MASTER THESIS

A Domain-Specific Language for Specifying Distributed Real-Time Software System Configurations

D.A. (Damiaan) van der Kruk

University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Department of Computer Science - Software Engineering Specialization

Abstract

Distributed real-time systems are complex systems to design and realize, because it involves extra dimensions of distributed and real-time concerns. The solution to realize these systems in a structured and manageable way is to utilize a component-based framework that is specially tailored for developing distributed real-time systems. Only a hand full of these frameworks are available and all of them work basically the same way of defining the structure and deployment settings of a target system in models, called software system configurations. These configurations can be provided to a code generator to create code and deployment setting files for the target system. This concept works very well, however defining correct software system configurations for any of these frameworks is pretty hard. This is because that those frameworks have two main shortcomings. The first one is that languages for defining software system configurations of these frameworks are too complex/hard to understand. The second one is that these frameworks don't provide any tooling with computer-aided features, like autocomplete and on-the-fly validation, to help with defining correct software system configurations.

In this thesis a domain-specific language (DSL) implementation is presented that overcomes the two mentioned shortcomings of other frameworks. This DSL implementation is called O2 DSL and is based on the existing component-based framework O2. O2 DSL is implemented with Eclipse Modeling Framework (EMF) and consist of three components; a textual part, a graphical part and transformations. The textual part is implemented with Xtext and consist of a textual grammar and editor with computer-aided features to fast and easily define software system configurations. The graphical part is implemented with Sirius and consists of graphical views with an editor. The transformations are implemented with ATL and Xtend and provides O2 DSL with a way to generate code and deployment settings files from defined software system configurations with the textual and/or graphical parts. This combination of textual and graphical views with computer-aided features, makes defining correct software system configurations much easier compared to existing frameworks and is an unique combination in the area of distributed real-time system development. O2 DSL is tested with a case study showing that with it a software system configuration can be defined for a target system. Also is the O2 DSL demonstrated to different developers/engineers that work in the area of distributed real-time system development. Their reactions where very positive and they see a lot of potential for O2 DSL. This says us that O2 DSL is possibility the solution to define correct software system configurations in an easy and fast way.

Contents

List of Figures

List of Tables

List of Code Snippets

List of Abbreviations

ATL	ATL Transformation Language	A model transformation language specified by AtlanMod research group used to specify the way to produce target models from a set of source models (model-to-model transformations).
DSL	Domain-Specific Language	A programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.
EMF	Eclipse Modeling Framework	The EMF project is a modelling framework and code generation facility for building tools and other applications based on a structured data model.
GPL	General Purpose Language	A programming language designed to be used for writing software in a wide variety of application domains. For example, C, Java and C#.
HUTN	Human-Usable Textual Notation	HUTN is an Object Management Group (OMG) standard for storing models in a human understandable format. In a sense it is a human-oriented alternative to XMI; it has a C-like style which uses curly braces instead of the verbose XML start and end-element tags.
IDE	Integrated Development Environment	A software application that provides comprehensive facilities to computer programmers for software development. A modern IDE normally consists of a source code editor, build automation tools and intelligent code completion.
MDE	Model-Driven Engineering	A software development methodology which focuses on creating and exploiting domain models, rather than on the computing concepts.
UML	Unified Modeling Language	A general purpose language for analysis, design, and implementation of software-based systems as well as for modelling business and similar processes.
XMI	XML Metadata Interchange	An OMG standard for exchanging metadata information via XML.
XML	EXtensible Markup Language	A language that was designed to describe data and is a software- and hardware-independent tool for carrying information.
XSD	XML Schema Definition	An XML Schema describes the structure of an XML document.

1 Introduction

Designing distributed real-time systems is very complex because not only the standard concerns of designing a system (like functionality, maintainability, supportability, etc.) are involved, but also the distributed and real-time concerns [1]. Real-time concerns are involved in systems where the correctness of their behaviour not only depends on the logical results of computations, but also on the physical time to produce these results. These results must be produced within a specific time or else the result is useless or even can result in dangerous real-world situations (safety-critical systems). Distributed concerns come into play when computation of a result is distributed using a communication network over multiple computer nodes, mainly for the reason that complete computation of the result on one computer is not efficient/fast enough. Systems which distribute computations over multiple nodes, like Folding@home [2], are called distributed systems and their distributed concerns are mainly about how computational data (inputs and results) can be correctly and/or rapidly distributed and collected. Both real-time and distributed concerns bring their own set of domain problems (and solutions), resulting in extra dimensions when designing distributed real-time systems and making them more complex.

Different (hard- and software) technologies are available to handle distributed and real-time concerns of systems. Selecting the most suited technologies and composing them in the right manner results in a (good) system which handles all its desired concerns. To explain this better we use a modern radar system (which is a distributed real-time system) as an example. Radar systems have real-time concerns because detection and identification of interesting objects (like airplanes) must happen within a specific time, so that a radar operator or other systems can react in time. Detection and identification of interesting objects costs a lot of processing power, so much that the processing must be distributed over multiple computers, involving distributed concerns. These two factors form that modern radar-systems are distributed real-time systems.

		Signal Processing	Data Processing	Mission Processing
Environment Domain		Data rates: 1 – 60 GBytes/sec Time domain: Mirco- to miliseconds	Data rates: 0.5 - 70 MBytes/sec Time domain: Mirco- to miliseconds	Data rates: 1 - 10 MBytes/sec Time domain: Miliseconds
Technology Domain	Design	Simulink	UML / XML	
	PL	C		Java
	OS	VDK	Linux	
	Proc.	DSP	GPP	
	Com.		UDP / TCP, DDS	SOAP

Figure 1.1: A subset of the processing phases of a radar system with their corresponding environment and technology domains.

In Figure 1.1 the technology domain of the digital processing part of a radar system is depictured, showing that it is also composed of multiple technologies. In this figure three processing phases with their environment domains (shown at the top of the figure) are depictured. In each phase different technologies are used (shown at the bottom of the figure) which works best for handling the data rates within the desired time domain of that phase. The flow of processing data begins left with Signal

Processing, that processes the raw digital signals in plots, and ends on the right-hand side with Mission Processing, where identified objects are send to a console for the operator or other systems. On the left side we see that more lower-level technologies are used for handling high data rates in the time domain where processing must happen in microseconds (µs), and on the right-hand side more higher-level (general-purpose) technologies are used where (more complex) processing with lower data rates must happen in the milliseconds (ms) time domain. Also on the right-hand side at the bottom we find transport protocol technologies (TCP/UDP, DDS and SOAP) used for handling the distributed concerns.

1.1 O2: A Component-Based Framework for Distributed Real-Time Systems

Distributed real-time systems consist of multiple technologies. Writing manually solutions with all these technologies and communication between them can be very hard, definitely when developing larger systems. There are different kinds of frameworks available to make development of distributed real-time systems easier. In this thesis we focus on in the domain of development of software designed to be deployed on computers with general-purpose processors (GPP). In Figure 1.1 this is the technology domain shown on the right-hand side used in the Data Processing and Mission Processing phases. O2 is a framework designed for this domain and has as extra feature that it is more focused on performance (handling high data-rates) than other kinds of frameworks [3]. O2 is proven (successfully used in practice) in the domain of radar systems, though it is also perfectly usable for other distributed real-time systems. O2 is also a platform-independent framework, meaning that is supports multiple operating systems and programming languages.

Developing a software system with O2 starts with defining a software system configuration, which specifies software components, the communication between these components and how these must be deployed on target computer nodes. Defining software system configurations for O2 was originally done with EXtensible Markup Language (XML) [4] files. XML is a textual mark-up meta language standardized by W3C. XML uses mainly tags for defining nested document structures. Software system configurations for O2 defined in XML files (O2 XML) can be provided to the code generator of O2 which parses and validates them, and then generates code and configuration files for the target system.

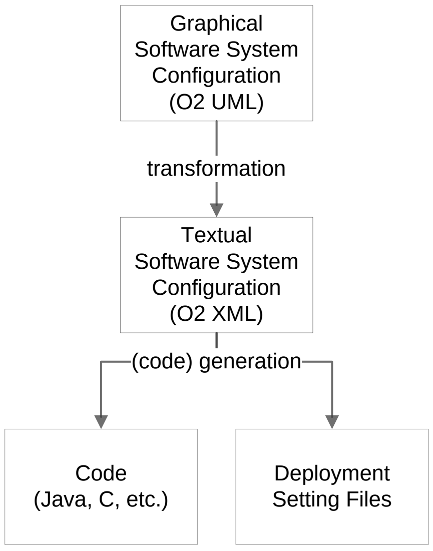

Graphical Software System Configuration (O2 UML)
transformation
Textual Software System Configuration (O2 XML)
(code) generation
Code (Java, C, etc.)
Deployment Setting Files

Figure 1.2: Process flow of how software system configurations in O2 UML are transformed into O2 XML and eventually resulting in generated code and deployment setting files.

XML provides a strong hierarchical structure, however its syntax is very noisy (verbose), making it hard for humans to read and analyse [5]. For small software system configurations, consisting of a few (small) XML files, this is less a problem, while it is for larger software system configurations, consisting of a lot of (big) XML files, a much bigger problem, for the simple reason that it is hard to read multiple XML files to understand a complete software system configuration. Graphical representations are in most cases much better suited for analysing large structures then textual representations [6]. Because of this reason a graphical-based way of defining software system configurations is realized for O2. This graphical-based way is a set of graphical representations which extends the Unified Modeling Language (UML) [7] diagrams using stereotypes. Each graphical representation is designed for describing a specific part of the software system configurations, which made is easier to understanding software system configurations.

A software system configuration defined in UML (O2 UML) is just a specification of a system and not runnable/deployable software. To get runnable software from a software system configuration with O2 the process as showed in Figure 1.2 should be followed. This process starts with transforming the graphical software system configuration defined in UML diagrams into XML files. These XML files conforms to the format of O2 XML for defining software system configurations for O2. Because of this the existing code generator can be utilized, generating in the same way code and deployment setting files. The transformation is only one way, meaning that software system configurations defined in O2 UML can be transformed into XML that conform to O2 XML, however not the other way around, O2 XML to O2 UML. This is because no transformation of implemented for O2 XML to O2 UML. This results in that existing software system configurations that are defined with O2 XML, must be completely rewritten in O2 UML.

1.2 Thesis Motivation

O2 and other component-based frameworks for distributed real-time systems greatly improved the way of developing distributed real-time systems, however making correct software system configurations with them is still pretty difficult. The main reason behind this is that current frameworks have two main shortcomings. The first shortcoming is that the languages for defining software system configurations of these frameworks are too complex/hard to understand. This has two main reasons. The first one is that most languages of frameworks are built on top of a complex and verbose host language, resulting in that that these languages directly inherit the verbose and complex structure of their host language. For example, O2 XML and a lot of other languages are based on XML. These languages inherit directly the complex and verbose structure of XML (with tags </>) [5]. Verbose and complex structures make languages much harder to understand and use.

The second reason that languages for software system configurations are hard to understand is because they are either completely graphical-based or completely textual-based. Some parts of a software system configuration can only be graphically represented in an unconventional way, while other parts are less well suited to be textually represented. When a graphical or textual representation is most suited highly depends on the task users want to perform. For example, defining data-types/message-types is much easier and faster done textually then graphically, but on the other hand a graphical representation is much better suited for analysing the composition of multiple software components in a system. This makes specific tasks with software system configurations more complex when a language is only graphical-based or textual-based.

The second shortcoming is that there is no computer-aided tool-support in form of on-the-fly validation and auto-completion for defining software system configurations. Modern Integrated Development Environments (IDEs) for programming languages (Java, C#, etc.) provide these computer-aided features, which greatly improved working with programming languages over the last decades. These computer-aided features can also greatly improve the way of defining software system configurations. For example, validation of software system configurations with O2 only happens when they are provided to the code generator, resulting in trail-and-error work costing a lot of time. On-the-fly validation and other computer-aided features can improve this by giving the user direct feedback during editing of software system configurations.

A Domain-Specific Language (DSL) is a programming language or specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [8]. *The goal of this thesis is to realize a DSL which overcomes the mentioned shortcomings (of existing languages of framework) that make defining software system configurations difficult, or in other words realizing a DSL for defining software system configurations with textual and graphical views including computer-aided tooling.* For this DSL the O2 framework is taken as base, because it is a proven-framework in the field of radar systems and is extensively used in the company where this thesis project was performed (Thales Nederland). Also no other frameworks are known so far which solve both mentioned shortcomings [9], making it no real alternative of choosing a different framework then O2 as base. How the new DSL is positioned is shown in Figure 1.3. On the left the current situation of defining software system configurations with O2 is shown and on the right the new DSL implementation. The new DSL (O2 DSL) includes both support for graphical and textual views and some kind of tooling or IDE support with computer-aided features like on-the-fly validation and autocomplete for this DSL. Other constraints/features of this new DSL are backward compatibility for support of existing/legacy systems made with XML of UML approaches of O2 and a notation that is not bound to any (complex) host language. The existing code generator of O2 is utilized in this approach elimination the need of implementing a new code generator.

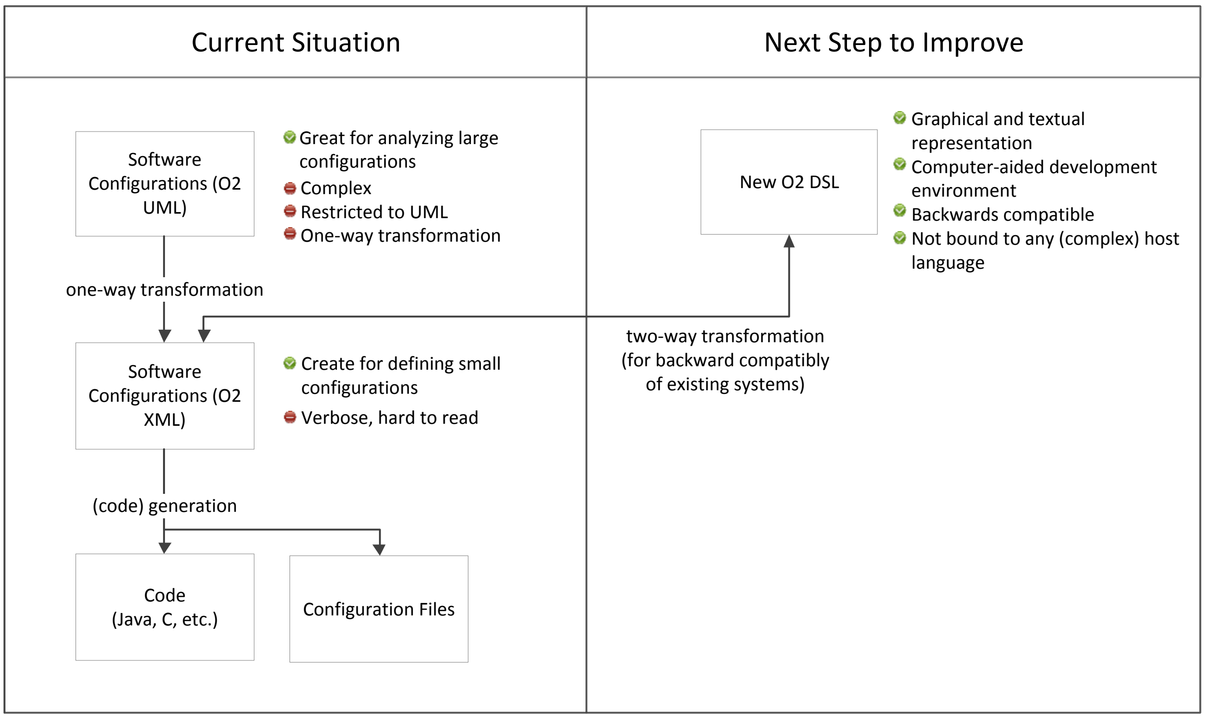

Current Situation
Next Step to Improve
Software Configurations (O2 UML)
Great for analyzing large configurations
Complex
Restricted to UML
One-way transformation
one-way transformation
Software Configurations (O2 XML)
Create for defining small configurations
Verbose, hard to read
(code) generation
Code (Java, C, etc.)
Configuration Files
New O2 DSL
Graphical and textual representation
Computer-aided development environment
Backwards compatible
Not bound to any (complex) host language
two-way transformation (for backward compatibly of existing systems)

Figure 1.3: On the left side the current situation of defining software system configurations for O2 and on the right side the next step of improving defining software system configurations for O2 using a new DSL.

1.3 Document Outline

This thesis report is structured as follows:

1. **Background Material:** Introduction to important concepts that are referred to throughout this report.
2. **Method of Implementing The DSL:** The method that was taken in realizing the new DSL, and overview of the architecture that was used.
3. **Implementation of The DSL:** Description of the implementation of the DSL.
4. **Discussion**: Discussion about the implementation of the DSL and its limitations.
5. **Conclusion and Recommendations:** Concluding comments and suggestions for future work.

2 Background Material

In this chapter, some of the more important concepts which been referred to later on in this thesis report are briefly described.

2.1 Domain-Specific Language Design

A Domain-Specific Language (DSL) is a programming language or specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [8]. In this section the different kinds of DSLs are discussed, advantages and disadvantages of DSLs, how to develop a DSL and tool support for creating DSLs.

2.1.1 Domain-Specific Languages

DSLs are not new. Most older programming languages (like Cobol, Fortran and Lisp) came into existence as DSLs for solving problems in a certain area (business processing, numeric computation, symbolic processing). Over time these programming languages have evolved into general-purpose languages (GPLs) [8]. Still new DSLs have surfaced and being used over the years, simply because they are doing a better job (when they are well-designed and implemented) of solving problems in certain domains then GPLs.

In the book “Domain Specific Languages” by M. Fowler [5] DSLs are distinguished in two groups, internal DSLs and external DSLs. An internal DSL is a form of an Application Programming Interface (API) in a host GPL. A technique that is used by a lot of internal DSLs is called fluent interface [10]. An example of an internal DSL that uses Fluent Interfaces is Mockito [11]. Mockito is a mocking library for defining expectations of (unit) tests with fluent interfaces and uses Java as host GPL. An advantage of internal DSLs is that they can be parsed by their host language, which is useful for DSLs that are designed to be directly executable. The disadvantage of this is that internal DSLs are restricted to their host GPL, which allows for no custom notations and platform switching. Basically an internal DSL is a DSL that is completely embedded in a host GPL, because of this internal DSLs are also referred as embedded DSLs.

External DSLs are DSLs having their own custom syntax and are parsed independently of a host GPL. Examples of external DSLs are CSS and HTML, also theO2 XML and O2 UML can also be seen as external DSLs. An external DSL is not restricted to a host GPL, allowing external DSLs to have custom structures which are better suited for the domain of the language. The disadvantage of this is that custom tools (such as the DSL compiler) need to be developed for enabling an external DSL. Although the time for developing custom tools for enabling an external DSL is greatly reduced by DSL tools like Xtext [12], it costs in most cases more time than creating an internal DSL. An example of SQL (an external DSL) and jOOQ (an internal DSL with Java as host language) [13] is shown in Code Snippet 2.1. Both DSLs are used for defining database queries.

SQL	jOOQ DSL
`SELECT * FROM BOOK`	`create.selectFrom(BOOK)`
`  WHERE BOOK.PUBLISHED_IN = 2011`	`  .where(BOOK.PUBLISHED_IN.eq(2011))`
`  ORDER BY BOOK.TITLE`	`  .orderBy(BOOK.TITLE);`

Code Snippet 2.1: Left an example of external DSL SQL and right of internal DSL jOOQ, both used for defining database queries.

Most common DSLs are textual DSLs, but a DSL can also be a graphical DSL. Graphical DSLs require a tool or IDE for editing and visualization. Graphical DSLs are usually better in representing structures of data, code or systems than textual DSLs. Most graphical DSLs are specialized views or have some filtering options for showing only the information that is needed by the user. For example, the class diagram of UML is specialized in showing the structure and dependencies of classes of OO applications. Specialized views with graphical navigation options make analysing large amounts of data, code, or large systems easier then would be possible with textual representations [6]. The UML approach of O2 for defining software system configurations can also be seen as a graphical DSL with specialized views to define different parts of software system configurations.

Most DSLs are supported by a DSL compiler which can generates applications based on a DSL instance. These kinds of DSLs are referred as application-specific languages and their DSL compilers as application generators [14]. Other DSLs are not aimed at specifying complete applications such as O2 XML and O2 UML, but rather at generating components or libraries. Some DSLs, like TeX and HTML, are aimed at specifying and generating documents. DSLs can also be simply a macro or scripting language enabling end-users to perform simple programming tasks. An example of this is Excel VBA enabling spread sheet programming. A common term also used for DSLs is 4th Generation Languages (4GL) [8].

2.1.2 Advantages and Disadvantages

Developing and using a new DSL have different advantages and disadvantages. Well-designed DSLs manage to find a proper balance between these two [8]. The advantages and disadvantages mentioned here should be interpreted as potential advantages and potential disadvantages (risks). This is because not all advantages and disadvantages apply to all situations in which a DSL is used. Potential advantages of DSLs include:

A1. **Productivity increase** – The main reason of introducing a DSL is to make it possible for programmers to develop applications faster. An aspect of DSLs that contributes to this is the reuse of DSL compiler code of the DSL. Every time an instance of the DSL is compiled/interpreted the code of the DSL compiler is reused which otherwise had to be manually developed.

A2. **Domain experts as programmers** – The syntax of a DSL can be designed to be understandable for non-programmers. This may allow domain experts to create applications by themselves without programming assistance of someone else. This eliminates the “middle man” and reduces the risk that the code not corresponds to the intentions of the experts.

A3. **Improve maintenance** – Back-end architectures consisting of hundreds of applications are hard to change. When applications are specified in DSLs, only the DSL compiler needs to be modified to obtain full conversion. This also improves portability, because moving to another platform can just be realized by re-implementing the DSL compiler. Maintenance can be improved even further when the DSL is designed to specify application in a self-documenting manner.

A4. **Faster applications** – DSL compilers can be designed to produce highly optimized applications for the desired domain. This is because of the fact that many more opportunities for optimization may be found in *domain-specific code* compared to code in a GPL. Applications generated by these compilers may outperform most hand-written solutions in practice.

A5. **Reliable applications** – DSLs can be constructed to support validation of (static) properties at domain level. This is usually a lot simpler to verify than in GPLs. A DSL compiler may also be constructed to prevent incorrect, unsafe and undesired structures in programs.

In most literature the usage of DSLs are praised, only highlighting their advantages. But a DSL is in no way a “silver bullet” for all software engineering problems. There are several disadvantages which have to be considered before choosing to develop and use a DSL. Potential disadvantages of DSLs include:

D1. **Education cost for DSL users** – Most DSLs are designed to be appealing for their intended users. Although this would normally make DSLs easier to work with, there is always some learning effort involved when using a new programming language.

D2. **Complexity of language design and implementation** – Creating a DSL involves designing a syntax and implementing a DSL compiler for it. This introduces an extra layer of complexity which costs initial effort to realize. For example realizing a DSL compiler introduces the complexity of handling with language-related problems (e.g. parsing, type-checking, etc.). Although creating a DSL has a higher initial cost, it normally reduces the cost of every application made with it afterwards. There are a lot of cases where the initial costs of a DSL are not worthwhile. This is one of the main reasons of not choosing for a DSL.

D3. **Harder maintenance** – Introducing a DSL can improve maintenance and portability as mentioned in potential advantage A3. But writing tools for enabling a DSL (like the DSL compiler) can be harder than just writing a number of applications. This is especially true for programmers that don't have experience with developing DSLs. For them maintaining a bunch of "normal" applications is easier than maintaining the tools for enabling a DSL.

D4. **Slower applications** – Some DSL compilers incorporates optimizations as mentioned in potential advantage A4. But sometimes DSL compilers are too complex, making optimizations hard to realize. When writing of these optimizations is too hard to realize (no time or budget is given), it can result in unacceptable efficiency loss.

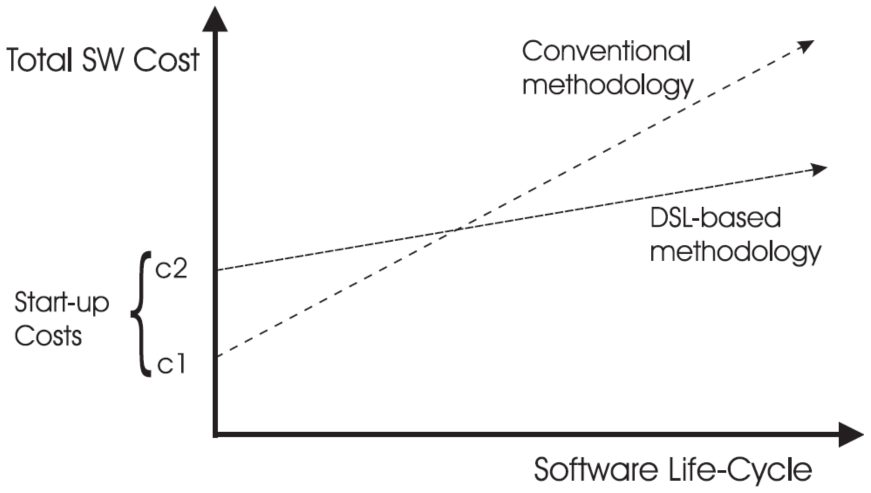


Figure 2.1: The payoff when using a DSL taken from **[15]**.

Most disadvantages are related to initial costs of introducing a DSL (realizing tools for enabling a DSL, training cost, etc.), while most advantages are related to improve efficiency in the long/continues term when using a DSL. The introduction of a DSL only pays off when over time the advantages outweigh the disadvantages [15]. When a DSL pays off over time is visualized in Figure 2.1. The initial costs (Start-up Costs) of developing applications the conventional way (c1) is lower than that of the DSL-based way (c2). When over time more and more applications are made (Software Life-Cycle) with the DSL, the total cost will be eventually lower than with the conventional way. When the total cost of the DSL-based way hits the total cost of the conventional way, it is better to start with a DSL right away.

The DSL implementation that is explained in this report mainly focuses on the advantages; increase productivity (A1), improve maintenance (A3) and realizing reliable applications (A5), however the disadvantages, education costs for users (D1) and complexity of language design and implementation (D2) are also involved in this implementation. Reducing the effect of each of these disadvantages to a minimum results into the best possible implementation of a DSL. In the following two paragraphs is discussed how the effects of the two involved disadvantages are minimized.

Reducing the effect of D1, a language have to been defined that is easy to learn and work with. Three action have been taken to accomplish this for the DSL implementation. The first one, is that a syntax is chosen that is familiar for the users of the DSL, yet is still small and with no verbose elements like with XML. Because most users of the DSL (probably) have a programming background with programming languages as C, C++, C# and Java, a DSL with a syntax that close the notation/style of these programming language should be picked up much easier by them. For the DSL implementation is chosen to use the Human-Usable Textual Notation (HUTN) as base to reduce the effect of D1, more about this notation and how it precisely is used in syntax of the DSL is discussed in 4.1. Secondly, in the DSL implementation the same concepts and their hierarchical order are used as in the existing XML and UML approaches of O2, this way the DSL implementation must be much easier to understand for existing users of O2. Thirdly, the syntax of the DSL provides some freedom for the user to structure elements where he wants, allowing them to easily experiment and prototype with it. For example, a complete system defined with the XML or UML approach of O2 consists of multiple XML files or UML diagrams, where with the new DSL implementation a complete system can be defined in only one file (or as many files the users wants). This freedom within the syntax of the DSL in addition with the computer-aided tooling results in an environment in which users easily can try out the features of the DSL implementation, without any extensive training.

