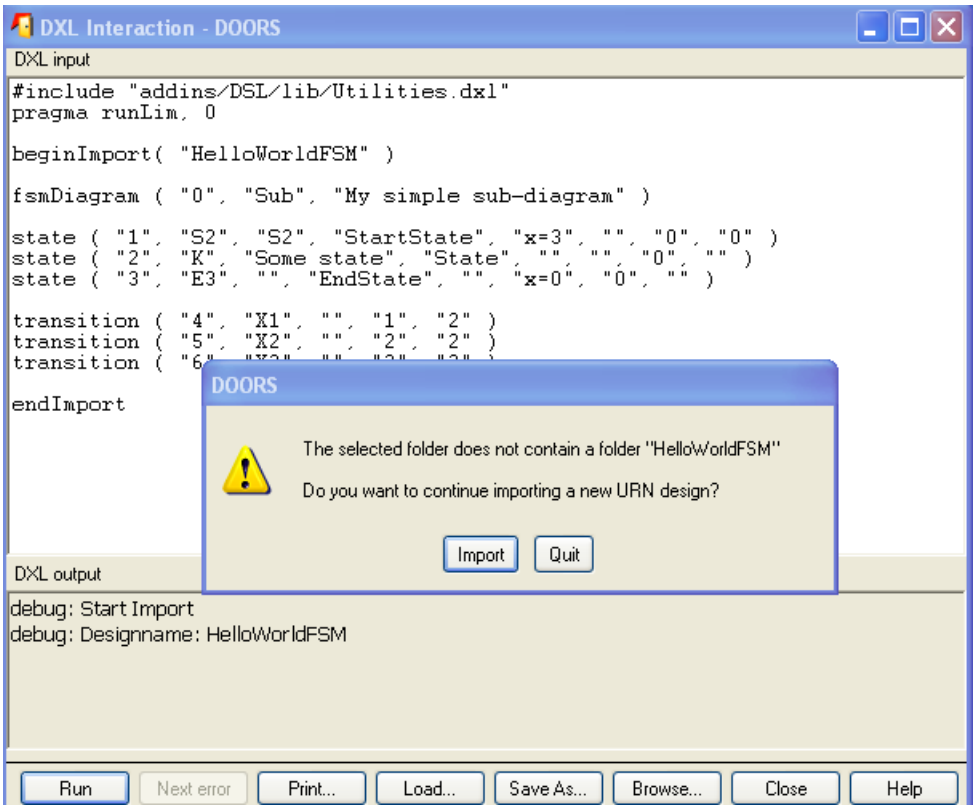

Figure 51 Confirmation window to run a DXL script

A confirmation window shows up as shown in Figure 51 (as a common mistake is to import the model in an incorrect folder). By clicking on 'Import', the script continues executing, and the required formal and link modules are populated in the DOORS database. The bottom of the DXL Interaction window provides feedback as the import progresses (see Figure 52).

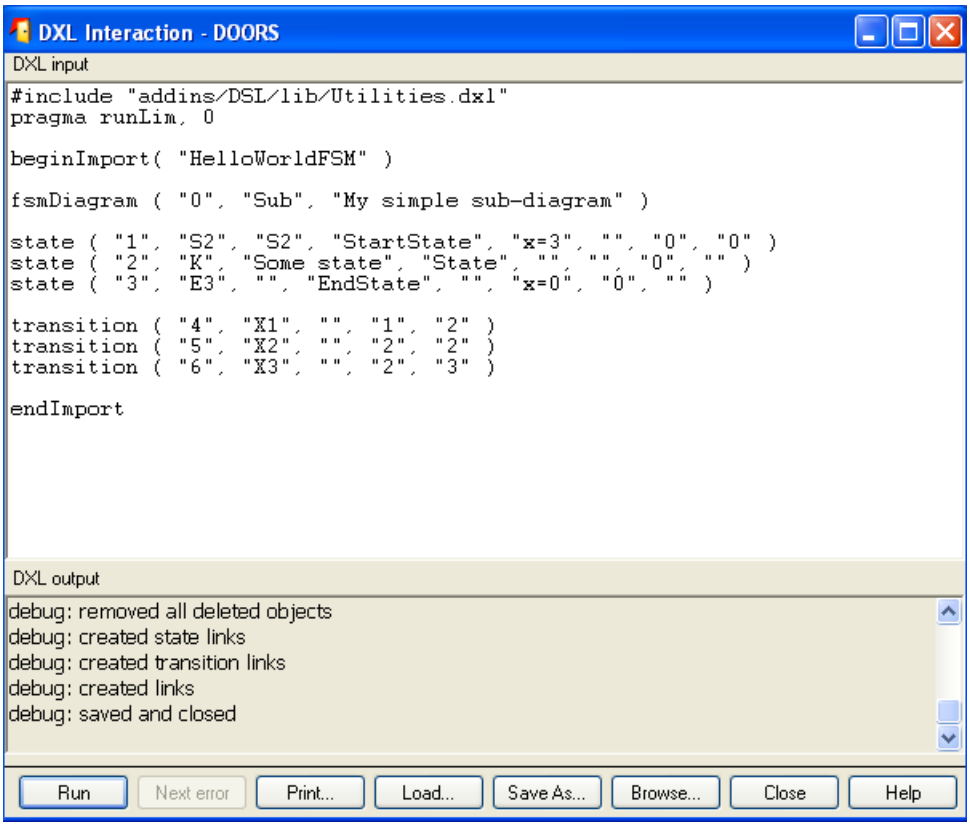

Figure 52 Completion of FSM model import

5.1.6 Results

At this point, a new subfolder is added for the model (HelloWorldFSM). By selecting 'View->Show Link Modules' one can see that this folder (displayed in Figure 53) has three formal DOORS modules for the three MT-DSL `modules`, as well as four DOORS link modules (one per MT-DSL `associationType`).

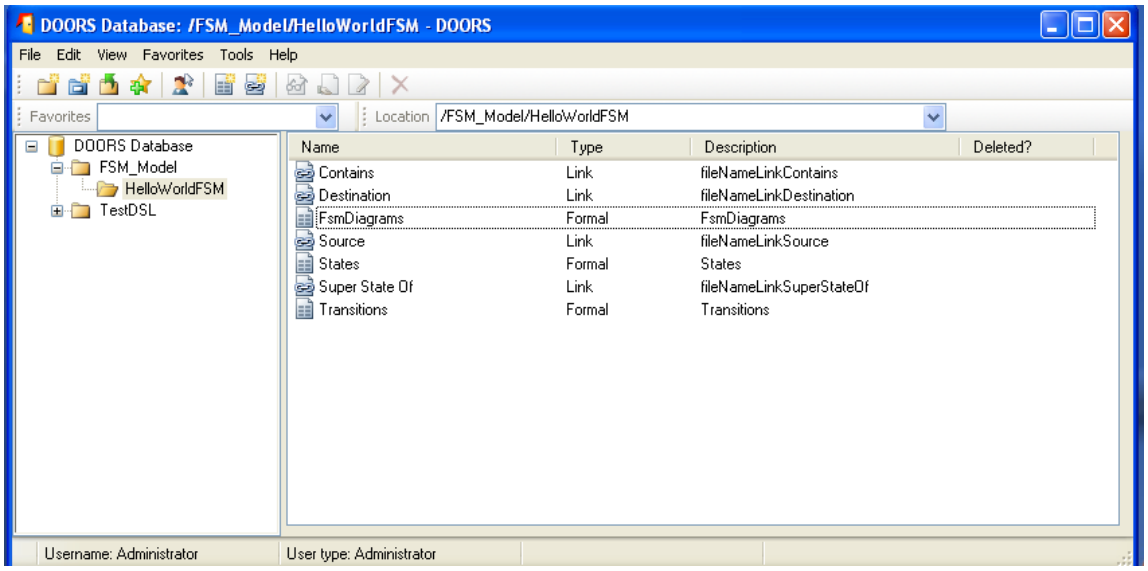

Figure 53 Formal and link modules in DOORS for the FSM model

By openingthe States formal module (Figure 54), we can see that the three states have been correctly imported, with their attributes. The yellow and red triangles indicate the presence of incoming and outgoing DOORS links.

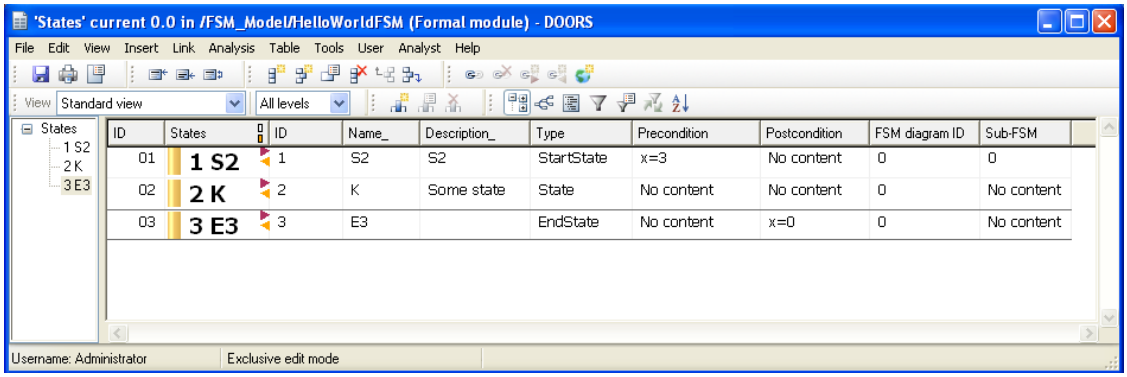

Figure 54 States formal module in DOORS

Similarly, Figure 55 shows the content of the Transitions module. The fsmDiagrams module contains only one object (Sub) and is not shown here.

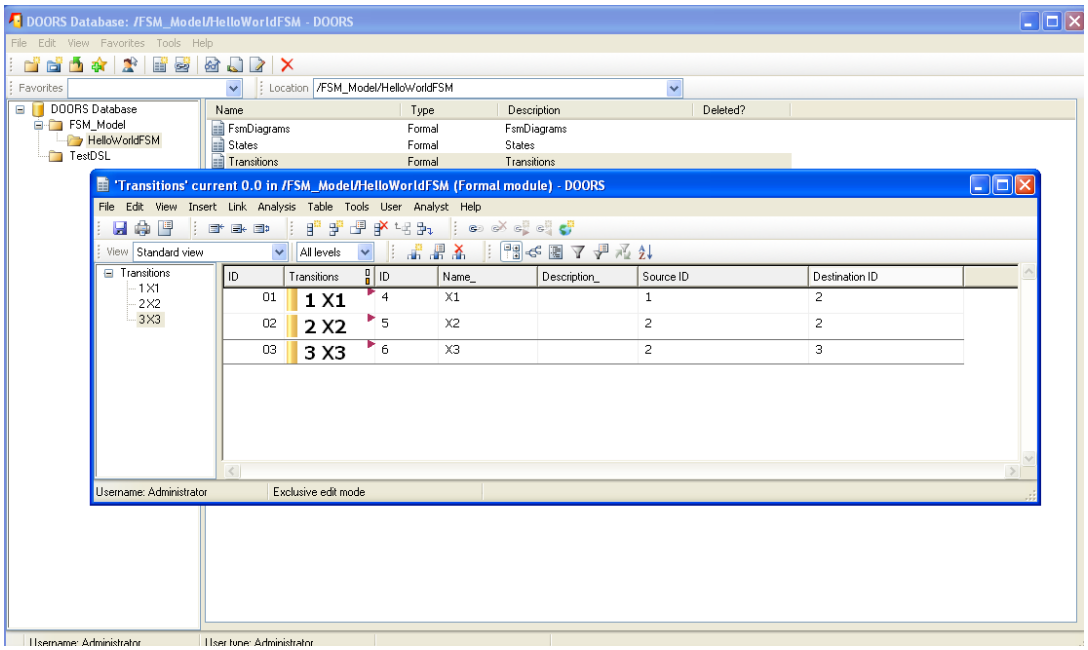

Figure 55 Transitions formal module in DOORS

To validate that the MT-DSL associations resulted in actual traceability links, DOORS allows to visualize link modules in a few ways, two of which are presented here. Figure 56 shows the links module Source with its links between Transtions and States.

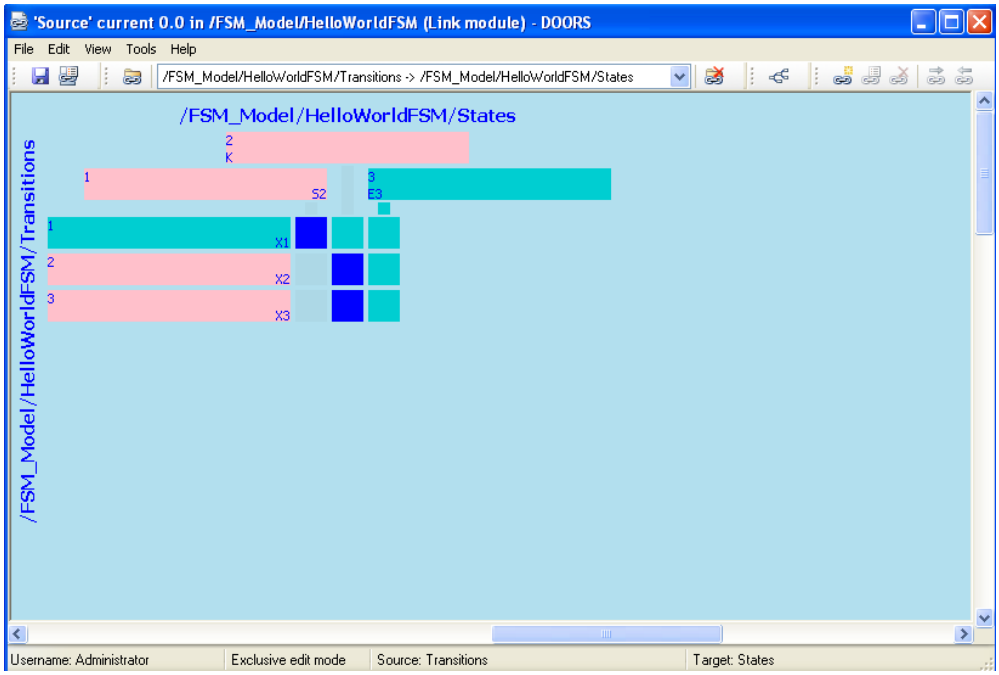

Figure 56 Source link module displaying links between transitions and states using a matrix in DOORS

In this traceability matrix view, a dark blue cell indicated the presence of a link, and the ones here correspond to the ones defined in the DXL script of the model. For instance, the source of transition X1 is S2.

Using a graphical traceability view, traceability links can also be displayed as a graph for a specific type of link. Figure 57 shows the Destination links between Transitions and States using this view. For example, the destination of transition X1 is S2, as specified in the DXL script.

