A DSL for Importing Models in a Requirements Management System

Anisur Rahman and Daniel Amyot
School of Electrical Engineering and Computer Science, University of Ottawa
Ottawa, Canada
anisur8@hotmail.com, damyot@eecs.uottawa.ca

***Abstract*—Requirements are artefacts often described with text and models. It is important to manage traceability between requirements and other software artefacts, including designs and test cases, also often captured with specialized models. Some Requirements Management Systems (RMS) support traceability relationships, between (textual) requirements artefacts in the RMS and model artefacts created outside the RMS, through complex standards or proprietary solutions. This paper proposes a new Domain-Specific Language (DSL) for describing the concepts of a modeling language intended to be traced using an RMS, with tool support handling the import and re-import of models and of their traceability links. The Model Import DSL (MI-DSL) is supported by an Xtext-based editor and the automatic generation of an import library targeting a leading RMS, namely IBM Rational DOORS. The language and the tools are demonstrated for model import and evolution scenarios with two different modeling languages. This work contributes a simple yet reliable mechanism to define and support traceability between requirements and models from different tools.**

***Index Terms*—DOORS, DSL, evolution, model, requirements management, traceability.**

I. Introduction

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 capture 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. A change to a model might also impact linked requirements.

In modern software engineering, models (e.g., in UML) are widely used artefacts that provide abstract representations of various aspects of software. This is especially important in the requirements and design phases. 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 [22], an Eclipse-based modeling tool for the User Requirements Notation (URN) [1][16], provides traceability support between the elements of one URN model. However, modeling tools typically do not support traceability 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. Two decades ago, Gotel and Finkelstein argued that the majority of poor requirements problems are due to inadequate specification traceability [9]. Traceability across modeling artefacts and tools is often achieved through the use of external tools, for example a Requirements Management Systems (RMS). An RMS is an application that provides support for managing and analyzing evolving requirements while providing traceability, change management, and impact analysis. The emerging *Open Services Lifecycle Collaboration* (OSLC) standard [23] also allows loose integration of modeling and requirements tools, enabling them to collaborate and synchronize changes, but usually at the cost of major development effort [28].

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 leading RMS tools [11]. 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) or scripting capabilities for extending its capabilities, for customization, and for integration with other tools. For example, DOORS provides the *DOORS eXtension Language* (DXL) [13], 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 [17][24], Kealey [18], Roy [26], and Ghanavati [8] 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 nature and sources.

However, such an approach suffers from many limitations: it is specific to a pair of tools (here, jUCMNav and DOORS), it

is specific to one language (URN), it is specific to a particular set of *elements to track* in URN, and the maintenance of the manually-created DXL library is difficult. Ideally, 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 here hence is: *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*?

This paper uses the Design and Creation research method [27] to tackle this problem. This is a common approach used for solving problems in Information Technology products, where the end result is a computer-based system. The method steps are the following (with correspondence to paper sections):

- *Awareness*: As explained above, 1) traceability is important in software engineering [4][9], 2) traceability between models, or between models and other types of artefacts, often requires the use of third-party tools such as an RMS, or complex integration à la OSLC, and 3) previous work is not usually generalizable to other subsets of a language or to other languages without having different import mechanisms (e.g., DXL libraries in DOORS), each requiring efforts to write and maintain.
- *Suggestion*: In section II, we survey existing approaches taken to provide traceability in modeling, and analyze current gaps. Our resulting suggestion is to create a *Model Import Domain-Specific Language* (MI-DSL) for describing the artefacts (elements, attributes and associations) in a modeling language to be tracked in an RMS, with support for automated model import.
- *Development*: In section III, we provide a formal definition of MI-DSL (with a metamodel and Xtext [30]), together with the implementation of an advanced editor. In section IV, we also define transformations from MI-DSL to a target an RMS scripting language (DOORS DXL), and implement them with Xtend [29] and Java.
- *Evaluation*: In section V, we validate the feasibility and practicality of MI-DSL and transformations to DXL using two different modeling languages, namely Finite State Machines and the User Requirements Notation. Section VI further highlights how this approach relates to and complements existing work.
- *Conclusion*: Section VII argues that MI-DSL, together with transformations, provides a cost-effective and maintainable solution for tackling the model import and traceability issues identified earlier.

II. Related Work

A. Models and Traceability

Traceability provides a logical connection between artifacts of the software development process, as well as 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 [4][20]. Traceability is an important feature for quality-oriented software development, e.g., during change management.

Unless imposed by regulations, 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. Mäder and Gotel describe a semi-automated way of maintaining traceability by capturing flows of events in a prototype modeling tool [20]. This approach depends on *Event Based Traceability* concepts described by Cleland-Huang et al. [3]. Mäder and Cleland-Huang also introduced an expressive *Visual Trace Modeling Language* (VTML) [19]. VTML simplifies traceability queries and avoids the redundant use of internal data structures compared to other technologies such as XQuery. Mirakhorli et al. [21] worked on developing a *Tactic-Centric Approach* for automating the traceability of quality concerns, in which 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.