To reduce the effect of D2 is chosen to use a framework and available tooling to implement the DSL. Frameworks and tools for implement DSLs can take care of generating DSL compilers, parsers, type-checking and other components and features, which greatly improves the way of realizing a DSL and maintaining one. Such a framework with tooling is also used for implement the DSL, the framework that is used (EMF) is discussed in 2.3 and how this framework is applied in the DSL implementation is discussed in 3.1. A set of mature frameworks for implementing DSLs is briefly discussed in 2.1.4.

2.1.3 Development Process of a DSL

The development of a DSL consists of the following five phases [16]:

1. **Decision** – Making the decision whether to create a new DSL or not. The most important question in this phase is "Will the investment of a DSL (probably) payoff?". Answering this question by comparing the advantages and disadvantages of a DSL (as discussed in 2.1.2) can be very difficult. Some prediction must be made if the cost reduction of the advantages will outweigh the costs of the disadvantages over a certain time.
2. **Analysis** – In this phase the problem domain is identified and domain knowledge is gathered. Gathering domain knowledge can come from various sources such as technical documentation, provided knowledge by domain experts and/or existing code. The result of the analysis consists basically of the domain-specific terminology and semantics in a more or less abstract form.
3. **Design** – Design a DSL which can specify applications/configurations in the problem domain. When designing a DSL it is good to look at other programming languages and reuse parts (like the notation) from them when it is possible.
4. **Implementation** – A decision must be made on how to implement the designed DSL. The main question which should be answered in this phase is "What is the most suitable way of implementing the DSL?".
5. **Deployment** – When the DSL is implemented it can be rolled-out to its intended end-users. In this phase probably a manual should be prepared or some training must be given to get the intended users familiar with the DSL. Also in this phase feedback can be captured from the users to improve the DSL implementation or design.

In the context of the thesis project we see that the decision of creating a new DSL is already made. The current ways of specifying *software systems configurations* for O2 in UML and XML are outdated and too complex and there are no other frameworks which overcome all the shortcomings mentioned in the introduction. Current technology, DSL tool support, knowledge and experiences about DSLs allow for the creation of much better DSLs, which are more efficient to work with and maintain. The advantages of a new DSL probably outweighs most disadvantages of the current way of developing systems with O2. How the other four phases are applied in the thesis project is discussed in section 3.2.

2.1.4 Frameworks for Implementing DSLs

Implementing a DSL completely manually is very hard and costs a lot of time. Luckily there are several well developed frameworks/tools out there for creating DSLs. In this section the better known frameworks for creating DSLs are briefly discussed.

Eclipse Modeling Framework (EMF) is an open-source modelling framework and code generation facility for building tools and other applications based on structured models. The core of EMF is a meta model (Ecore) for describing models and provides runtime support such as change notifications and persistence. EMF promotes interoperability between different Eclipse plug-ins for code generation, model transformations and development of textual and graphical DSLs [17]. EMF is chosen as framework for implementing the DSL discussed in this thesis report and is more extensity discussed in section 2.3.

MetaEdit+ is a commercial tool for creating and using graphical DSLs. MetaEdit+ is originally based on the concept that all CASE tools are the same: you can put objects on a diagram, fill in their properties, connect them with relationships, and move them around. The MetaEdit+ toolset includes generic CASE behaviour for objects and relationships, including a Diagram Editor, Object and Graph Browsers, and property dialogs [18]. MetaEdit+ is widely used and would be considered as the most easiest way of making (standalone) graphical DSLs with it [18] [19].

Spoofax is a platform for developing textual domain-specific languages with full-featured Eclipse editor plugins. With Spoofax you can write the textual grammar of your language using the high-level SDF grammar formalism. Based on this grammar, basic editor services such as syntax highlighting and code folding are automatically provided. Using high-level descriptor languages, these services can be customized. More sophisticated services such as error marking and auto completion can be specified using rewrite rules in the Stratego language [20]. The latest version of Spoofax was released in 2013, after that release the community behind it died and so it not advisable to Spoofax to create any new DSL.

Modeling SDK for Visual Studio (MSDK) enables to create model-based development tools that you can integrate into Visual Studio (VS) of Microsoft. As an example the UML tools of VS are created with MSDK. The heart of MSDK is the definition of a model that you create to represent concepts in your business area/domain. MSDK provide a variety of tools to implement a model with a textual grammar and graphical notation [21]. In a lot of concepts and patterns used in MSDK look much like ones used in EMF, but then focused on the environment of Microsoft's own products. In the company where is project was performance high performance systems (based on Linux) must be realized, which is something that cannot be done on the large OS of Microsoft (Windows). Because of this MSDK is not an option for the DSL implementation presented in this report.

2.2 O2 Framework

O2 is a generic middleware and service framework designed by Thales Nederland and is used in their (radar) products. With O2 it is possible to specify software system configurations for distributed real-time computer systems. Software system configurations in O2 are expressed in XML configuration files or in UML diagrams and are called O2 system models (or in short XML models or UML models). From these systems models O2 can generate code and deployment setting files for a target system. Code can be generated for different programming languages, including C, Java and MATLAB. The generated code includes so called containers with stubs for yet-to-be-developed code. These stubs can later be implemented with desired applications, algorithms or subsystems. The generated container code handles all the communication with other containers in the system using the O2 communication & management framework. The O2 framework can be divided in the following four components:

- **O2 UML** – A set of graphical representations defined in UML profiles for specifying O2 system models and transformation rules for converting O2 system models defined in O2 UML to O2 XML.
- **O2 XML** – A set of textual representations defined in XML Schema Definitions (XSDs) for specifying O2 system models in XML. These XSDs specify what kind of XML elements are allowed to specify in O2 system models in XML.
- **Code Producer Factory** – A code generator which parses and validates XML models and generates code from these models when they are valid. The code producer factory uses code generation templates to generate actual code.
- **Communication & Management Framework** – A framework for handling communication between and management of generated software components. The code producer factory generates the code of these software components that make use of the communication & management framework.

2.2.1 Workflow

The main workflow of creating software systems with O2 is shown in Figure 2.2. Creating a software system begins with the software system designer. The software system designer is responsible for specifying O2 system models in either UML of XML. In these models the system configuration of the target system is defined. Elements defined by a software system designer in an O2 system model are message types, software components, software systems, hardware components, hardware systems and deployment settings. These elements are in more detail explained in section 2.2.2.

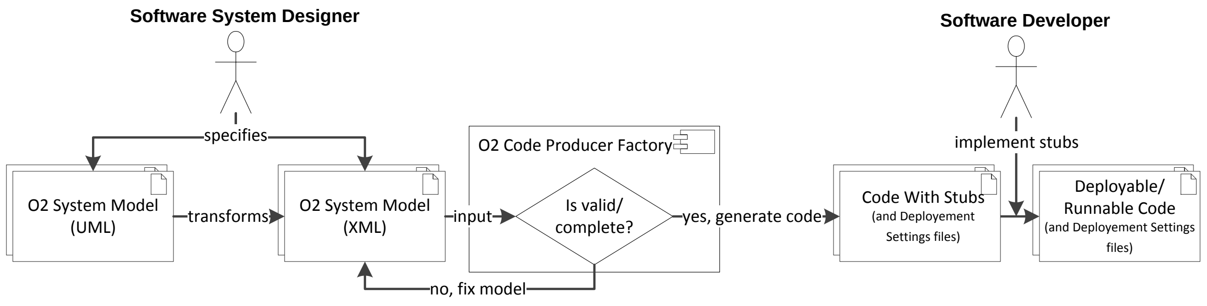


Figure 2.2: The main workflow with O2 for creating software systems.

Complete O2 system models in UML can be transformed in XML models. These XML models can be provided to the code producer factory of O2. The code producer factory validates the O2 system model for correctness and completeness. When the O2 system model is valid and complete, the O2 code producer factory will produce code with stubs and deployment setting files. When the provided O2 system model is not valid (e.g. incomplete) the O2 code producer factory will return an error, meaning that the model must be fixed by the software system designer. After fixing the model it can once more provided to the O2 code producer factory. This a trail-and-error trip that must be performed for every incorrectness in the O2 system model, which can be very time-consuming.

Stubs are places in the code for yet-to-be-developed code. These stubs must be implemented by a *software developer* with the desired algorithms, applications or sub systems. When all stubs are implemented the software system implementation is complete and can it be deployed to target computers.

2.2.2 O2 System Models

O2 XML and O2 UML both use the same concepts/elements for defining software system configurations. These elements can be defined in the following groups:

- **Software** – Used for defining software components, software templates and management interfaces.
- **Software System** – Used for composing software components to form a software system.
- **Hardware** – Used for defining hardware components/boards and hardware templates.
- **Hardware System** – Used for composing hardware components for defining the hardware system.
- **Mapping** – Used for defining routing of messages between software components, how software systems must be mapped/deployed onto hardware systems and how thread scheduling of each software component instance should be handled.
- **Technology** – Used for defining a collection of technologies (like protocols and programming languages) and templates. This specifies the technology space which O2 supports. O2 comes with its own set of supported technologies and templates which can be extended.
- **Message** – Used for defining message types used as format for sending messages between software components.

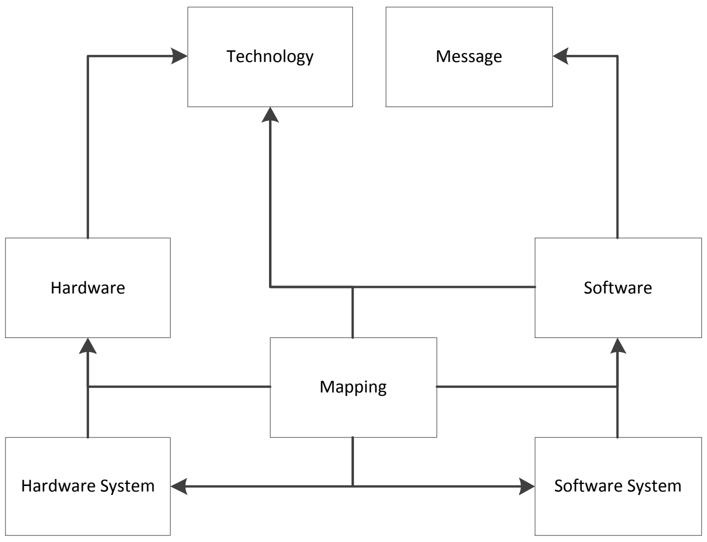


Figure 2.3: Relations/dependencies of element groups of O2.

Elements of these groups have different relations/dependencies between them, which are shown in Figure 2.3. In this figure you can see that elements of the Software group depend on elements of the

Message and Technology groups. A metamodel/domain model for O2 is presented in Appendix A, showing in detail the elements of each group including their dependencies. The DSL implementation discussed in this report is largely based on this metamodel of O2.

2.2.3 O2 XML And O2 UML

O2 XML is the original way to define software system configurations with O2. Configurations are mainly defined in XML files with O2 XML (messages are defined in XSD files as complex and simple types) and must conform to the XSDs of O2 XML. These simple and complex types can be used in the other As mentioned in the introduction is XML a verbose language which makes analysing and writing software system configurations with O2 XML hard. O2 UML is developed to tackle these problems. O2 UML consist of different UML diagrams that are extended with stereotypes. In Table 2.1 an example of a message defined with O2 XML and O2 UML is shown. In this table is shown that a message in O2 XML is defined as a `complexType` with a `name` and a `sequence` with three elements; **`a`**, **`b`** and **`c`** of type **`float`**. In O2 UML a message is defined as an interface with a name and the stereotype «`message`» with a set of attributes; `a`, `b` and `c` of type `float`.

O2 XML	O2 UML

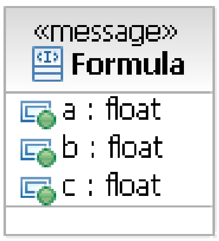
`<xs:complexType name="Formula">`
`    <xs:sequence>`
`            <xs:element name="a" type="xs:float"/>`
`            <xs:element name="b" type="xs:float"/>`
`            <xs:element name="c" type="xs:float"/>`
`    </xs:sequence>`
`</xs:complexType>`
	

Table 2.1: Left a message defined in O2 XML and on the right one defined in O2 UML.

O2 UML improves understanding and analysing of large software system configurations compared to O2 XML with graphical views. This is because graphical properties, like position and shape, can be used to directly see what a specific element means. Also can these properties be used to create a better overview for a set of elements in the configurations. This improves the way of analysing configurations, however it does not mean that graphical views are also faster in defining software system configurations. Compared to O2 XML, is defining software system configurations with O2 UML not much faster. This because that some elements of configurations can be faster defined textually than graphically. For example, defining a simple algorithm with UML diagrams will cost more time than defining it directly in an OO programming language. This is because you need multiple kinds of UML diagrams to express an algorithm, while with a programming language an algorithm can just be defined in with a few lines of code. Also cost mouse actions for creating graphical elements is most case more time than just typing some small text/code. So basically O2 UML only improved the analysability of (large) software system configurations compared to O2 XML.

2.3 Eclipse Modeling Framework (EMF)

Eclipse Modeling Framework (EMF) is a modelling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XML Metadata Interchange (XMI - the default language for models in EMF), EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor [17]. During the thesis project EMF is used for implementing the DSL, based on what requirements EMF is chosen as framework for the DSL implementation is discussed in section 3.1. In the following subsections the important concepts of EMF are discussed including the Eclipse plug-ins that are use with it for implementing the DSL.

2.3.1 Ecore Metamodel and Genmodel

Ecore is the metamodel provided by EMF for defining other more domain-specific metamodels [17] and because of this it is also referred as a meta-metamodel [22]. Ecore shares a lot of similarities with the class diagram of UML, however Ecore only consists of 17 components [23] and can basically be seen as a simplified subset of the UML class diagram. This makes Ecore a small model that is easy to understand and working with [24].

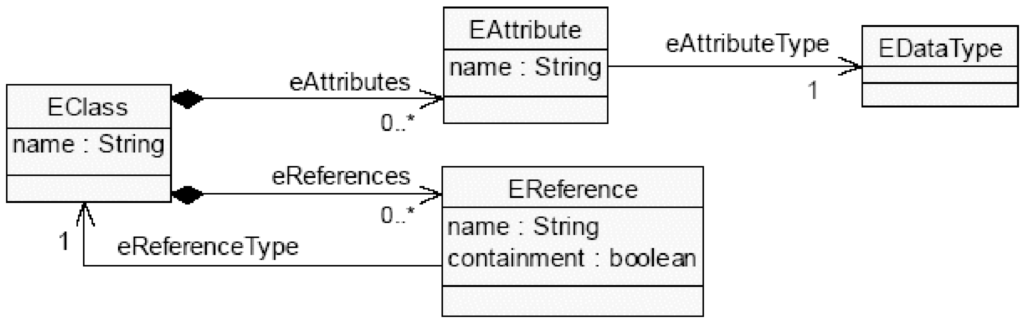


Figure 2.4: A simplified subset of the Ecore metamodel taken from [24].

The four most used components of Ecore are shown in Figure 2.4 and are briefly described below:

- **EClass** – is used to represent a modelled class. It has a name, zero or more attributes, and zero or more references.
- **EAttribute** – is used to represent a modelled attribute. Attributes have a name and a type.
- **EReference** – is used to represent one end of an association between classes. It has a name, a boolean flag to indicate if it represents containment, and a reference (target) type, which is another class.
- **EDataType** – is used to represent the type of an attribute. A data type can be a primitive type like int or float or an object type like java.util.Date.

A metamodel defined with Ecore is captured in an Ecore file (`.ecore`), which is basically a XML file that conforms to the XSD of Ecore. Writing an Ecore file manually is very hard and not advisable to do, it is better to use the Sample Ecore Model Editor directly provided by EMF or the Ecore Diagram Editor provided by EcoreTools [25] (an Eclipse plug-in). Both these editors are graphical-based and enable easy creation of Ecore-based metamodels, however the Ecore diagrams of EcoreTools give a much better overview of the structure of metamodels then the tree-view in the Sample Ecore Model Editor and therefore is it seen as the better tool for defining Ecore-based metamodels.

To give a better view of how an Ecore-based metamodel looks like, one is given as Ecore diagram in Figure 2.5. This metamodel can be used for managing bowling languages and tournaments. It consists of five EClasses (instances of EClass), League, Player, Tournament, Matchup and Game. Some of these EClasses have EAttributes, for example Player got four EAttributes with each an EAttribute name and EAttribute EDataType. The arrows between EClasses are EReferences and they come in two flavours; containment references (the ones with diamond shapes) and non-containment/normal references (the ones without diamonds shapes). Also the multiplicity of every reference is shown, for example a Matchup consists of two Games. Besides the five EClasses there is also the TournamentType EEnum (marked with <<enumeration>>), which is actually a special EDataType. This TournamentType consists of two EEnumLiterals, Pro and Amateur.

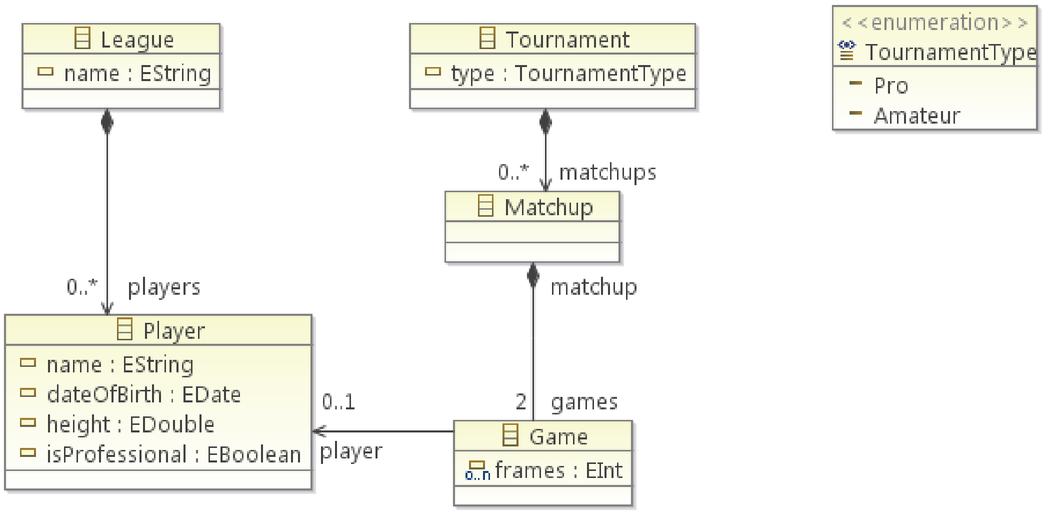


Figure 2.5: An Ecore-based metamodel for managing bowling leagues and tournaments taken from [26].

In the DSL implementation an Ecore-based metamodel is used to capture the configuration elements of a software system configuration. This metamodel is not manually created (with one of the explained editors), but generated using Xtext (see 2.3.2). EcoreTools is however used to make Ecore diagrams to visualize and explaining different parts of the DSL implementation in this report.

EMF comes with the support for generating a simple tree-view editor for Ecore-based metamodels. With this editor instances can made in XMI format that conform to the metamodel. This tree-view editor is a very basic implementation for a DSL and can be useful for small models, however for the desired DSL implementation more advanced features are needed, like a custom textual grammar, a multi-view graphical-based grammar, an editor with computer-aided features and so on. EMF can luckily be extended with so called Eclipse Modeling Components (Eclipse plug-ins) that enabling more advanced features. The Eclipse plug-ins that are used in the actual DSL implementation are briefly described in the following subsections. The criteria why is chosen for the Eclipse plug-ins discussed here is explained in section 3.1.

2.3.2 Xtext

Xtext is an Eclipse framework for implementing textual DSLs. It is made for quick and easy implementation of languages, however more importantly it covers all aspects of a complete language infrastructure, starting from the parser, interpreter, code generator, up to a complete Eclipse IDE with syntax highlighting, code completion, error markers and other computer-aided features [27]. In this section is discussed how the write grammar definitions with Xtext and how scoping and validation can be customized in Xtext.

Writing Grammar Definitions With Xtext

Implementing a DSL with Xtext starts by defining a grammar in a Xtext file (`.xtext`) using the Xtext grammar editor. In this editor Xtext uses its own notation for defining grammars, which is an extended version (with cross-references) of the Extended Backus–Naur Form (EBNF) notation [28]. In Table 2.2 the symbols of this notation that Xtext uses for defining grammars are shown.

To give a better idea of how a grammar definitions in Xtext look like, one is given in Code Snippet 2.2. This grammar is accepts the instance Hello world Bye world. All grammar definitions in Xtext starts with a name and namespace, shown on the first line of the example. The name of the example grammar is `MyDsl` with the namespace `org.xtext.example.mydsl.MyDsl`. Behind the name and namespace of the grammar the word **`with`** is shown, which is used for referencing to another grammar for reusing elements/rules of that grammar. In the example the `org.eclipse.xtext.common.Terminals` grammar is used, which is the standard grammar provided by Xtext for defining other grammars and can be found at the end of Appendix B in this report.

Line 3 of Code Snippet 2.2 starts with the **`generate`** word and is used to indicate that Xtext must generate an Ecore-based metamodel for this grammar. The metamodel that should be generated gets the name and namespace Uniform Resource Locator (URL) followed by the **`generate`** word, which in this case are `myDsl` (for the name) and `http://www.xtext.org/example/mydsl/MyDsl` (for the namespace URL). The Ecore-based metamodel that should be generated is eventually used to capture instances of the grammar. Xtext can however also be used for defining a grammar for an existing Ecore-based metamodel, instead of generating one. This can be done by using the **`import`** word followed by the location of the Ecore file.

Usage	**Notation**
rule definition	:
rule assignment	=
alternation	|
terminal string	'...'
option	?
grouping	( )
one or more	+
multiple	*
cross reference	[...]
order negation	&

Table 2.2: Symbols of the notation Xtext uses for defining grammars.

A huge drawback of using an existing Ecore-based metamodel and not letting Xtext generating one is, that it introduces an extra place where manually modifications must happen. For example, when the used metamodel is modified also the grammar definition of that model must be modified, resulting in that on two place manually modifications must happen and eventually will result in higher maintenance costs. This drawback does not exist when one letting Xtext handle the generation of the metamodel, however the notation for defining grammars in Xtext is not expressive enough to define advanced features of Ecore, like default values and abstract EClasses. If these advanced features are needed, one can include them by creating a post processor for the metamodel generator of Xtext. This post processor can be created using a Java class implementing the interface `IXtext2EcorePostProcessor`.

```
01 grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals
02
03 generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"
04
05 Model:
06   greetings+=Greeting*
07   farewells+=Farewell*;
08
09 Greeting:
10   'Hello' name=ID honest?='!'?;
11
12 Farewell:
13   'Bye' greeting=[Greeting];
```

Code Snippet 2.2: A simple grammar definition in Xtext taken from [29].

On the lines 5, 9 and 12 of Code Snippet 2.2 three grammar rules are defined. The first rule, `Model` is the root rule, the starting point of the grammar. Below it, on lines 6 and 7 the definition of this rule is shown, `greetings+=Greeting*` and `farewells+=Farewell*`. The definition for line 6 basically means that a `Model` can consists of multiple greeting rules and for line 7 basically means that multiple of greetings can be followed by multiple farewell rules. The greeting rule is the second rule of grammar and stated that a `Greeting` consist of a `Hello` string followed by a `name` that is an `ID` (a terminal that consist of letters and numbers used to identify an element - which is defined in the `org.eclipse.xtext.common.Terminals` grammar, the one provided by Xtext) and optionally ending with an exclamation mark (`!`). The `?=` assignment after `honest` is a binary assignment meaning that when a greeting ends with an exclamation mark that the greeting is honest and otherwise it is just a normal greeting. The last rule is the farewell rule and stated that a `Farewell` begins with a `Bye` string followed by a cross-reference to a greeting. The full syntax of a cross-reference is `[TypeName|RuleCall]` where `RuleCall` defaults to `ID`. This mean that for the farewell rule a `Bye` must be followed by the `name` of an earlier defined greeting. With this small grammar Xtext can generate a full language infrastructure, including an editor which accepts the following instances:

- Hello world
- Hello world! Bye world
- Hello world! Hello universe Bye universe

However not:

• hello world!	(missing capitol letter H of Hello)
• Bye nobody	(missing greeting with a name nobody)
• Hello world! Bye world Hello universe!	(farewells must come after all greetings)

Scoping and Validation Rules

Xtext standardly generates a language infrastructure with basic restrictions derived from the grammar definition. Xtext is very customizable and allows one the implement more advanced restrictions and/or changing the basic restrictions. There are basically two kinds of restrictions in Xtext. The first one is restricting the scope (also known as visibility) of name resolving for references. In Code Snippet 2.2 this kind of restriction could be that farewells can only be instantiated for honest greetings. For this restriction the scope of the cross-reference of greeting of farewell must met restricted. This restriction can be implemented by creating a Java class extending `AbstractDeclarativeScopeProvider` and adding the method as shown in Code Snippet 2.3. This method is defined in Xtend [30], a multi-functional language provided by Xtext and more briefly discussed in section 2.3.5. The signature of the method consist of a return type `IScope` with the method name `scope_Farewell_greeting` followed by two parameters, `farewell` of type `Farewell` and `eReference` of type `EReference`. In the body of this method the scope restriction is defined by calling the (static) method `scopeFor` of class `Scopes` and getting all greetings that are honest using the `filter` method. With this code now only farewells can be defined for greetings that are honest (the greetings ending with an exclamation mark).

```
def IScope scope_Farewell_greeting(Farewell farewell, EReference eReference) {
     Scopes::scopeFor((farewell.eContainer as Model).greetings.filter [ honest ])
}
```

Code Snippet 2.3: A scope restriction in Xtext defined with Xtend.

The second kind of restrictions that can be made in Xtext are validation rules. With these validations rules, domain specific constraints of the language can be statically checked. An example of a validation rule can be that, every name of a greeting must start with a capitol letter. To implement this validation rule one must create a Java class that extends `AbstractDeclarativeValidator` and add the method as shown in Code Snippet 2.4 to it. A validation rule method must be marked with the annotation `@Check`, so that Xtext knows that this method must be used for validation. The body of the method in the code snippet simply takes the first character of the name of a greeting and checks if its uppercase, if it is not uppercase than a warning will be created. Besides warnings, also error can info level notifications can be raised by a validation rule.

```
@Check
def void checkNameStartsWithCapital(Greeting greeting) {
    if (!Character::isUpperCase(greeting.getName().charAt(0))) {
        warning("Name should start with a capital");
    }
}
```
Code Snippet 2.4: A validation check in Xtext defined with Xtend.

2.3.3 Sirius

Sirius is an Eclipse project which allows you to create your own graphical modeling workbench by leveraging the Eclipse Modeling technologies. It is built upon EMF and Graphical Modeling Framework (GMF) [31] (see Figure 2.6) and provides a generic workbench for model-based architecture engineering that could be easily tailored to fit specific needs. Based on a viewpoint approach, Sirius makes it possible to equip teams who have to deal with complex architectures on specific domains [32].
Sirius is pretty new in the EMF eco-system, however it is already widely adopted in different kinds of tools/products like, EcoreTools [25], UML Designer [33], Mobile Multimodality Creator (MIMIC) [34] and Capella [35].