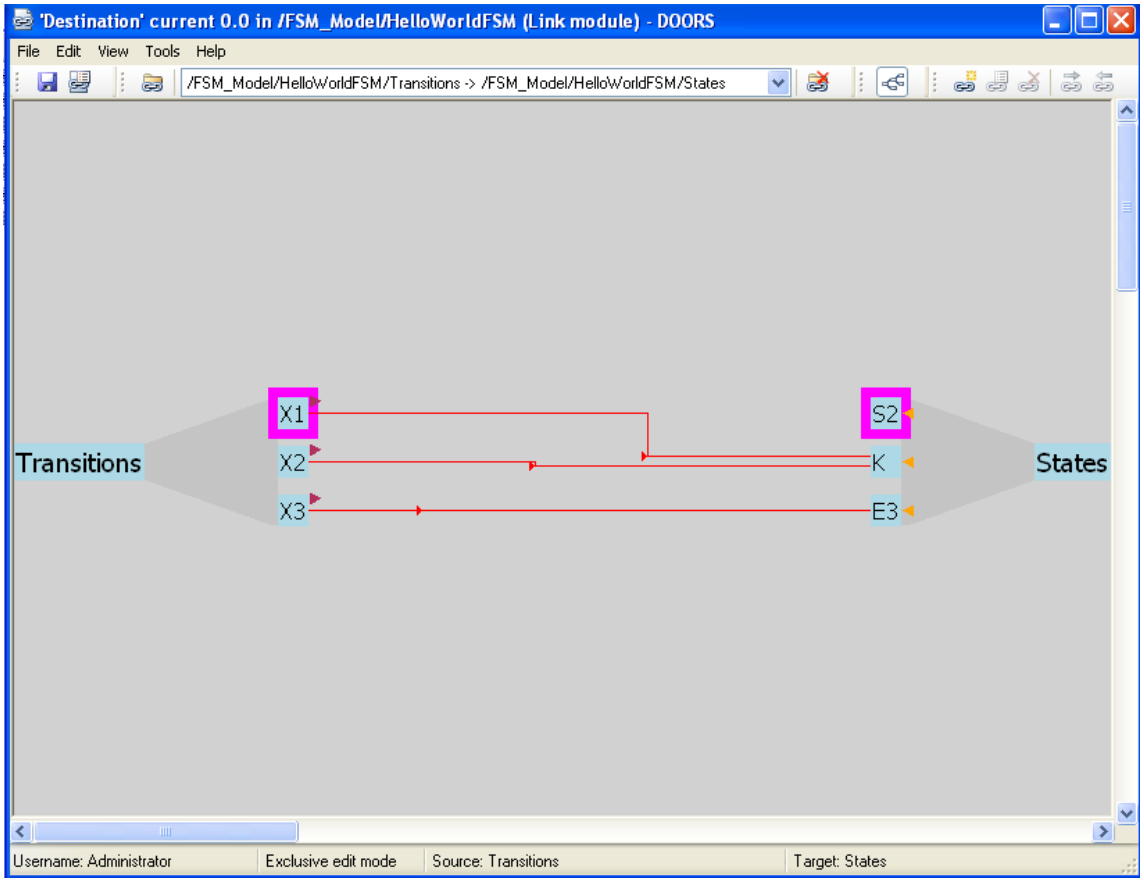

Figure 57 Destination link module displaying links between transitions and states in DOORS' graphics mode

Using this approach, we have validated, by inspection, that all DOORS modules, objects attributes and links were properly populated by the import based on the DXL library.

5.1.7 Re-importing Changes

As many of the DXL functions generated are meant to support the evolution of models and of associated requirements artefacts in DOORS [27], we have modified the initial DXL test script used to import the first version of the FSM model to verify the changes in DOORS when re-importing models.

Table 4 Changes to initial FSM model to validate re-importing in DOORS

Model (#ID)	**Attribute Changed**	**Previous value**	**Modified value**
State (ID #1)	Name_	S2	S2_modified
State (ID #1)	Description_	S2	S2_modified
State (ID #1)	Precondition	x=3	x=4
Transition (ID #4)	Description_		'Updated Description'

We have modified the test script with the changes described in Table 4. When re-importing (Figure 58), the import process provides feedback asking confirmation for up-dating *existing* modules.

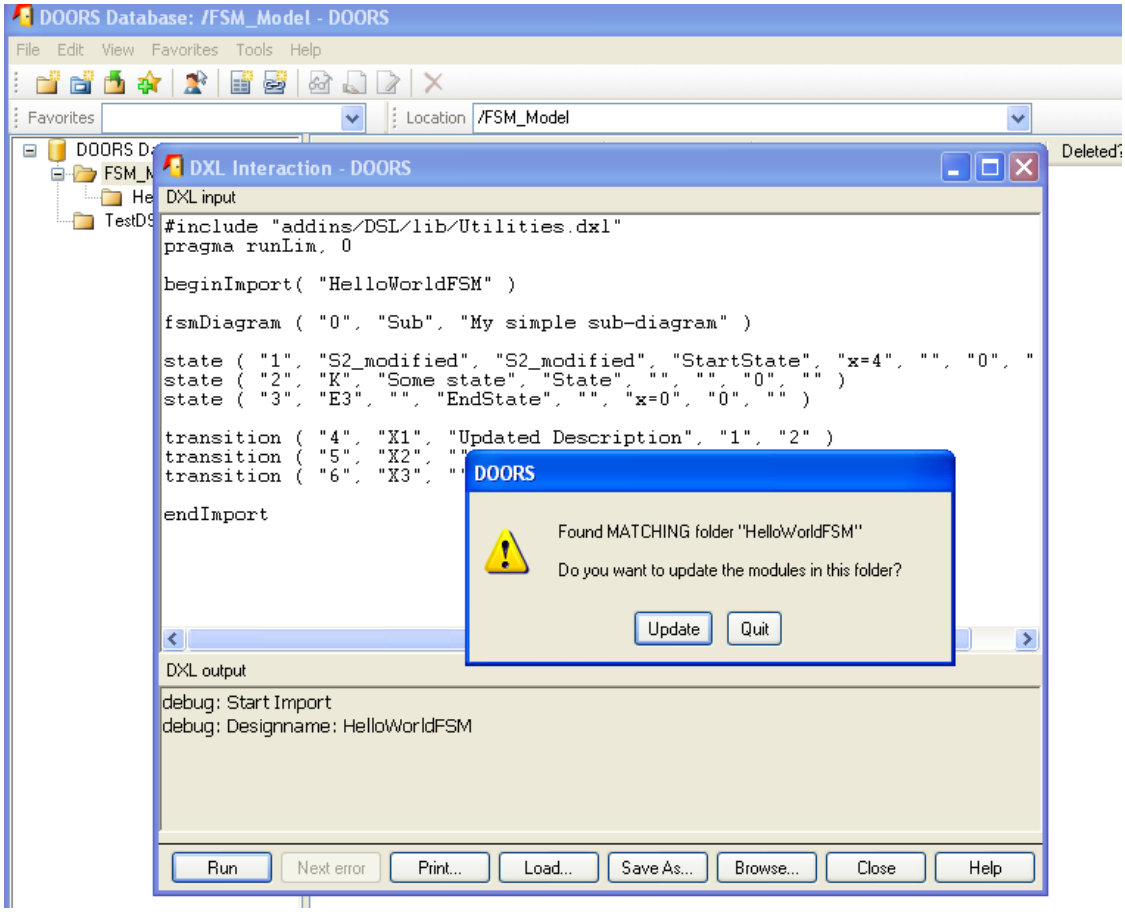

Figure 58 DXL interaction window showing the re-imported FSM model

Inspection of the revised States formal module (Figure 59) shows that the attribute values have been properly modified. Modifications to the Transitions model are also correct.

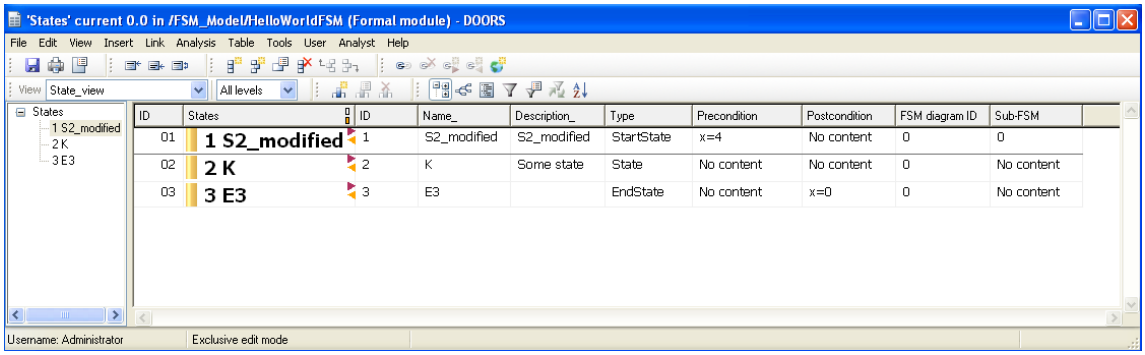

Figure 59 States formal module in DOORS after re-import the revised model

The changes in models imported in DOORS can also be traced using the History tab of the properties window for an object. For example, Figure 60 shows the history of the

S2_modified object in the States formal module. The name, heading, description, and precondition attributes were updated, but not the other attributes (e.g., type). The history view tracks who did these modifications and when. Modifications can also be seen using redlining, à la Microsoft Word. Traceability and tracking over time are essential in a requirements management context, and this feature of the DXL libraries generated from MT-DSL descriptions enable fine-grained support for models in that context.

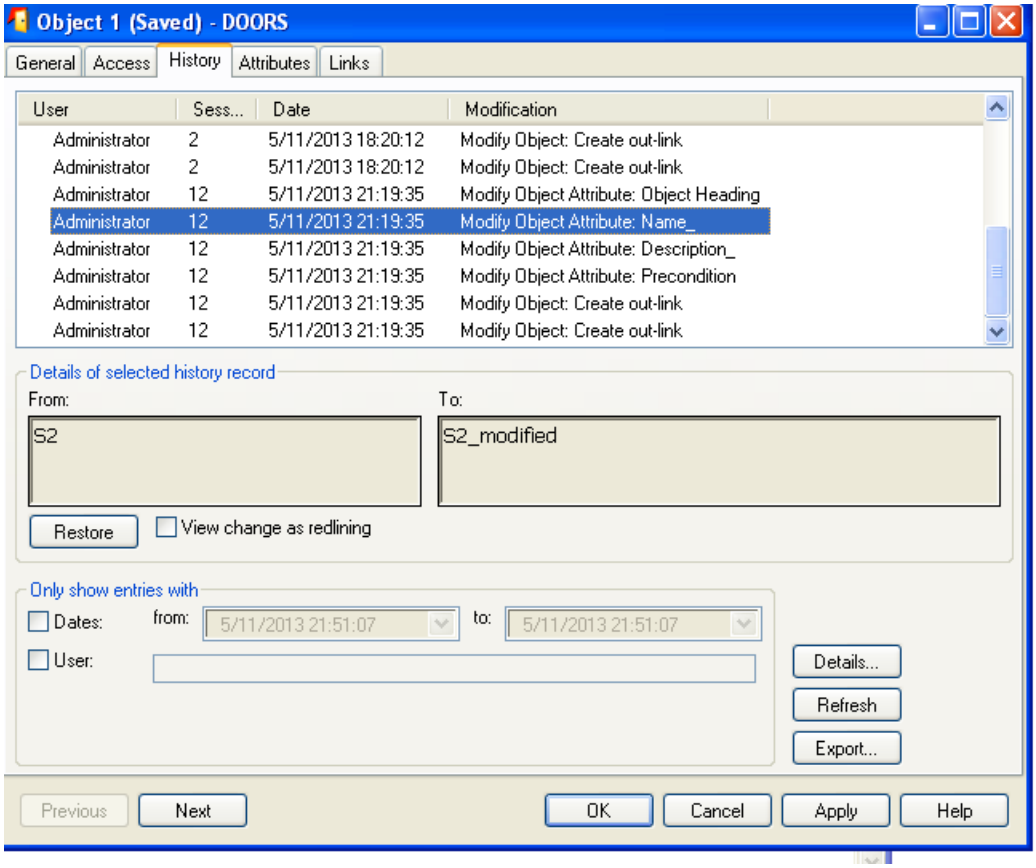

Figure 60 History tab on Properties window for a module in DOORS

5.2 URN Models

The subset of User Requirements Notation (URN) described in Appendix B is reused here for validation. This view on URN corresponds to what was supported in Roy's thesis, without Jian's scenario definitions and without Ghanavati's extensions for compliance (see Section 2.6.3). In total, 7 UCM concepts, 10 GRL concepts, and 8 types of associations are being tracked in this MT-DSL view.

The generated DXL library for the subset of URN model has been described in details in Chapter 4, and the complete list of generated DXL files and descriptions are available in Table 2.

5.2.1 Test Scenarios

We have used following DXL script to invoke the generated library and to import a sample URN model in DOORS. This is a non-trivial model composed of UCM/GRL elements and diagrams.

```
#include "addins/DSL/lib/Utilities.dxl"
pragma runLim, 0

beginImport( "HelloWorId" )

actor( "12", "Vendor", "" )
actor( "14", "Customer", "" )
actor( "18", "System", "" )

component( "58", "System", "", "Team", "" )
component( "60", "Vendor", "", "Actor", "" )
component( "64", "Database", "", "Object", "" )
component( "68", "Customer", "", "Actor", "" )

responsibility( "107", "CheckAvailability", "", "0" )
responsibility( "121", "SetupProduct", "", "0" )

intentionalElement( "22", "Provide Some Functionality", "", "Goal",
                                                              "Or" )
intentionalElement( "24", "First Option", "", "Task", "And" )
intentionalElement( "26", "Second Option", "", "Task", "And" )
intentionalElement( "33", "Make Profits", "", "Softgoal", "And" )
intentionalElement( "35", "High Quality of Product", "", "Softgoal",
                                                            "And" )
intentionalElement( "39", "Revenues", "", "Indicator", "And" )
intentionalElement( "47", "High Degree of Customer Satisfaction",
                                          "", "Softgoal", "And" )

elementlink("30","Decomposition30","", "decomposition", "24", "22" )
elementlink("31","Decomposition31", "", "decomposition", "26","22" )



contribution("43","Contribution43", "", "contribution", "Make", "0",
                                                          "39", "33" )
contribution("45", "Contribution45", "","contribution",
```

```
                                    "SomePositive", "0",  "26", "35" )
contribution("51", "Contribution51","", "contribution",
                                    "SomeNegative", "0",  "24", "33" )
contribution("53", "Contribution53", "","contribution", "Make", "0",
                                                          "35", "47" )

grldiagram( "2", "Goals", "", "HelloWorId-GRLGraph2-Goals.bmp",
                                                              "Goals" )

actorRef( "13","Vendor",  19,20,348,189,"12", "" )
actorRef( "15", "Customer",399,17,277,193,"14",  "" )
actorRef( "19", "System",97,233,436,177,"18",  "" )

intentionalElementRef( "23",  "Provide Some Functionality",
                               "",246,255,"19", "22", "None", "None" )
intentionalElementRef( "25", "First Option", "",159,331,"19", "24",
                                                          "None", "None" )
intentionalElementRef( "27", "Second Option", "",358,325,"19", "26",
                                                          "None", "None" )
intentionalElementRef( "34", "Make Profits", "", 76,48,"13", "33",
                                                          "None", "None" )
intentionalElementRef( "36", "High Quality of Product", "",
                                  473,75,"15", "35",  "None", "None" )
intentionalElementRef( "40", "Revenues", "",74,132,"13", "39",
                                                          "None", "None" )
intentionalElementRef( "48", "High Degree of Customer Satisfaction",
                               "",199,67,"13", "47",  "None", "None" )

map( "3", "Scenarios", "", "HelloWorId-Map3-Scenarios.bmp",
                                                            "Scenarios" )
respRef( "108", "CheckAvailability", "", 236,149,"65", "107" )
respRef( "122", "SetupProduct", "", 412,237,"61", "121" )
compRef( "59", "System",165,82,164,216,"no", "58",  "", "" )
compRef( "61", "Vendor",358,85,136,214,"no", "60",  "", "" )
compRef( "65", "Database",180,115,135,74,"no", "64",  "", "System" )
compRef( "69","Customer", 11,83,124,217,"no", "68",  "", "" )

strategy( "5", "1-FirstOption", "", "damyot" )
strategy( "134", "2-SecondOption", "", "damyot" )

endImport
```

The script describes that the URN models is to be imported in DOORS in a folder called ‘HelloWorld’. The DXL script describes many URN components including 3 actors, 4 components, 2 responsibilities, 7 intentional elements, 1 GRL diagram and 1 UCM map

object. This script refers to two images (exported by jUCMNav) that need to be imported in the RMS (Table 5).