The problem of creating and maintaining traceability is further exacerbated by the difficulties in incorporating models created with specialized tools into an RMS. In terms of the traceability lifecycle defined in [20], this paper 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 [20] focus on a semi-automated way of *maintaining* traceability (in general, not just for models), VTML focuses on *using* traceability (through visual queries), and Mirakhorli et al. [21] focus on *creating* traceability links for software quality aspects.

Note that there already exist DSLs targeting traceability. For instance, Drivalos et al. propose the *Traceability Metamodeling Language* (TML), a DSL for describing traceability metamodels at a high level of abstraction [5]. They argue that TML enables the construction and maintenance of traceability metamodels and accompanying constraints with reduced effort. TML is defined with an Ecore metamodel (from the Eclipse Modeling Framework [6]) and includes concepts such as traces, contexts, context data, tracelinks, and tracelink ends. TML is not used to import models in an RMS; it is closer to 3rd-party traceability, with one tool for models, one tool for requirements management, and a third tool for handling traceability. The traceability is generic, but the part between a model and other types of requirements is handled outside of the RMS, which could negatively impact usability and utility (e.g., one would want to reuse the existing RMS analysis and reporting functionalities as much as possible).

Graf et al. present an Eclipse-based DSL of ReqIF, the *Requirement Interchange Format* [10]. ReqIF is an emerging standard for requirements interchange, which is driven by the German automotive industry. Their solution is a mapping table with three element types: source element, target element, and additional information such as descriptions. The elements are identified by a data structure called *tracepoint*, whose inner structure depends on the metamodel being used. Graf et al. describe that their technical solution is often based on the import of requirements into the target models of other tools. They see this as an important drawback as not all models support extensions.

In terms of general tool integration, OSLC uses linked data (based on the Resource Description Framework) and a resource management protocol to enable tools to interchange data and invoke mutual services [23]. OSLC goes beyond more specialized technologies such as IBM Rational Jazz [14] in that the former can handle tools based on different development platforms (e.g., non-EMF). Elaasar and Neal recently demonstrated that EMF-based RMS and UML modeling tools can be integrated using OSLC [7]. However, as noticed by Wolvers and Seceleanu, who went through a similar exercise [28], developing the required tool adapters and data definitions requires much effort, to the point where they claim that this is difficult to do as an afterthought once a tool exists. In addition, OSLC does not solve the problem of modeling which model entities, links, and attributes to track for analysis.

B. Background on Relevant Technologies

Requirements Management Systems, in addition to their core functionalities (i.e., managing and analyzing evolving requirements and their traceability links), often support extensibility mechanisms [15] useful for integrating models generated by specialized modeling tools. We have chosen DOORS as the target RMS tool for this work given its previously demonstrated capability to support the import of models. DOORS already comes with an external component (*DOORS Analyst* [12]) that allows UML models to be created and linked to other requirements. *MDConnect* [2] is another tool that enables the integration of UML models (in IBM Rational Software Architect) with DOORS. However, these are proprietary solutions that cannot be customized to other subsets of UML, to other modeling languages, or to other tools.

DOORS' scripting language, DXL, has features for capability extensions, customization, automation, and linking to other tools. DXL offers predefined functions to manipulate the various kinds of objects in its requirements database (projects, folders, formal modules, link modules, objects, properties, links, baselines, etc.), as well as ways to define new functions. Previous work led to the definition of a DXL library of functions for importing URN models from the jUCMNav tool (as well as its predecessor UCMNav). The main concepts and process are illustrated in Fig. 1. A model in a modeling tool (Fig. 1a) is first converted to a DXL script (Fig. 1b). jUCMNav has an export plug-in that generates DXL that enumerates the elements and links of the scenario view [8][24] and the goal view [17][26] of a given URN model. These include actors, scenarios, maps, goals, etc., and a subset of their attributes and relationships. Even the diagrams themselves are exported (as bitmaps). The generation of DXL scripts is automated in jUCMNav (implemented in Java), but such automation is outside the scope of this paper.

A generated DXL script is a sequence of function calls to a DXL library that handles the creation of new (or update/deletion of existing) projects, modules, objects, links and attributes corresponding to the model elements. Running such a script results in importing the model into the DOORS database (Fig. 1c), with one module per type of object, their attributes, their diagrams (wherever appropriate), and their typed traceability links. The library also enables models to be re-imported after modifications, while flagging modifications in DOORS, hence enabling change analysis.

The solution developed in this paper goes beyond this existing work by enabling requirements engineers and other analysts to specify the parts of a modeling language (elements, attributes and associations) that deserve tracking in an RMS, and by automatically generating a corresponding DXL library that enables importing the models into DOORS, with functions that even support the evolution of models and their linked artifacts, including requirements, after re-imports.

In order to do so, the next section presents a new *Model Import Domain-Specific Language* (MI-DSL), which is specified using Eclipse's *Xtext* [30]. Xtext is a framework to help implement a DSL with proper Integrated Development Environment support. Xtext-based editors include many language usability features such as syntax coloring, content assistance, validation and quick fixes. Xtext can also infer an Ecore metamodel automatically from the DSL syntax. In section IV, the transformation of an MI-DSL description into a DXL library of functions is implemented with *Xtend* [29] and Java. Xtend is an expressive dialect of Java, which compiles into readable Java 5 compatible source code. Xtend improves upon Java with a simpler syntax and with many features useful for handling languages and editors, including: lambda expressions, improved operator overloading, type-based switch expressions, multiple dispatch, template expressions with white space handling, and shorthands for accessing and defining getters and setters. Xtend is also promoted as an Xtext validation and transformation language.