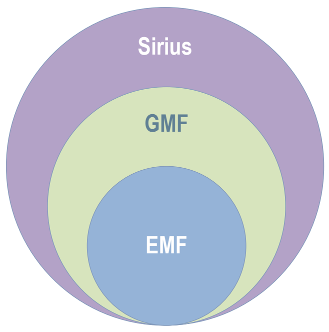


Figure 2.6: Hierarchy of Sirius.

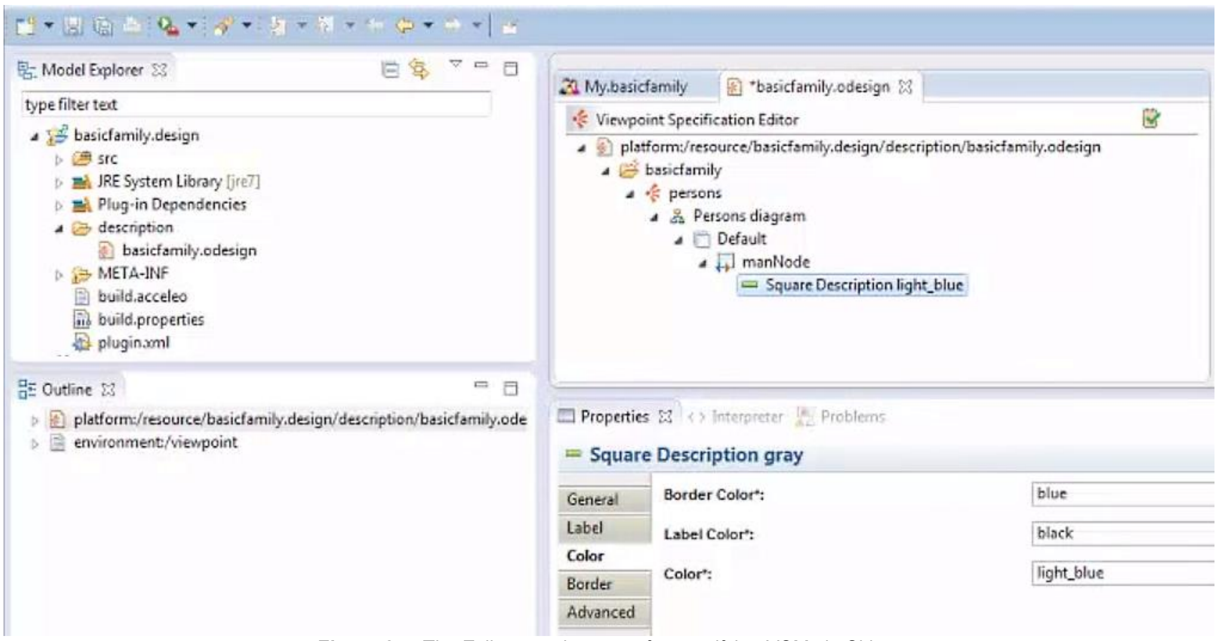
Sirius provides three different kinds of dialects (kinds of representations), diagrams, tables and trees, to create your own graphical views. These dialects uses the same core principles and concepts of Sirius, like viewpoint, mapping, style and tool, briefly discussed below:

- **Viewpoint** – A core element of Sirius containing custom representation specifications and specifies what file extensions of models these representations are associated to. For example, in the UML Designer the extension is `.uml` the representation specifications are the different diagrams of UML.
- **Mapping** – Identifies a (sub-)set of the semantic model elements that should appear in a representation and indicates how they should be represented. Each dialect offers different kinds of mappings. For example, diagrams provide container mappings, which can be used to represent semantic elements with graphical containers, where other elements (specified by other mappings) can appear.
- **Style** – Each mapping can have several styles associated to it, which are used to configure the visual appearance of the elements they represent. For example, for diagram elements this would include their shape and colors.
- **Tool** – Without tools, representations are only visualizations without any edition capabilities. While this can be sometimes useful, most representations will allow users to create, edit and delete elements. Tools are associated to mappings to describe their behaviors. Each kind of user interaction supported by Sirius (e.g. creation or deletion, label editing, edge reconnection...) is specified by a different kind of tool, but they all follow the same principles in their definition.

In this repot mainly the diagram dialect of Sirius is discussed. This is not for the fact that it is the most used dialect of Sirius in existing tools, but because it is the only used dialect in the DSL implementation.

Specifying a Diagram Editor

Specifying any editor with Sirius starts with a viewpoint specification project (VPS), which is basically an Eclipse (plug-in) project and a container for all specification elements and code extensions of Sirius. By convention this project should be suffixed with `.design`. A VPS contains one or more viewpoint specification models (VSMs) and are store in `.odesign` files. Editing a VSM is done by using the viewpoint specification editor (VSE), a tree view editor, and the properties pane in Eclipse as showed in Figure 2.7. In the Eclipse environment of this figure we see the VPS (in this example with the name `basicfamiliy.design`) in the Model Explorer on the top left, the VSE for the VSM in the top right and the properties pane at the bottom right. The VSE is used to add, remove or organize any element to the VSM and the properties pane is used to edit properties of these elements in the VSM.


Model Explorer
type filter text
basicfamily.design
src
JRE System Library [jre7]
Plug-in Dependencies
description
basicfamily.odesign
META-INF
build.acceleo
build.properties
plugin.xml

Outline
platform:/resource/basicfamily.design/description/basicfamily.ode
environment:/viewpoint

My.basicfamily
*basicfamily.odesign
Viewpoint Specification Editor
platform:/resource/basicfamily.design/description/basicfamily.odesign
basicfamily
persons
Persons diagram
Default
manNode
Square Description light_blue

Properties
Interpreter
Problems
Square Description gray

General	Border Color*:	blue
Label	Label Color*:	black
Color	Color*:	light_blue
Border		
Advanced		

Figure 2.7: The Eclipse environment for specifying VSMs in Sirius.

A VSM contains one or more viewpoints, which are visualized with the -icon in the VSE. Every viewpoint contains one or more representations defined in one of the provided dialects. A representation defined with the diagrams dialect (called a diagram description) is indicated with the -icon in the VSE. A diagram description consist of always a default layer, but can have multiple additional layers. A layer contains different graphical elements of the diagram, including mappings, tools and styles. Layers can be enabled or disabled (expect the default layer, which is always enabled) by the end-user (who will work with the diagrams) to show or hide different concerns/information in the diagram.

An example with two different layers of a diagram is given in Figure 2.9. In this example we see two views of the same diagram with different layers enabled. The view of the diagram at the top shows only the diagram with the default layer enabled and the one at the bottom shows the same diagram but with both the default and temperatures layers enabled. With only the default layer enabled we see only the structure of the two presented example units and their internals, however when the temperature layer is enabled we see the same structures of the units but also the temperature information about them. Also the temperature layer include extra internals like cooling elements, like fans with their speeds (RPM). One can imagine that defining the structure of a unit in this diagram is much easier and cleaner with the default layer only enabled, but for handling the specialized concerns with temperature it can be better done with the temperature layer enabled. Separating concerns into the right layers can make diagram usable for multiple disciplines.

As mentioned earlier a layer includes mappings. Mappings specify what graphical elements of the diagram description are associated to elements of the Ecore-based model behind the diagram. Graphical elements that can be mapped come in three flavours; nodes, containers and edges. Nodes are solid elements in the diagram and can have different styles/forms like, squire, note, image and custom graphic. Containers are used for encapsulating other graphical elements, like nodes and edges. Containers come in two forms; the first one, is a normal container in which elements can be freely ordered. The second one, is a list container in which graphical elements are ordered in a list. Edges are connectors between source and target elements, like nodes and containers. Edges come in three styles; straight, Manhattan (draw edges only using vertical and horizontal segments) and tree.

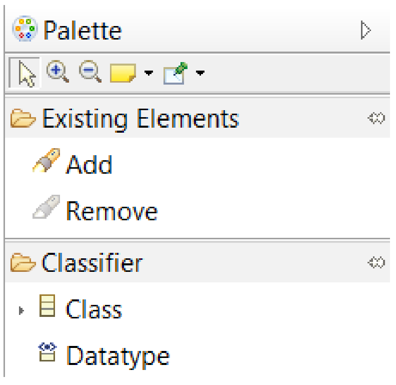


Figure 2.8: The palette with tools of EcoreTools.

Besides mappings a layer can also include tools. Tools are used to add, modify of remove graphical elements in the diagram. Tools come in a lot of flavours, some of them are; node creation, container creation, edge creation, reconnect edge, delete element, edit label and selection wizard. Most tools elements will be showed as a button in the palette (toolbox) of the diagram. This palette is standardly showed on the right side of the diagram editor. A subset of such a palette is showed in Figure 2.8.

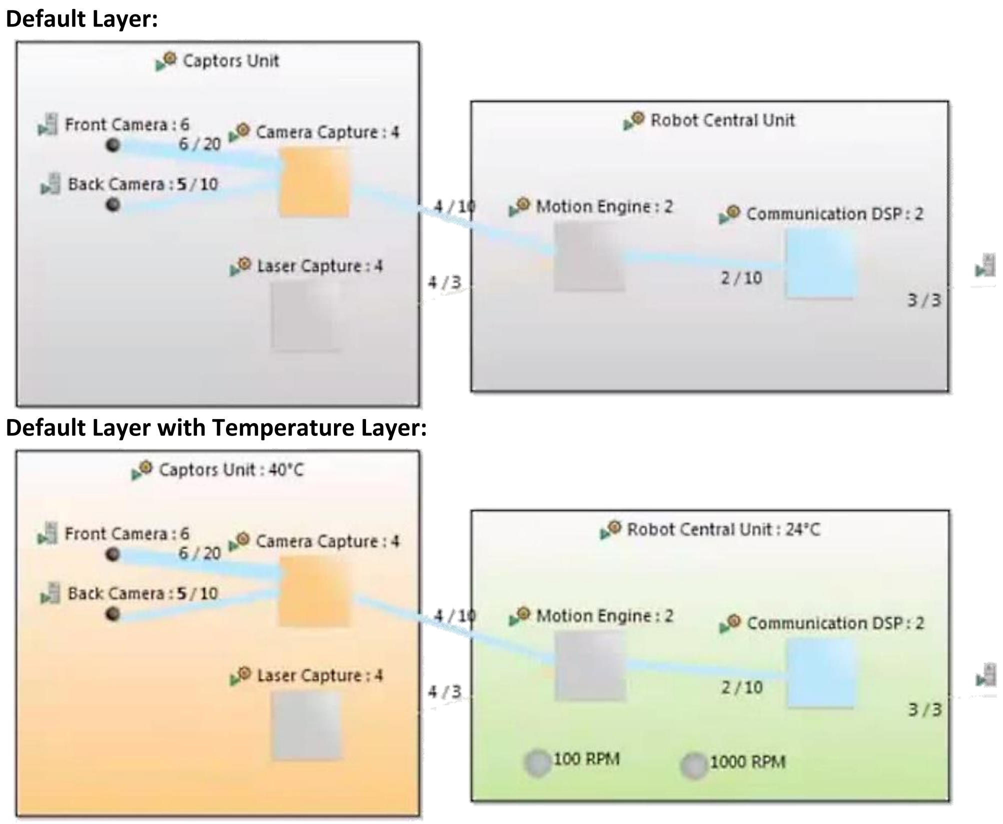


Figure 2.9: An example of a diagram with two layers; the top one without the temperature layer enablable and the bottom one with the temperature layer enabled.

2.3.4 ATL

ATL Transformation Language (ATL) is a language designed by AtlanMod research group and is used to specify a way to produce target models from a set of source models. This kind of transformation is also called a model-to-model (M2M) transformation [36]. In this (sub-) section the transformation approach of ATL is discussed and how to define transformation rules with ATL.

ATL Transformation Approach

In Figure 2.10 an overview of the transformation approach of ATL is showed. At the top of this overview we see a meta-metamodel depictured, which is in this case of the DSL implementation; Ecore. Below the meta-metamodel two metamodels, A and B, are depictured with ATL between them. ATL and both metamodels A and B are languages/metamodels that conforms to the meta-metamodel. At the bottom of the overview we see two models, A1 and B1, and transformation rules. Model A1 conforms to metamodel A and model B1 conforms to metamodel B. The transformation rules conforms to ATL and define how a transformation must happen from models that conforms to metamodel A (source) to models that conforms to metamodel B (target). By executing the transformation rules with the ATL virtual machine/execution environment on model A1, results into the model B1. In this overview only an example is given for a transformation of a model that conforms to one source metamodel into a model that conforms to a target metamodel, however with ATL it is also possible to define transformations for models of multiple source metamodels into models of multiple target metamodels. With this approach of ATL it is possible to do simple and advanced M2M transformations.

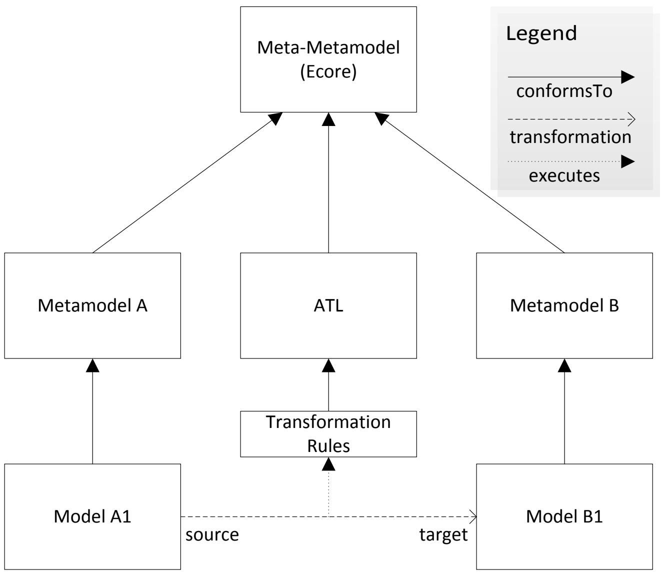


Figure 2.10: Overview of ATL transformation approach.

Defining Transformation Rules

As earlier mentioned the transformations rules define how a transformation with ATL must happen. Transformation rules for ATL are stored in `.atl` files and can easily be edited with the Eclipse IDE for ATL. In this section we explain briefly how transformation rules are defined in ATL using the example “Families to Persons” from [37]. In this example an ATL transformation is defined for transforming models conforming to the families meta-model into models conforming to the persons metamodel. The families metamodel is used to capture people as members of families and the persons metamodel for capturing people as male or female persons. In Figure 2.11 is an overview given of how the ATL transformation works with the families to persons example.

A subset of the Families2Persons.atl file, pictured at the bottom middle of Figure 2.11, is showed in Code Snippet 2.5. In this subset the basic elements for defining transformations with ATL are showed and in this paragraph and the coming ones these elements are explained. Starting with line 1 and line 2 on which the paths/locations of the target and source metamodels of the transformation are defined, in this case that are the families (line 1) and persons (line 2) Ecore-based metamodels. On line 4 the name of the ATL module is defined. In a module the transformation direction must always be defined, which is showed in the code snippet on line 5. The transformation direction of this module is to create persons models (OUT) from families models (IN).

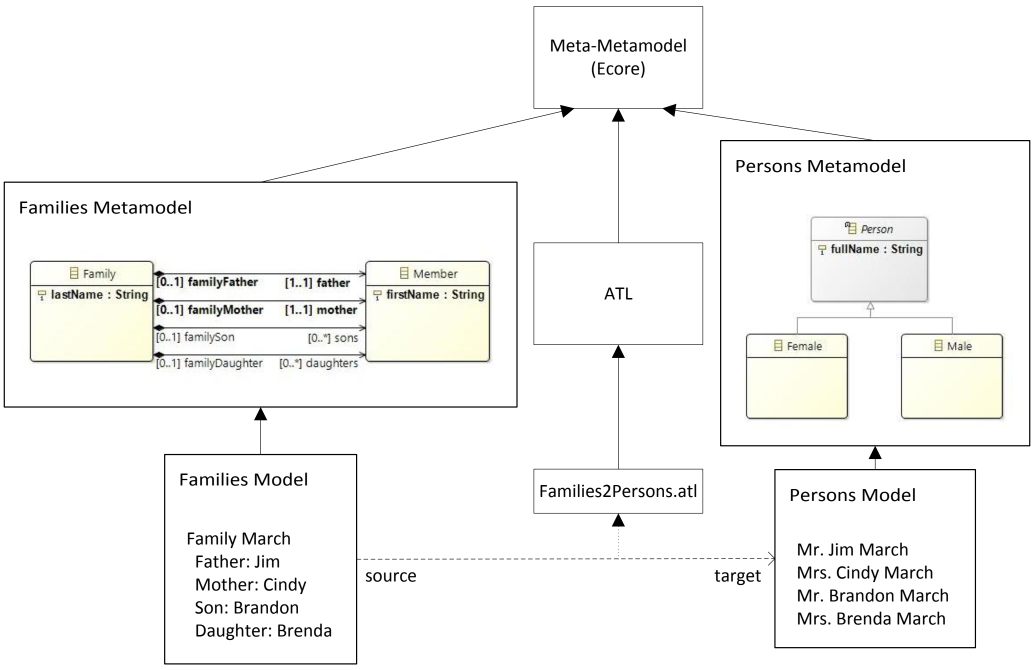


Figure 2.11: Overview of the families to persons transformation with ATL.

Besides the defining the transformation direction in a module, it is also used as container for the transformation rules and helper functions of the transformation. On line 7 until 11 is a transformation rule defined with the name Member2Male. As the name suggests this rule is used to transformation male members of a families model into male persons of a person model. What this rule specifies is showed on line 8 until 11, starting with that from the families model all members are selected that are not female. For every member in this selection than a male person is created in the target persons models. The rule presented is this subset is a normal rule that always be executed, however there are also so called lazy rules which are rule that only executed when they are called from another rule. Both normal and lazy transformation rules are used in the DSL implementation.

On line 9 in the rule Member2Male is the helper function isFemale() used. The definition of this helper function is showed in lines 13 until 19. Helper functions are used to make definition of rules simpler. In this case the helper function is used to check if a member is either a mother or daughter. The advantage of using this helper function in the code snippet is now that only one rule for creating male persons have to defined instead of two rules, one for father members to male persons and the other one for son members to male persons.

```
-- @path Families=/Families2Persons/Families.ecore
-- @path Persons=/Families2Persons/Persons.ecore

module Families2Persons;
create OUT : Persons from IN : Families;

rule Member2Male {
    from
            s : Families!Member (not s.isFemale())
    to
            t : Persons!Male (fullName <- s.firstName + ' ' + s.familyName) }

helper context Families!Member def: isFemale() : Boolean =
    if not self.familyMother.oclIsUndefined() then true
    else
            if not self.familyDaughter.oclIsUndefined() then true
            else false
            endif
    endif;
```
Code Snippet 2.5: A subset of the Families2Persons.atl file.

2.3.5 Xtend

Xtend is a flexible and expressive dialect of Java, which compiles into Java compatible source code. Xtend can use any existing Java library seamlessly and provides many improvements compared to plain Java, like extension methods, lambda expressions, operator overloading and more [30]. Xtend comes along by installing Xtext and is used in Xtext to defined different elements, like scopes and validations rules. In this section the two aspects of Xtend that are most used in Xtext, lambda expressions and template expressions, are discussed.

Lambda Expressions

A lambda expression is basically a piece of code wrapped into an object to pass it around. It is best to think of a lambda expression as an anonymous class with a single method. This kind of anonymous classes can be found everywhere in Java code of programs and is the poor-man's replacement for lambda expressions in Java [30]. An example of an anonymous class in Java is showed in Code Snippet 2.6.

```
final JTextField textField = new JTextField();
textField.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    textField.setText("Something happened!");
  }
});
```
Code Snippet 2.6: The poor-man's replacement of lambda expressions in plain Java.

Code Snippet 2.6 can be much shorter written in Xtend using lambda expressions as showed in Code Snippet 2.7. In this snippet we see that the anonymous class is replaced for a lambda expression. A lambda expression is defined between square brackets (`[]`) like in Smalltalk [38]. A lambda expression may declare parameters, in the code snippet the expression only has one parameter called `e` of type `ActionEvent`. This parameter can be used in the body of the lambda expression, the part behind the vertical line (`|`). In the body of the lambda expression the actual method calls and variable assignment can be declared. Parameters of lambda expressions can also be left out if they are not needed or if one does not want custom names for the parameters. Without naming the parameters they can be accessed using the variable `it` when one parameter is available and for more parameters by using the `it1, it2, …, itn` variables.

```
val textField = new JTextField
textField.addActionListener([ ActionEvent e |
    textField.text = "Something happened!"
])
```

Code Snippet 2.7: A lambda expression in Xtend.

In Code Snippet 2.7 we see also the properties aspect of Xtend. The get and set of property `text` of `textField` can used be to set the property by assigning it to a variable and can be get by just calling the property `text` of `textField`. Also no ending semicolons (`;`) are needed in Xtend. In Xtext lambda expressions are mostly used to filter objects from lists or sets in definitions for scopes or validations rules.

Template Expressions

Templates in Xtend allow for readable string concatenation and are defined in blocks surrounded by tripe quotes (`'''`). Expressions within the template can be accessed by surrounding them with guillemets («»). In Code Snippet 2.8 an example is given of a template expression. In the snippet we see a method (defined with **`def`** in Xtend) with the name `someHTML` and one parameter, `p` of type `Paragraph` (line 1). After this signature we see the starting triple quotes of the template expression block. In the template expression block we see that a simple HTML structure with a body is defined. In this body we see that there is an `IF` expression to check if `p.headLine` is not null and when true put `p.headLine` in a H1 block. On line 6 we see a simple expression that put `p.text` in a paragraph block.

```
1 def someHTML(Paragraph p) '''
2     <html>
3         <body>
4             «IF p.headLine != null»<h1>«p.headLine»</h1>«ENDIF»
5             <p>«p.text»</p>
6         </body>
8     </html> '''
```

Code Snippet 2.8: An example template expression in Xtend.

In Xtext template expression can be used to do model-to-text (M2T) transformations. Results of a template expression is a string and can be easily written to a file. In the DSL implementation template expressions are used to produce XSD files.

3 Method of Implementing The DSL

In this chapter, the architecture that was used for the implementation is described, as well as the approach that was taken for implementing the DSL and transformation rules.

3.1 Architecture Overview

As mentioned in 2.1.4, there are different frameworks for implementing DSLs. During the selecting of what framework is best suited for the DSL implementation, the following requirements were decided to be a minimum:

- **R1** The framework should have support for implementing the
  - **R1a** textual grammar of the DSL.
  - **R1b** graphical grammar of the DSL.
- **R2** The framework should support generation of some editor with computer-aided features for the
  - **R2a** textual grammar of the DSL, so that users can easily work with the textual part of the DSL.
  - **R2b** graphical grammar of the DSL, so that users can easily work with the graphical part of the DSL.
- **R3** The framework should support some kind of transformation of one grammar into another, including the transformation of one DSL into another DSL.

If we look at the available frameworks we see that only EMF satisfies all the above requirements, and so would be the best option for the DSL implementation. EMF can satisfy all these requirements due to fact that it comes with so called Eclipse Modeling Components, which are basically Eclipse plug-ins extending EMF with specific features. These plug-ins can be found and installed using the Eclipse Modeling Components Discovery catalogue [39]. In this discovery catalogue there are multiple Eclipse Modeling Components which can satisfy one or more of the above requirements, so an selection is made for choosing the most suited combination of plug-ins for the DSL implementation. Eventually four plugins where selected, the first one is Xtext [12] for handling the textual part of the DSL (R1a and R2a), the second one is Sirius [32] for handling the graphical part of the DSL (R1b and R2b), and the last two are ATL [36] and Xtend [30] for the transformations (R3). EMF with these four plug-ins covers all of the above mentioned requirements for implementing the DSL.

When we combine Xtext, Sirius, ATL and Xtend we get an architecture as shown in Figure 3.1. This is also the architecture that is used as solution to implement the actual DSL. On the left side the four artefacts, DSL metamodel, textual DSL part, graphical DSL part and transformation rules are shown. These are the artefacts that need to be created to implement the DSL. The DSL metamodel is a model to capture all elements of a software system configuration for O2 and can be seen as an abstract syntax tree of a programming language. This DSL metamodel conforms to the Ecore model, which is the (meta-)metamodel provided by EMF and every component of EMF is based around this. The textual part of the DSL is defined in Xtext and takes care of the textual grammar implementation including the textual editor with type checking, auto-completion, error highlighting and other computer-aided features. The graphical part of the DSL is defined in Sirius, handling the graphical grammar implementation including the graphical editor. The last artefact that realizes the DSL are the transformation rules that are defined in ATL and Xtend. In the architecture is chosen for transformation to O2 XML system models so that the original code generator of the O2 framework can be utilized and no new code generator has to been written.

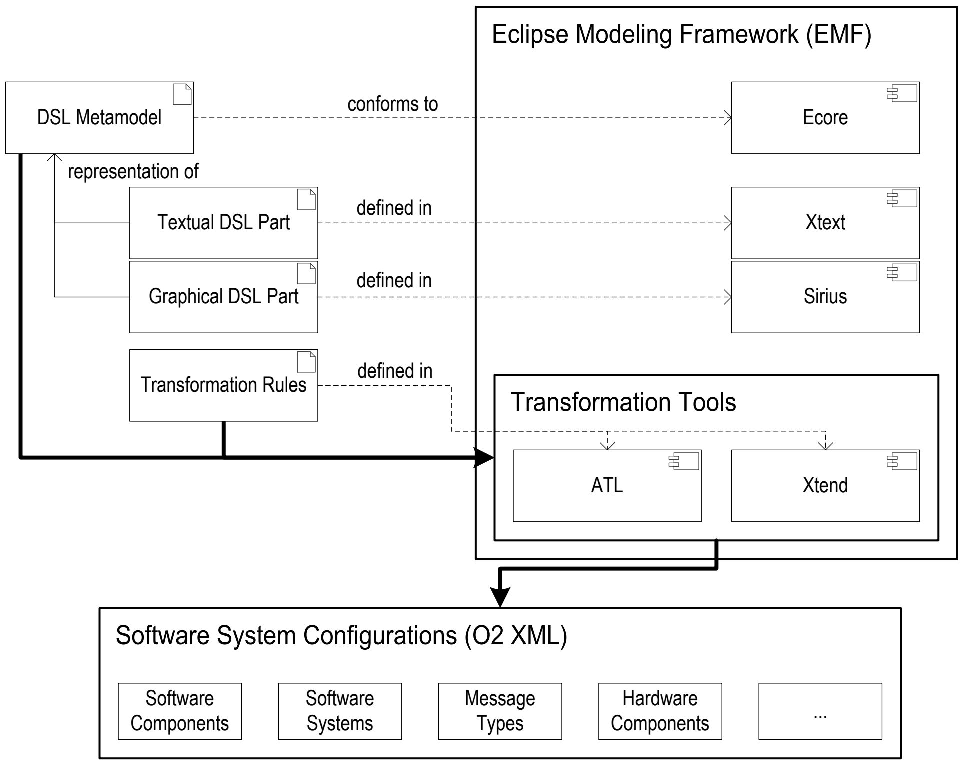


Figure 3.1: Architecture overview of the solution used to implement the DSL.