Table 5 Images included in DXL test script for URN

URN Object	ID	Name	Image file name
grldiagram	2	Goals	HelloWorId-GRLGraph2-Goals.bmp
map	3	Scenarios	HelloWorId-Map3-Scenarios.bmp

5.2.2 Results

The DOORS formal and link modules resulting from the imported URN/jUCMNav model are shown in Figure 61.

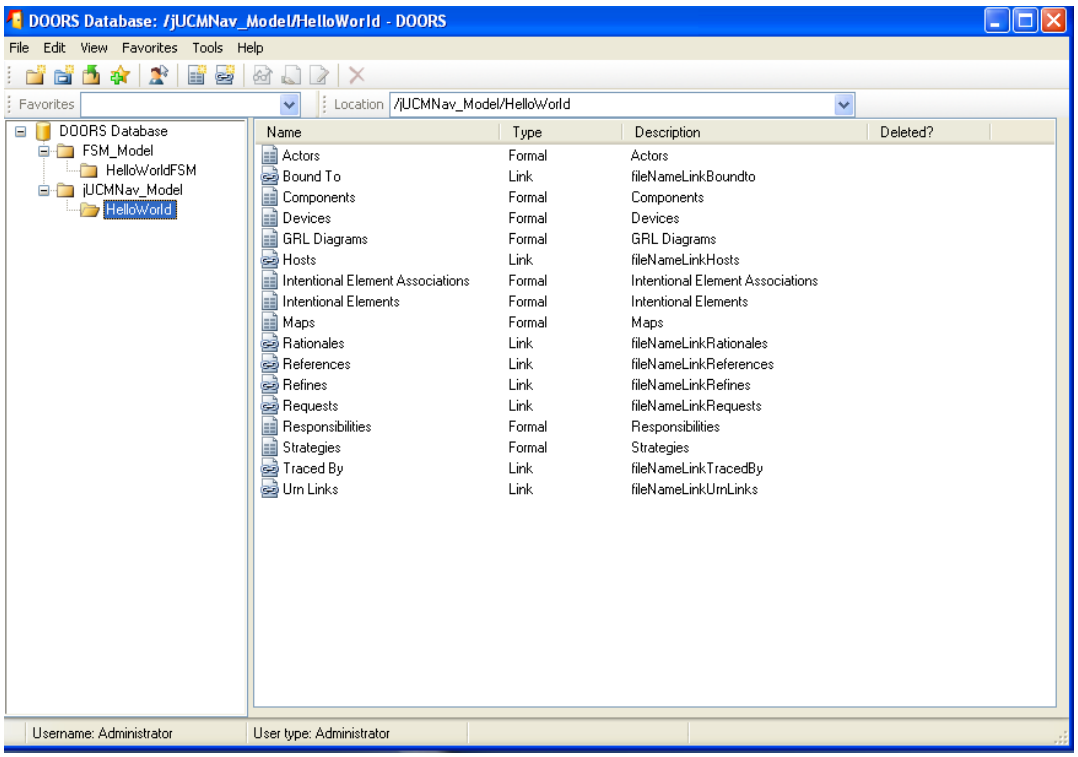

Figure 61 URN model imported in DOORS

These formal and link modules were inspected manually, and no missing element or link was detected. For the sake of illustration however, two formal modules are shown here

(as they involve graphics). Figure 62 shows the imported (UCM) Map module, which displays the content of the image file ‘HelloWorId-Map3-Scenarios.bmp’ described in Table 5.

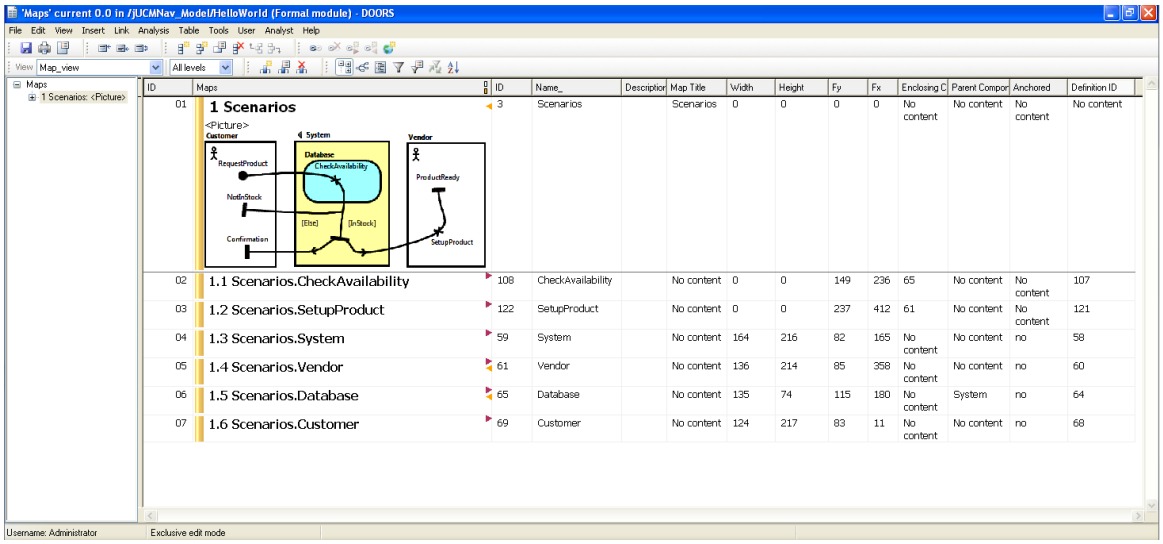

Figure 62 Map formal module (with image) in DOORS

Similarly, Figure 63 shows the imported GRL diagram formal modules in DOORS. The module contains the image file ‘HelloWorId-GRLGraph2-Goals.bmp’ described in Table 5.

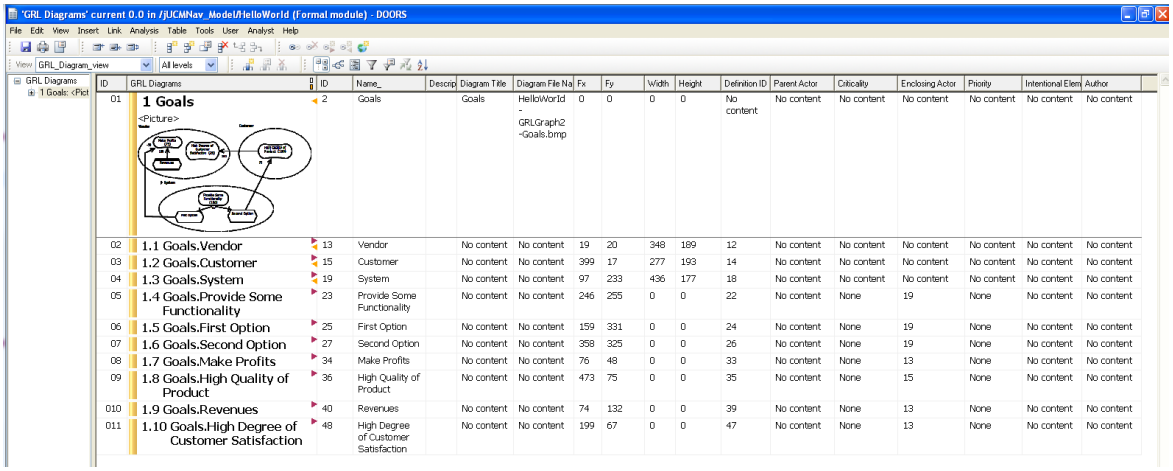

Figure 63 GRL Diagram formal module (with image) in DOORS

5.2.3 Re- importing Changes

We have modified the initial test script used to import the URN model to verify the handling of changes in DOORS when re-importing models. In addition to object attributes being modified as in the FSM example, this time entire objects are also added/deleted, images are modified, and links are also changed.

Re-importing Content Changes in Image Files

In this section, we will re-import the models with changes in the content of an imported image file in DOORS. Table 6 shows changes in the URN model to validate this test scenario. As shown on Table 6, there are no changes in the name of the image files. For the purpose of this test, we have manually edited the two image files to alter the content showing the text 'Changed' in the image (in red color).

Table 6 Changes to URN model to validate re-importing changes in DOORS for image file content changes

Model (#ID)	Attribute Changed	Previous value	New value
grldiagram (ID #2)	Name_	Goals	Goals Modified
grldiagram (ID #2)	Diagram File Name	'HelloWorId-GRLGraph2-Goals.bmp'	'HelloWorId-GRLGraph2-Goals.bmp'
map (ID #3)	Name_	Scenarios	Scenarios Modified
map (ID #3)	Diagram File Name	'HelloWorId-Map3-Scenarios.bmp'	'HelloWorId-Map3-Scenarios.bmp'

Figure 64 shows the modified image file content for the GRL Diagram after the re-import, which demonstrates that images can be updated, just like any other DOORS object. The same result was observed for the other (UCM) image. Both histories were also updated properly.

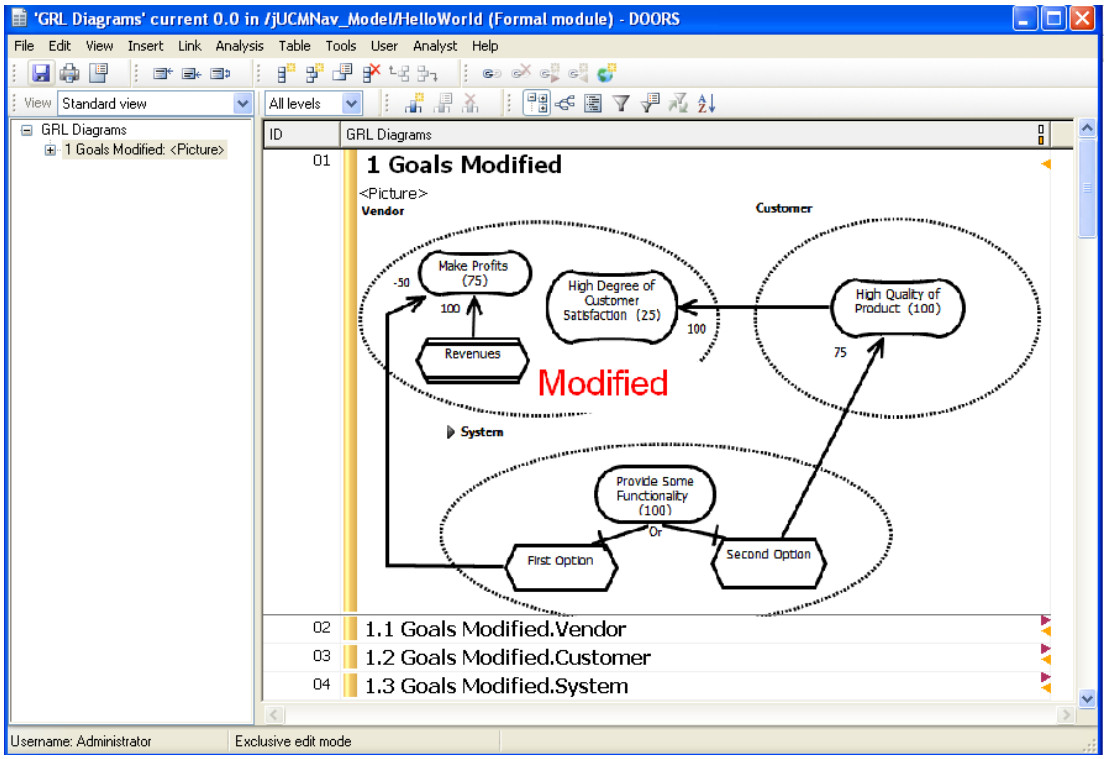

Figure 64 Modified image file content for GRL Diagram after the re-import

Re-importing a Model with Deleted Links

In this test, we re-import the model after *deleting* a link from the DXL script. Table 7 shows the corresponding change in the URN model used to validate this test scenario.

Table 7 Changes to URN model to validate re-importing changes in DOORS with a link removal

Model (#ID)	Attribute Changed	Previous value	New value
actorRef (ID #15)	Definition ID	14	‘’

We have modified the DXL script with the changes described in Table 7, as shown in Table 8.

Table 8 Changes to DXL scripts to validate re-importing changes in DOORS with deletion of a link

DXL before change	DXL after change
actor("14", "Customer", "") actorRef("15", "Customer", 399, 17, 277,193,**"14"**, "")	actor("14", "Customer", "") actorRef("15", "Customer", 399, 17, 277,193,**""**, "")

Figure 65 shows the 'References' links between GRL Diagrams and Actors created with the initial URN model imported using our first test script, and *before* any changes made to the links.

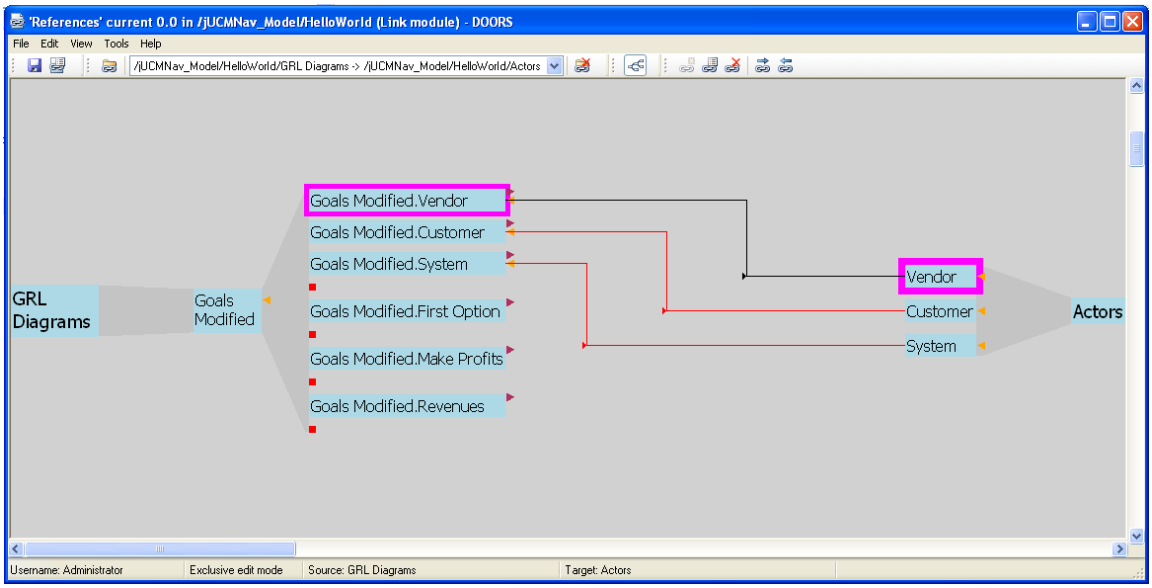

Figure 65 'References' links between GRL Diagrams and Actors created with the initial URN model import

Figure 66 shows the 'References' link module *after* the link with the initial actor (with id #14, i.e., the Customer) is deleted in the re-imported file. Again, the history was inspected manually and no problem was detected.