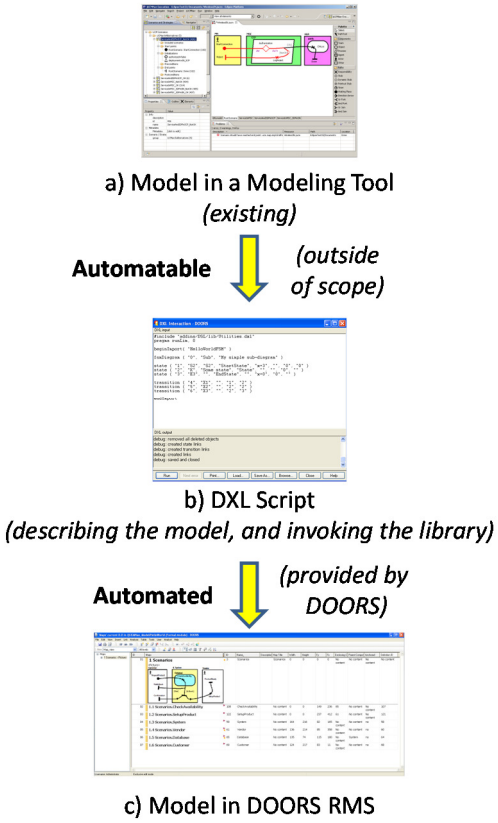

Fig. 1. DXL-based import of models in the DOORS RMS.

III. Model Import Domain-Specific Language

Modeling languages are often complex. For example, URN and UML have metamodels with hundreds of associations, classes, and attributes. Not all of them deserve being tracked in an RMS, as this could be distracting to users and lead to a large negative impact on the RMS performance. Analysts often need to focus on the types of model elements susceptible of being useful as source or target of traceability links. Different contexts may also call for different subsets of a same language. The purpose of MI-DSL is to enable the description (Fig. 2b) of a subset of concepts from an existing modeling language (Fig. 2a) while enabling the automated generation of a DXL library (Fig. 2c) enabling one to (re-)import models in that language according to the process illustrated in Fig. 1. In a sense, MI-DSL formalizes the notion of *Traceability Information Model* (TIM) used in traceability planning [4].

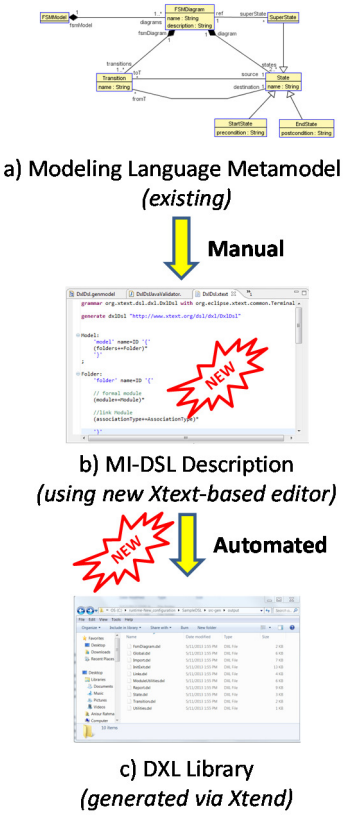

Fig. 2. Generation of a DXL library for the MI-DSL description of a traceability-oriented view of a modeling language.

A. MI-DSL Definition

In order to capture the structure of models, one needs a description language that includes concepts often used to specify modeling languages themselves (classes, attributes, associations, data types, some packaging facility, etc.). Figure 4 presents the MI-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 the Meta Object Facility (MOF) or Ecore. Figure 3 specifies the corresponding Xtext-based syntax. The DSL uses `Model` as a top-level element (with only one instance of `Model` per language/notation for which we want to generate a library). A `Model` contains `folders`, which contain local definitions of `Modules` and `AssociationTypes`. A `Module` can contain any number of `Classes` that often share similar attributes. A `Class` typically captures a language concept (a metaclass in the language's metamodel) one wants to trace in an RMS. `Classes` may contain typed `Attributes`, where each attribute is of a basic `DataType` (Boolean, string, integer, long text with many sentences, or diagram). An `Attribute` can have a `default` value, and modifications to its value can be `ignored` when the model is re-imported. An `Attribute` can also contain typed `Associations` to other `Classes`, which can be instantiated as traceability links. Most elements support unique identifiers (`IDs`) for external traceability, as well as textual descriptions. The remaining attributes of these elements are used to override default file names in the generated DXL library.

```
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, 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' ;
   // 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)? ;
   // the classType must be defined in the moduleType.
```

Fig. 3. Xtext description of MI-DSL.