The workflow how to use the above pictured architecture is described with bold solid arrows. It starts at the DSL metamodel or more precise at a software system configuration that conforms to the DSL metamodel. The model of a software system configuration can be edited and viewed with both textual and graphical DSL parts. Both editors can be used for editing a model simultaneously, because Ecore on which the model indirectly is based supports change notifications. This means that any changes done to a model by an editor will be notified to the other editor. This way the editors can update their representation of the model and are the representations of both editors consistent with the actual model.

A complete model of a software system configuration along with the transformation rules (an earlier defined artefact of the DSL implementation) can be provided to the transformation tools, which transforms the model into a O2 XML system model. Note that the transformation rules are only specified once and can be reused for every transformation of a model that conforms to the DSL metamodel to a O2 XML system model. Also note that the transformation consists of two plug-ins, ATL and Xtend. ATL is used for the transformation to XML files of the O2 XML system model, however the transformation to message types, which are XSD files, are handled by Xtend. This is because there exists no good transformation model to the XSD format with ATL, so Xtend is used for simple model-to-text (m2t) transformation using self-specified templates. A nice addition is that Xtend is a component of the Xtext plug-in and so by installing Xtext directly Xtend can be utilized, this way introduction of more plug-ins is avoided.

3.1.1 Alternatives

EMF is the only framework which covers all minimum requirements for implementing the DSL, however EMF can also be used as enabler for connecting different frameworks with each other. For example, it would be possible to use MetaEdit+ for implementing the graphical parts of the DSL together with EMF or other frameworks for implementing the other parts of the DSL. MetaEdit+ is currently considered as the best framework for implementing (standalone) graphical DSLs [18] [19] and so the graphical part of the DSL could be faster implemented with it than with Sirius. So a serious alternative for the presented solution in Figure 3.1 is to use MetaEdit+ in combination with EMF, however for this combination extra code/transformations have to been written for MetaEdit+ to work with EMF. Most likely this extra code/transformations will take more time to realize, than the time that is gained to implement the graphical part of the DSL compared to implementing it with Sirius. Also this extra code/transformations must be maintained, resulting in besides the extra overall initial costs also in extra future maintenance costs. An extra advantage for choosing for Sirius instead of MetaEdit+ is that it is used in the company where this thesis project is performed and so experience about Sirius is already within the company. Summing everything up, we can conclude that using only EMF for implementing the DSL is the best option compared to every other framework or combinations of them.

Picking Xtext for the textual part of the DSL was easy, because it is the only plug-in provided in the discovery catalogue of EMF that covers both R1a and R2a. But for implementing the graphical part of the DSL and the transformations, there are more plug-ins in the discovery catalogue that cover the same minimum requirements as Sirius or ATL. First the alternatives for the graphical part of the DSL, there are two other plug-ins besides Sirius that both satisfy R1b and R2b; these are Graphical Modeling Framework (GMF) [40] and Graphiti [41]. Graphiti has a basic set of features which is useful when creating very simple graphical presentations, GMP on the other hand is very feature rich and used for more advanced graphical representations [42]. As we look at the graphical part of the DSL, we most likely are implementing graphical views with advanced features, so this is way Graphiti is not suited for the job [43]. If we compare GMF and Sirius we see that Sirius is the better option, because Sirius is built upon GMF and so got the same feature-rich set for implementing graphical representations, however Sirius is much easier to setup and work with than just plainly using GMF [44].

Secondly the alternative for implementing the transformations, which is QVT Operational [45]. Both ATL and QVT Operational do so called model-to-model (M2M) transformations and fulfil the same purpose, however ATL can be used for more advanced transformations than QVT Operational because of its declarative and imperative programming hybrid nature. Also ATL is in most cases faster in doing transformations and easier to define transformation rules with [46]. Because of these extra advantages of ATL compared to QVT Operational, ATL is chosen as best suited for the DSL implementation. Either way if QVT Operational or ATL was chosen both had to be combined with Xtend, due to that fact that there is no good transformation model available for doing transformations to the XSD format. Writing a transformation model for QVT Operational or ATL would cost a lot more time than implementing this small part of the complete transformation in Xtend.

In the selected architecture solution for the DSL implementation (Figure 3.1) is chosen for a transformation from models conforming to the DSL metamodel to O2 system models in XML format. From these O2 system models, system code can be produced by utilizing the existing code generator of O2, however system code can also be directly generated from models conforming to the DSL metamodel using EMF plug-ins. Code generation with EMF can either be realized with M2M transformations (ATL) using metamodels of the desired target (programming) languages (a lot of them can be found in the ZooFoundation of AtlanMod [47]) or with M2T transformations (Xtend) using code templates. When in fact both are viable options, realizing and testing a new code generator costs a lot of time. In the case of the DSL implementation much more time than making M2M (and partially M2T) transformations to O2 system models in XML format. The fastest solution is chosen so that a DSL implementation could be realized within the short time period of the project, that could handle the whole process of defining software system configurations until (indirectly) code generation. Within the same time period this is not possible to realize with the other two mentioned options.

Although there have to be noted that the existing code generator of O2 doing besides code generation also model validations, making any modification to it very complex. Because of this it is advisable when a decent-sized modification is needed in the future, to leave the existing one as is and create a complete new code generator using one of the mentioned options (with M2M or M2T transformations). With tooling nowadays a code generator can relatively fast realized, but more importantly is it much easier to maintain than the existing code generator. Additionally the model validations can be completely left over to the graphical and textual parts of the DSL implementation, this simplifies the code generators implementation drastically and improves the centralization of validations instead of having it in two places (in the existing code generator of O2 and the new DSL implementation).

3.2 Implementation Approach

In section 2.1.3 is discussed that the development of a DSL consists of five phases, decisions, analysis, design, implementation and deployment, along with how the decision phase is handled during the thesis project. How the other four phases are handled during the thesis project is shown in Figure 3.2, which is an iterative process looking much like a scrum process. The process that is used initially starts with the domain analysis (shown in the analysis phase on the left) to identify and understand the concepts of mainly O2, but also other component-based frameworks for distributed real-time systems. The results of the domain analysis are already briefly discussed in section 2.2.

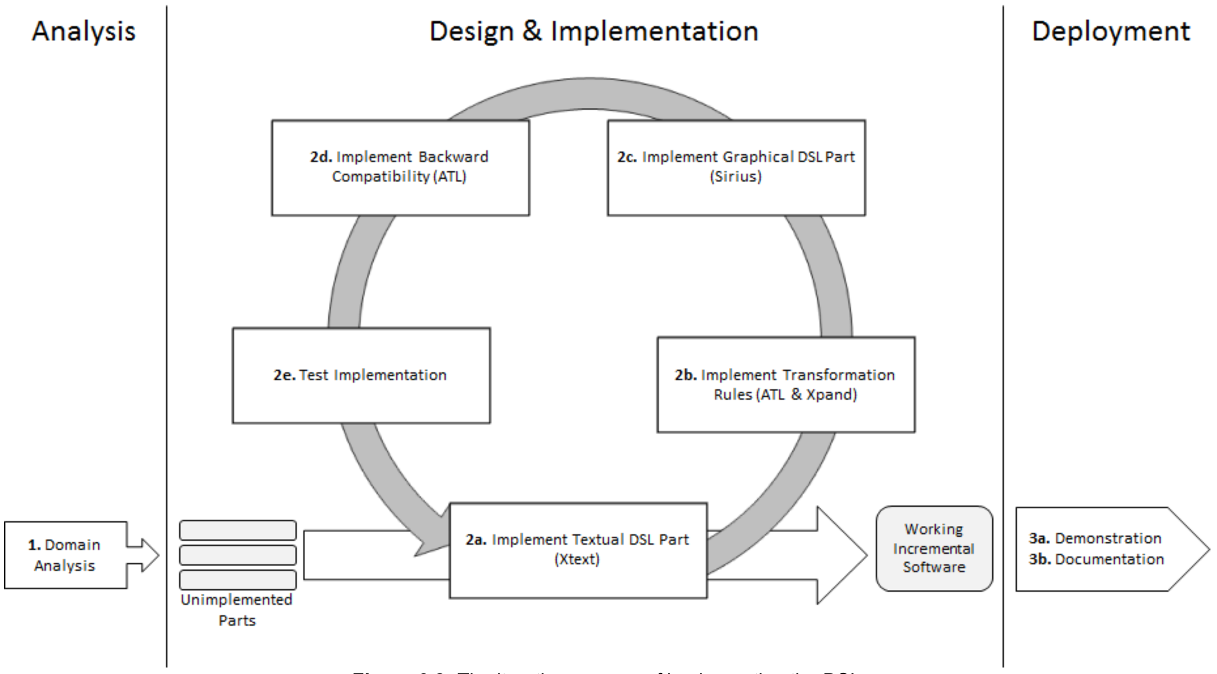

Analysis
Design & Implementation
Deployment
2d. Implement Backward Compatibility (ATL)
2c. Implement Graphical DSL Part (Sirius)
2e. Test Implementation
2b. Implement Transformation Rules (ATL & Xpand)
1. Domain Analysis
2a. Implement Textual DSL Part (Xtext)
Working Incremental Software
3a. Demonstration
3b. Documentation
Unimplemented Parts

Figure 3.2: The iterative process of implementing the DSL.

The second part (the middle part of Figure 3.2) is the iterative element of the process and handles the design and implementation phases. In this part the domain concepts are known and so a set can be defined of what parts should be implemented for the DSL. These parts start as unimplemented parts and from them every time one is picked and implemented following the steps of the iterative cycle (2a, 2b, 2c, 2d and 2e). The iterative cycle starts with implementing the textual part of the DSL in Xtext (2a), then realizing the transformation to O2 XML system models with ATL and/or Xtend (2b), after that implementing the graphical part in Sirius (2c), then the backward compatibility with ATL (2d) and eventually testing the implementation by creating various cases (2e). Completing this cycle for an unimplemented part results in working incremental software, which is a part of the whole software product of the DSL implementation. When all unimplemented parts have completed the iterative cycle, we eventually end up with a complete DSL implementation.

The last part of the process is the demonstration and documentation of the results that are realized in the design and implementation phases. The demonstration and documentation are part of the deployment phase and are used as information transfer of the thesis project to the end-users and the company where this project is executed. The demonstration used to show the end result of the project to the company and also used to get input for improvements and ideas for future work. The documentation will result in some kind of manual of how to work with the end result and how to extend/modify it.

In an early stage of implementing the DSL, it became clear that caring out all the iterative steps of the process for every unimplemented part would cost a lot of time, more time than is set for a thesis project. This resulted in that not all iterative steps could be carried out for every unimplemented part. Because of this a compromise was set that only the iterative steps 2a, 2b and 2e are mandatory for every unimplemented part and the other steps are marked as optional/extra. This way the end result would at least be a working textual DSL implementation with some graphical and backward compatibility features. The optional steps are less extensively implemented during the thesis process and also the documentation part (3b) is less extensively worked out, due to the time constraints of the thesis project.

4 Implementation of The DSL

As discussed in the previous chapter the implementation of the DSL consists of five parts; implementing textual DSL part, implementing graphical DSL part, implementing transformation rules, implementing backward compatibility and testing the implementation. In this chapter the implementation of these parts are discussed, focusing on the different features of the DSL. The implementation of the textual DSL part is discussed in 4.1, the implementation of the graphical DSL part in 4.2, the implementation of the transformation rules and backward compatibility in 4.3, and the implementation is tested using a case study discussed in 4.4.

4.1 *Textual DSL Part (Xtext)*

The textual part of the DSL consist of a grammar definition and an Eclipse IDE with computer-aided feature for it and are both implemented with Xtext. Also with Xtext the metamodel of the DSL is generated (from the grammar definition) and is used to capture software system configuration made with the DSL. The concepts and hierarchy used in this DSL are mainly derived from O2. This means that most elements in the DSL will be very familiar for users who worked with O2. In the following subsections the specific constructs of the textual part of the DSL, like the grammar definition and implementation of validation rules and scoping are discussed. In most of these subsections simplify parts of the grammar are used, the complete grammar of the DSL implementation can be found in Appendix C.

4.1.1 Basic Grammar

For the textual grammar of the DSL the following constraint is defined; users of O2 must easily understand and work with the language. This constraint comes from the company where the thesis project was performed, with in mind that O2 users within company should easily pick the language without extensive training of learning a new language. Eventually is chosen to use the Human-Usable Textual Notation (HUTN) [48] as base for the textual grammar. This notation uses a C-like (and also Java-like) bracket notation (`{ }`) for organizing language elements and class and package style like notation for language elements (`element <<element name>> { }`). Where C and Java are both focused on defining programs is HUTN focused on defining structured data, which software system configurations basically are. This makes HUTN well suited for defining software system configuration which are easy to write and read than their XML equivalent, but also make it more recognizable for O2 users because of the reuse of language structures of Java and C (which are commonly known by most O2 users).

```
Timer:
    'timer' name=ID '{'
          'mode:' mode=TIMERMODE
          'timeout:' timeout=INT
          ('autostart:' autostart?='true'|'false')?
    '}';
```

Code Snippet 4.1: The grammar definition for defining a timer of management interface.

In Code Snippet 4.1 the textual grammar definition of the DSL for defining timers shown(see section 2.3.2 how to read these grammars). This snippet is used to show how the HUTN style is applied in the DSL. Every element starts with the element type, in this case `timer`, followed by the `name` of the element, and then the opening bracket (`{`) followed by the attributes of the element and eventually a closing bracket (`}`). In this code snippet we see that the timer has two mandatory attributes, `mode` and `timeout`, and one optional attribute `autostart`. This is the basic grammar of how elements are structured in the DSL.

In Code Snippet 4.2 is shown how a timer is defined in XML as well with the DSL implementation. This small snippet is used for showing the differences between these two languages. At first we see that in the XML definition start with a timers tag in which a timer must be defined. In the DSL implementation this part is not need, a timer can be created anywhere within the attributes block of its parent, the (software) component. Looking at the timer definition itself we see that in XML the timer is defined in one tag and with the DSL in one element block. The all attributes including the id/name of the timer in XML are defined with tag attributes, which have the following format; the attribute name followed by a equals sign and the attribute value between double quotes. In the DSL implementation we see that the timer id/name comes directly after the timer indicator followed by the attribute block between brackets. Attribute values in the DSL implementation do not need any double quotes around unless the attribute expects a string as value type. It is also good to note here is that the mode is defined as an enum in the grammar with two values, PERIODIC and SINGLESHOT, and will be shown in the auto completion of the IDE for the DSL.

XML	HUTN-like DSL
`<timers>`	`timer timer1 {`
`    <timer id="timer1"`	`    timeout: 1000`
`        timeout="1000"`	`    mode: PERIODIC`
`        mode="PERIODIC"/>`	`}`
`</timers>`	

Code Snippet 4.2: Left the a timer defined in XML and right the same timer defined in the HUTN-like DSL.

Taking the timer definition in XML from Code Snippet 4.2 and format it in the smallest possible way that it is still parsable, will end up in a 82 symbols definition with the timers tags and in 65 symbols without it. If we do the same for the timer definition of the DSL implementation we only end up with 51 symbols. This is reduction of 38% of symbols compared to the same definition in XML with the `timers` tags and 22% compared to only the `timer` definition itself (without the `timers` tags) in XML. This is a huge reduction of symbols and shows that with the DSL implementation less verbose and more understandable software system configurations can be made than with the XML way of O2.

4.1.2 Defining Packages

In the DSL implementation a package is used to organize software system configuration elements for O2 (O2 elements). In Code Snippet 4.3 is the grammar shown for defining packages. A package embodies a namespace (`QualifiedName`) and is similarly used as a package in Java. Accessing an O2 element from a different package is done by prefixing the namespace of the package followed by the name of the O2 element. For example, accessing a simple type with the name `simpleFloat` of package `example.types` can be done with the full QualifiedName `example.types.simpleFloat`. The import construction in the DSL is used to access an O2 element without prefixing the namespace of the package. If we for example want to access the earlier mentioned simple type `example.types.simpleFloat`, we can import the package of the simple type first (`import example.types.*`) and then the simple type can be simply accessed by its short QualifiedName `simpleFloat` within the package.

```
Package:
  'package' name=QualifiedName '{'
    imports+=Import*
     elements+=O2Element*
  '}'
;

Import:
  'import' importedNamespace=QualifiedNameWithWildcard
;

O2Element:
    DataType | ManagementInterface | Component | Template | SoftwareSystem |
    AbstractHardwareBuildingBlock | HardwareSystem | Mapping
;
```
Code Snippet 4.3: The grammar defining for defining packages.

In a package all kinds of software system configuration elements (as shown in Figure 4.1) can be defined. For example, datatypes and software components can be defined in the same package. This is something that is not possible with O2 XML, where software components and datatypes should be defined in different files. The packages in this DSL give software system designers more freedom of how to define and organize software systems configurations. For example it is possible to quickly define a complete software system configuration in one file (package) and check if everything works, and later split it up into multiple files (packages). This is great for prototyping and creating quick examples to see if something works without switching between different files.

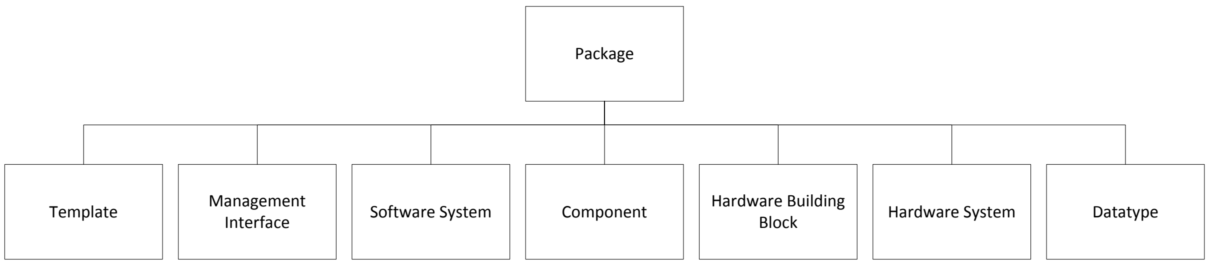


Figure 4.1: Overview of system configuration elements that can be defined in a package.

4.1.3 Assigning Data Types and Templates

In earlier shown snippets can be seen that the colon symbol (`:`) is used to assign values to attributes of elements, however in the DSL implementation the colon is also used in more places. This means that the colon symbol can be interrupted differently on different places, but still in a confined way. This language concept comes from the programming language C# [49], where the colon symbol is also multi interpretable. So is the colon symbol in C# used for both defining that a class extends another class and for defining that a class implements interfaces. It seems inconvenient to use one symbol with multiple interpretations, however with the C# language it is proven that most users with a programming background will rightly interpreted these symbols. This is because the symbols are only used on places where the symbol will likely rightly interpreted to due to the language structures in which the symbol is used. For example, taking the assignment of timeout attribute in Code Snippet 4.2 on line 3 (`timeout: 1000`). Here we see that the colon symbol is used for assigning a value to an attribute. In most cases the colon symbol will be rightly interpreted by an user with a programming background, because one recognizes this structure or can substitute into a recognizable structure (replacing the colon symbol with a equals symbol for example) with the same meaning.

One of the other places where the colon symbol is used, is for the assignment of a type to an element that must be specifically bounded to one type. For example, when defining a component instance of a software system definition with the DSL one must specify also of what type (software component) the instance should be or another example, the inputs and outputs of a software component are bound to a message type (data type) specifying what type/kind of message it eventually should sent or receive.

```
ComponentInstance:
    'instance' name=ID ':' component=[Component|QualifiedName]
;
```

Code Snippet 4.4: Grammar for defining a component instance of a software system.

In Code Snippet 4.4 is the grammar shown for defining a (software) component instance of a software system. In the snippet we see that an instance is defined by the first the keyword instance followed by a name and then the colon symbols, which can be in this case (and most other cases) interpreted as; is of type. For example, `instance RC1 : RouteCalculator` means instance RC1 is of type RouteCalculator. After the colon symbol comes the reference to the component, specifying that the instance is of that component type. This way the colon is also used for assigning types to inputs, outputs, connectors, parameters, recording points and notifications.

```
01 SoftwareSystem:
02    'softwareSystem' name=ID (':' realizes=[Template|QualifiedName])?
'{'
03          componentInstances+=ComponentInstance*
04          (delagations+=Delegation)*
05    '}'
06 ;
07
08 Delegation:
09    InputDelegation | OutputDelegation
10 ;
11
12 InputDelegation:
13    'inputDelegate' templateInput=[Input|ID] ':'
14          componentInstance=[ComponentInstance|ID]'.'input=[Input|ID]
15 ;
16
17 OutputDelegation:
18    'outputDelegate' templateOutput=[Output|ID] ':'
19          componentInstance=[ComponentInstance|ID]'.'output=[Output|ID]
20 ;
```

Code Snippet 4.5: A simplified grammar for defining a software system with delegations.

Besides assigning data types to elements and values to attributes the colon symbol is also used to indicate the realization of a template. In Code Snippet 4.5 a simplified grammar is shown for defining a software system with delegations. On the second line is shown that defining a software system starts with the `softwareSystem` word followed by its `name` and then the option for realizing a template with the colon symbol. In case the colon symbol can be interpreted as; realizes. For example, `softwareSystem TomTom : NavigationTemplate` means software system TomTom realizes NavigationTemplate. On line 13 and 18 is seen that the colon symbol also is used in delegations. With delegations the colon symbol can be interpreted as; delegates to. For example, `inputDelagate DesiredDestination : RC1.Destination` means that template input DesiredDestination is delegated to input Destination of component instance RC1.

4.1.4 Defining Protocol Attributes

The structure of setting attribute values for a protocol in O2 is very generic. Setting an attribute value in O2 is done by defining the name of the desired attribute and then a value. With this structure values for any attribute can be defined even for attributes that do not exist. Per protocol different sets of attributes are allowed which can be found in Appendix B. With this generic structure it is up to the software system designer to know which set of attributes he/she can use. In the DSL implementation these restrictions are directly embedded in the language. By supporting these restrictions directly in the language makes it for software system designers easier to see which attributes can be used where. A simplified grammar part for defining attributes of UDP sources is shown in Code Snippet 4.6.

```
NamedUDPSourceAttributes:
    'UDPSourceAttributes' name=ID '{'
          attributes=UDPSourceAttributes
    '}';

UDPSourceAttributes:
    'address:' address=STRING
    'port:' port=INT
    'idNoId:' (idNoId?='true'|'false');
```

Code Snippet 4.6: Simplified grammar for defining attributes of UDP (*channel*) sources.

In Code Snippet 4.7 an example is given of UDP protocol attributes defined in both XML and the implemented DSL format. In the XML example is directly seen that is it much bigger/verbose van the same definition of protocol attributes as in the DSL format. In the XML example every attribute can have manually chosen name followed by an value, both are specified as strings. In the DSL format we see that it is much more compact and has highlighting for different value types. This make the DSL much easier to read and write than with XML. Also on on-the-fly validation is added for these attributes, so is for example check that value assigned to a port is within the range of 0 up to 65536. Quickly counting the characters of the two examples, we see that the XML example consists of 167 characters and the DSL example of 74 characters, difference of 57% in the amount of used characters.

XML	DSL
`<attributes id="src1">`	`UDPSourceAttributes src1 {`
`  <attribute name="sendaddress" value="localhost"/>`	`  address: "localhost"`
`  <attribute name="sendport" value="50001"/>`	`  port: 50001`
`  <attribute name="idnoid" value="1"/>`	`  idNoId: true`
`</attributes>`	`}`

Code Snippet 4.7: Left a definition of protocol attributes in XML and right the same definition in the DSL implementation.

4.1.5 Scoping

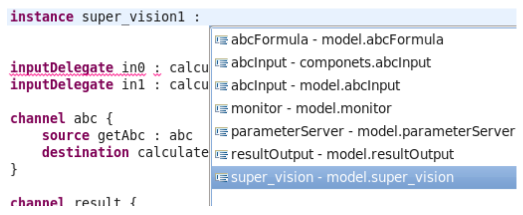
Scopes in Xtext are used for validation of references, but also for the auto completion feature in the IDE for the DSL. Standardly every element can be easily referenced to within a package by using it is name with the constraint that is conforms to the type/rule defined in the grammar. In most case this is fine but there are several cases in the DSL where the scopes are restricted to create valid software system configurations. A element in the language where the scope is restricted is the assignment of sources (outputs of component instances) and destinations (inputs of component instances) within a channel. A channel defined within a software system definition and is used to send messages from sources to destination, however only one type of message is allowed to send over the channel. What type of message this is, is determined by the first source defined in the channel. This means that every other source or destination with in that channel must be of the same type as the first defined source in the channel.

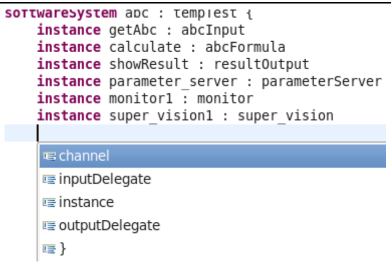
The standard scope of Xtext for sources and destinations of a channel, allows all inputs and outputs of component instances within a software system. This is one of the scopes that are restricted in the DSL implementation by introducing kinds of functions/code including the one showed in Code Snippet 4.8. In this code snippet the scope restriction of assignable component instances for destinations are handled. The function starts on line 3, getting the type of the channel by using a helper function `getChannelType` that gets the type of the first source defined in the channel. On line 4 the software system in which the channel is defined is retrieved. On line 7 and 8 the filtering of component instances with at least an input with the type conforming to the type of the channel is applied, but only if there is a channel type available, else a scope will be returned that of all component instances of the software system in which the channel is defined. Other places in the DSL where scope restrictions are applied include delegations and hardware link connections.

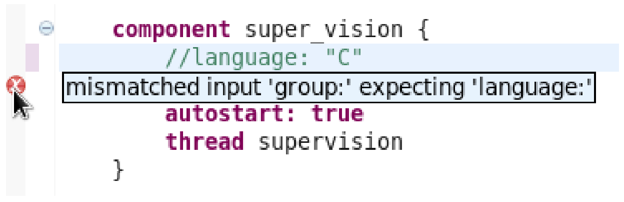
```
def IScope scope_Destination_componentInstance(Destination destination,
            EReference eReference) {
    val channelType = getChannelType(destination.eContainer as Channel, null);
    val softSys = destination.eContainer.eContainer as SoftwareSystem

    if (channelType != null) {
            return Scopes::scopeFor(softSys.componentInstances.filter[
                    hasComponentInputsWithType(component, channelType) ])
    }

    return Scopes::scopeFor(softSys.componentInstances)
}
```
Code Snippet 4.8: Code for restricting the scope of component instances for destinations of channels.

4.1.6 Eclipse Editor (IDE)

With Xtext a full-fledged Eclipse IDE is created for the textual part of the DSL. This IDE greatly improves defining software system configurations because the of computer-aided features, like auto completion, syntax highlighting and on-the-fly validation, it comes with. In Table 4.1 some of these computer-aided features of the IDE for the DSL are briefly described.

 `instance super_vision1 :`
`inputDelegate in0 : calcu`
`inputDelegate in1 : calcu`
`channel abc {`
`    source getAbc : abc`
`    destination calculate`
`}`
`channel result {`
abcFormula - model.abcFormula
abcInput - componets.abcInput
abcInput - model.abcInput
monitor - model.monitor
parameterServer - model.parameterServer
resultOutput - model.resultOutput
super_vision - model.super_vision
	Auto Completion for References
	The IDE supports auto completion for references. In the example left the auto completion shows that there are seven candidates for assign to the component type of the instance.