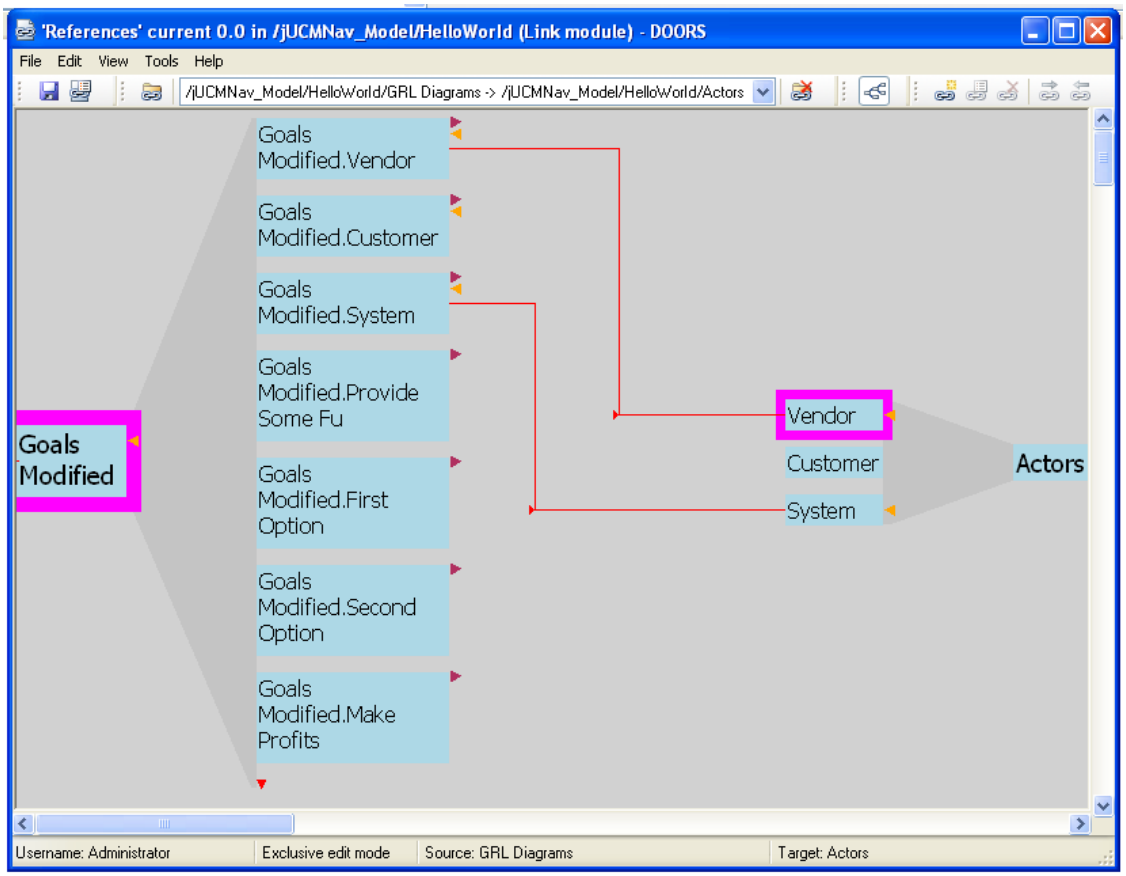

Figure 66 'References' links between GRL Diagrams and Actors after re-importing a model with a deleted link

Re-importing Model with New Links Created

In this test, we re-import the model after adding new model element (a New Customer actor with ID 114) and a new link from a previous actor reference. We have modified the test script with the changes described shown in 0. Changes to DXL scripts to validate re-importing changes in DOORS with a new link

DXL before change	DXL after change
actor("14", "Customer", "") actorRef("15", "Customer", 399, 17, 277,193,"", "")	actor("14", "Customer", "") **actor("114", "New Customer", "")** actorRef("15", "Customer", 399, 17, 277,193,**"114"**, "")

Figure 67 shows the ‘References’ links between GRL Diagrams and Actors in graphic mode after the changes are imported. A new ‘References’ link was created between the appropriate source actor reference (Customer) in the Goals Modified diagram and the actor definition (New Customer).

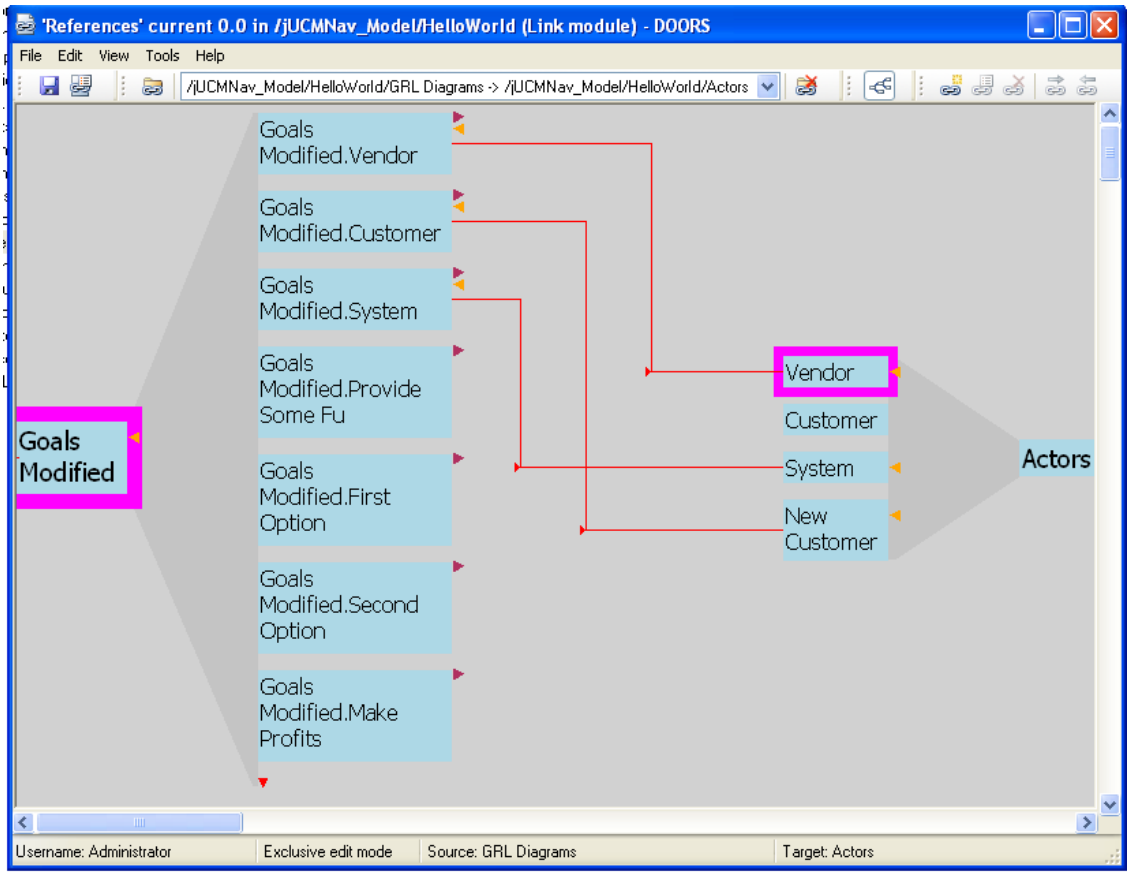

Figure 67 ‘References’ links between GRL Diagram and Actors after re-importing a model with a new object and a new link, in graphic mode

5.3 External Links

In this section, the tool-supported approach based on MT-DSL is validated by *evolving* model elements linked from and to external requirements. In order to check both traceability directions, two models (one FSM and one URN) are linked to each other, and then evolved. It is important here to validate the impact of modifications to the source/target element of a link (leading to *suspect* links in DOORS).

5.3.1 MT-DSL and DXL library

The languages chosen for the validation of external links (FSM and URN) have already been used for testing this thesis in Sections 5.1 and 5.2. The MT-DSL scripts for both languages have already been described, and we will be reusing their existing DXL libraries for testing external links. This also demonstrates that our tools support the concurrent usage of multiple modeling languages in an RMS.

5.3.2 Creating External Links

We have used a DXL script to invoke the generated library and to import the FSM model in DOORS (Figure 68).

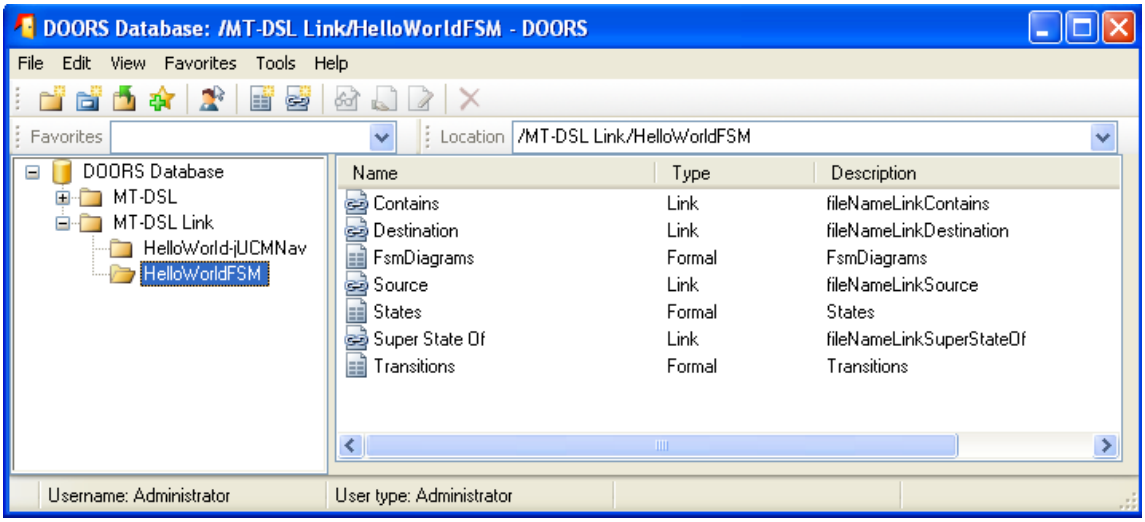

Figure 68 FSM models imported in DOORS to test External Links

Figure 69 shows the 'States' model of the imported FSM model that will be used in the testing of external links.

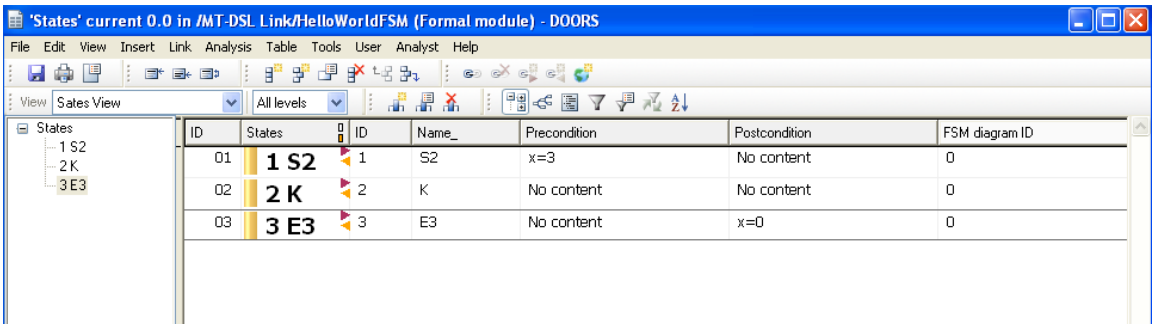

Figure 69 'States' module of imported FSM model used for testing external links

We have used another script to import a URN/jUCMNav model (composed of a goal model only). Figure 70 describes the actors of the imported URN model in DOORS.

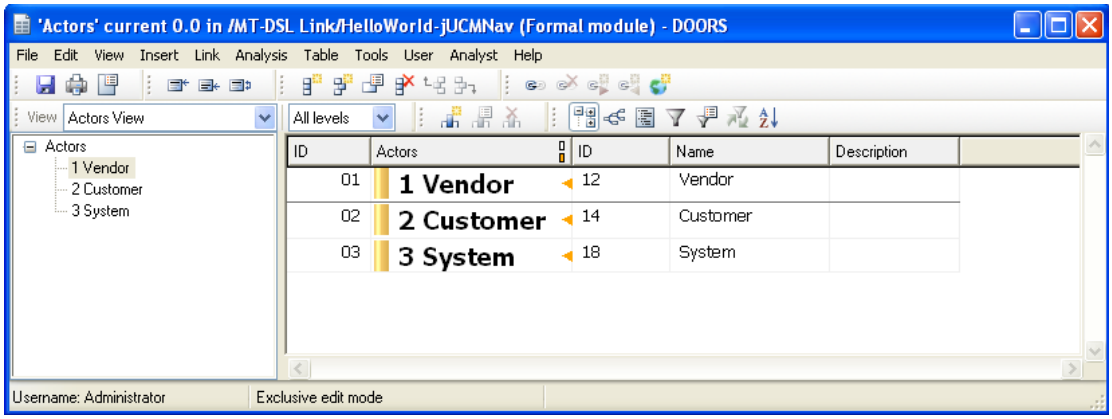

Figure 70 'Actors' module of imported URN model used for testing external links

We have created a new link module (SourceToActor, in the FSM folder) for external links from 'States' in FSM models to 'Actors' in URN models. The nature of this type of link is irrelevant and is only used to illustrate and validate the correctness of the import libraries in an evolution context. Figure 71 shows creating a new link module to represent this external link.