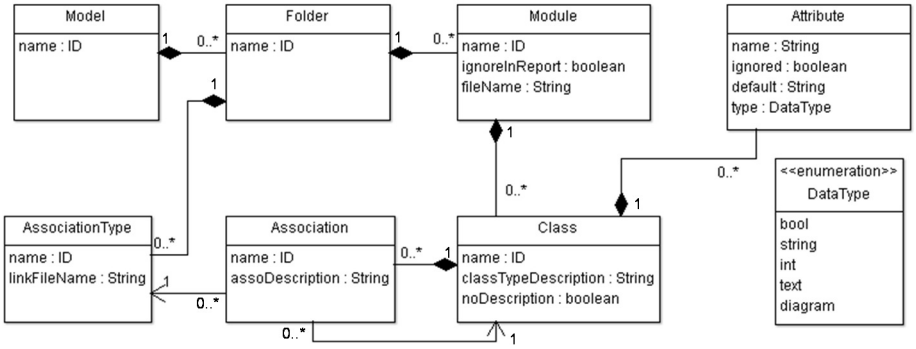

Fig. 4. Metamodel for MI-DSL shown as a UML class diagram.

B. Example of a Language Subset in MI-DSL

Figure 5 presents the metamodel of a simple Finite State Machine (FSM) language, where a model contains diagrams, each composed of states connected by transitions. Start and end states can be used, and super states (referencing a diagram) can be used to decompose complex models hierarchically.

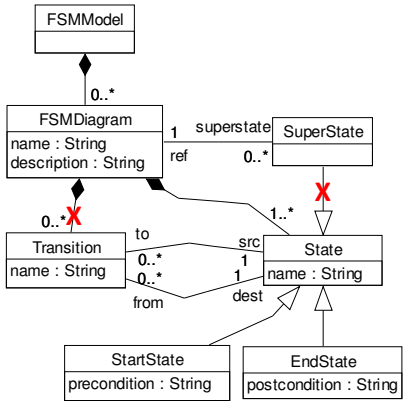

Fig. 5. FSM metamodel with a subset of interest for traceability.

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 metamodel. For example, the associations and inheritance relationships marked with '**X**' in Fig. 5 may not need to be tracked. Also, to avoid the proliferation of classes caused by inheritance (e.g., for states in this meta-model), a flatter class representation with the union of the sub-classes attributes can be used in MI-DSL and in the RMS. The corresponding MI-DSL description is found in the left part of Fig. 6. Note that IDs, names and descriptions are implicit attributes of any class in MI-DSL, so they do not need to be declared. The errors reported in that figure are explained in the next sub-section.

C. Xtext-Based MI-DSL Editor

We produced an Xtext-based MI-DSL editor for the Eclipse environment (Fig. 6). The editor comes with syntax highlight (with Eclipse preferences for font, color, and style for comments, keywords, numbers and Strings), predefined and selectable templates, a tree-like Outline view (on the right), and content assistance (for code/bracket completion). Default error handling (underlining, error symbol on the far left side, and error details in the Problems view) is also available. In addition, we added custom error handling that statically validates domain-specific constraints, including that an association references a class that exists in the described MI-DSL module. Such violation is also shown in Fig. 6, as the association `included` cannot find a reference to the association type `superStates`. The resolution here is to remove the extra "`s` " between `superState` and `Of`.

IV. Traceability Library Generation

A DXL traceability library is generated from a MI-DSL model, using Xtend (from *b* to *c* in Fig. 2). The DXL library is a collection of files containing functions meant to be invoked by a DOORS DXL script representing a model and generated by a modeling tool (Fig. 1b). We have used Xtend to produce a DXL code generator (model-to-text transformation) from MI-DSL descriptions that follow the syntax presented in Fig. 3.

Several assumptions were made in this transformation, including these three important ones:

- To simplify the use of MI-DSL by analysts, it is assumed that each `Class` contains implicitly three attributes by default: name, id, and description. The language exceptionally allows `Classes` without a description attribute by using the `noDescription` flag.
- By default, each `Module` is included in the DOORS report that is generated after each model import (to provide feedback about the import operation, and on re-import operations for models that have been updated or modified), unless the flag `ignoreInReport` is used.
- Any changes to an attribute of a `Class` flags the `Class` as modified in the generated report by default. The language however ignores the attributes flagged as `ignored`.

The library generated from a single MI-DSL description results in a collection of interrelated DXL files. Seven files (in italic in Table I, with the number of functions each file contains and a brief description of the nature of the file) are always generated for every MI-DSL description. One additional file is also produced per MI-DSL module (in bold in Table I). 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.