 `softwareSystem abc : tempTest {`
`    instance getAbc : abcInput`
`    instance calculate : abcFormula`
`    instance showResult : resultOutput`
`    instance parameter_server : parameterServer`
`    instance monitor1 : monitor`
`    instance super_vision1 : super_vision`
channel
inputDelegate
instance
outputDelegate
}
	Auto Completion for Syntax Elements
	The IDE support auto completion a every point in an instance of the DSL. In the example left is shown that there are four options to define software system elements and one option to close the software system definition with a bracket.

 `component super_vision {`
`    //language: "C"`
mismatched input 'group:' expecting 'language:'
`    autostart: true`
`    thread supervision`
`}`
	On-The-Fly Validation
	Software system configurations are directly validated while an user defines them in the IDE. In the example on the left there is syntax error shown, stating that it expects the assignment of a language attribute, which is in this case commented out.

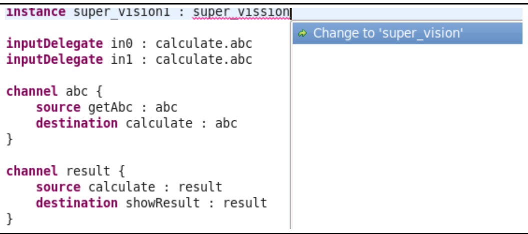
 `instance super_vision1 : super_vission`
Change to 'super_vision'
`inputDelegate in0 : calculate.abc`
`inputDelegate in1 : calculate.abc`
`channel abc {`
`    source getAbc : abc`
`    destination calculate : abc`
`}`
`channel result {`
`    source calculate : result`
`    destination showResult : result`
`}`
	Quick Fixes
	Some small errors, like the typo as shown in the example on the left, can be detected by the IDE. In most cases the IDE will also suggests a quick fix to directly solve the problem.

Table 4.1: Some of the computer-aided features that are provided by IDE of the DSL.

4.2 Graphical DSL Part (Sirius)

The DSL implementation has besides the textual DSL part also a graphical DSL part. This graphical part is implemented with Sirius and is used to specify different diagram representations/descriptions for the DSL. All visual mapping elements in these diagram description are associated to elements of the Ecore-based metamodel that is generated with Xtext from the textual DSL part. Every element in the metamodel can be accessed in diagram descriptions, meaning that basically any kind of graphical representation can be developed. In Figure 4.2 the mapping between graphical elements in a diagram view of a component and the elements of (a subset of) the generated metamodel for defining components is visualized with the red arrows. In this example every graphical element is mapped to one element of the metamodel, however it is also possible to map one graphical element to a group of elements in the metamodel.

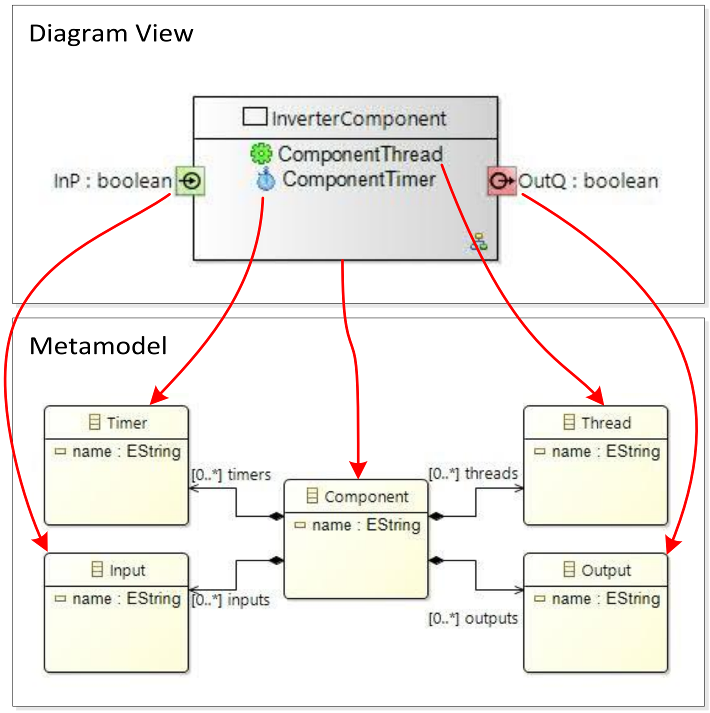


Figure 4.2: Visualization of mappings of graphical elements to elements of the metamodel.

Implementing graphical representations for the DSL costs more time than implementing their equivalent in the textual DSL part. This is because not only structural and textual aspects are involved but also graphical aspects, like colours, shapes and sizes. These graphical aspects results in more options that need to be considered, costing more time to implementing graphical representation. Because of the limited time of the these project only a few (basic) graphical representations are created for the DSL. However as earlier mentioned it is possible to create any graphical view in the future, because mappings can be associated to any element of the metamodel of the textual DSL part. In the following subsections is the implementation of the software system diagram of the DSL explained and how diagrams of the DSL can be used in the Eclipse IDE.

4.2.1 Software System Diagram View

The software system diagram is used to compose different instances of components within a software system. In Figure 4.3 an example software system with the software system diagram view of the DSL is showed. In this diagram we see a software system called `SwTimeSystem` and it contains three component instances, `TimeGen` of component type `TimeGenerator`, and `TimeDsplyA` and `TimeDsplyB` of component type `TimeDisplay`. The instance of component type

TimeGenerator has one output, SendTime and the two instances of component type TimeDisplay have one input, ReceiveTime. The arrows shown in this diagram are the direction in which messages/data is send. In this case the TimeGen instance sends a message of type Time to both the TimeDsplyA and TimeDsplyB instances.

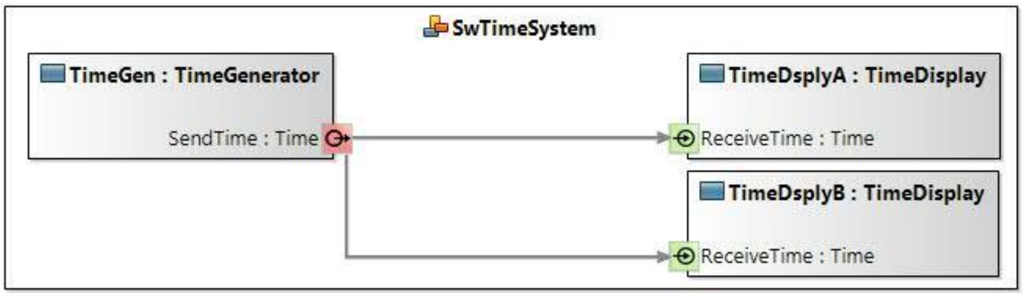
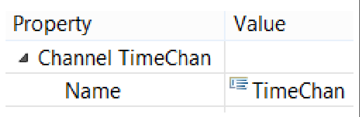
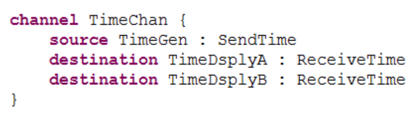
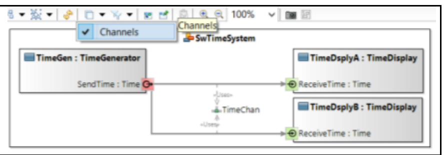


Figure 4.3: The software system diagram view of the DSL, showing the composition of component instance within a software system.

In Figure 4.3 the flow/direction of messages are clearly represented, but what is hidden in the diagram is what channels are used in the software system for sending messages. There are three option available to find out this kind of detailed information of software system configuration with the DSL implementation. The first option is to select the a element in the diagram and look in the properties pane for detailed information of that element. The second option is to open the textual file in which the element defined and search for more information there. And the last option is to enable available layers to show different views of the same diagram. In Table 4.2 these three options are shown to find the used channel of the software system presented in Figure 4.3. Also in the presented option the software system configuration can be edited.

		
Properties Pane	Textual File	Enable Layers

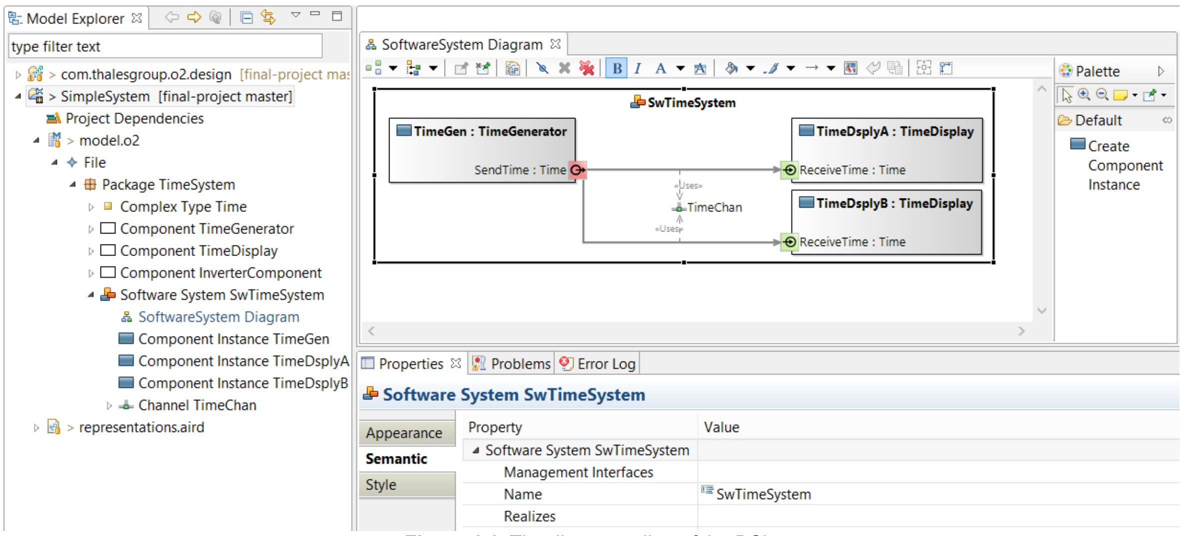
Property	Value
◢ Channel TimeChan	
Name	TimeChan

```
channel TimeChan {
    source TimeGen : SendTime
    destination TimeDsplyA : ReceiveTime
    destination TimeDsplyB : ReceiveTime
}
```

Table 4.2: The three options to find detailed information of an element in a software system configuration.

4.2.2 Eclipse IDE For Working With Diagrams

Diagrams of the DSL can only be applied in a so called modelling project and within this project there must be at least one .o2 file. When these two conditions are met, one can activate the O2 Design viewpoint of the DSL by right clicking on the modelling project, than clicking Viewpoints Selection and enabling the O2 Design viewpoint. When this viewpoint is enabled one can used the diagram views of the graphical DSL part in combination with the textual grammar and editor of the textual DSL part. Changes made in the graphical views are now automatically updates the textual representation of software system configurations and vice versa. This way the graphical and textual parts are always in sync. Also all .o2 files in the project are mark as models when the viewpoint is activated, this is indicated a -icon in the Model Explorer of Eclipse (showed on the left side in Figure 4.4). When a file is marked as model is possible to browse its content using a tree view in the Model Explorer. In this tree view an element can be selected to create specific diagram representation for that element. For example, in the model explorer of Figure 4.4 is shown that a software system diagram is created for the SwTimeSystem software system. Per element it is possible to created multiple diagram representations.



> com.thalesgroup.o2.design [final-project mas
> SimpleSystem [final-project master]
Project Dependencies
> model.o2
File
Package TimeSystem
Complex Type Time
Component TimeGenerator
Component TimeDisplay
Component InverterComponent
Software System SwTimeSystem
SoftwareSystem Diagram
Component Instance TimeGen
Component Instance TimeDsplyA
Component Instance TimeDsplyB
Channel TimeChan
> representations.aird

Software System SwTimeSystem

Property	Value
Software System SwTimeSystem	
Management Interfaces	
Name	SwTimeSystem
Realizes	

Figure 4.4: The diagram editor of the DSL.

All graphical information of diagrams, like size and position of elements in a diagram, are stored in a `.aird` file. By storing this information in a separate file ensures that the `.o2` files will stay clean of graphical information and so only contain information about software system configurations. This has the advantage that when in a shared project one software system designer works in the textually editor on a model is not bother by someone else who creates diagram representations of the same model for analysis.

On right-hand side of the model explorer in Figure 4.4 is the diagram editor showed. This editor basically consist of the three elements. The first element is the diagram view, showed in at the top middle. In the die view the diagram can be viewed and edited. The second element is the palette, showed on the right-hand side of the editor. This pallet contains different tool to edit the diagram in the diagram view. This palette is different for every kind of diagram in the DSL and can also change when a layers are enabled or disenabled. The last element is the properties pane, showed at bottom of the editor. This properties pane is as earlier mentioned used to showed en edited detailed information of a selected element in the diagram view.

4.3 *Transformations (ATL & Xtend)*

Transformations in the DSL implementation is used to produce XML files that are O2 compatible. With these XML files O2 can generate code for the desired target system. An overview of this approach and the UML approach of O2 is showed in Figure 4.5.On the left side of this overview the different meta-levels are shown. Each meta-level has its own horizontal lane, starting at the top with the meta-metamodel level (M3) and ending at the bottom with the real world level (M0). This overview can vertically be divided into three lanes, on the left side the UML approach of O2, in the middle the XML approach of O2 and on the right-hand side the implemented DSL approach. Between these approaches there are two transformations one from UML to XML and one from DSL to XML. As earlier mentioned this transformation to XML is done to utilize the code generator of the XML approach to produce code, so that no new code generator have to be created.

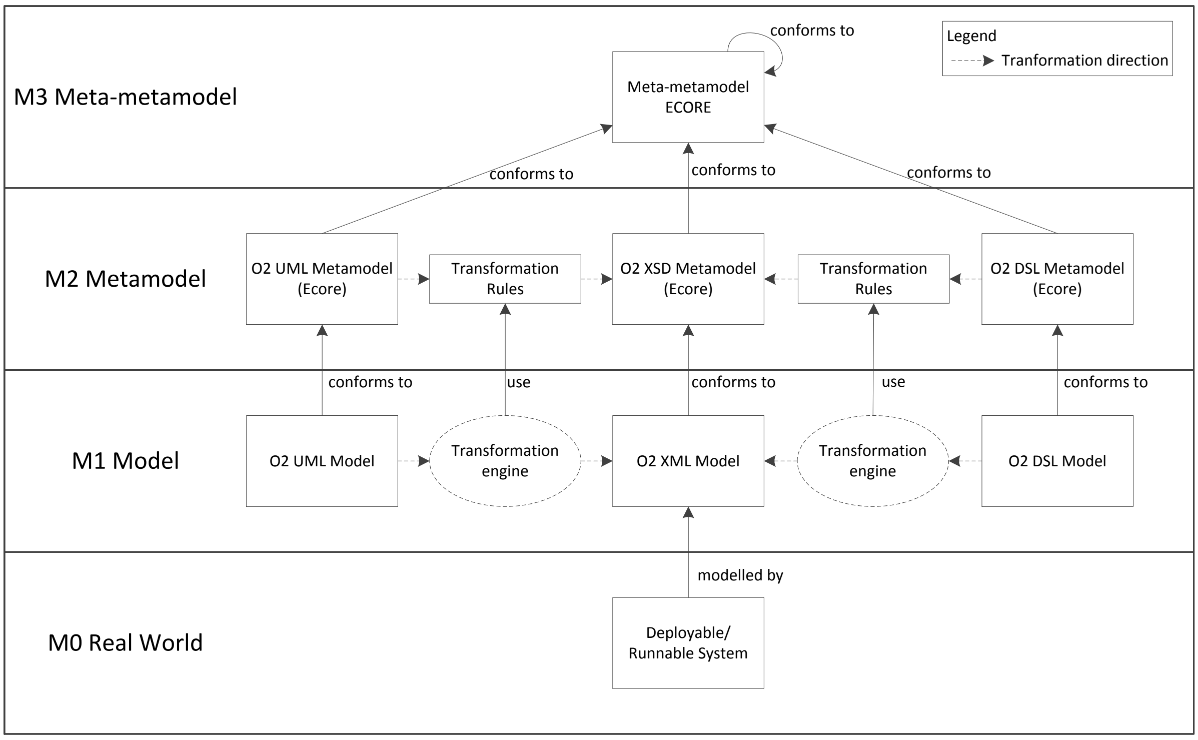


Figure 4.5: Overview of the UML and O2 DSL transformation approaches.

4.3.1 Model-to-Model Transformations (O2 DSL to O2 XML - ATL)

The transformation of O2 DSL to O2 XML consists of M2M and M2T transformations. The M2M transformations are discussed in this section and the M2T transformations in section 4.3.2. Almost all elements of a software system configuration defined with the DSL implementation are transformed into XML files using M2M transformations, except for the message types which are handled with M2T transformations. For M2M transformations at least two metamodels are needed, one source metamodel and one target metamodel. In the DSL to XML transformation the metamodel of the DSL is the source metamodel and the target metamodels are the XSDs of O2 XML (this are not the XSDs that are used to define message types). XSDs are basically metamodels (in non-Ecore format) for XML, however for M2M transformations in EMF Ecore-based target metamodels are needed. Luckily the modelling plugins of eclipse provide a XSD to Ecore converter. This converter is used to create Ecore-based metamodels of every XSD of the XML approach of O2. These (converted) Ecore-based metamodels of the XSD are used in the M2M transformations.

The M2M transformations of the DSL are implemented with ATL. Most elements defined with the DSL implementation have hierarchically the same structure as their equivalent in XML. This into a basic set of one-to-one transformation rules in ATL. However there are two big structural different that made the transformation more complex. The first one is that all elements of a specific type must be defined in their own type block. For example, threads, timers and usage of management interfaces in a component must be defined in their own blocks. These block cannot be empty following the XSDs, so either no block must be defined of the block must contain at least one element. This structure with type blocks is not present in the DSL metamodel and because these blocks cannot be empty an extra check have to be added to the transformation rules in ATL. This check must look if there is any element of a specific type and when there is one or more elements of that specific type, then first a block have to be created before adding any elements to it. An example of this check for threads and timers of a component is showed in Code Snippet 4.9. In this example an `do` block is added to the transformation rule with two `if` blocks (line 6 until 13). The first `if` (starting at line 7) checks if there are any threads defined in the DSL model and when true the `ThreadType` transformation rule is call, which handles the creation of the threads block and its content in the target XML model. The second `if` (starting at line 10) does the same but then for timers that are designed in the DSL model.

```
01 lazy rule XO2Component2Component {
02     from
03             xc : XO2!Component
04     to
05             c : S!Component  ( id <- xc.name )
06     do {
07             if (xc.threads.notEmpty()) {
08                     c.threads <- thisModule.ThreadsType(xc);
09             }
10             if (xc.timers.notEmpty()) {
11                     c.timers <- thisModule.TimersType(xc);
12             }
13     }
14 }
```

Code Snippet 4.9: A simplified transformation rule in ATL for producing (software) components in XML from components defined with the DSL implementation.

Another problem that had to be tackled is the transformation of a package to XML model. In the DSL implementation all system configuration elements are defined in packages, however this concept of a package is not available in a XML model. To preserve the names of packages, there are basically two options. One option is the adopted the package name in the file name of the target model and because a package can have multiple target models, the file should be suffixed with the name of the target XSD. For example, a package with the name components will result in the name components.software.xml when software is the target XSD. The other option is to prefix all software system configuration elements in the target XML model with the package name in which they are defined. However this options make the XML models much bigger and even more unreadable, so the first option is chosen and implement for the DSL.

4.3.2 Model-to-Text Transformation (DSL to XSD - Xtend)

In the DSL implementation M2T transformations are used for producing message types in O2 XML. In contrast to the other software system elements in O2 XML are message types not defined in a XML format, but in XSD format. The M2T transformation for the DSL are implemented in Xtend by defining templates. Basically only two templates are defined for the transformation of the DSL, one for simple types and one for complex types. In Code Snippet 4.10 a simplified template for producing simple types in XSD format are showed. In this example is can be seen that for every package a schema block is created and within this block for every `SimpleType` in the package a `simpleType` in XSD format is created (line 4 until 11). This way for every package with simple types a XSD can be produced.

```
01 def CharSequence compileXSD(Package p) '''
02    <?xml version="1.0" encoding="UTF-8"?>
03    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" id="«p.name»">
04    «FOR st : p.eContents.filter(SimpleType)»
05          <xs:simpleType name="«p.name».«st.name»">
06                <xsd:restriction base="«st.base»">
07                      <xsd:minInclusive value="«st.minInclusiveValue»"/>
08                      <xsd:maxInclusive value="«st.maxInclusiveValue»"/>
09                </xsd:restriction>
10          </xs:simpleType>
11    «ENDFOR»
12    </xs:schema> '''
```

Code Snippet 4.10: A simplified template for producing simple type in XSD format.

4.3.3 Execution of Transformations

For every ATL transformation to XML model file a plugin is created. This plugin can be executed from the command line by executing the jar of the plugin and giving the path of the source model and the path of the target model location. To create a complete XML model every plugin has to be executed on a source model. This way manually one-by-one transformations can be executed, however there is also a custom command line interface (cli) implemented for the DSL that can validate the source model and then execute all transformations at once on this source model. This execution of all transformations on a source model is defined in the `O2Generator` class and is also used in the on-the-fly transformation when a model is modified or created with graphical or textual editors of the DSL. With on-the-fly transformations the target models are automatically produced when a software system configuration in the editor doesn't have any errors. This way software system developers can directly see to what XML models their software system configuration defined with the DSL are transformed in. The default location where automatically generated XML models are stored is the `src-gen` directory in the project.

4.3.4 Approach for Backward Compatibility (O2 XML to O2 DSL)

A shortcoming of the implementation of O2 UML is that it did not provide backward compatibility for earlier defined software system configurations with O2 XML. This resulted in that software system configurations defined with O2 XML must be manually redefined in O2 UML. This can be a time-consuming task and also introduces an extra place to make errors. Due to time restrictions of the thesis project no backward compatibility with O2 XML is implemented for O2 DSL, however the approach to accomplish this backward compatibility is very similar to the approach as showed in Figure 4.5. The difference is that the transformation from O2 DSL to O2 XML is than the other way around, from O2 XML to O2 DSL. When such a transformation is realized legacy configurations made with O2 XML can then be edited with O2 DSL, however this also enables compatibility with O2 UML because there are already transformation available for O2 UML to O2 XML. So only one new transformation, O2 XML to O2 DSL is needed to realized compatibility for O2 DSL with O2 XML and O2 UML.

There are few challenges that are involved for implementing the O2 XML to O2 DSL transformation. These challenges are mainly the inverse of the challenges that are tackled in the O2 DSL to O2 XML transformation. One of these challenges is assigning names of packages in O2 DSL. Because O2 XML does not have packages, the name for the package should be derived from some something else. The file name of software system configurations defined with O2 XML is an element that is always available and so can be used as package name in O2 DSL. For all other elements the transformation should be really straightforward, because the hierarchical structure of most elements in O2 XML and O2 DSL are the same.

4.4 Case Study: ABC Formula Example

In this section the ABC Formula Example (or just ABC Example) defined with the DSL implementation is discussed. This ABC Example is used in the O2 manual [3] to explain and demonstrate all elements of software system configurations that can be defined and used with O2. By successfully implementing the ABC Example with the DSL implementation we can conclude that the DSL is complete enough for defining software system configurations of most target systems. In this case study also a comparisons between O2 DSL and O2 XML, and O2 DSL and O2 UML is presented. The full textual software system configuration of the ABC Example defined with O2 DSL is showed in Appendix C.

4.4.1 Overview of The ABC Formula Example

The ABC formula, also called square root formula, is used to find the roots of a quadratic equation. The quadratic equation has the form of $ax^2 + bx + c = 0$, with the idea is to find an `x` for a given `a`, `b` and `c`. Solving `x` can be done with the following formula: $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$. For this formula we need three input variables `a`, `b` and `c` and results into one output variable `x`. The ABC Example consists of a system that follows the characteristics for the square root formula. This system basically consist of three core components as showed in Figure 4.6. The first component, `abcInput` is used to provide massages that contain an `a`, `b` and `c`. These messages are send to the second component, `abcFormula`. This second component is used to calculate the square root formula for the provided `a`, `b` and `c` of received messages. The result of this component is send to the last component of the example, `resultOuput`. This last component is used to print the result to a screen. Note that the square root formula is only as context for the software system configuration. The actually code for executing the square root formula is not discussed in this chapter.

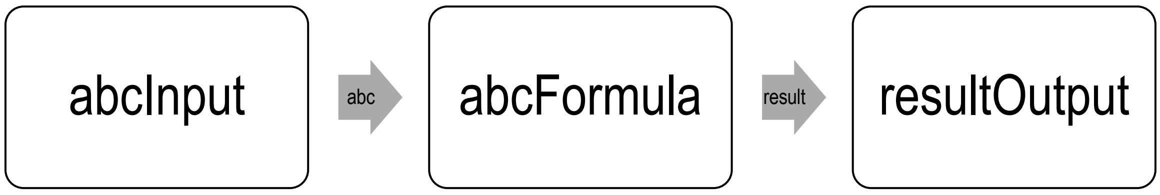


Figure 4.6: The (software) system description of the ABC example.

4.4.2 Textual Comparison

The ABC Example is completely manually written with the textual part of the implemented DSL. The ABC Example could be written within a half an hour, which is pretty fast and something that definitely cannot be achieved with the O2 UML or O2 XML. This is mainly because of the computer-aided features that are provided by the editor of the textual DSL part. The auto-completion feature really speeds up writing software system configurations and the on-the-fly validation shows directly if something is wrongly defined in the software system configuration.

With O2 XML the elements of the same group (see section 2.2.2) can only be defined in the same file. This results in that the ABC Example consists of multiple files. With O2 DSL it is possible to define the whole software system configuration of the ABC formula in one file. To make the comparison between O2 DSL and O2 XML more clear, is chosen the use the file structure of the ABC Example defined with O2 XML in the ABC Example defined with O2 DSL. In Table 4.3 a comparison is shown between the O2 XML and O2 DSL software system configurations of the ABC Example. This comparison is based on the two metrics; lines and length. Lines are the amount of lines excluding blank lines in a files (with pretty print format). The length is the amount of characters in a file without blanks. If we look at the numbers of both approaches we see directly that the ABC Example defined with O2 DSL is in total 47% smaller in length and contains 26% less lines compared to O2 XML. Looking at every file separately we see that all files in the DSL implementation score better than O2 XML on length metric, however on lines metric we see that we see that four files score better with O2 DSL than O2 XML. The hardware system and mapping defined with O2 DSL score worse on the line metric comparted to O2 XML. The explanation for this is that the pretty print view of files defined with O2 DSL uses more blank lines than the pretty print of XML files. This results in more lines for O2 DSL, however the length of the files are still shorter comparted to their equivalent in O2 XML.

Element Group/File	O2 XML		O2 DSL		Diff	
	Lines	Length	Lines	Length	Lines	Length
Mesage type	26	845	13	286	50%	66%
Software	134	4538	85	2247	37%	50%
Software system	31	960	19	437	39%	54%
Hardware	31	839	12	296	61%	65%
Hardware system	4	132	5	91	-25%	31%
Mapping	51	1847	72	1498	-41%	19%
Total	**277**	**9161**	**206**	**4855**	26%	47%

Table 4.3: A comparison based on the amount of lines and length between the O2 XML and O2 DSL definitions of the ABC Example.