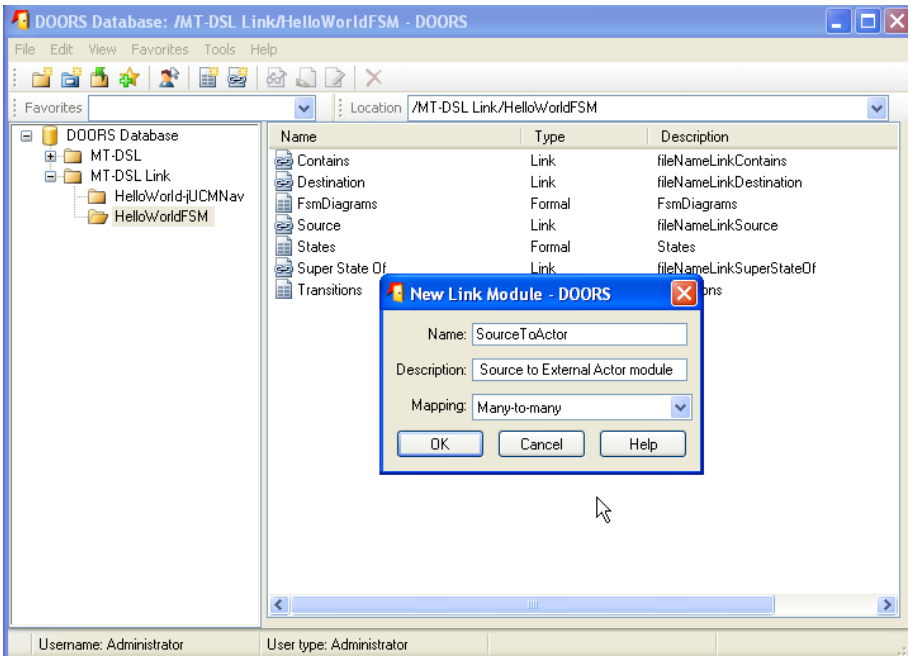

Figure 71 Creating a link module in DOORS

We have created some external links from States to Actors in DOORS for testing external links (Table 9).

Table 9 External links of type SourceToActor

Link Source			**Link Destination**		
Module	Object ID	Object Name	Module	Object ID	Object Name
States	1	S2	Actors	12	Vendor
States	2	K	Actors	12	Vendor
States	2	K	Actors	14	Customer
States	3	E3	Actors	14	Customer
States	3	E3	Actors	18	System

Figure 72 shows the external links described in the above table.

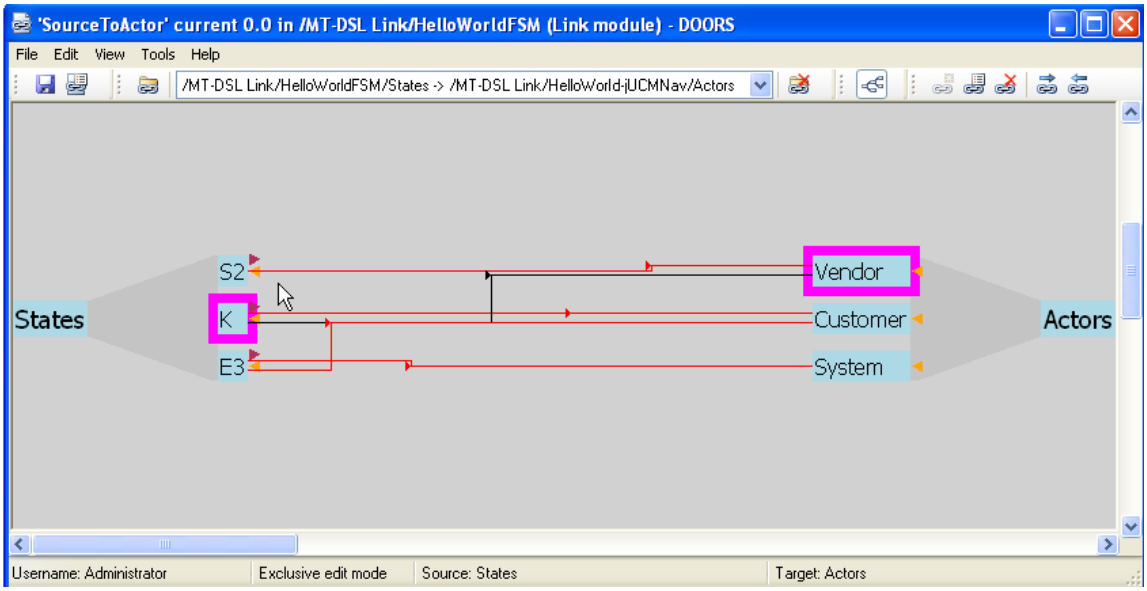

Figure 72 External links from FSM States to URN Actors in DOORS

5.3.3 Test Scenarios

We have updated the test scripts as described in Table 10 to modify the models connected by external links. We have updated one FSM model element and one URN model element.

Table 10 Update in DXL script to test external links

DXL before change	DXL after change
state ("2", "**K**", "Some state", "State", "", "", "0", "")	state ("2", "**K_new**", "Some state", "State", "", "", "0", "")
actor("18", "**System**", "")	actor("18", "**System_new**", "")

The object name for State with id #2 is modified to “K_new” while the object name for actor with id#18 is modified to “System_new”. We will explore the effect on the external links after the changes are imported in DOORS.

5.3.4 Test Result

We have imported the update in FSM and URN models as described in the test case. Figure 73 shows updated ‘Actors’ module after re-importing the model in DOORS.

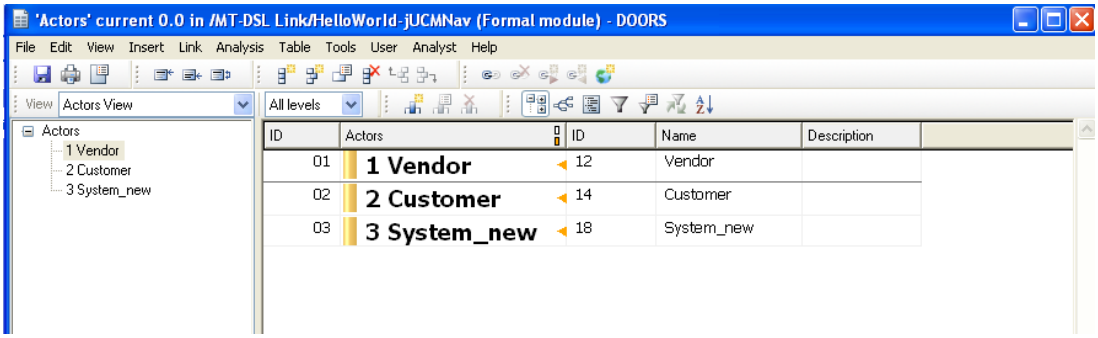

Figure 73 Updated ‘Actors’ module after changes in the URN model

Figure 74 shows the updated ‘States’ module after importing the updates in DOORS.

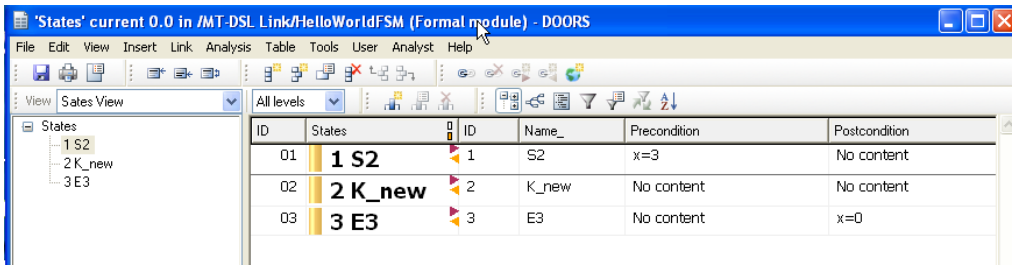

Figure 74 Updated ‘States’ module after changes in the FSM model

Figure 75 shows the external links after the update of both models.

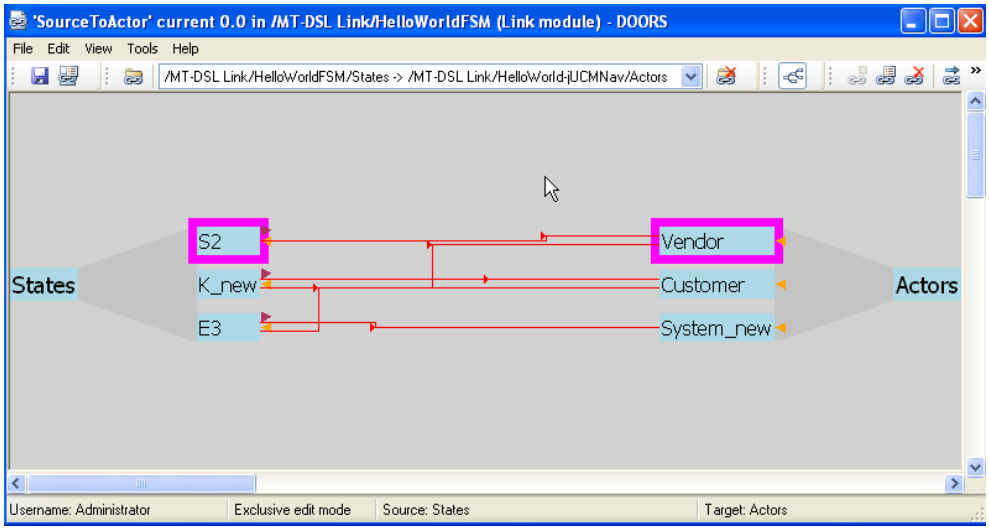

Figure 75 External links from States to Actors in DOORS after model updates

Table 11 shows all the links after the model updates. All original links are still present, i.e., modifying the source and target objects of links did not delete the links.

Table 11 External Links SourceToActor after model update

Link Source			**Link Destination**		
Module	Object ID	Object Name	Module	Object ID	Object Name
State	1	S2	Actor	12	Vendor
State	2	**K_new**	Actor	12	Vendor
State	2	**K_new**	Actor	14	Customer
State	3	E3	Actor	14	Customer
State	3	E3	Actor	18	**System_new**

DOORS indicates linked objects with associated changes in formal models with *suspect links*. Figure 76 shows the outgoing suspect links in the 'States' module using the '**?**' mark beside the links.

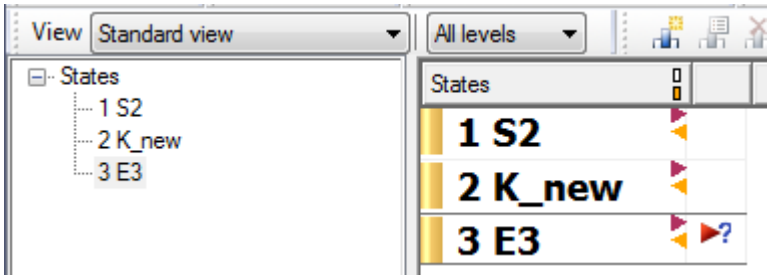

Figure 76 Suspect Links for 'States' model

Similary, Figure 77 shows the incoming suspect links for the 'Actors' module of the URN model. These are indeed the links that involve modified objects.

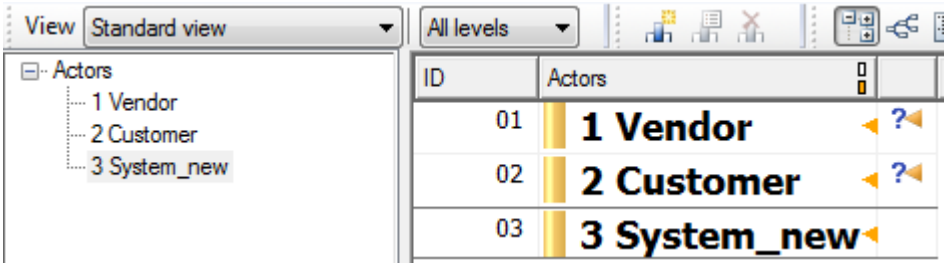

Figure 77 Suspect Links for 'Actor' Model

The details on the model changes for suspect links can be analyzed using the DOORS menu item *Analysis → Suspect Links → Display all changes → Out-links (all modules)*, as shown in Figure 78.

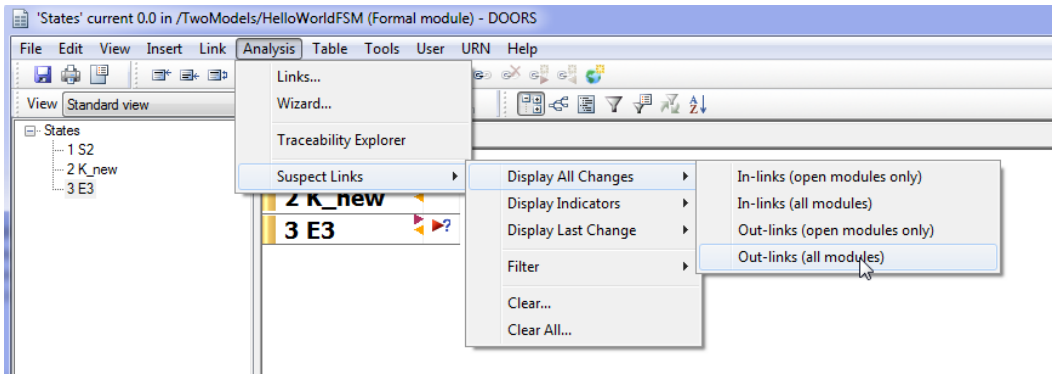

Figure 78 Menu item to display changes for suspect links in DOORS

Figure 79 shows the details of the model changes for suspect links, which can help users determine whether they should investigate changes further, or simply clear suspect links.

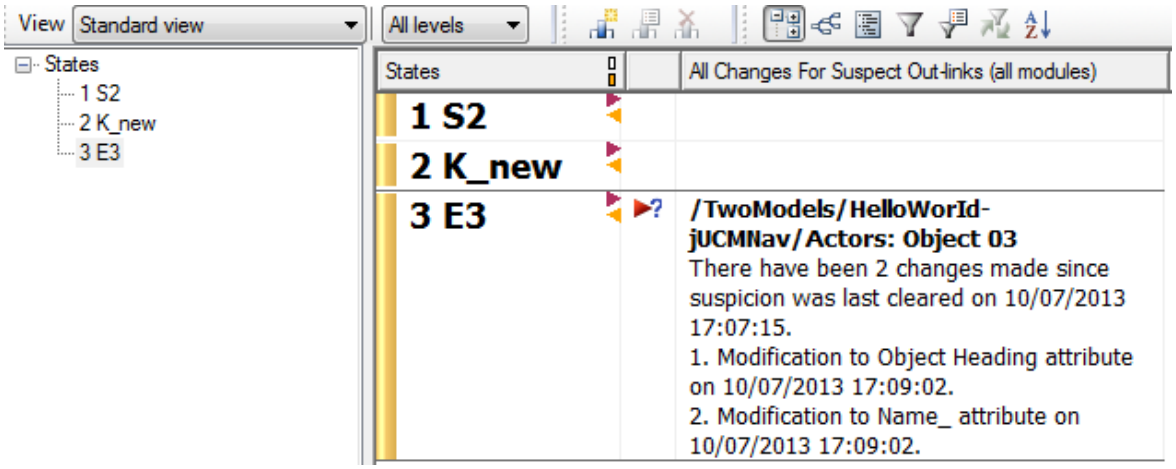

Figure 79 Details on changes in DOORS for suspect links

5.3.5 Deleting Linked Models

In this section, we explore changes in link modules when objects in the linked formal modules are deleted. We will first test our tool by deleting the *source* object for an external links, and then will be trying to delete the *target* object of such external links. Table 12 shows the change in our DXL script for testing this type of evolution by deleting the source element (state S2) of an external link.