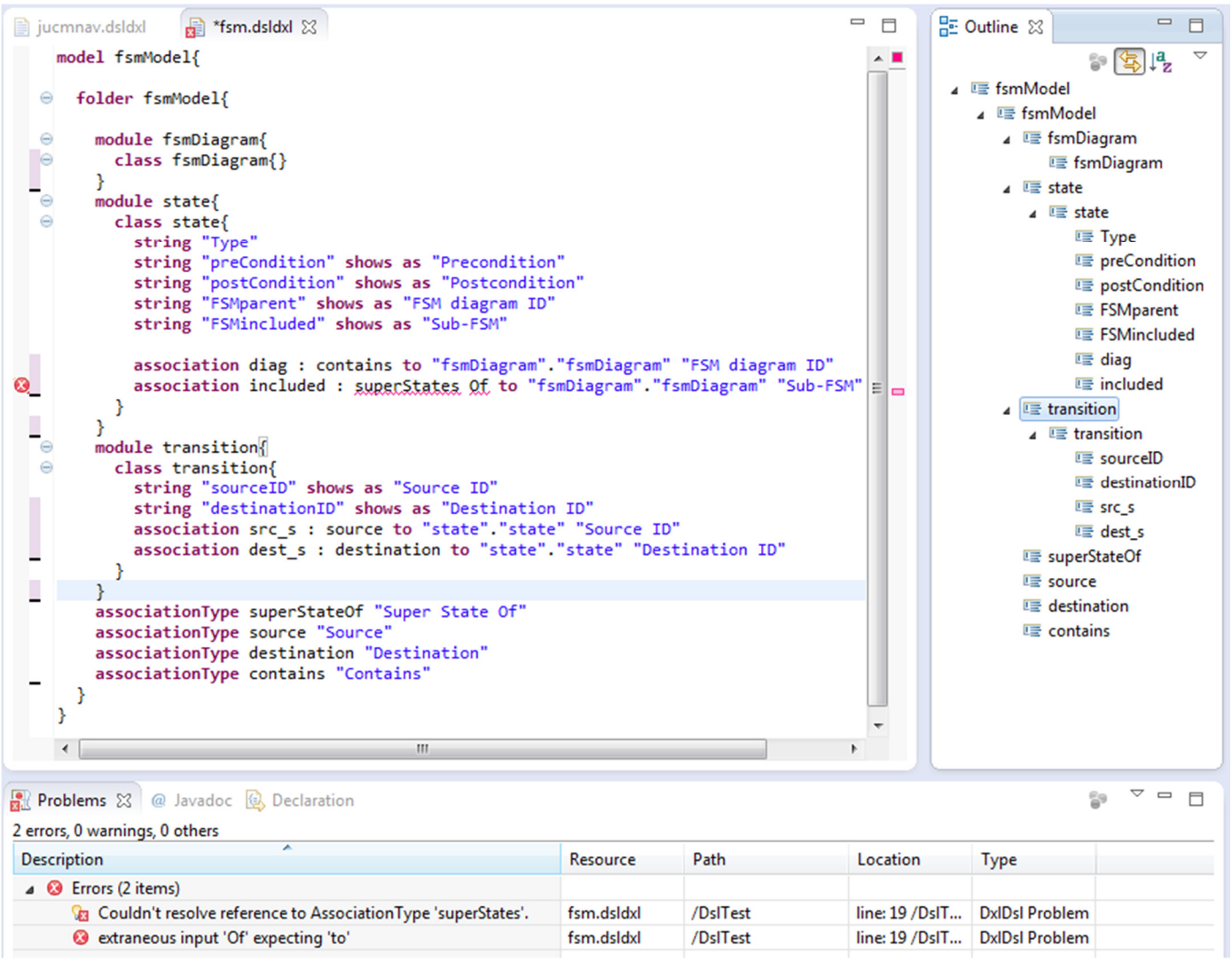

Fig. 6. MI-DSL description of the selected subset of the FSM language, with errors reported by the Xtext-based MI-DSL editor.

TABLE I GENERATED DXL LIBRARY FOR JUCMNAV/URN MODELS

Library File Name	Size	Func	Description of DXL Library File
import.dxl	192	2	Provides the utility methods that would be invoked to start the model import process in RMS. This includes the DXL function `beginImport` to start the import process.
InitExit.dxl	596	11	Contains all the DXL functions to initialize and finalize the import process (including GUI interactions).
ModuleUtilities.dxl	169	7	Includes the helper DXL functions that are invoked during the model import process in DOORS.
Utilities.dxl	21	0	Contains the list of import statements to import all other library files.
Report.dxl	199	6	Contains generated DXL code for creating a report at the end of the import process.
Links.dxl	275	4	Contains DXL library code for the links described in the modules.
Global.dxl	136	0	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

```
// State.dxl: Imports state (updates object if it exists otherwise creates new one). Always returns true
// Assumptions for this function: stateModule exists and is ready to be used

bool state(string stateID, string stateName, string stateDescription, string stateType,
       string statePreCondition, string statePostCondition, string stateFSMparent, string stateFSMincluded)
{
       Object foundObject
       foundObject = findObject( stateID, stateModule )
       if ( null foundObject ) {
               foundObject = createNewObject(stateModule )
               foundObject."ID" = stateID
               foundObject."Object Heading" = stateName
               foundObject."Name_" = stateName
               foundObject."ObjectType_" = "state"
               foundObject."Description_" = stateDescription
               foundObject."Type" =  stateType
               foundObject."Precondition" =  statePreCondition
               foundObject."Postcondition" =  statePostCondition
               foundObject."FSM diagram ID" =  stateFSMparent
               foundObject."Sub-FSM" =  stateFSMincluded
               foundObject."New" = true
               foundObject."Deleted" = false
        } else {
               if( foundObject."Name_" "" != stateName ) {
                        foundObject."Object Heading" = stateName
                        foundObject."Name_" = stateName
               }
               if( foundObject."Description_" "" != stateDescription )
                        foundObject."Description_" = stateDescription
               if( foundObject."Type" "" !=  stateType  )
                        foundObject."Type" = stateType
               if( foundObject."Precondition" "" !=  statePreCondition  )
                        foundObject."Precondition" = statePreCondition
               if( foundObject."Postcondition" "" !=  statePostCondition  )
                        foundObject."Postcondition" = statePostCondition
               if( foundObject."FSM diagram ID" "" !=  stateFSMparent  )
                        foundObject."FSM diagram ID" = stateFSMparent
               if( foundObject."Sub-FSM" "" !=  stateFSMincluded  )
                        foundObject."Sub-FSM" = stateFSMincluded
               foundObject."Deleted" = false
       }
       debug("imported state " foundObject."ID" "\n",3)
       return true
}
```

Fig. 7. Sample DXL function for a class with attributes (State).