4.4.3 Graphical Comparison

The graphical DSL part of O2 DSL is during the thesis period not completely implemented for all elements of a software system configuration. This means that no complete comparison can be made between the graphical DSL part and O2 UML, however views that are implemented can be compared. For the comparison discussed here, the software system diagram views of both the textual DSL part and O2 UML are taken, which are presented in Table 4.4.

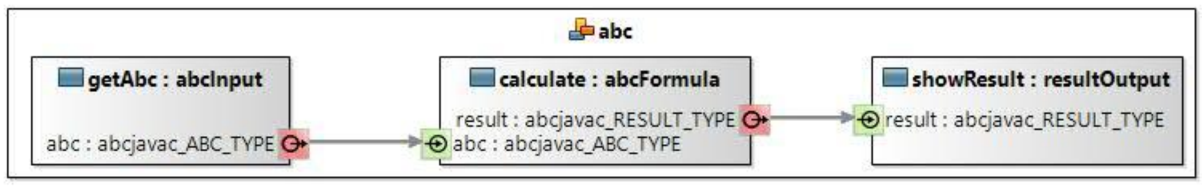
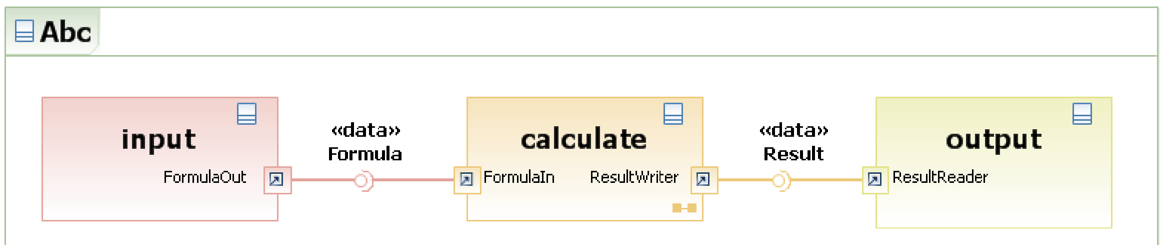
O2 DSL
 abc getAbc : abcInput abc : abcjavac_ABC_TYPE calculate : abcFormula result : abcjavac_RESULT_TYPE abc : abcjavac_ABC_TYPE showResult : resultOutput result : abcjavac_RESULT_TYPE
O2 UML
 Abc input FormulaOut «data» Formula calculate FormulaIn ResultWriter «data» Result output ResultReader

Table 4.4: The software system diagram views of both the implemented DSL and O2 UML.

Comparing both diagrams we see that they globally have the same structure, however there are differences in the amount of displayed information and the usage of graphical elements. One of the differences is that the inputs and outputs in O2 DSL are indicated with icons and colours, green for inputs and red for outputs. In the O2 UML diagram inputs and outputs are recognized with provided and required interface symbols. Another difference is that the message type in the DSL diagram is directly showed behind the label of an input or output and in the O2 UML diagram this is showed at the connection point of the provided and required interface symbols. In O2 UML this point must be annotated with the stereotype «data», and extra action that is not needed when defining a connection with O2 DSL. A small differences, but makes easier to work with O2 DSL than with O2 UML.

Besides graphical differences, there are also functional differences between O2 UML and the graphical part of O2 DSL. The biggest of those differences are the use of layers. The software system diagram view of O2 DSL provides the option to enable a channels layer. With this layer the channels are shown in the diagram and not only the direction of the messages as in O2 UML. This layer is useful when channels with multiple sources and/or destinations are used in a software system configuration. An example of a software system diagram with the channels layer enabled is earlier showed in Figure 4.4.

4.4.4 Result

The ABC Example defined with O2 DSL can be transformed into a XML model, which is substantially the same as the O2 XML definition of the ABC Example. This XML model can be provided to the code generator O2 to eventually produce runnable/deployable code. With this last step is showed that he ABC Example can be defined with O2 DSL, suggesting that software system configurations for most target systems can be made with it. Also is showed with this case study that O2 DSL is in most cases better than O2 XML and O2 UML. So is defining correct software system configurations with O2 DSL faster, are the textual files smaller and are graphical views more advanced with layers.

Beside this case study is O2 DSL also demonstrated several times to users that work with O2 XML and/or O2 UML. Reactions from these users on this demonstration were very positive. The most common reaction from them was that they were positively surprised by how fast and easy a correct software system configuration could be defined with O2 DSL. They also see a lot of potential in O2 DSL, because it allows to work both in textual and graphical views. At the end of the demonstrations all attended users agreed that O2 DSL is much better than O2 XML and O2 UML for defining software system configuration and they would defiantly want to work with it in their projects.

5 Discussion

In this chapter is shortly discussed the comparison of the DSL implementation in contrast to other frameworks and what extensive evaluations have to be performed to see if O2 DSL is really a success.

5.1 Comparison With O2 UML and O2 XML

On different places in this report is O2 DSL compared with O2 XML. So is multiple times the length of software system configurations defined with O2 DSL compared to the same configurations defined with O2 XML. From these comparisons can been seen that the textual grammar of O2 DSL is much smaller and easier to read and write than O2 XML. The results of these comparisons shows that software system configurations defined with O2 DSL are between 19% and 66% smaller. The textual grammar of O2 DSL also looks more like the programming languages C and Java. A style that did get a lot of positive feedback from software system designers, while O2 DSL was demonstrated to them.

The comparison between O2 DSL and O2 UML are less extensively presented in this report. This is because that the graphical views of O2 DSL are not completely implemented and optimized, due to time limits of the thesis project. This resulted in that the graphical views of O2 DSL look much like the diagrams used in O2 UML. However O2 DSL have two great advantages compared to O2 UML which are that graphical views of O2 DSL can utilize advanced graphical features like layers. The second advantage of O2 DSL is that with it graphical views can be created that are not bounded to any complex host language. This allows for the creation of more advanced and flexible graphical views for O2 DSL. This is something that can be used to improve O2 DSL with in the future, by creating more easier to use and/or advanced diagram views. Looking at both comparisons of O2 DSL and O2 XML, and O2 DSL and O2 UML, the conclusion can be safely made that O2 DSL is the better that the other two solutions.

5.2 Comparison With Other Frameworks

As mentioned before is O2 DSL only compared with O2 DSL and O2 UML, however there are more component-based frameworks for realizing distributed real-time software systems. At beginning of this thesis project a small research was performed to find different component-based frameworks for creating real-time software systems. The result of this search was that most of these frameworks work the same way and distinguishes themselves to focus on a specific characteristic, like performance, interoperability between a lot of different languages or creating systems with a small footprint for small embedded systems. One thing that was found during this research was that defining software system configurations with the founded frameworks must either happen completely graphical with diagrams or completely textual. None of the founded frameworks has both an easy way to define configurations both graphically and textually. Most of the founded framework do not come with tooling or only very limited tooling for defining software system configurations. None of these framework comes with such a feature rich IDE as O2 DSL provides. The combination of both providing an easy way of defining software system configuration textually and graphically and providing tooling with computer-aided features was not found in any framework. This means that O2 DSL is probably the only solution with a textual and graphical way, and computer-aided support, to define software system configurations for real-time software systems.

5.3 Evaluation

In this report the DSL implementation is evaluated using the ABC Formula Example case study. From this case study we can conclude that the DSL implementation is complete enough to defined software system configurations for most target systems. Besides the case study is the DSL implementation also multiple times demonstrated to different software system designers which work with O2. The reactions from them were very positive and the most noticeable thing they mentioned was that the DSL implementation is must faster for defining software system configurations compared to the current solutions they use.

Both the case study and the positive reactions of software system designers indicates that the DSL implementation has a lot of potential, however to see if the DSL implementation is really a success more extensive evaluation must be performed. For example, a bigger case study must be handling of defining a big software system configuration to see how the implementation scales. Another thing that have to be evaluated is how the DSL implementation works with shared projects on which multiple people work at the same time.

6 Conclusion

In this chapter a summarizing conclusion of delivered thesis work is given including recommendations and ideas for future work.

6.1 Summary

In this report a DSL implementation (O2 DSL) for defining software system configurations in both textual and graphical ways with computer-aided tooling is presented. This implementation consists of three parts; a textual part, a graphical part and transformations. The first part, the textual part, consists of a textual grammar with an (Eclipse) IDE for defining software system configurations and is realized with Xtext. This textual grammar is mainly composed of different language constructs of GPLs and DSLs that are familiar for users with a programming background (the main user group that define software system configurations). This makes the grammar easy to use and understand for this group of users. It also resulted in a grammar with no complex and verbose syntactic elements, making it a much cleaner and smaller language compared to O2 XML. The IDE for editing configurations in this textual grammar provides different computer-aided features, like syntax highlighting, autocomplete and on-the-fly validation. These features improve the speed and ease of defining (syntactically and semantically) correct software system configurations compared to O2 XML and O2 UML. For example, with on-the-fly validation software system designers can while editing directly see what is correct and incorrect in their software system configurations. This feature alone already eliminates the time-consuming trail-and-error trips of O2 XML and O2 UML.

The second part, the graphical part, consists of a set of graphical views and an editor for these views. The graphical views are mapped to the metamodel of the textual grammar of O2 DSL. This mapping makes it possible to edit and view software system configurations both in a textual and graphical view, allowing software system designers to choose the best view for their needs. The current set of graphical views implemented in O2 DSL look much like the views used in O2 UML, but with more advanced features such as layers. Sirius, the chosen tool for implementing the graphical views of O2 DSL, is very flexible and allows for the creation of custom graphical views not bounded to any complex host language/notation, such as UML. This makes it possible to create more specialized views in the future for O2 DSL and improve defining software system configurations with it even further.

The third and last part of O2 DSL are the transformations. The transformations are implemented for O2 DSL to provide a way to (indirectly) generate code from software system configurations. These transformations are implemented in ATL and Xtend and transformation software system configurations defined with O2 DSL in XML files that conform to O2 XML. These files can be provided to the existing code generator of the O2 framework to generate code for the desired target system. This makes O2 DSL basically a complete working framework/solution for defining software system configurations, but also for generating executable/deployable code from these configurations. Besides the transformations for code generation purposes, is in this report also a backward compatibility approach for supporting existing defined software system configurations (of legacy systems) that conform to O2 XML or O2 UML discussed. This approach is basically the transformation of O2 XML to O2 DSL.

All the parts of O2 DSL are tested with a case study called ABC Example. In this case study shows that the complete ABC Example can be defined with O2 DSL, from which we can conclude that O2 DSL is suited for defining software system configurations for most target systems. In the case study is also a comparison made with the ABC Example defined with O2 DSL and the ones defined with O2 XML and O2 UML. In this comparison is showed that the software system configuration defined with O2 DSL are almost 50% smaller than the same one in O2 XML. From the comparison with O2 UML is shown that the graphical views of O2 DSL provide the same information as O2 UML, but then with layers. A demonstration of O2 DSL is given to different software system designers with experience and knowledge of O2 XML and O2 UML. The reactions and feedback from these designers were very positive and they were most pleasantly surprised of how fast correct software system configurations could be made with O2 DSL compared to O2 XML and O2 UML. Both the results of the demonstration and the case study shows that O2 DSL is an good addition in the world of frameworks for creating distributed real-time software systems.

As formulated in the instruction the goal of this thesis is; *realizing a DSL for defining software system configurations with textual and graphical views including computer-aided tooling*. The idea behind this goal is overcome the two main shortcomings of existing languages (of frameworks) that makes defining software system configurations hard. These shortcomings are; that existing languages for defining software system configurations are too complex/hard to understand and that most of these languages do not have tooling with advanced computer-aided features/guidance. O2 DSL overcomes the first shortcoming by providing an easy to use textual language with graphical views. The second shortcoming overcomes O2 DSL by providing an IDE with a set of computer-aided features similar to the ones provide by IDEs of GPLs. The transformations makes O2 DSL complete, by making it not just a solution to fast and correctly define software system configurations with, but also to generate code from these configurations. With the mentioned shortcomings covered by O2 DSL, the conclusion can be made that the goal of the thesis is achieved and that O2 DSL is the only solution that support textual and graphical views with computer-aided tooling for development of distributed real-time systems.

6.2 Recommendations

The implementation of O2 DSL shows a lot of potential to be a good solution that improve the development of distributed real-time software systems. O2 DSL as new solution open doors for further improvements and research directions. In the following subsections three of these research directions are discussed.

6.2.1 Traceability of Requirements

Engineering a real-time system involves a lot of requirements. These requirements define what functions a system should have and the communication between high-level system components. A requirement is fulfilled by one or more (software and/or hardware) components in a system. The mapping of what component or set of components should fulfil a specific requirement is traditionally specified in text documents. Tracing a requirement that is only defined in text documents to the specific code (or hardware elements) that fulfil the requirement of a system is very hard, because manually tracing requirements in a system is time-consuming. And even more time-consuming when software developers/engineers use different terminology in the implementation than used int the textual description of the requirement. For example, a software developer can use a different name for a type in code than is used in the requirement specification. This makes finding back the requirement in de code very hard, because the type cannot be found in de code with the name that is specified in the requirement.

Luckily there are several developments that promote specification of system requirements in more formal models. Capella [50] is such an approach to specify system requirements in models. The way how these models are structured should improve the way of tracing requirements, however they are still no directly linked to actual software components as defined in software system configurations. An interested research would be to look how O2 DSL can be linked to such models, to improve traceability of requirements. Looking at if it is possible to generate skeletons of software system configuration from models with system requirements is one of the research questions than can be included. Also can be looked at tracing of requirements in software system configurations to form a set of features that can be used in a product line engineering approach.

6.2.2 Quality of Service Specification

Channels in software system configurations defined with O2 DSL, O2 XML and O2 UML are directly mapped to a (communication) protocol, like TCP, UDP or shared memory. Every mapping of a protocol to a channel (protocol mapping) consist of a set of (protocol) attributes. Each type of protocol has its own set of allowed attributes and every attribute has its own range of allowed values. With O2 XML and O2 UML the software system designer needs to learn all these attributes and their ranges by heart or look on a cheat sheet every time to correctly define attributes of a protocol mapping This makes defining protocol mappings with O2 XML and O2 UML hard. O2 DSL provides computer-aided features that help the software system designers with correctly defining attributes of protocol mappings. So provides O2 DSL autocomplete to see what attributes are allowed per protocol and on-the-fly validation gives software system designers direct feedback if values are in the ranges of their corresponding attributes. This makes defining protocol mappings already much easier than with O2 XML and O2 UML.

So O2 DSL improves defining protocol mappings of software system configurations, however it still involves a lot of attributes and technical knowledge about the mapped protocol types. This is a part of the software system configuration that can be simplified/improved by replicating it with some kind of quality of service assignment to channels. The idea is that per channel speed and reliability levels can be assigned. Based on these levels the desired communication technology will be selected. For example, a channel assigned to the levels fast and unreliable will result in the target system as an UDP protocol. This is a very basic example, however more advanced solutions can be researched that also take into where physically software components are placed to determine what protocol can be best used in the target system. Besides only looking at simplifying protocol mappings, can also be looked at what other elements of software system configurations can be improved.

6.2.3 Dynamic Validation

The environment of O2 DSL provides a good base for performing validation of dynamic properties, like system performance or optimal deployment of software components on hardware/computers. Performing dynamic validation most likely involves a model checker/some kind of simulation tooling. There are different researches done like [51], showing approaches for checking dynamic properties. An interesting research area is to look if performance bottlenecks can be found from software system configurations defined with O2 DSL. This way these bottlenecks can be found and possibly eliminated before deploying it on the actual computer system. Another interesting area is to look if evolution of software system configurations can be simulated and tracked to evaluate design requirements/patterns of a system. Both these research areas require to extends O2 DSL with extra information. For example, to find performance bottlenecks in software system configurations, some kind of information should be provide. How this information is provided, for example with annotations or a separate file, would be an important part of the research.

References

[1] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, New York: Springer, 2011.

[2] Stanford University, "Folding@home," Stanford University, 2015. [Online]. Available: http://folding.stanford.edu/home/. [Accessed 04 01 2015].

[3] R. van Hees, T. de Goede, F. Schopbarteld and H. ten Brugge, "Software User Manual (SUM) for The O2 of The SR3D Platform," THALES NEDERLAND B.V., Hengelo, 2014.

[4] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," W3C, 2008.

[5] M. Fowler, Domain Specific Languages, Addison-Wesley Professional, 2010.

[6] I. Dejanovic, M. Tumbas, G. Milosavljevic and B. Perisic, "Comparison of Textual and Visual Notations of DOMMLite Domain-Specific Language," in *ADBIS (Local Proceedings)*, 2010.

[7] Object Management Group, "OMG Unified Modeling LanguageTM (OMG UML), Infrastructure Version 2.4.1," 2011.

[8] A. van Deursen, P. Klint and J. Visser, Domain-specific languages: an annotated bibliography, ACM Sigplan Notices, 2000.

[9] P. Hosek, T. Pop, T. Bures, P. Hnetynka and M. Malohlava, "Comparison of Component Frameworks for Real-Time Embedded Systems," in *Component-Based Software Engineering*, Berlin , Springer Berlin Heidelberg, 2010, pp. 21-36.

[10] M. Fowler, "FluentInterface," 20 December 2005. [Online]. Available: http://martinfowler.com/bliki/FluentInterface.html. [Accessed 12 February 2015].

[11] Mockito, "mockito.org," Mockito, 2015. [Online]. Available: http://mockito.org/. [Accessed 23 February 2015].

[12] The Eclipse Foundation, "Xtext - Language Development Made Easy!," The Eclipse Foundation, 2014. [Online]. Available: https://eclipse.org/Xtext/. [Accessed 3 December 2014].

[13] Data Geekery, "jOOQ: The easiest way to write SQL in Java," Data Geekery, 2015. [Online]. Available: http://www.jooq.org/. [Accessed 11 March 2015].

[14] J. C. Cleaveland, "Building application generators," in *IEEE Software 5.4* , 1988.

[15] P. Haduk, "Modular Domain Specific Languages and Tools," in *Software Reuse, 1998. Proceedings. Fifth International Conference on*, New Haven, 1998.

[16] M. Mernik, J. Heering and A. M. Sloane, "When and How to Develop Domain-Specific Languages," in *ACM computing surveys (CSUR)*, Maribor, 2005.

[17] The Eclipse Foundation, "Eclipse Modeling Framework (EMF)," The Eclipse Foundation, 2015. [Online]. Available: https://www.eclipse.org/modeling/emf/. [Accessed 02 03 2015].

[18] S. Kelly, "Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM," in *19th annual ACM conference on object-oriented programming, systems, languages, and applications, workshop on best practices for model driven software development*, 2004.

[19] H. Kern, A. Hummel and S. Kühne, "Towards a Comparative Analysis of Meta-Metamodels," *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11,* no. ACM, pp. 7-12, 2011.

[20] Spoofax, "The Spoofax Language Workbench," Spoofax, 2013. [Online]. Available: http://strategoxt.org/Spoofax. [Accessed 21 January 2015].

[21] Microsoft, "Modeling SDK for Visual Studio - Domain-Specific Languages," Microsoft, 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/bb126259. [Accessed 24 January

2015].

[22] J. Bezivin, F. Jouault and D. Touzet, "Principles, standards and tools for model engineering," in *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on. IEEE*, 2005.

[23] The Eclipse Foundation, "org.eclipse.emf.ecore (EMF Documentation)," The Eclipse Foundation, 22 January 2015. [Online]. Available: http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html. [Accessed 28 January 2015].

[24] D. Steinberg, F. Budinsky and E. Merks, EMF: eclipse modeling framework, Pearson Education, 2008.

[25] The Eclipse Foundation, “EcoreTools,” The Eclipse Foundation., 2013. [Online]. Available: http://eclipse.org/ecoretools. [Accessed 26 November 2014].

[26] M. Koegel and J. Helming, "EMF Tutorial," EclipseSource, 14 April 2015. [Online]. Available: http://eclipsesource.com/blogs/tutorials/emf-tutorial/. [Accessed 19 April 2015].

[27] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing Ltd, 2013.

[28] R. Pattis, "Ebnf: A notation to describe syntax," 1980.

[29] B. Brodski, "#17 Restricting Scope," XtextCasts, 27 March 2013. [Online]. Available: http://xtextcasts.org/episodes/17-restricting-scope. [Accessed 2015 April 28].

[30] The Eclipse Foundation, "Xtend," The Eclipse Foundation, 4 June 2015. [Online]. Available: https://eclipse.org/xtend/. [Accessed 2 April 2015].

[31] V. Vujović, M. Maksimović and B. Perišić, "Sirius: A rapid development of DSM graphical editor," in *Intelligent Engineering Systems (INES), 2014 18th International Conference on*, 2014.

[32] The Eclipse Foundation, "Serius - The easiest way to get your own Modeling Tool," The Eclipse Foundation, 2014. [Online]. Available: https://eclipse.org/sirius/. [Accessed 11 December 2014].

[33] Obeo, "UML Designer," Obeo, 2014. [Online]. Available: http://www.umldesigner.org/. [Accessed 21 December 2015].

[34] N. Elouali, J. Rouillard, X. Le Pallec and J.-C. Tarby, "MIMIC: Leveraging Sensor-based Interactions in Multimodal Mobile Applications," in *CHI'14 Extended Abstracts on Human Factors in Computing Systems*, 2014.

[35] The Eclipse Foundation, "Capella," The Eclipse Foundation, 2014. [Online]. Available: https://polarsys.org/capella/. [Accessed 21 December 2014].

[36] The Eclipse Foundation, "ATL," The Eclipse Foundation, 2014. [Online]. Available: https://eclipse.org/atl/. [Accessed 12 December 2014].

[37] F. Allilaire and F. Jouault, ""Families to Persons" - A simple illustration of model A simple illustration of model," ATLAS group, INRIA & University of Nantes, France, Nantes, 2007.

[38] Smalltalk.org™, "Smalltalk.org," Smalltalk.org™, 2010. [Online]. Available: http://www.smalltalk.org/. [Accessed 11 05 2015].

[39] The Eclipse Foundation, "Modeling Amalgamation," The Eclipse Foundation, 2015. [Online]. Available: https://eclipse.org/modeling/amalgam/. [Accessed 2 April 2015].

[40] The Eclipse Foundation, "Graphical Modeling Framework (GMF) Tooling," The Eclipse Foundation, 2015. [Online]. Available: http://eclipse.org/gmf-tooling/. [Accessed 2 April 2015].

[41] The Eclipse Foundation, "Graphiti Home," The Eclipse Foundation, 2015. [Online]. Available: https://eclipse.org/graphiti/. [Accessed 2 April 2015].

[42] K. Aers, "Graphiti and GMF Compared," 4 April 2011. [Online]. Available: http://www.slideshare.net/koentsje/graphiti-and-gmf-compared. [Accessed 17 4 2015].

[43] V. Vujović, M. Maksimović and P. Branko, "Comparative analysis of DSM Graphical Editor frameworks: Graphiti vs. Sirius".

[44] M. Barbero, "Sirius: Graphical Editors for your DSLs," 3 December 2013. [Online]. Available: http://www.slideshare.net/mikaelbarbero/sirius-graphical-editors-for-your-dsls. [Accessed 20 December 2014].

[45] The Eclipse Foundation, "QVT Operational," The Eclipse Foundation, 2015. [Online]. Available: https://projects.eclipse.org/projects/modeling.mmt.qvt-oml. [Accessed 17 4 2015].

[46] I. Kurtev, "Alignment of ATL and QVT," ATLAS Nantes, France, 2006.

[47] AtlanMod, "ZooFederation," AtlanMod, 19 February 2010. [Online]. Available: http://www.emn.fr/z-info/atlanmod/index.php/ZooFederation. [Accessed 20 April 2015].

[48] Object Management Group, "Human-Usable Textual Notation," 2004.

[49] Microsoft, "Visual C# resources," Microsoft, 2015. [Online]. Available: https://msdn.microsoft.com/en-us/vstudio/hh341490. [Accessed 30 April 2015].

[50] PolarSys, "Capella," The Eclipse Foundation, 2015. [Online]. Available: https://www.polarsys.org/projects/polarsys.capella. [Accessed 15 March 2015].

[51] D. A. van der Kruk, "A Formal Model to Evaluate Evolution of Design Patterns," Enschede, 2014.

[52] The Eclipse Foundation, "Eclipse," The Eclipse Foundation, 2014. [Online]. Available: http://eclipse.org. [Accessed 25 November 2014].

[53] I. Jacobson, G. Booch and J. Rumbaugh, The Unified Software Development Process, Addison-Wesley Professional, 1999.

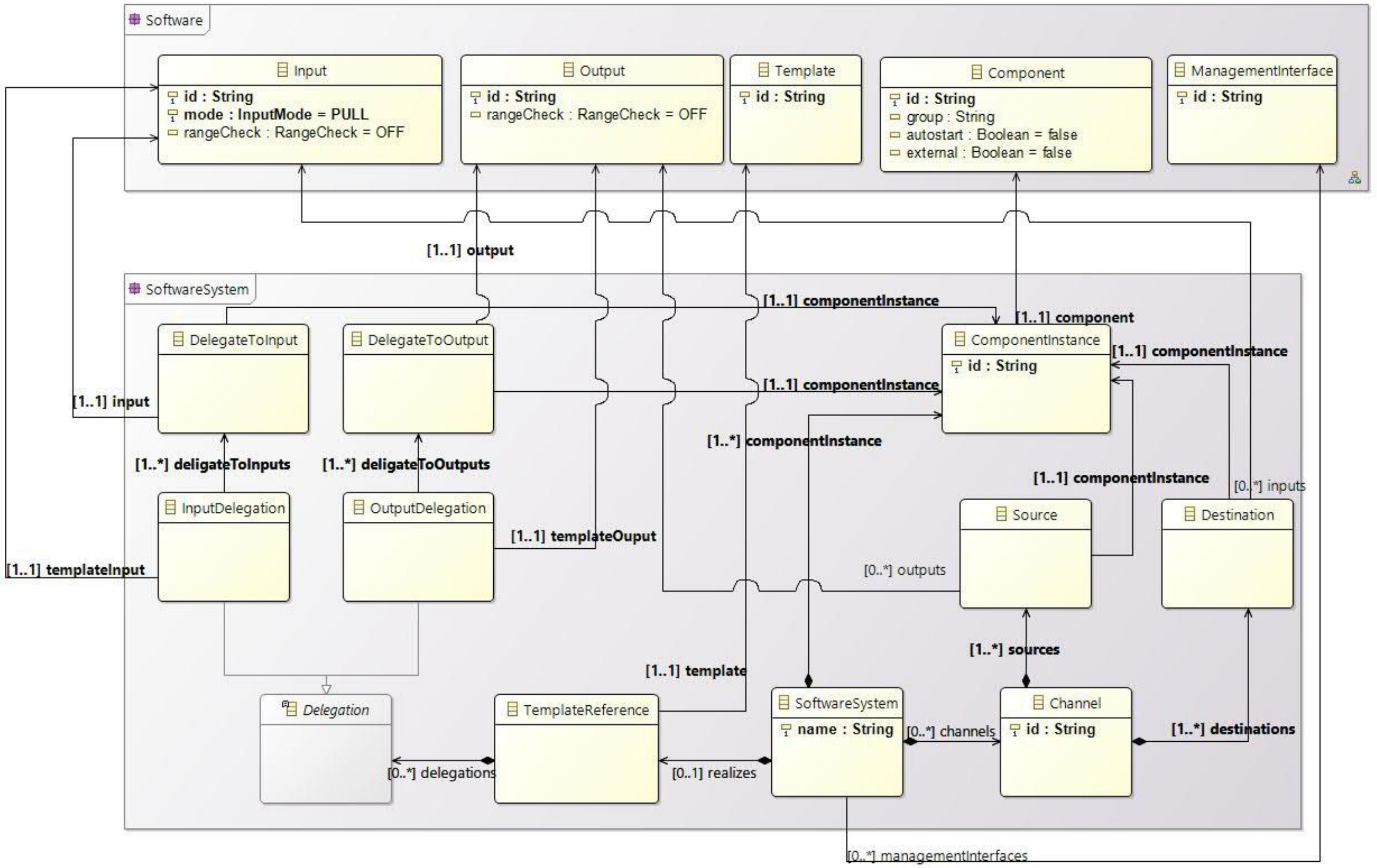
[54] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler and S. Völkel, "Design Guidelines for Domain Specific Languages," arXiv, Orlando, 2014.