Table 12 Changes in DXL script to delete source model with External Links

DXL before change	DXL after change
state ("1", "S2", "S2", "StartState", "x=3", "", "0", "0")	***/* removed! */***
state ("2", "K_new", "Some state", "State", "", "", "0", "")	state ("2", "K_new", "Some state", "State", "", "", "0", "")
state ("3", "E3", "", "EndState", "", "x=0", "0", "")	state ("3", "E3", "", "EndState", "", "x=0", "0", "")

DOORS provides feedback to the user while deleting objects with links and asks for user confirmation prior to deleting such objects, as shown in Figure 80.

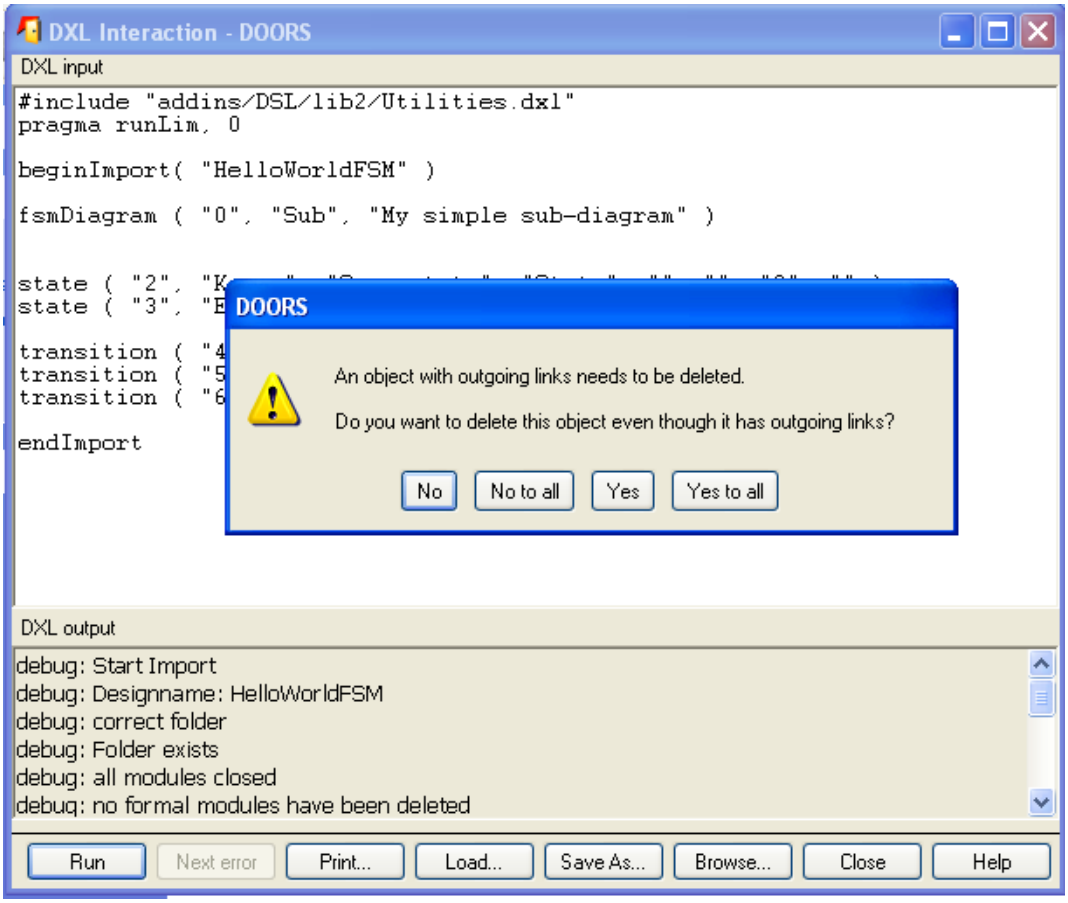

Figure 80 User confirmation to delete models with External links

Figure 81 shows the ‘State’ module after deleting one of the states (E2) with external links. E2 is indeed no longer in the module.

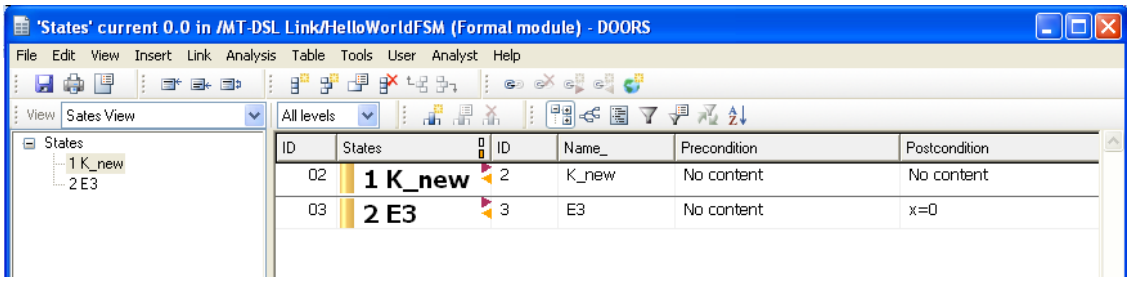

Figure 81 ‘States’ module after deleting one of the states with external links

Figure 82 shows the links after deleting the source object of an external link; the associated link was inded deleted from DOORS.

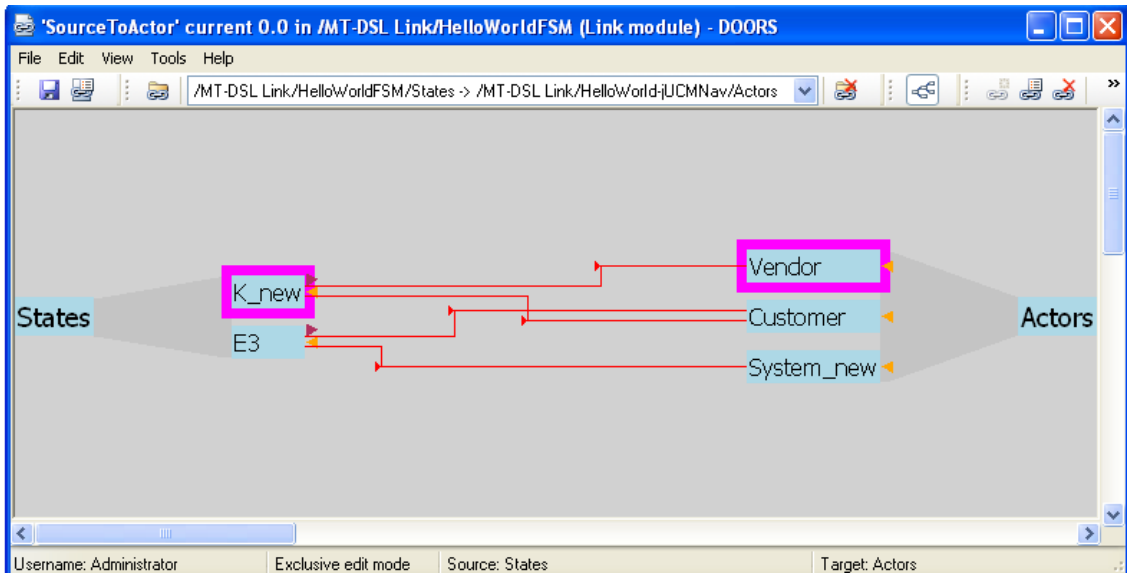

Figure 82 External links from ‘States’ to ‘Actors’ in DOORS, after deletion

We must also explore the deletion of the target object of an external link (Table 13). The actor object with id #14 is deleted. This object is in a folder (URN) that is different from that of the the link module (FSM). In DOORS, only the owner of a link can delete it, and this is not the case here.

Table 13 Update in DXL script to delete a target model of an External Links

DXL before change	DXL after change
actor("12", "Vendor", "")	actor("12", "Vendor", "")
actor("14", "Customer", "")	***/* deleted! */***
actor("18", "System_new", "")	actor("18", "System_new", "")

DOORS indeed displays an error message while trying to execute the script, as shown in Figure 83. This is because the links are owned by a different module.

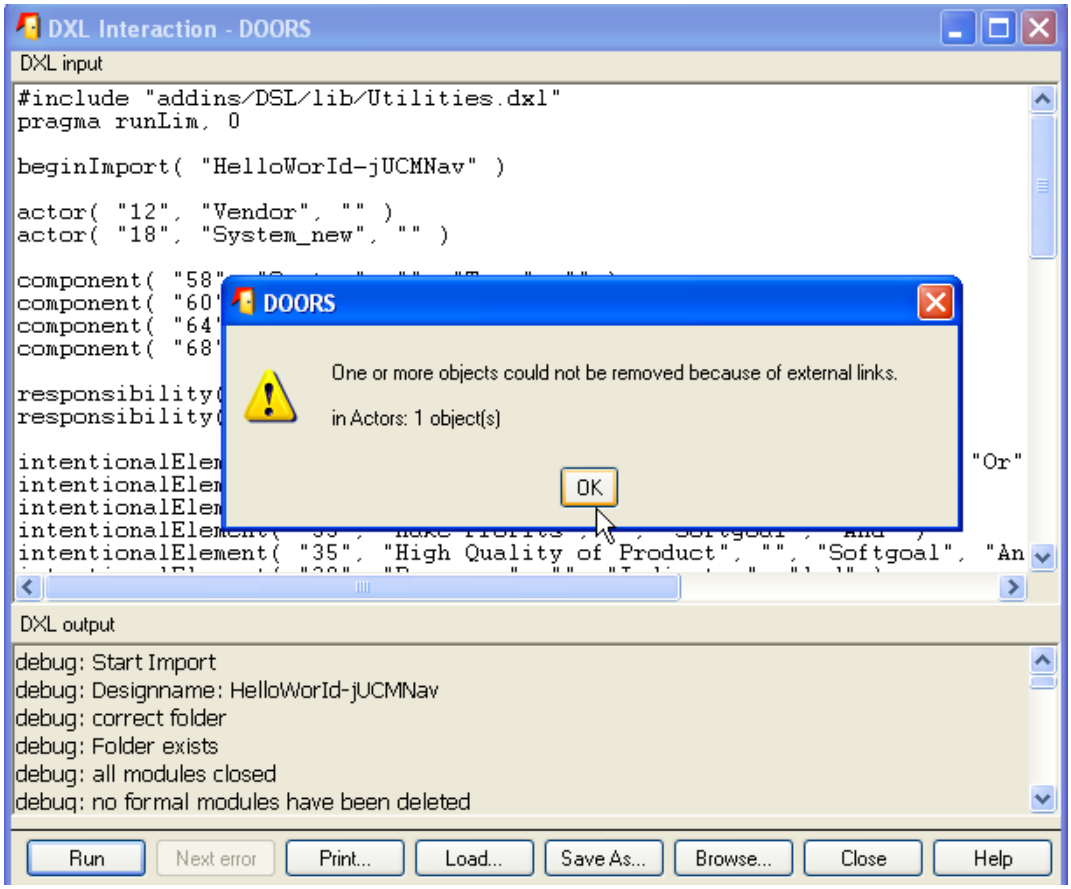

Figure 83 External links from ‘States’ to ‘Actors’ in DOORS, after deletion attempt

These successful test cases illustrate that behavior of the library in terms of handling external links is as expected.

5.4 Threats to Validity and Limitations

In this section, we discuss the main threats to the validity and limitations based on the roadmap presented by Perry et al. [40], where they define the following three types of validity:

- **Construct Validity:** Examine to what extent the case studies actually measure the answers to the questions. This is to validate that the independent and dependant variables accurately model the abstract hypotheses.
- **Internal Validity:** Estimate to what extent the cause and effect be made based on measures used, the research setting and design. This is to validate that that changes in the dependent variables can be safely attributed to the changes in the independent variables.
- **External Validity:** Verify the extent to which the results of the studies can be generalized.

Construct Validity

One threat is that we could have tested this approach on only one simple modeling language. This was mitigated by having two languages tested (FSM and UCM), with multiple models for each (especially for URN).

The DXL scripts could have deviated from the initial models. This could be the case for the FSM example (although it is so short that mistakes are unlikely). However, to mitigate this with the longer script for the URN model, this script was generated with jUCMNav's existing export and hence was not produced manually. This also shows that we were able to automatically replicate the behaviour of the manually generated DXL library in jUCMNav.

The tests could also have not covered important functionalities of the DXL library generated. To mitigate this, we have systematically covered additions, deletions, and modifications of objects and links in the many test cases done (including on objects linked to external objects), and did additional ad hoc testing outside of what was reported here. History properties were inspected manually. We have also validated the modification of an image file in the model.

We have not demonstrated nor tested that DXL scripts can actually be generated from modeling tools in general. However, we believe this is easily feasible in many cases based on the prior experience of three students (Jiang, Roy, and Ghanavati) who created such export mechanisms for two different tools (UCMNav and jUCMNav). The complexity resides in the construction of the DXL library (automated here) rather than on the generation of fairly declarative scripts in DXL.

Internal Validity

In order to avoid bias, the languages (one is an international standard, and one came from a graduate course assignment) and the initial models also existed before the experiment. More experiments would however improve our level of confidence in the results.

External Validity

We have validated that the thesis can be generalized for two input languages, one of which (URN) being non-trivial. We believe that the common language construction concepts used in MT-DSL (classes, attributes, associations, data types and modules) are generic enough (as they are used in meta-metamodeling frameworks such as MOF, Ecore and Z.111) to capture many interesting views of existing modeling languages, especially those that are defined with a metamodel (including MT-DSL!). However, we have no proof that we will not face languages one day that will require improvements to MT-DSL itself.

Although MT-DSL is meant to be independent of Requirements Management Systems, our automation currently only targets one RMS, namely, DOORS. Even if DOORS is very common, this is certainly a threat to the validity of this claim. Also, DOORS supports a very open API with a mature and expressive scripting language, which is not the case for all RMS. There could also exist features in other RMS that would lead to additions to MT-DSL so they can be exploited properly.

5.5 Summary

In this chapter, we have validated the thesis hypothesis, MT-DSL, and the DXL code generation. Two languages and several models (with different versions where changes covered typical addition, deletion, and modification changes when modeling) were used,

and the results demonstrated. We have also validated the support for the evolution of model elements linked from/to external objects. This also demonstrated that our tools can support multiple modeling languages concurrently in an RMS. No major problem was detected along the way.

The next chapter will conclude this thesis with its main contributions and future work items.

6 Conclusions

This chapter summarizes the main contributions of the thesis and potential future works items.

6.1 Contributions

This thesis provides the following contributions:

- The Model Traceability Domain-Specific Language (MT-DSL) for describing models in a way that enables import and traceability management in an RMS. We have implemented the DSL independently of the target Requirement Management System. The DSL is supported by a sophisticated Eclipse-based editor that provides content assistance, syntax highlighting, and error highlighting.
- The second main contribution is the tool-supported transformation of MT-DSL descriptions to a DXL library enabling models to be seamlessly imported in DOORS, a very popular RMS. Traceability is maintained in DOORS by re-importing models when changed.

These contributions allow describing the traceability model using a domain-specific language. As a result, the supported traceability model for any tool can be changed easily by simply updating the traceability description file. No manual work is required to re-write/update the library for the RMS. For example, jUCMNav currently supports a fixed subset of URN for export to DOORS. Using this thesis approach, support can be extended for new jUCMNav features (e.g., aspect-oriented UCM, indicators in GRL, etc.) without manually rewriting the DXL library for jUCMNav. Only the export of a model as DXL scripts (which is simple to do) needs updating. In fact, many libraries can be generated that provide different views on what must be tracked. As such, there may exist many TM-DSL scripts for one modeling language (for different people, or even for one person).