One of our case studies (section V-B) is a MI-DSL description of a subset of URN (189 lines), for which a total of 16 DXL files were produced (2467 lines of commented DXL code), with a total of 47 DXL functions. MI-DSL descriptions are much more concise than the DXL equivalent (in addition to being less prone to errors given the support from the Eclipse-based editor), and much easier to maintain. The transformation code provided by us needed to be only created once and can be reused for all MI-DSL models. Also, the DXL script generator must be coded for a modeling tool whether MI-DSL is used or not, so the effort on that side is the same.

Since not all functions can be discussed here (the details are available in [25]), only one representative function generated from the FSM example is illustrated in Fig. 7. `state(…)` handles the creation and update of state objects. The function signature contains all attributes, including the implicit `ID`, `name`, and `description`, and the IDs of linked elements, with names prefixed by that of the class. If no object with the same `ID` exists in the corresponding DOORS module, then a new object is created and its attributes (including predefined ones for this RMS, e.g., `Object Heading`) are set. If an object with the same `ID` already exists, then only its attributes of interest (i.e., `ID`, `name`, and `description`, and additional MI-DSL attributes not prefixed with `ignored`) are updated while keeping track of the history of changes in DOORS. Such a function can then be invoked by a DXL script describing a particular model element. The generation of links between model elements (e.g., state and a FSM diagram) follows a similar logic: for each state in the module, the function essentially tries to find an FSM diagram with the corresponding `ID` and, if it exists and if the FSM diagram is not logically deleted, then a link of the required association type is created.

V. Experiments and Validation

We have experimented with MI-DSL and the generated DXL libraries with two modeling languages: the FSM language used in the last two sections (to check import and reimport functionalities rigorously), and the subset of the URN language already supported by jUCMNav (for a more direct comparison with existing work and to get confidence that we can replicate a manually generated library with similar results). Further details (tools, examples, and results) are available in [25].

To test each library, a model is created as a DXL script with the modeling language (this is done manually as the export of DXL scripts from existing modeling tools is outside the scope of this paper, as explained in Fig. 1) 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. Tests involving models and links to external requirements or models are also performed.

A. FSM Models

Figure 8 is a DXL script describing a simple FSM model. It contains an FSM diagram, three states of different types and preconditions (with invocations to `state()` from Fig. 7), and three transitions (S2→K, K→K, and K→E3). Running this script invokes the library generated for this language (Fig. 6) and results in a DOORS folder with three formal DOORS modules for the three MI-DSL modules, as well as four DOORS link modules for the four association types. Figure 9 highlights the content of the States formal module, with the three states, their attributes, and link indicators (the triangles).

```
#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
```

Fig. 8. Sample DXL script for an FSM with 3 states and 3 transitions.

ID	States	ID	Name_	Description_	Type	Precondition	Postcondition	FSM diagram ID	Sub-FSM
01	1 S2	1	S2	S2	StartState	x=3	No content	0	0
02	2 K	2	K	Some state	State	No content	No content	0	No content
03	3 E3	3	E3		EndState	No content	x=0	0	No content

Fig. 9. States formal module in DOORS, with attributes and links.

To test the capability of the generated DXL functions to handle change, a new transition (ID 7) was added, one was removed (X2), and 4 attributes were modified (name, description, and precondition of state S2, and description of transition. The model was re-imported in DOORS, and an inspection of the modules and of the object histories confirmed that the functions worked as expected: histories captured the modifications (with timestamps), and what needed to be added/deleted actually was, hence enabling various kinds of change/impact analyses on linked requirements in DOORS.

B. URN Models

URN [1][16] is a language whose metamodel contains nearly 90 classes and hundreds of attributes and associations. jUCMNav has an existing DXL export mechanism that targets a subset of these language constructs. We reverse-engineered this subset into an MI-DSL description highlighted in section IV and Table 1. The DXL code generated from the MI-DSL description was compared (by inspection and by using a *diff* engine) against the code manually produced. Beside the comments and the ordering of some parameters, no significant differences were encountered, hence suggesting that a complex language can be handled reliably and efficiently (several minutes for writing an MI-DSL description compared to days of manual development and debugging for the conventional DXL-only approach, when DXL expertise is available).

An existing URN model composed of 3 actors, 4 components, 2 responsibilities, 7 intentional elements, 1 goal diagram and 1 scenario diagram was described as a DXL script. This script was actually generated from jUCMNav directly, and then modified slightly to accommodate the strict ordering of parameters expected for IDs, names and descriptions by MI-DSL generated libraries (jUCMNav's DXL library uses an inconsistent ordering for some elements). The model was imported successfully, including the images, into DOORS.