[55] Mono Project, "WinForms Designer," Mono Project, 2015. [Online]. Available: http://www.mono-project.com/archived/winforms_designer/. [Accessed 20 January 2015].

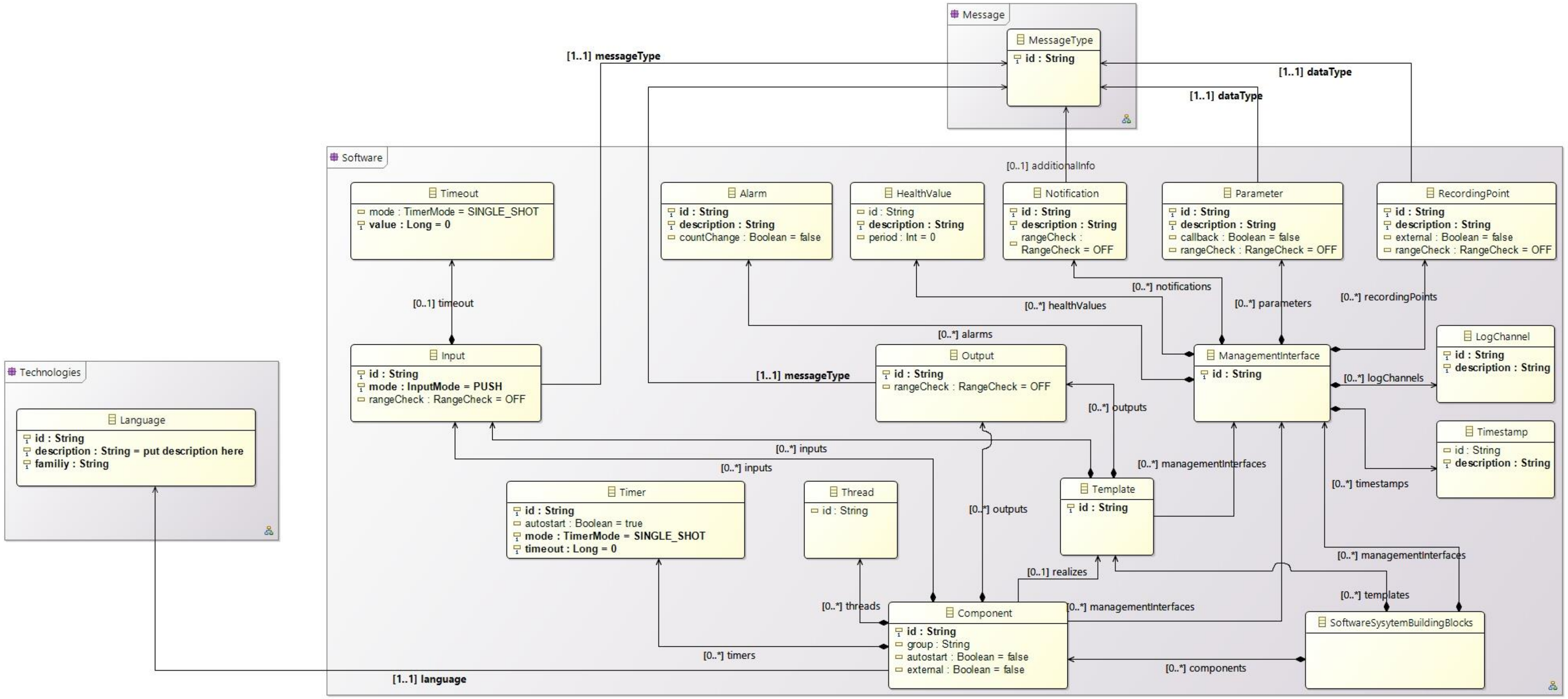
Appendix A O2 Domain Models

In this appendix the metamodel for O2 system models shown, which is divided in the groups; Software System, Software, Hardware System, Hardware, Mapping and Technology. Every model shows in detail the concepts/elements that can be used and the relations/dependencies between them.

Software System

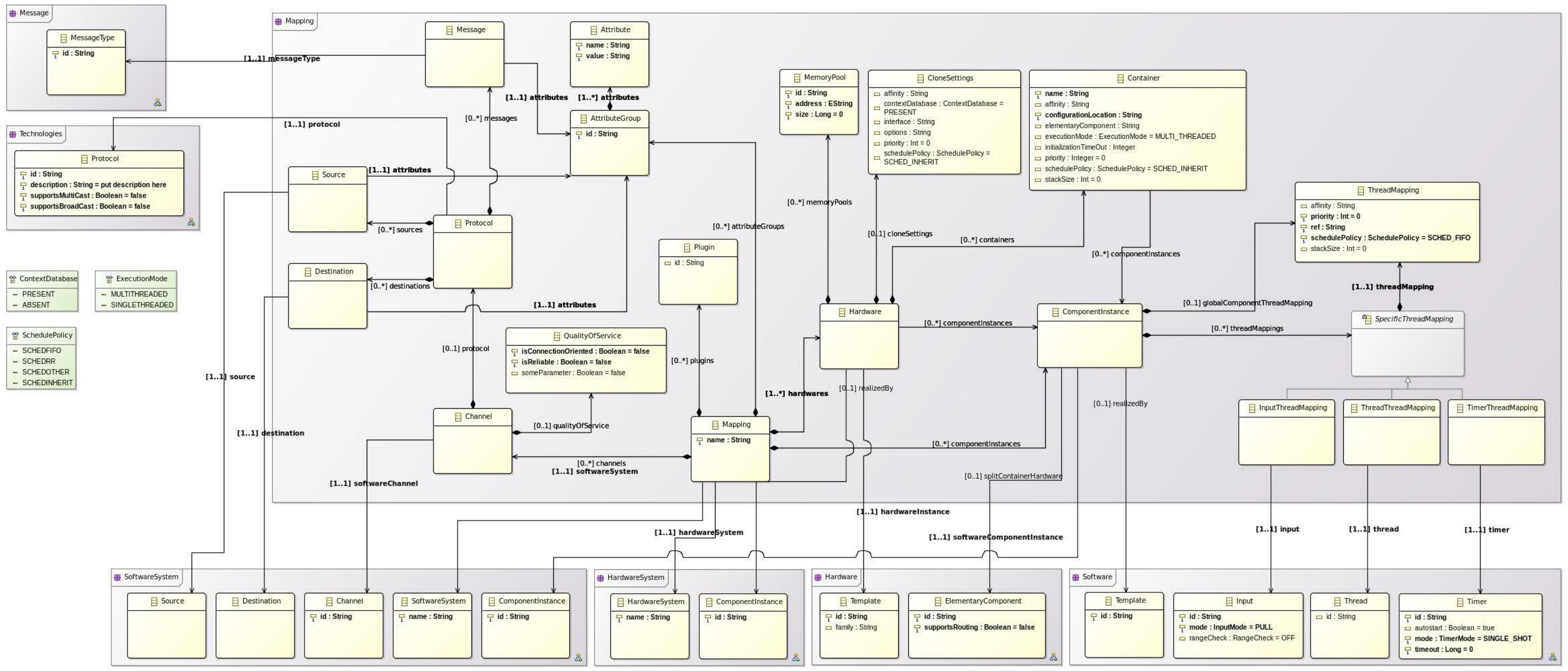


Software

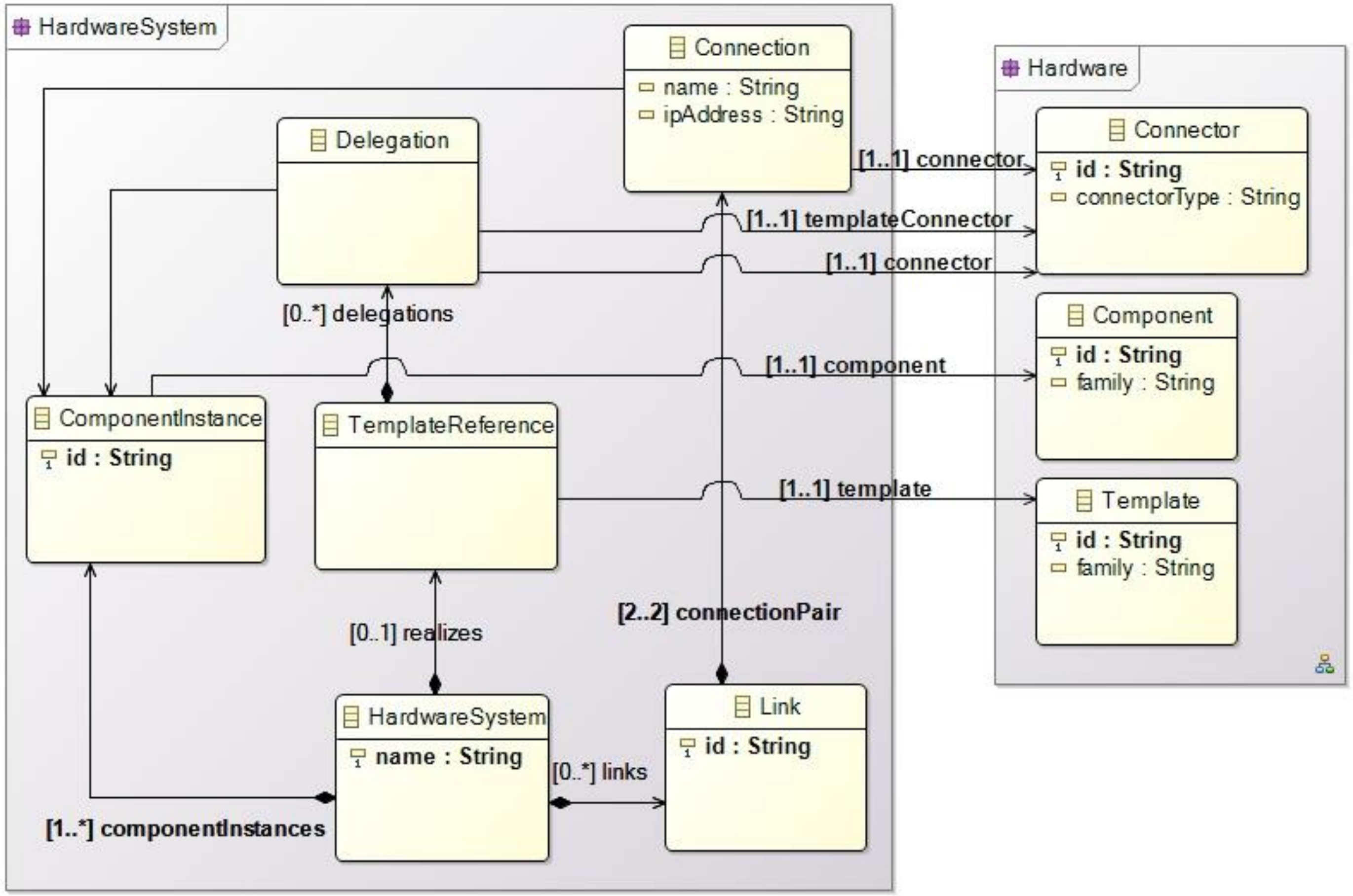


Message
MessageType
id : String
[1..1] messageType
[1..1] dataType
[1..1] dataType
[0..1] additionalInfo
Software
Timeout
mode : TimerMode = SINGLE_SHOT
value : Long = 0
Alarm
id : String
description : String
countChange : Boolean = false
HealthValue
id : String
description : String
period : Int = 0
Notification
id : String
description : String
rangeCheck : RangeCheck = OFF
Parameter
id : String
description : String
callback : Boolean = false
rangeCheck : RangeCheck = OFF
RecordingPoint
id : String
description : String
external : Boolean = false
rangeCheck : RangeCheck = OFF
[0..1] timeout
[0..*] notifications
[0..*] healthValues
[0..*] parameters
[0..*] recordingPoints
[0..*] alarms
Technologies
Language
id : String
description : String = put description here
familiy : String
Input
id : String
mode : InputMode = PUSH
rangeCheck : RangeCheck = OFF
[1..1] messageType
Output
id : String
rangeCheck : RangeCheck = OFF
ManagementInterface
id : String
LogChannel
id : String
description : String
[0..*] logChannels
[0..*] outputs
Timestamp
id : String
description : String
[0..*] inputs
[0..*] inputs
[0..*] managementInterfaces
[0..*] timestamps
Timer
id : String
autostart : Boolean = true
mode : TimerMode = SINGLE_SHOT
timeout : Long = 0
Thread
id : String
[0..*] outputs
Template
id : String
[0..*] managementInterfaces
[0..1] realizes
[0..*] templates
[0..*] threads
[0..*] managementInterfaces
Component
id : String
group : String
autostart : Boolean = false
external : Boolean = false
SoftwareSysytemBuildingBlocks
[0..*] timers
[0..*] components
[1..1] language

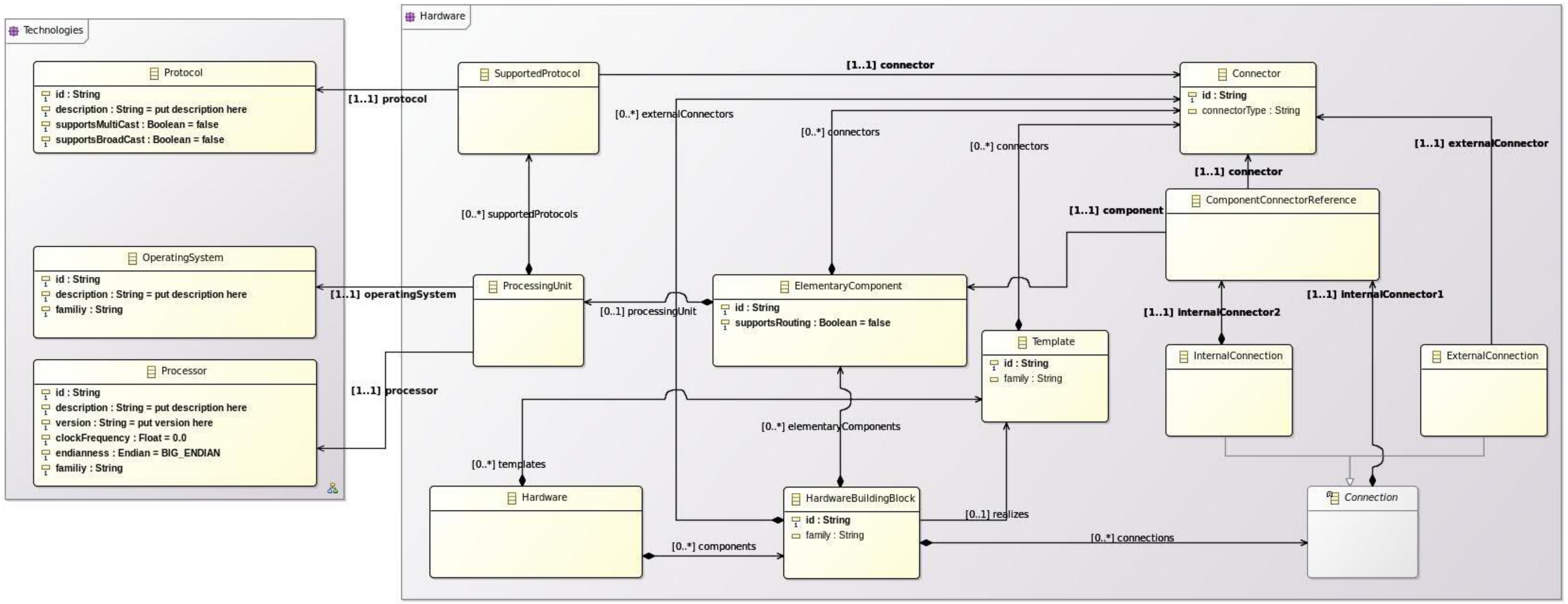
Mapping


Message
MessageType
id : String
[1..1] messageType
Mapping
Message
Attribute
name : String
value : String
[1..1] attributes
[1..*] attributes
[0..*] messages
AttributeGroup
id : String
Technologies
Protocol
id : String
description : String = put description here
supportsMultiCast : Boolean = false
supportsBroadCast : Boolean = false
[1..1] protocol
Source
[1..1] attributes
[0..*] sources
Protocol
Destination
[0..*] destinations
[1..1] attributes
ContextDatabase
PRESENT
ABSENT
ExecutionMode
MULTITHREADED
SINGLETHREADED
SchedulePolicy
SCHEDFIFO
SCHEDRR
SCHEDOTHER
SCHEDINHERIT
[0..1] protocol
QualityOfService
isConnectionOriented : Boolean = false
isReliable : Boolean = false
someParameter : Boolean = false
Plugin
id : String
[0..*] plugins
[0..*] attributeGroups
MemoryPool
id : String
address : EString
size : Long = 0
[0..*] memoryPools
CloneSettings
affinity : String
contextDatabase : ContextDatabase = PRESENT
interface : String
options : String
priority : Int = 0
schedulePolicy : SchedulePolicy = SCHED_INHERIT
[0..1] cloneSettings
Container
name : String
affinity : String
configurationLocation : String
elementaryComponent : String
executionMode : ExecutionMode = MULTI_THREADED
initializationTimeOut : Integer
priority : Integer = 0
schedulePolicy : SchedulePolicy = SCHED_INHERIT
stackSize : Int = 0
[0..*] containers
[0..*] componentInstances
ThreadMapping
affinity : String
priority : Int = 0
ref : String
schedulePolicy : SchedulePolicy = SCHED_FIFO
stackSize : Int = 0
[1..1] threadMapping
Hardware
[0..*] componentInstances
ComponentInstance
[0..1] globalComponentThreadMapping
[0..*] threadMappings
SpecificThreadMapping
[1..*] hardwares
[0..1] realizedBy
[0..1] realizedBy
InputThreadMapping
ThreadThreadMapping
TimerThreadMapping
Channel
[0..1] qualityOfService
Mapping
name : String
[0..*] componentInstances
[0..*] channels
[1..1] softwareSystem
[1..1] source
[1..1] destination
[1..1] softwareChannel
[0..1] splitContainerHardware
[1..1] hardwareInstance
[1..1] hardwareSystem
[1..1] softwareComponentInstance
[1..1] input
[1..1] thread
[1..1] timer
SoftwareSystem
Source
Destination
Channel
id : String
SoftwareSystem
name : String
ComponentInstance
id : String
HardwareSystem
HardwareSystem
name : String
ComponentInstance
id : String
Hardware
Template
id : String
family : String
ElementaryComponent
id : String
supportsRouting : Boolean = false
Software
Template
id : String
Input
id : String
mode : InputMode = PULL
rangeCheck : RangeCheck = OFF
Thread
id : String
Timer
id : String
autostart : Boolean = true
mode : TimerMode = SINGLE_SHOT
timeout : Long = 0

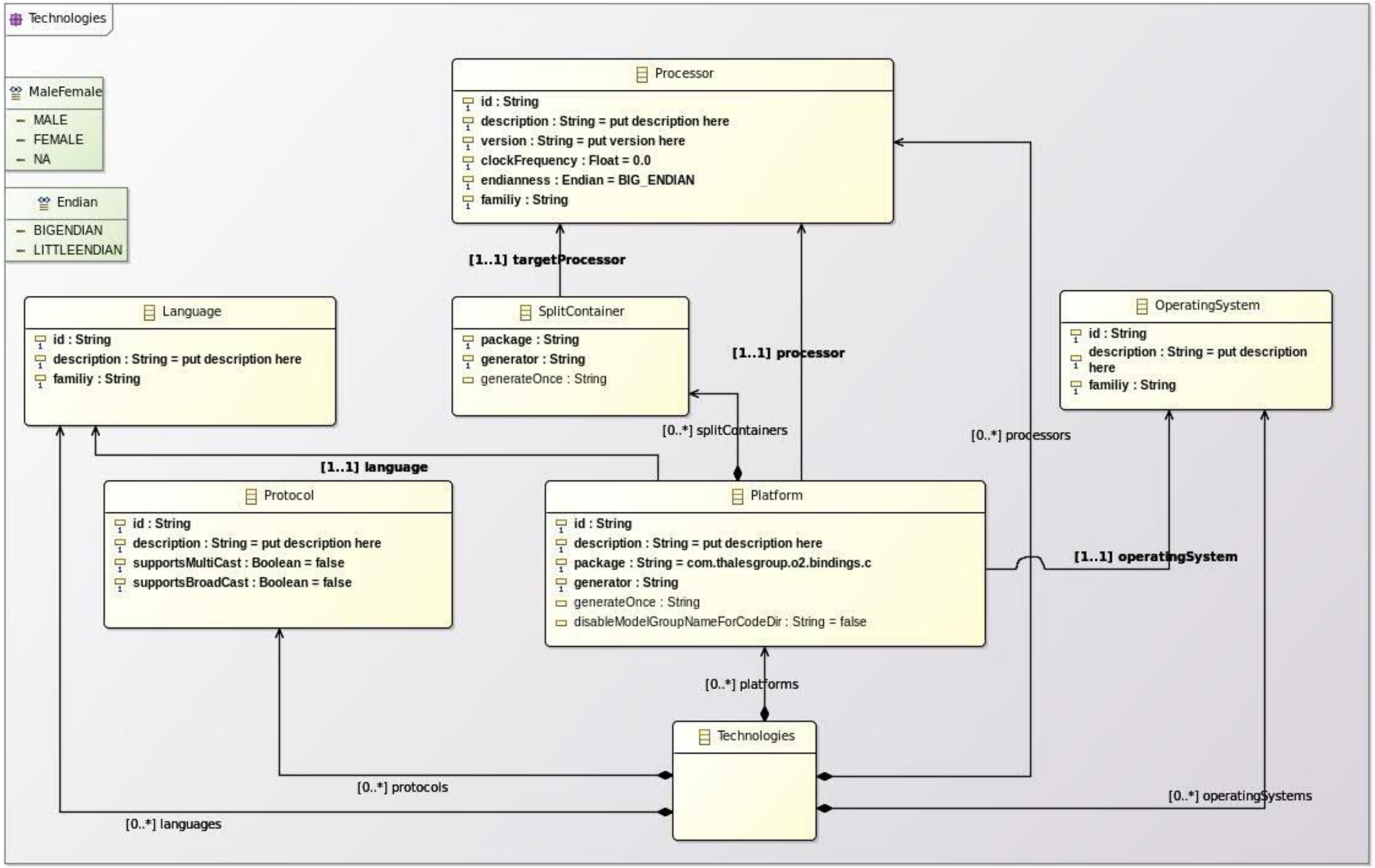
Hardware System



Hardware


Technologies
Protocol
id : String
description : String = put description here
supportsMultiCast : Boolean = false
supportsBroadCast : Boolean = false
OperatingSystem
id : String
description : String = put description here
familiy : String
Processor
id : String
description : String = put description here
version : String = put version here
clockFrequency : Float = 0.0
endianness : Endian = BIG_ENDIAN
familiy : String
Hardware
SupportedProtocol
[1..1] protocol
[1..1] connector
Connector
id : String
connectorType : String
[0..*] externalConnectors
[0..*] connectors
[0..*] connectors
[1..1] externalConnector
[1..1] connector
ComponentConnectorReference
[1..1] component
[0..*] supportedProtocols
ProcessingUnit
[1..1] operatingSystem
[0..1] processingUnit
ElementaryComponent
id : String
supportsRouting : Boolean = false
Template
id : String
family : String
[1..1] internalConnector2
[1..1] internalConnector1
InternalConnection
ExternalConnection
[1..1] processor
[0..*] elementaryComponents
[0..*] templates
Hardware
HardwareBuildingBlock
id : String
family : String
[0..1] realizes
[0..*] components
[0..*] connections
Connection

Technology


Technologies
MaleFemale
MALE
FEMALE
NA
Endian
BIGENDIAN
LITTLEENDIAN
Processor
id : String
description : String = put description here
version : String = put version here
clockFrequency : Float = 0.0
endianness : Endian = BIG_ENDIAN
familiy : String
[1..1] targetProcessor
Language
id : String
description : String = put description here
familiy : String
SplitContainer
package : String
generator : String
generateOnce : String
[1..1] processor
OperatingSystem
id : String
description : String = put description here
familiy : String
[0..*] splitContainers
[0..*] processors
[1..1] language
Protocol
id : String
description : String = put description here
supportsMultiCast : Boolean = false
supportsBroadCast : Boolean = false
Platform
id : String
description : String = put description here
package : String = com.thalesgroup.o2.bindings.c
generator : String
generateOnce : String
disableModelGroupNameForCodeDir : String = false
[1..1] operatingSystem
[0..*] platforms
Technologies
[0..*] protocols
[0..*] operatingSystems
[0..*] languages

Appendix B Protocol Attributes

The possible attributes that can be defined per protocol/communication technology.