6.2 Comparison with Related Work

While other researchers are working on traceability for models, this thesis provides a different and unique approach that complements the approaches introduced in Section 2.1.

Mäder and Gotel [30] describe a semi-automated way of updating traceability relations between requirements and other software artifacts expressed with UML. The approach is currently limited that one language. This approach is meant to convert part of the manual effort necessary for traceability maintenance into a computational effort. Their approach is complementary to ours as we are focusing on a way to bring models (including but not limited to UML models) and their internal traceability links into an RMS, a prerequisite to their work. The model elements can then be linked to other types of requirements artefacts (including textual ones), with some traceability links being inferred automatically through their approach.

Similarly, the work of Mirakhorli et al. [32] on automating the traceability of quality concerns can be seen as complementary to our work. Models (including quality models expressed with goal languages such as GRL) can be first imported in the RMS and then quality-related traceability links can be established with their approach. The change management features of the generated DXL libraries will help manage the links between modified elements (from a re-imported model) and related quality artefacts (which may also be evolving).

The Visual Trace Modeling Language (VTML) of Mäder and Cleland-Huang [29] is actually a tool that can be used to *query* an RMS' database. Such database can be populated through our approach, which again shows complementarities.

For the purpose of the thesis we have introduced the new MT-DSL language. MT-DSL is complementary to other existing traceability languages such as TML and VTML. Existing languages do not really support the import of models in an RMS. MT-DSL allows describing the models to be imported in the DOORS RMS, as well as different configuration options on what views of a model to track in the RMS. On the other hand, MT-DSL does not formalize links between a model and external requirements, which is what TML aims to achieve.

6.3 Future Work

The work in this thesis can be further extended in various directions.

- We have used DOORS, a very popular RMS, for maintaining traceability. This was a good choice given the expressiveness of this DXL scripting language. However, our implementation does not currently support generating libraries for any other RMS other than DOORS. Replicating this work with another RMS (e.g., Requisite Pro) still needs to be explored.
- In terms of replicating the existing DXL library for jUCMNav, a future work item would involve support for the modeling of laws and regulations, compliance with institutional policies, and government legislation, as implemented by Ghanavati [11][12][13][14]. Two interesting things to explore in this context is the generation of pre-defined views for formal modules (i.e., which columns to show and in which order), and the generation of additional "auto-tracing" functions to create new links automatically based on transitive traceability relations.
- We have validated the thesis approach against FSM and URN languages and models. Future work could further validate the approach through other modeling languages (e.g., UML, BPMN, SysML, etc.) and recent extensions to existing ones (e.g., Aspect-oriented User Requirements Notation).
- We can also improve the Eclipse plugin editor used in this implementation to allow the configuration items in Section 4.3 to be set using Eclipse's Properties page (rather than changing Xtend code manually as currently done).
- The current MT-DSL views on a language target a filtering on the types of elements (e.g., desired classes, attributes, and associations). For example, we can track the Actors in a URN model. A complementary kind of filtering could be done on *subsets* of the elements to be tracked (e.g., actors A, B, and C, but not D). This would likely involve tool-supported interactions with the users.
- Our work could further be extended by automating the creation of a MT-DSL script from the meta-model of a language, which could then be tailored (e.g., likely through reductions) to lead to a desired view on the language.

- In addition, once this is done, if the metamodel of the language evolves (e.g., with the addition of new concepts), then the difference between the metamodels could be computed and perhaps transformed into patches or increments to existing MT-DSL views of the original metamodel (leading to the automated, incremental evolution of the views).

References

[1] Amyot, D. (2003) Introduction to the user requirements notation: learning by example. *Computer Networks*, 42(3), pp. 285-301.

[2] Amyot, D., Horkoff, J., Gross, D., and Mussbacher, M. (2009) A lightweight GRL profile for i* modeling. *Advances in Conceptual Modeling - Challenging Perspectives*. ER 2009 Workshops. LNCS 5833, Springer, pp. 254-264.

[3] Amyot, D. and Mussbacher, G. (2011) User Requirements Notation: The First Ten Years, The Next Ten Years. Invited paper, *Journal of Software (JSW)*, Vol. 6, No. 5, Academy Publisher, May 2011, pp. 747-768.

[4] Cleland-Huang, J., Chang, C.K., Christensen, M.J. (2003) Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering* 29 (9), pp. 796-810.

[5] Cleland-Huang, J (2006) Requirements Traceability - When and How does it Deliver more than it Costs? *14th IEEE Int. Conf. on Requirements Engineering (RE'2006)*, IEEE CS, p. 323.

[6] Deursen, A. (1998) Domain-Specific Languages, *ACM Computing Classification System : D3.*

[7] Drivalos, N., Kolovos, D. S., Paige, R. F., and Fernandes, K. J. (2008) Engineering a DSL for Software Traceability. *1st Int. Conf. on Software Language Engineering (SLE'2008)*, LNCS 5452, Springer, pp. 151-167.

[8] *Eclipse Modeling Framework Project* (2013) http://www.eclipse.org/emf. Accessed January 2013.

[9] *Epsilon Validation Language* (2013) http://www.eclipse.org/epsilon/doc/evl. Accessed January 2013.

[10] Fowler, M. (2010) *Domain-Specific Languages.* Addison-Wesley Professional. ISBN: 0321712943 978-0321712943

[11] Ghanavati, S. (2007) *A Compliance Framework for Business Processes Based on URN*. M.Sc. in System Science, University of Ottawa, Canada, May 2007.

[12] Ghanavati, S., Amyot D., and Peyton, L. (2007) Towards a Framework for Tracking Legal Compliance in Healthcare. *19th Int. Conf. on Advanced Information Systems Engineering (CAiSE'07)*, Trondheim, Norway, June. LNCS 4495, Springer, pp. 218-232.

[13] Ghanavati, S., Amyot D., and Peyton, L. (2008) Comparative Analysis between Document-based and Model-based Compliance Management Approaches. *First*

Int. Workshop on Requirements Engineering and Law (RELAW 2008), Barcelona, Spain, September. IEEE CS, pp. 35-39.

[14] Ghanavati, S., Amyot, D., Peyton, L., and Mussbacher, G. (2007) A Compliance Framework for Business Processes Based on URN and DOORS. *2007 Telelogic User Group Conference*, Atlanta, USA, October.

[15] Gotel, O., Cleland-Huang, J., Hayes, J. H., Zisman, A., Egyed, A., Grünbacher, P., and Antoniol, G. (2012) The quest for ubiquity: A roadmap for software and systems traceability research. *20th IEEE Int. Requirements Engineering Conf. (RE'2012)*, Chicago, USA. IEEE CS, pp. 71-80.

[16] Gotel, O.C.Z. and Finkelstein, A.C.W. (1994) An analysis of the requirements traceability problem. *Proceedings of the First Int. Conf. on Requirements Engineering (ICRE 1994)*. IEEE CS, pp. 94-101.

[17] Graf, A., Sasidharan, N., and Gürsoy, Ö (2011) Requirements, Traceability and DSLs in Eclipse with the Requirements Interchange Format (ReqIF). *Proc. Second Int. Conf. on Complex Systems Design & Management (CSDM 2011)*, Paris, France. Springer, pp. 187-199.

[18] IBM: *Rational DOORS*. http://www.ibm.com/software/awdtools/doors. Accessed January 2013.

[19] IBM: *Rational DOORS API Manual*, Release 9.2 http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.doors.doc/pdf92/doors_api_manual.pdf. Accessed January 2013

[20] IBM*: Getting Started with Rational DOORS*, Release 9.2. http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.doors.doc/pdf92/doors_getting_started.pdf. Accessed January 2013

[21] IBM: *DXL Reference Manual*, Release 9.2. http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.doors.doc/pdf92/dxl_reference_manual.pdf. Accessed January 2013.

[22] IEEE (1990) Std 610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, New York.

[23] INCOSE (2013) Requirements Management Tools Survey. http://www.incose.org/productspubs/products/rmsurvey.aspx. Accessed May 2013.

[24] ITU-T (2008) *Recommendation Z.111 (11/08): Notations and guidelines for the definition of ITU-T languages* , Geneva, Switzerland, November.

[25] ITU-T (2012) *Recommendation Z.151 (10/12): User Requirements Notation (URN) - Language Definition*, Geneva, Switzerland, October.

[26] Java (2013) http://www.oracle.com/us/technologies/java/overview/index.html. Accessed April 2013.

[27] Jiang, B. (2005) *Combining Graphical Scenarios with a Requirements Management*. Master of Computer Science (M.C.S.), University of Ottawa, Canada, June.

[28] jUCMNav (2013) *Eclipse plug-in for the User Requirements Notation*, http://softwareengineering.ca/jucmnav/. Accessed January 2013.

[29] Mäder, P. and Cleland-Huang, J. (2010) A Visual Traceability Modeling Language. *Model Driven Engineering Languages and Systems (MoDELS)*. LNCS 6394, Springer, pp. 226-240.

[30] Mäder, P. and Gotel, O. (2012) Towards automated traceability maintenance. *Journal of Systems and Software*, 85(10), pp. 2205-2227.

[31] Mernik, M. (2005) When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4), December, pp. 316-344.

[32] Mirakhorli, M., Shin, Y., Cleland-Huang, J., and Çinar, M. (2012) A tactic-centric approach for automating traceability of quality concerns. *2012 Int. Conf. on Software Engineering (ICSE)*. IEEE CS, pp. 639-649.

[33] Mussbacher, G., Amyot, D. Araújo, J., and Moreira, A. (2010) Requirements Modeling with the Aspect-oriented User Requirements Notation (AoURN): A Case Study. *Transactions on Aspect-Oriented Software Development VII*, LNCS 6210, Springer, 2010, pp. 23-68.

[34] Mussbacher, G. and Amyot, D. (2009) Goal and Scenario Modeling, Analysis, and Transformation with jUCMNav. *31st Int. Conf. on Software Engineering (ICSE'09)*, Vancouver, Canada, May 2009. *ICSE Companion 2009*, IEEE CS, pp. 431-432.

[35] Mussbacher, G., Jiang B., Amyot, D., and Woodside, M. (2005) Importing and Updating of Scenario Models in DOORS. *Telelogic User Group Conference*, Hollywood, USA, October.

[36] OMG (2011) Meta Object Facility, Version 2.4.1, August 2011. http://www.omg.org/spec/MOF/2.4.1 . Accessed May 2013

[37] OMG (2010) *Object Constraint Language*, Version 2.2, February 2010. http://www.omg.org/spec/OCL/2.2/ . Accessed January 2013

[38] OMG (2011) *Unified Modeling Language*, Version 2.4.1, August 2011. http://www.omg.org/spec/UML/2.4.1/ . Accessed January 2013.

[39] openArchitectureware (2008) an earlier Eclipse project, http://www.openarchitectureware.org. Accessed January 2013.

[40] Perry, D.E., Porter, A. A., and Votta, L. G. (2000) Empirical Studies of Software Engineering: A Roadmap. *Proceedings of the Conference on The Future of Software Engineering*. ICSE'00, ACM, pp. 345-355.

[41] Petriu, D.B., Amyot, D., Woodside, M, and Jiang, B. (2005) Traceability and Evaluation in Scenario Analysis by Use Case Maps. *Scenarios: Models, Algorithms and Tools*. LNCS 3466, Springer, pp. 134-151.

[42] Roy, J.-F., Kealey, J and Amyot, D. (2006) Towards Integrated Tool Support for the User Requirements Notation. *SAM 2006: Language Profiles - Fifth Workshop*

on System Analysis and Modelling, Kaiserslautern, Germany, May. LNCS 4320, Springer, pp. 198-215.

[43] Roy, J.-F. (2007) *Requirement Engineering with URN: Integrating Goals and Scenarios*. Master of Computer Science (M.C.S.), University of Ottawa, Canada, March.

[44] Vaishnavi, V. and Kuechler, W. (2008) *Design Science Research Methods and Patterns: Innovating Information and Communication Technology*. Auerbach Publications, Boston, MA, USA. ISBN:1420059327 9781420059328.

[45] Xtend (2012) http://www.eclipses.org/xtend. Accessed January 2013

[46] Xtend User Guide (2012) http://www.eclipse.org/xtend/documentation/2.3.0/Documentation.pdf. Accessed January 2013

[47] Xtext (2012) http://www.eclipse.org/Xtext. Accessed January 2013

[48] Xtext 2.3 Documentation (2012) http://www.eclipse.org/Xtext/documentation/2.3.0/Documentation.pdf. Accessed January 2013

Appendix A: Xtext Definition of MT-DSL

This appendix describes the model traceability DSL definition using Xtext's syntax. This grammar captures the concrete syntax of the MT-DSL metamodel in Figure 22. The Xtext file used for this thesis is called DxlDsl.xtext and is included in package org.xtext.dsl.dxl. Note that the ID and STRING types are predefined in an imported module (common terminals).

```
grammar org.xtext.dsl.dxl.DxlDsl with org.eclipse.xtext.common.Terminals

generate dxlDsl "http://www.xtext.org/dsl/dxl/DxlDsl"


Model:
	'model' name=ID '{'
		(folders+=Folder)*
	'}'
;

Folder:
	'folder' name=ID '{'

		// formal modules
		(module+=Module)*

		// link modules
		(associationType+=AssociationType)*
	'}'
;

Module:
	'module' (ignoreInReport?='ignoreInReport')? name=ID '{'

	// Option to declare a file name: default convention used 'Maps' for
	// map, 'Devices' for 'device', etc.
		('fileName' fileName=STRING)?
		(classes+=Class)*
	'}'
;
```

```
Class:
       'class' (noDescription?='noDescription')? name=ID
           ('shows as' classTypeDescription=STRING)? '{'
              (attributes+=Attribute)*
              (associations+=Association)*
       '}'
;

Attribute:
       (ignored?='ignored')? type=DataType name=STRING
           ('shows as' default=STRING)?
;

DataType:
       'bool' | 'string' | 'int' | 'text' | 'diagram'
       // Note: not more than 1 diagram attribute per class
;

AssociationType:
       'associationType' name=ID linkFileName=STRING
;

Association:
       'association' name=ID':'assoType=[AssociationType] 'to'
       moduleType=STRING('.'classType=STRING)?
           (assoDescription=STRING)?
       // Note: the classType must be defined in the moduleType.
;
```