In addition to a suitable coverage of object additions, deletions and modifications, we deleted a link from the DXL script and added a new one, again with successful results.

C. External Links

The tool-supported approach based on MI-DSL was further validated by evolving model elements linked from and to *external* requirements. In order to check both traceability directions (incoming and outgoing), two models (one FSM and one URN) were linked to each other, and then modified. It is important here to validate the impact of modifications on the source/target objects of a link, leading to *suspect links* in DOORS, whose purpose is to attract human attention for verification. This test also demonstrates that our tools support the concurrent usage of many modeling languages in an RMS, and hence a limited form of inter-model tracing.

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. After importing an FMS model and a URN model, we manually created five links between different states and actors. We then modified the original DXL scripts (we changed the names of one state and of one actor involved in external links) and reimported the two models. As expected, the links were still present, but the links in which the modified state/actor were involved were flagged by DOORS as suspect links, indicating that the source or the destination of the link was modified.

We further changed the input DXL scripts to delete an object involved as the source of an external link and another object involved as the target of an external link. As expected with DOORS, the first case resulted in a dialog box appearing and asking to confirm the deletion of the object with outgoing links. In the second case, DOORS provides a notification box indicating that the object could not be removed because of the presence of incoming links (links are owned by the source object in DOORS). This behavior was hence the one expected.

D. Threats to Validity and Limitations

In terms of *correctness*, one threat is that the DXL scripts could deviate from the initial models. This could be the case for

the FSM example (although it is so short that mistakes are unlikely). To mitigate this threat with the longer DXL script for the URN model, the script was generated using jUCMNav's existing export mechanism and not manually.

The tests could also miss important functionalities of the generated DXL libraries. We have however systematically covered additions, deletions, and modifications of objects and links in the many test cases done (including on links to external objects), and did additional ad hoc testing outside of what was reported here. History properties were inspected manually.

In terms of *internal validity*, there could be bias in our choice of languages and models used as test cases. We have partially mitigated this threat using one international standard and one language that came from a graduate course assignment. Also, the initial models existed before the experiments. More experiments by others would however improve our level of confidence in the results.

As for *external validity*, although we have checked that the approach can be generalized for two input languages, we have neither demonstrated nor tested that DXL scripts can actually be generated from modeling tools in general. Still, we believe this is easily feasible in many cases based on the prior experience of at least four studies [8][17][22][26], where researchers 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. We also believe that the common language construction concepts used in MI-DSL (classes, attributes, associations, data types and modules) are generic enough (as they are used in frameworks such as MOF and Ecore) to capture many interesting views of existing modeling languages. However, we have no proof that we will not face languages that will require improvements to MI-DSL itself. Improving MI-DSL and the editor is easy, but evolving the DXL code generation engine is more difficult.

Although MI-DSL is meant to be independent of Requirements Management Systems, our automation currently only targets one RMS, namely, DOORS. Even if DOORS is popular, this is certainly a threat to the validity of our work. Also, DOORS supports 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 MI-DSL so they can be exploited properly.

VI. Comparison with Related Work

As seen in section II-A, Mäder and Gotel [20] describe an approach to update traceability relations between requirements and UML artefacts (only). 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. Some external traceability links can then be inferred through their approach. Similarly, the work of Mirakhorli et al. [21] on automating the traceability of quality concerns is complementary to our work. Models (including quality models) can be first imported in the RMS and then quality-related external traceability links can be established with their approach. VTML [19] is actually a tool that can be used to query an RMS database. Such database can be populated through our approach, which again shows its complementarity.

MI-DSL is also complementary to other existing traceability languages such as TML [5] and ReqIF [10], which do not really support the import of models in an RMS. MI-DSL allows describing the views of a model to track in an RMS, but it does not formalize links between a model and external requirements, which is what TML aims to achieve.

The use of MI-DSL with generated DXL libraries represents an alternative to the use of OSLC. Using MI-DSL, our approach enables the simple selection of elements, attributes and links to be tracked in a RMS. This is not easily feasible with OSLC. Whereas the development of a DXL export for a modeling tool is also simple (boiling down to listing the elements and links of a model, but using a DXL syntax), the development of tool adapters in OSLC is difficult and time consuming. Although there are several RMS that now support some version of OSLC, very few modeling tools (outside of IBM's) currently support this standard. An OSLC approach however supports interactivity and immediate updates better in cases of change, and tools can talk to each other directly, even in the absence of an RMS.

VII. Conclusions and Future Work

This paper provides two main contributions, highlighted in Fig. 2. The first is the Model Import DSL for describing models in a way that enables model import and traceability management in an RMS. We have implemented this DSL independently of the target RMS. The DSL is supported by an Eclipse-based editor that provides content assistance, syntax highlighting, and error handling. The second main contribution is the tool-supported transformation of MI-DSL descriptions to a DXL library enabling models to be seamlessly imported in DOORS, a popular RMS. Traceability is maintained in DOORS by reimporting models as they evolve.