Appendix B: jUCMNav/URN Model using MT- DSL

This appendix describes the subset of the jUCMNav metamodel that is currently supported by the existing DXL library using our new Model Traceability DSL. This is specified in file `model.dxldsl` in the run-time project. As currently supported models in the existing tool are described here using a simple DSL file, any modification to the existing DXL library for jUCMNav can be performed with minimum effort by changing only this file.

```
model myModel{

    folder MyFolder{

        module responsibility{
            //default name, id, description
            class responsibility{
                string "processorDemand" shows as "Processor Demand"
            }
        }

        module ignoreInReport actor{
            class actor{}
        }

        module ignoreInReport intentionalElement{
            fileName "Intentional Elements"

            class intentionalElement{
                string "type"
                string "decompositionType" shows as "Decomposition Type"
            }
        }
```

```
module map{
    class map{
        diagram "graphFileName" shows as "Map File Name"
        string "title" shows as "Map Title"
    }

    class respRef{
        int "Fx"
        int "Fy"
        string "enclosingComponent" shows as "Enclosing Component"
        string "referenceID" shows as "Definition ID"

        association respID1 : references to
                              "responsibility" "Definition ID"
        association respID2 : boundto to
                              "map"."compRef"  "Enclosing Component"
        association respID3 : refines to "map"."map"
    }

    class noDescription compRef{
        int "Fx"
        int "Fy"
        int "Width"
        int "Height"
        string "Anchored"
        string "referenceComponent" shows as "Definition ID"
        string "role" shows as "Component Role"
        string "parentComponent" shows as "Parent Component"

        association compID1 : references
                               to "component"  "Definition ID"
        association compID2 : boundto
                               to "map"."compRef"  "Parent Component"
        association compID3 : refines  to "map"."map"
    }
```

```
    class noDescription stub{
        int "Fx"
        int "Fy"
        string "stubType" shows as "Stub Type"
        string "submapID" shows as "Plugins"
        association stubID1 : refines to "map"."map"
    }
}

module ignoreInReport grlDiagram{
    fileName "GRL Diagrams"

    class grldiagram shows as "grl diagram"{
        diagram "graphFileName" shows as "Diagram File Name"
        string "title" shows as "Diagram Title"
    }

    class parentActor{
    }

    class noDescription actorRef{
        int "Fx"
        int "Fy"
        int "Width"
        int "Height"
        string "referenceActor" shows as "Definition ID"
        string "parentActor" shows as "Parent Actor"

        association actorRef1 : references
                                to "actor"."actor" "Definition ID"
        association actorRef2 : boundto
                        to "grlDiagram"."parentActor"  "Parent Actor"
        association actorRef3 : refines to "grlDiagram"."grldiagram"
    }

    class intentionalElementRef{
        int "Fx"
        int "Fy"
        string "enclosingActor" shows as "Enclosing Actor"
        string "defID" shows as "Definition ID"
        string "priority" shows as "Priority"
        string "criticality" shows as "Criticality"
```

```
        association ieAsso1 : references
            to "intentionalElement"."intentionalElement" "Definition ID"
        association ieAsso2 : boundto
            to "grlDiagram"."actorRef" "Enclosing Actor"
        association ieAsso3 : refines to "grlDiagram"."grldiagram"
    }

    class belief{
        int "Fx"
        int "Fy"
        string "enclosingActor" shows as "Enclosing Actor"
        string "linkedElementId" shows as "Intentional Element ID"
        string "author" shows as "Author"

        association beliefAsso1 : boundto
                        to "grlDiagram"."actorRef" "Enclosing Actor"
        association beliefAsso2 : rationales
            to "grlDiagram"."intentionalElementRef"
                                              "Intentional Element ID"
        association beliefAsso3 : refines to "grlDiagram"."grldiagram"

    }
}

module component{
    class component{
        string "Type" shows as "Type"
        string "hostedDeviceID" shows as "Device ID"

        association compAsso1: hosts to "device"."device" "Device ID"
    }
}

module device{
    class device{
        string "speedFactor" shows as "Speed Factor"
    }
}
```

```
/*
 * Renaming the module intentionalElementAssociations to elementlink
 */
module elementlink{
    fileName "Intentional Element Associations"

    class elementlink{
        string "Type"
        string "sourceID" shows as "Source ID"
        string "destinationID" shows as "Destination ID"

        association elemlinkAsso1 : refines
           to  "intentionalElement"."intentionalElement" "source ID"

        association elemlinkAsso2 : refines
           to "intentionalElement"."intentionalElement" "destination ID"
    }

    class contribution{
        string "Type"
        string "contributionType" shows as "Contribution Type"
        string "Correlation"
        string "sourceID" shows as "Source ID"
        string "destinationID" shows as "Destination ID"

        association contributionAsso1 : refines
           to "intentionalElement"."intentionalElement" "source ID"
        association contributionAsso2 : refines
           to "intentionalElement"."intentionalElement" "destination ID"
    }
}

module strategy{
    class strategy{
        string "Author"
    }
}
```

```
        // Association type declarations
        associationType hosts "Hosts"
        associationType requests "Requests"
        associationType references "References"
        associationType refines "Refines"
        associationType tracedBy "Traced By"
        associationType boundto "Bound To"
        associationType rationales "Rationales"
        associationType urnLinks "Urn Links"
    }
}
```

Appendix C: Implementation Details

This online appendix includes the implementation details for the model traceability DXL library generation. This is packaged as Eclipse projects, one with the code itself, one with test cases, and one with the MT-DSL editor.

Available online at: http://www.eecs.uottawa.ca/damyot/pub/Rahman/AppendixC.zip

Appendix D: Generated Traceability Library

This appendix includes the DXL library generated for the subset of the URN notation used in this thesis and specified in Appendix B using MT-DSL. This library can be used as is in DOORS.

Available online at: http://www.eecs.uottawa.ca/damyot/pub/Rahman/AppendixD.zip

Appendix E: How to Run the MT-DSL Editor

In order to run the MT_DSL editor based on the packages provided in Appendix C, one first needs to configure the run time project. In Eclipse, this is done by selecting "Run As -> Run Configurations…" on a project. The 'Argument' and 'VM Argument' should be set as shown in Figure 84.

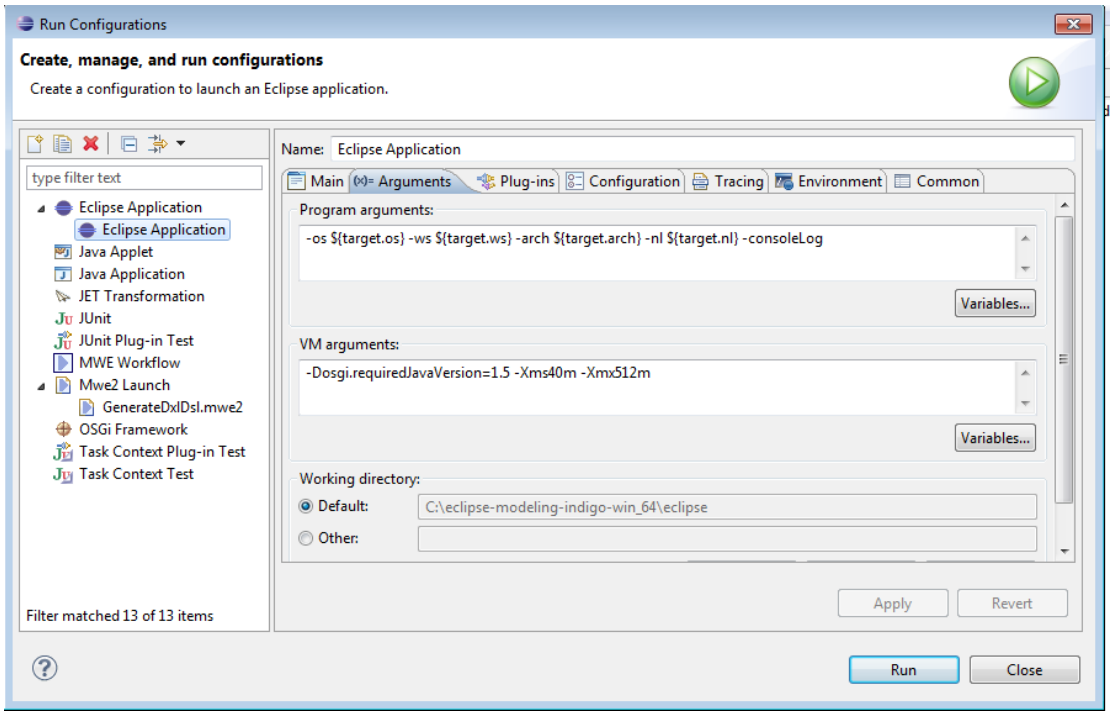

Figure 84 How to configure 'Run Configurations'

To run the editor, right-click on the name of the project in the project explorer. Select 'Run As' and then choose 'Eclipse Application' (Figure 85). A new Eclipse instance will be launched and will allow testing the new editor.

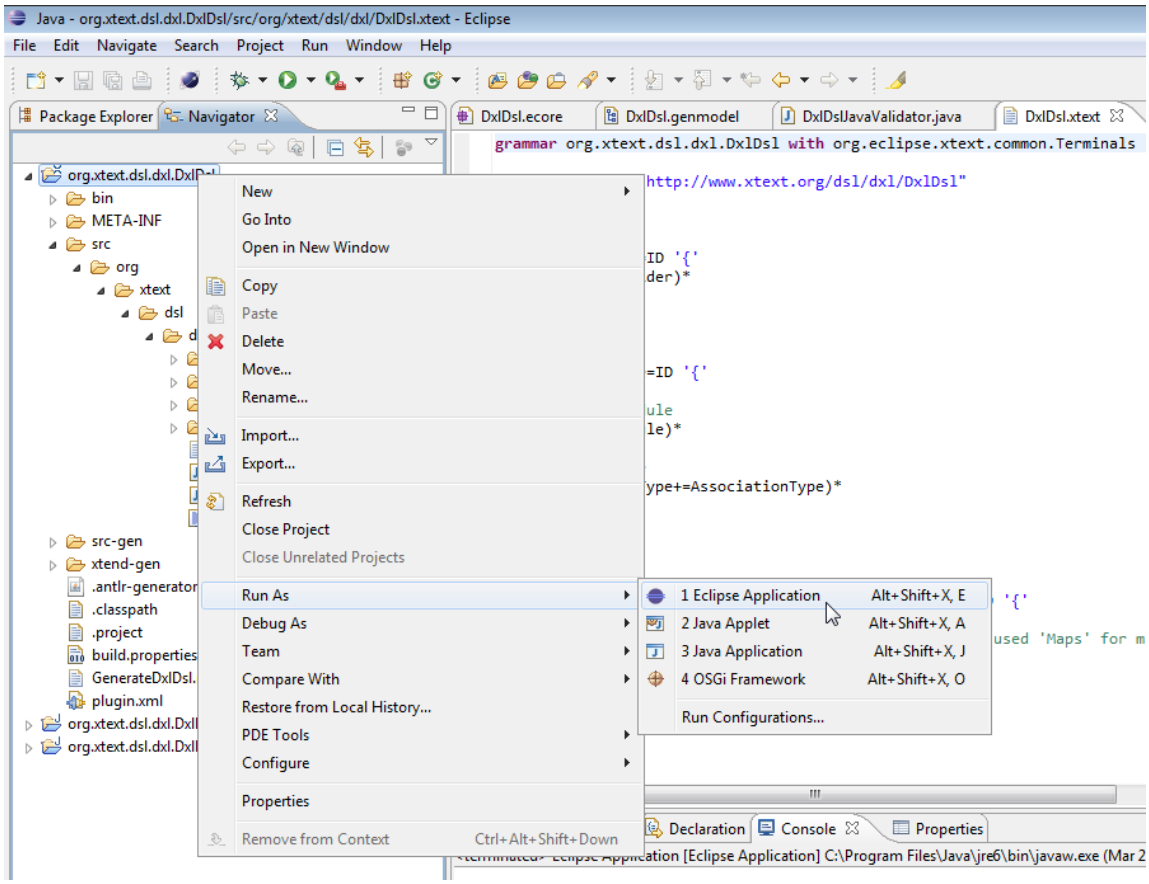

Figure 85 How to launch the editor as an Eclipse Application

A new project needs to exist before creating a file for the new language (Figure 86).

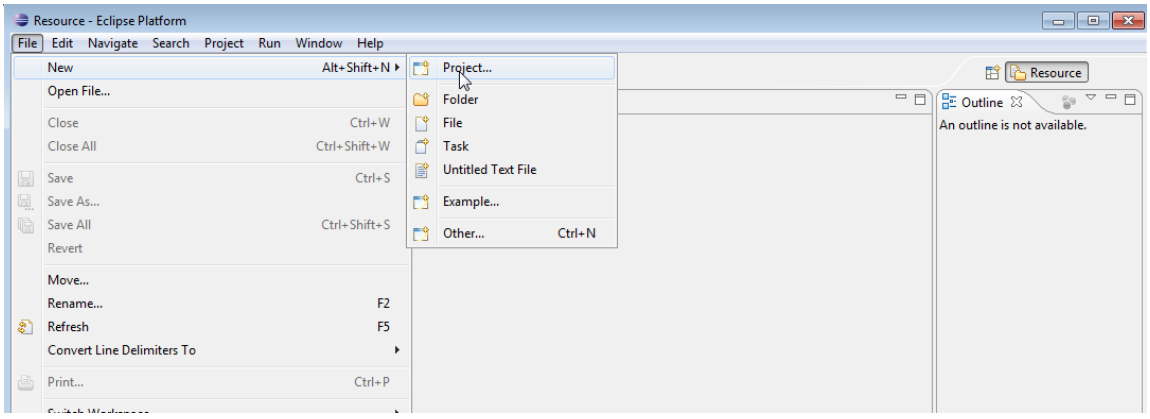

Figure 86 Create a new project using the File menu

An appropriate project type needs to be selected on the new project wizard (e.g., *Java Project* can be used). The project must be given a name, e.g., *SampleDSL*. In this project, create a new file (*SampleDSL.dxldsl*) for the DSL language in the source (*src*) folder. The extension (**.dxldsl**) is the key piece of information that will link the file with the MT-DSL editor.

Eclipse then opens the file in the Eclipse-based editor as shown in Figure 87.

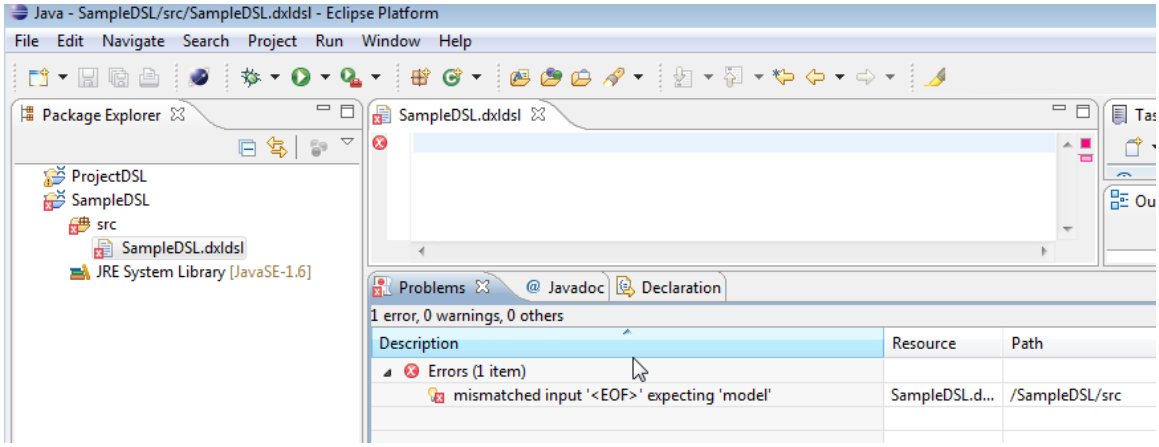

Figure 87 A new DSL language file opened in the MT-DSL Eclipse-based editor (with error marks)