These contributions allow describing the traceability model using a DSL. As a result, the supported traceability model for any tool can be changed easily by simply updating the MI-DSL description file. No manual work is required to rewrite/update the library for the RMS. 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 MI-DSL descriptions for one modeling language (for different people, or even for one person with different needs).

Regarding our initial research problem, MI-DSL, the DXL generator, and the their application in case studies demonstrate that we can characterize formally the input modeling language and the traceability relationships of interest such that we can automate the import of models in an RMS. In addition, this approach offers an alternative to tool integration exploiting standards such as OSLC, with different trade-offs in terms of complexity, tracking of subsets of modeling languages, and reactiveness to changes.

Our work can be further extended in various directions:

- This approach should be validated on real-world, industrial systems, to study benefits in the field.
- Although DOORS was a simple choice given the expressiveness of its DXL scripting language, there is a need to study the applicability of MI-DSL to other RMS, and replicate these experiments.
- Advanced features of the current DXL library in jUCMNav support the generation of predefined 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 [8][18]. Such advanced features could justify extensions to MI-DSL.
- We have validated our approach against FSM and URN languages and models. Future work could further validate the approach through other modeling languages and tools.
- Our work could also be extended by automating the creation of a MI-DSL description from the metamodel of a language (if available), which could then be tailored to lead to a desired view on the language.

REFERENCES

[1] D. Amyot and G. Mussbacher, "User Requirements Notation: The First Ten Years, The Next Ten Years", Journal of Software, 6(5), 2011, pp. 747–768.

[2] S. Boucar, Traceability between DOORS requirements and RSA UML elements, Sodius, Dec. 2011, http://sodius.com/team-blog/2011/doors/traceability-doors-requirements-rsa-uml.

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

[4] J. Cleland-Huang, O. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability: trends and future directions". Future of Software Engineering, ACM, 2014, pp. 55–69.

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

[6] Eclipse Modeling Framework, http://www.eclipse.org/emf. Accessed June 2014.

[7] M. Elaasar and A. Neal, "Integrating Modeling Tools in the Development Lifecycle with OSLC: A Case Study". MoDELS 2013, LNCS 8107, Springer, 2013, pp. 154–169.

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

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

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

[11] IBM, Rational DOORS V9.5.1. http://www.ibm.com/software/awdtools/doors. Accessed June 2014.

[12] IBM, Rational DOORS Analyst Ad On. http://www-03.ibm.com/software/products/en/ratidooranaladdon. Accessed June 2014.

[13] IBM, DXL Reference Manual, Release 9.2, http://bit.ly/1iaEtvj. Accessed June 2014.

[14] IBM, Rational Jazz. http://www-01.ibm.com/software/rational/jazz/. Accessed June 2014

[15] INCOSE, Requirements Management Tools Survey, http://bit.ly/1iheCBZ. Accessed June 2014.

[16] International Telecommunication Union, Recommendation Z.151 (10/12), User Requirements Notation (URN) – Language Definition, Geneva, Switzerland, October 2012.

[17] B. Jiang, Combining Graphical Scenarios with a Requirements Management, Master of Computer Science, University of Ottawa, Canada, June 2005.

[18] J. Kealey, Y. Kim, D. Amyot, and G. Mussbacher, "Integrating an Eclipse-Based Scenario Modeling Environment with a Requirements Management System", 2006 IEEE CCECE, IEEE CS, 2006, pp. 2432–2435.

[19] P. Mäder and J. Cleland-Huang, "A visual language for modeling and executing traceability queries", Software and System Modeling, 12(3), 2013, pp. 537–553.

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

[21] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Çinar, "A tactic-centric approach for automating traceability of quality concerns", 34th Int. Conf. on Software Engineering (ICSE'12) IEEE CS, 2012, pp. 639–649.

[22] G. Mussbacher and D. Amyot, "Goal and Scenario Modeling, Analysis, and Transformation with jUCMNav", 31st Int. Conf. on Software Engineering (ICSE'09), ICSE Companion, IEEE CS, 2009, pp. 431–432. http://softwareengineering.ca/jucmnav/

[23] Open Services Lifecycle Collaboration, http://open-services.net. Accessed June 2014

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

[25] A. Rahman, A Domain-Specific Language for Traceability in Modeling, master's thesis, Electrical and Computer Engineering, University of Ottawa, Canada, July 2013. http://hdl.handle.net/10393/24346

[26] J.-F. Roy, J. Kealey, and D. Amyot, "Towards Integrated Tool Support for the User Requirements Notation", SAM 2006: Language Profiles - Fifth Workshop on System Analysis and Modelling, LNCS 4320, Springer, 2006, pp. 198–215.

[27] V. Vaishnavi and W. Kuechler, Design Science Research Methods and Patterns: Innovating Information and Communication Technology, Auerbach Publications, Boston, MA, USA. 2008.

[28] R. Wolvers and T. Seceleanu. "Embedded Systems Design: Integrating Requirements Authoring and Design Tools". 39th Euromicro Conference Series on Software Engineering and Advanced Applications, IEEE CS, 2013, pp. 244–251.

[29] Xtend, http://www.eclipse.org/xtend. Accessed June 2014.

[30] Xtext, http://www.eclipse.org/Xtext. Accessed June 2014.