UDP

Attribute	Key	Value	Example	Type
Buffer size maximum of a channel	buffermaximum	integer [bytes] Must be equal or greater than the real message size (Note: 65535 is maximum) **65535**	4096	D
Socket write buffer size (OS platform dependant)	bufferreceive	integer [bytes]		D
Socket read buffer size (OS platform dependant)	buffersend	integer [bytes]		S

Whether bytebuffer type is sent completely or buffer only.
struct {
unsigned int _maximum;
unsigned int _length;
char *_buffer;
} bb;
	bytebuffer	boolean
		0 : send as specified
		1 : first 8 bytes (fields *length* and *maximum*) are removed, only send *buffer* part over the line.
			1	S, D

Endianness	endian	**b** : big endian l : little endian	b	S, D
Address where the data is send from	**fromaddress**	String	localhost	S
Port where the data is sent from	**fromport**	Integer (0 – 65535)	50003	S
Header size	**headersize**	Extra header size at start of message. This can be used for the msgid. **0**	1	S, D
Message discriminator choice	**idnoid**	boolean **0** : msgid, idoffset and idsize must be specified per messagetype 1 : only one type of message on the channel	1	S, D
Message discriminator position (OS platform dependant). Needed when a channel contains multiple types.	**idoffset**	integer [bytes] (after encoding to bytebuffer) Must be specified if idnoid=0	0	S, D
Message discriminator size. Needed when a channel contains multiple types.	**idsize**	integer [bytes] (values 1,2 or 4) Must be specified if idnoid=0	4	S, D
Message discriminator value. Needed for types that are part of multi type channels.	**msgid**	integer (unique value per messagetype) Must be specified if idnoid=0	20	T
Allow to have multiple messages in one UDP frame	multiplemessage	boolean **0** : disabled 1 : enabled	0	D
Queue size of a channel (valid in PULL mode only)	queuesize	integer [number of messages] (**10**)	1	D
Address of destination of a channel	receiveaddress	String (equal to sendaddress)	localhost	D
Port of destination of a channel	**receiveport**	Integer (0 – 65535) (equal to sendport)	50002	D
Address of destination of a channel	**sendaddress**	String (equal to receiveaddress)	localhost	S
Port of destination of a channel	**sendport**	Integer (0 – 65535) (equal to receiveport)	50001	S
Optimize (compress) data on interface	optimize	boolean **0** : disabled 1 : enabled	0	S, D

UDP MULTICAST

Attribute	Key	Value	Example	Type
Buffer size maximum of a channel	buffermaximum	integer [bytes] Must be equal or greater than the real message size (Note: 65535 is maximum) **65535**	4096	D
Socket write buffer size (OS platform dependant)	bufferreceive	integer [bytes]		D
Socket read buffer size (OS platform dependant)	buffersend	integer [bytes]		S

Whether bytebuffer type is sent completely or buffer only.
struct {
unsigned int _maximum;
unsigned int _length;
char *_buffer;
} bb;
	bytebuffer	boolean
		0 : send as specified
		1 : first 8 bytes (fields *length* and *maximum*) are removed, only send *buffer* part over the line.
			1	S, D

Endianness	endian	**b** : big endian l : little endian	b	S, D
Address where the data is send from	**fromaddress**	String	localhost	S
Port where the data is sent from	**fromport**	Integer (0 – 65535)	50003	S
Header size	**headersize**	Extra header size at start of message. This can be used for the msgid. **0**	1	S, D
Message discriminator choice	**idnoid**	boolean **0** : msgid, idoffset and idsize must be specified per messagetype 1 : only one type of message on the channel	1	S, D
Message discriminator position (OS platform dependant). Needed when a channel contains multiple types.	**idoffset**	integer [bytes] (**0**) (after encoding to bytebuffer) Must be specified if idnoid=0	0	S, D
Message discriminator size. Needed when a channel contains multiple types.	**idsize**	integer [bytes] (values 1,2 or 4) Must be specified if idnoid=0	4	S, D
Message discriminator value. Needed for types that are part of multi type channels.	**msgid**	integer (unique value per messagetype) Must be specified if idnoid=0	20	T
Allow to have multiple messages in one UDP frame	multiplemessage	boolean **0** : disabled 1 : enabled	0	D
Multicast address to listen to	**multicastaddress**	string starting with 224.	224.0.2.4	D
Multicast interface to listen to	**multicastinterface**	string	testpc1	D
Multicast port to listen to	**multicastport**	integer (0 - 65535) (equal to sendport)	50001	D
Queue size of a channel (valid in PULL mode only)	queuesize	integer [number of messages] (**10**)	1	D
Address of destination of a channel	receiveaddress	String (equal to sendaddress)	localhost	D
Port of destination of a channel	**receiveport**	integer (0 - 65535)	50002	D
Address of destination of a channel	**sendaddress**	String (equal to receiveaddress)	localhost	S
Interface of destination of a channel (multicast only)	**sendinterface**	string	testpc1	S
Loop (multicast)	sendloop	integer boolean (0 or 1)	0	S
Port of destination of a channel	**sendport**	Integer (0 – 65535) (equal to multicastport)	50001	S
Time to live (multicast)	sendttl	integer [hops] (**0**)		S
Optimize (compress) data on interface	optimize	boolean **0** : disabled 1 : enabled	0	S,D

TCP

Attribute	Key	Value	Example	Types
Buffer size maximum of a channel	buffermaximum	integer [bytes] Must be equal or greater than the real message size **65535**	4096	D
Connection server, client must be paired per connection	**connection-type**	string Push : client → server Pull : server → client	"server" "client"	S, D
Endianness	endian	**b** : big endian l : little endian	b	S, D
Address where the data is send from	**fromaddress**	String	localhost	S
Port where the data is sent from	**fromport**	Integer (0 – 65535)	50003	S
Header size	**headersize**	Extra header size at start of message. This can be used for the msgid, length and synccode. **0**	1	S, D
Message discriminator position (OS platform dependant). Needed when a channel contains multiple types.	**idnoid**	integer [bytes] **0** : msgid, idoffset and idsize must be specified per messagetype 1 : only one type of message on the channel	1	S, D
Message discriminator size. Needed when a channel contains multiple types.	**idoffset**	integer [bytes] (after encoding to bytebuffer) Must be specified if idnoid=0	0	S, D
Message discriminator value. Needed for types that are part of multi type channels.	**idsize**	integer [bytes] (values 1,2 or 4) Must be specified if idnoid=0	4	S, D
Message length is fixed size	lengthfixed	Send fixed size messages with lengthfixed size.	off	S, D
Points to a length value in the message	**lengthoffset**	Integer [bytes] (after encoding to bytebuffer) Must be specified if lengthfixed is not specified.	4	S, D
Message length	**lengthsize**	Integer [bytes] (values 1,2 or 4) Must be specified if lengthfixed is not specified.	4	S, D
Message discriminator value	**msgid**	integer (unique value per messagetype) Must be specified if idnoid=0	20	T
Queue size of a channel (valid in PULL mode only)	queuesize	integer [number of messages] (**10**)	1	D
Address of destination of a channel (client side)	**sendaddress**	string	localhost	S or D
Port of destination of a channel (client side)	**sendport**	integer (0 - 65535)	50001	S or D
Synchronization code in hex	**tcpsynccode**	String with hex values	ABC123	S, D
Points to a unique value in the message (for tcp synchronization purposes in multitype channels).	**tcpsync-position**	Integer [bytes] (after encoding to bytebuffer)	0	S, D
Write ignore error	**writeignoreerror**	When set ignore all write errors (**0**)	0	S
Tcp max output queue	**tcpmaxoutputqueue**	The maximumum queue size after which a tcp connection is close	16384	S

POOL (SHARED MEMORY)

Attribute	Key	Value	Example	Type
Number of entries in shared memory pool	**poolsize**	Integer (**10**)	10	S, D
Unique name of memory pool *	**poolname**	string	my-pool	S, D
Queue size of a channel (valid in PULL mode only)	**queuesize**	integer [number of messages] (**10**)	1	D

* pool name must match pool id in <mapping.xml><mapping><hardware ><pool id="">

CLONE

Attribute	Key	Value	Example	Type
Clone domain	**domain**	string (must be unique)	result_domain	T
Clone keys	**keys**	string (can be empty) (see explanation below)	""	T
Quality-of-service	**qos**	string (empty string gives "TH_DEFAULT_QOS") (see explanation below)	"TH_DEFAULT_QOS"	T
Queue size of a channel (valid in PULL mode only)	**queuesize**	integer [number of messages] (**10**)	1	T

Clone Keys
The *keys* attribute can hold names of fields of the message types that are transported in the *domain*. Several names can be put in the key attribute, separated by comma's. They can be used as sort keys for the data in the clone database (see also below at the TREE keyword). When the database type is QUEUE or ONEPLACE the key is ignored.
Use a point separator for nested data structures e.g. "position.X, position.Y"

Clone quality of service
The behavior of Clone can be tuned by means of its quality of service key. The default value "TH_DEFAULT_QOS" is just one set out of many possibilities. Below all tuning parameters are listed:

Delivery:
- **GUARANTEED** Data is sent reliable and is received in the same sequence as transmitted.
- **BEST EFFORT** Data that is send can be lost.

Persistency:
- **VOLATILE** Data is not saved in a context memory database.
- **TRANSIENT** Data is saved in a context memory database as long as the software is running. When the database is full new messages are no longer stored. Messages defined as transient are always send to subscribers even if the subscriber is added after the message is available.
- **PERSISTENT** Data is saved in a memory database and on disk. When the database is full new messages are no longer stored. Messages defined as persistent are always send to subscribers even if the subscriber is added after the message is available.

Database:
- **TREE** Database is tree sorted (on key). This key is a specific message field defined by the user. If no key is defined only 1 element of a specific message type will exist (see ONEPLACE).
- **QUEUE** Database is sorted on sequence number. This sequence number is an internal number generated by Clone.
- **ONEPLACE** Database contains only 1 element of a specific message type. Multiple messages can exist in the database but only 1 of each type.

Latency budget time - Data is stored in a memory buffer for a maximum of latency budget time or if this buffer is full. After the time field the value ns (nano second), us (u second), ms (milli second), of s (second) can be specified. Default is 0ns. Age time Data is stored in the subject database with a maximum of Age time. After this time the garbage collector will remove the data. After the time field the value ns (nano second), us (u second), ms (milli second), of s (second) can be specified. Default is 0 which means unlimited.

Size size - The maximum number of subjects that can be stored in the memory database. New subjects will be deleted. This is useful for a queue database. After the size field the value k (1024), m (1024*1024) or g (1024*1024*1024) can be specified. Default is 0 which means unlimited.

Memory size - The maximum memory size of the subjects in the memory database. New subjects will be deleted. This is useful for a queue database. After the size field the value k (1024), m (1024*1024) or g (1024*1024*1024) can be specified. Default is 0 which means unlimited.

There are 4 predefined types:
TH_DEFAULT_QOS: GUARANTEED, VOLATILE, TREE, Latency budget 0, Age 0, Size 0, Memory 0
TH_SAMPLE_QOS : BEST EFFORT, VOLATILE, TREE, Latency budget 0, Age 0, Size 0, Memory 0
TH_STATE_QOS : GUARANTEED, TRANSIENT, TREE, Latency budget 0, Age 0, Size 0, Memory 0
TH_SETTING_QOS: GUARANTEED, PERSISTENT, TREE, Latency budget 0, Age 0, Size 0, Memory 0

Appendix C Textual Grammar of O2 DSL

The complete grammar definition of O2 DSL in Xtext notation.

O2.xtext

```
grammar com.thalesgroup.o2.xtext.O2 with org.eclipse.xtext.common.Terminals

generate o2 "http://www.thalesgroup.com/o2/xtext/O2"


O2File:
    {O2File}
    packages+=Package*
;

Package:
  'package' name=QualifiedName '{'
    imports+=Import*
    elements+=O2Element*
  '}'
;

Import:
  'import' importedNamespace=QualifiedNameWithWildcard
;

O2Element:
    DataType | ManagementInterface | Component | Template | SoftwareSystem | AbstractHardwareBuildingBlock | HardwareSystem | Mapping
;

Mapping:
    'mapping' name=ID '{'
            (hardware+=Hardware | channels+=ProtocolChannel | protocolAttributes+=NamedProtocolAttributes)*
    '}'
;

Hardware:
    'hardware' componentInstance=[HardwareComponentInstance|QualifiedName] '{'
            (containers+=Container)*
    '}'
;

ProtocolChannel:
    'channel' channel=[Channel|QualifiedName] 'use' protocol=ProtocolAttributeAssignment
;

ProtocolAttributeAssignment:
    UDPProtocolAttributeAssignment
;

UDPProtocolAttributeAssignment:
    'UDP' '{'
            'sourceAttributes:' sourceAttributes=[NamedUDPSourceAttributes|QualifiedName]
            'destination' componentInstance=[ComponentInstance|QualifiedName]':'input=[Input|QualifiedName] 'useAttributes:' destinationAttributes=[NamedUDPDestinationAttributes|QualifiedName]
    '}'
;

NamedProtocolAttributes:
    NamedUDPSourceAttributes | NamedUDPDestinationAttributes
;

NamedUDPDestinationAttributes:
    'UPDDestinationAttributes' name=ID '{'
            attributes=UDPDestinationAttributes
    '}'
;

UDPDestinationAttributes:
    'port:' port=INT
    'idNoId:' (idNoId?='true'|'false')
;

NamedUDPSourceAttributes:
    'UDPSourceAttributes' name=ID '{'
            attributes=UDPSourceAttributes
    '}'
;

UDPSourceAttributes:
    'address:' address=STRING
    'port:' port=INT
    'idNoId:' (idNoId?='true'|'false')
;

CloneChannel:
    'CLONEChannel' channel=[Channel|QualifiedName] '{'
            'type:' type=AssignableDataType
            'domain:' domain=STRING
            'keys:' keys=STRING
            'qos:' qos=STRING
    '}'
;

UDPDestionationSettings:
    'destionation' componentInstance=[ComponentInstance|QualifiedName]'-'input=[Input|QualifiedName] '{'
            'port:' port=INT
            'idNoId:' (idNoId?='true'|'false')
    '}'
;

Container:
    'container' name=ID '{'
            'configurationLocation:' configurationLocation=ID
            ('initializationTimeOut:' initializationTimeOut=INT)?
            ('priority:' priority=INT)?
            'componentInstances:' componentInstances+=[ComponentInstance|QualifiedName] (',' componentInstances+=[ComponentInstance|QualifiedName])*
    '}'
;

HardwareSystem:
    'hardwareSystem' name=ID (':' realizes=[HardwareBuildingBlockTemplate|QualifiedName])? '{'
            (componentInstances+=HardwareComponentInstance | links+=HardwareLink | delegations+=HardwareTemplateDelegations)*
    '}'
;

HardwareTemplateDelegations:
    'delegate' templateConnector=[HardwareBuildingBlockTemplateConnector|QualifiedName] ':' componentInstance=[HardwareComponentInstance]'.'connector=[ExternalConnector|QualifiedName]
;

HardwareLink:
    'link' name=ID '{'
            connection1=HardwareLinkConnection
            connection2=HardwareLinkConnection
    '}'
;

HardwareLinkConnection:
    'connection' name=ID '{'
            'ipAddress:' ipAddress=STRING
            'connector:' componentInstance=[HardwareComponentInstance]'.'connector=[ExternalConnector|QualifiedName]
    '}'
;

HardwareComponentInstance:
    'hardware' name=ID ':' component=[HardwareBuildingBlock|QualifiedName]
;

AbstractHardwareBuildingBlock:
    HardwareBuildingBlock | HardwareBuildingBlockTemplate
;

HardwareBuildingBlockTemplate:
    'hardwareBuildingBlockTemplate' name=ID '{'
            ('family:' familiity=STRING)?
            (connectors+=HardwareBuildingBlockTemplateConnector)*
    '}'
;

HardwareBuildingBlock:
    'hardwareBuildingBlock' name=ID (':' realizes=[HardwareBuildingBlockTemplate|QualifiedName])? '{'
            ('family:' familiity=STRING)?
            (connectors+=HardwareBuildingBlockConnector | components+=AbstractHardwareComponent | connections+=AbstractConnection)*
    '}'
;

AbstractConnection:
    InternalConnection | ExternalConnection
;

InternalConnection:
    'internalConnection' internalConnector=[InternalConnector|QualifiedName] 'to' toInternalConnector=[InternalConnector|ID]
;

ExternalConnection:
    'externalConnection' internalConnector=[InternalConnector|QualifiedName] 'to' toExternalConnector=[ExternalConnector|ID]
;

AbstractHardwareComponent:
    ElementaryComponent | ProcessingUnitComponent
;

ProcessingUnitComponent:
    supportsRouting?='routingSupported processingUnitComponent'|'processingUnitComponent' name=ID '{'
            'processor:' processor=ID
            'operatingSystem:' operatingSystem=ID
            (connectors+=ProcessingUnitComponentConnector)*
     '}'
;

ElementaryComponent:
     supportsRouting?='routingSupported elementaryComponent'|'elementaryComponent' name=ID '{'
            (connectors+=ElementaryComponentConnector)*
     '}'
;

InternalConnector:
    ProcessingUnitComponentConnector | ElementaryComponentConnector
;

ExternalConnector:
    HardwareBuildingBlockTemplateConnector | HardwareBuildingBlockConnector
;

ProcessingUnitComponentConnector:
    'connector' name=ID (':' type=ID)? ('supports' supportedProtocol+=ID (',' supportedProtocol+=ID)*)?
;

HardwareBuildingBlockTemplateConnector:
    'connector' name=ID (':' type=ID)?
;

ElementaryComponentConnector:
    'connector' name=ID (':' type=ID)?
;

HardwareBuildingBlockConnector:
    'connector' name=ID (':' type=ID)?
;
```

```

SoftwareSystem:
    'softwareSystem' name=ID (':' realizes=[Template|QualifiedName])? '{'
            ('use:' managementInterfaces+=[ManagementInterface|QualifiedName] (','
managementInterfaces+=[ManagementInterface|QualifiedName])*)?
            componentInstances+=ComponentInstance*
            (delagations+=Delegation)*
            channels+=Channel*
    '}'
;

Delegation:
    InputDelegation | OutputDelegation
;

InputDelegation:
    'inputDelegate' templateInput=[Input|ID] ':'
componentInstance=[ComponentInstance|ID]'.'input=[Input|ID]
;

OutputDelegation:
    'outputDelegate' templateOutput=[Output|ID] ':'
componentInstance=[ComponentInstance|ID]'.'output=[Output|ID]
;

Channel:
    'channel' name=ID '{'
            sources+=Source+
            destinations+=Destination+
    '}'
;

Destination:
    'destination' componentInstance=[ComponentInstance] ':' input=[Input|ID]
;

Source:
    'source' componentInstance=[ComponentInstance] ':' output=[Output|ID]
;

ComponentInstance:
    'instance' name=ID ':' component=[Component|QualifiedName]
;

Component:
    external?='external component'|'component' name=ID (':'
realizes=[Template|QualifiedName])? '{'
            'language:' language=STRING
            ('group:' group=ID)?
            ('autostart:' autostart?='true'|'false')?
            ('use:' managementInterfaces+=[ManagementInterface|QualifiedName] (','
managementInterfaces+=[ManagementInterface|QualifiedName])*)?
            threads+=Thread*
            timers+=Timer*
            inputs+=Input*
            outputs+=Output*
    '}'
;

Output:
    'output' name=ID ':' messageType=AssignableDataType ('{'
            ('rangeCheck:' rangeCheck=RANGECHECK)?
    '}')?
;

Input:
    'input' name=ID ':' messageType=AssignableDataType '{'
            'mode:' mode=INPUTMODE
            ('timeout:' timeout=INT)?
            ('timerMode:' timerMode=TIMERMODE)?
            ('rangeCheck:' rangeCheck=RANGECHECK)?
    '}'
;

Timer:
    'timer' name=ID '{'
            'mode:' mode=TIMERMODE
            'timeout:' timeout=INT
            ('autostart:' autostart?='true'|'false')?
    '}'
;

Thread:
    'thread' name=ID
;

Template:
    'template' name=ID '{'
            ('use:' managementInterfaces+=[ManagementInterface|QualifiedName] (','
managementInterfaces+=[ManagementInterface|QualifiedName])*)?
            (inputs+=Input|outputs+=Output)+
    '}'
;

DataType:
    SimpleType|ComplexType;

SimpleType:
    'simpleType' name=ID ':'  base=XSDDATATYPE '{'
            ('documentation:' documentation=STRING)?
            'minInclusive:' minInclusiveValue=STRING
            'maxInclusive:' maxInclusiveValue=STRING
    '}'
;

ComplexType:
    'complexType' name=ID '{'
            ('documentation:' documentation=STRING)?
            elements+=DataElement+
    '}'
;

DataElement:
    'element' name=ID ':' type=AssignableDataType ('{'
                    'minOccurs:' minOccurs=INT
                    'maxOccurs:' maxOccurs=INT
            '}')?
;

AssignableDataType:
    xsdType=XSDDATATYPE | otherType=[DataType|QualifiedName]
;

ManagementInterface:
    'managementInterface' name=ID '{'
    (alarms+=Alarm | healthValues+=HealthValue | notifications+=Notification |
parameters+=Parameter
            | recordingPoints+=RecordingPoint | logChannels+=LogChannel |
timeStamps+=Timestamp
    )*
    '}'
;

Timestamp:
    'timestamp' name=ID '{'
            'description:' description=STRING
    '}'
;

LogChannel:
    'logChannel' name=ID '{'
            'description:' description=STRING
    '}'
;

RecordingPoint:
    external?='external recordingPoint'|'recordingPoint' name=ID ':'
dataType=AssignableDataType '{'
            'description:' description=STRING
            ('rangeCheck:' rangeCheck=RANGECHECK)?
    '}'
;

Parameter:
    (callback?='callback parameter'|'parameter') name=ID ':' dataType=AssignableDataType
'{'
            'description:' description=STRING
            ('rangeCheck:' rangeCheck=RANGECHECK)?
    '}'
;

Notification:
    'notification' name=ID (':' additionalInfoType=AssignableDataType)?'{'
            'description:' description=STRING
            ('rangeCheck:' rangeCheck=RANGECHECK)?
    '}'
;

HealthValue:
    'healthValue' name=ID '{'
            'description:' description=STRING
            ('period:' period=INT)?
    '}'
;

Alarm:
    'alarm' name=ID '{'
            'description:' description=STRING
            ('countChange:' countChange?='true'|'false')?
    '}'
;

enum RANGECHECK: OFF | VALIDATE | ERROR;

enum XSDDATATYPE: string | boolean | decimal | float | double | duration | dateTime | time
| date | long | int | short | byte;

enum TIMERMODE: SINGELESHOT | PERIODIC;

enum INPUTMODE: PUSH | PULL;

QualifiedName:
  ID ('.' ID)*
;

QualifiedNameWithWildcard:
  QualifiedName '.*'?
;
```

Grammar org.eclipse.xtext.common.Terminals

The basis grammar (provided by Xtext) that is inherited by the grammar of O2 DSL.

Terminals.xtext

```
/*******************************************************************************
 * Copyright (c) 2008 itemis AG and others.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *******************************************************************************/
grammar org.eclipse.xtext.common.Terminals hidden(WS, ML_COMMENT, SL_COMMENT)

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

terminal ID               : '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
terminal INT returns ecore::EInt: ('0'..'9')+;
terminal STRING   :
                  '"' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|'"') )* '"' |
                  "'" ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|"'") )* "'"
           ;
terminal ML_COMMENT       : '/*' -> '*/';
terminal SL_COMMENT       : '//' !('\n'|'\r')* ('\r'? '\n')?;

terminal WS                      : (' '|'\t'|'\r'|'\n')+;

terminal ANY_OTHER: .;
```

Appendix D ABC Example With The DSL Implementation

In this appendix the software system configuration of the ABC example defined with O2 DSL is presented. This software system configuration is defined in six files; messages.o2, software.o2, softwareSystem.o2, hardware.o2, hardwareSystem.o2 and mapping.o2. The content of each of these files are shown below:

messages.o2

```
package messages {
    complexType abcjavac_ABC_TYPE {
            documentation: "Input for the abc formula"
            element a : float
            element b : float
            element c : float
    }

    complexType abcjavac_RESULT_TYPE {
            documentation: "Output of the abc formula"
            element result1 : float
            element result2 : float
    }
}
```

software.o2

```
package software {

    import messages.*

    managementInterface abcInput {
            alarm userInput { description: "Example Alarm" }
    }

    managementInterface abcFormula {
            notification divide_by_zero : abcjavac_ABC_TYPE {
                    description: "divide by zero"
            }
            notification negative_discriminant : abcjavac_ABC_TYPE {
                    description: "discriminant is negative"
            }
    }

    managementInterface debug {
            logChannel abc { description: "logging abc input" }
            logChannel result { description: "logging result output" }
    }

    managementInterface params {
            callback parameter testShared : int {
                    description: "Parameter to test that parameter handling is functioning."
            }
            callback parameter test : int {
                    description: "Parameter to test that parameter handling is functioning."
            }
    }

    managementInterface rec {
            recordingPoint point1 : abcjavac_ABC_TYPE {
                    description: "records the abc values after they have been received."
            }
            recordingPoint point2 : abcjavac_RESULT_TYPE {
                    description: "records the result values before they are send."
            }
    }

    managementInterface ps {
            notification ERROR_READING_XML {
                    description: "An error has occurred while reading the parameter values
from the xml file."
            }
            notification ERROR_PROCESSING_XML {
```

```
            description: "An error has occurred while processing the parameter values that have been read from the xml file."
        }

        logChannel debug {
            description: "Debug channel for the parameter server"
        }
    }

    component abcInput {
        language: "Java"
        group: abc
        use: abcInput, debug, params

        thread abcInputThread
        timer abcInputTimer { mode:PERIODIC timeout:1000000000 }

        output abc : abcjavac_ABC_TYPE
    }

    component abcFormula {
        language: "C"
        group: abc
        use: debug, rec, abcFormula

        input abc : abcjavac_ABC_TYPE { mode:PUSH }
        output result : abcjavac_RESULT_TYPE
    }

    component resultOutput {
        language: "Java"
        group: abc
        use: debug, params

        thread esultOutputThread

        input result : abcjavac_RESULT_TYPE { mode:PULL }
    }

    component parameterServer {
        language: "C"
        group: abc
        use: ps

        thread parameter_server
    }

    component monitor {
        language: "Java"
        group: abc
        use: abcInput, abcFormula, debug, params, ps, rec

        thread monitor
    }

    component super_vision {
        language: "C"
        group: abc
        autostart: true

        thread supervision
    }
}
```

softwareSystem.o2

```
package softwaresystem {

    import software.*

    softwareSystem abc {

            instance getAbc : abcInput
            instance calculate : abcFormula
            instance showResult : resultOutput
            instance parameter_server : parameterServer
            instance monitor1 : monitor
            instance super_vision1 : super_vision

            channel abc {
                    source getAbc : abc
                    destination calculate : abc
            }

            channel result {
                    source calculate : result
                    destination showResult : result
            }
    }
}
```

hardware.o2

```
package _hardware {
    hardwareBuildingBlock PClinux_mainboard {
            family: "famility0"

            connector eth0 : RJ45

            processingUnitComponent component0 {
                    processor: Pentium4
                    operatingSystem: Linux
                    connector connector0 : _UDP supports _UDP
            }

            externalConnection component0.connector0 to eth0
    }
}
```

hardwareSystem.o2

```
package hardwaresystem {
    hardwareSystem abc {
            hardware PC : _hardware.PClinux_mainboard
    }
}
```

mapping.o2

```
package _mapping {
    import hardwaresystem.abc.*
    import softwaresystem.abc.*
    import software.abcFormula.*
    import software.resultOutput.*

    mapping abc {
            hardware PC {
                    container abc {
                            configurationLocation: abc
                            initializationTimeOut: 5
                            priority: 0
                            componentInstances: getAbc
                    }
                    container calc {
                            configurationLocation: abc
                            initializationTimeOut: 5
                            priority: 0
                            componentInstances: calculate
                    }
                    container result {
                            configurationLocation: abc
                            initializationTimeOut: 5
                            priority: 0
                            componentInstances: showResult
                    }
                    container ps {
                            configurationLocation: abc
                            initializationTimeOut: 5
                            priority: 0
                            componentInstances: parameter_server
                    }
                    container mon {
                            configurationLocation: abc
                            initializationTimeOut: 5
                            priority: 0
                            componentInstances: monitor1
                    }
                    container super {
                            configurationLocation: abc
                            initializationTimeOut: 5
                            priority: 0
                            componentInstances: super_vision1
                    }
            }

            channel abc use UDP {
                    sourceAttributes: src1_attr
                    destination calculate:abc useAttributes: dst1_attr
            }
            channel result use UDP {
                    sourceAttributes: src2_attr
                    destination showResult:result useAttributes: dst2_attr
            }

            UDPSourceAttributes src1_attr {
                    address: "localhost"
                    port: 50001
                    idNoId: true
            }
            UDPSourceAttributes src2_attr {
                    address: "localhost"
                    port: 50002
                    idNoId: true
            }
            UPDDestinationAttributes dst1_attr {
                    port: 50001
                    idNoId: true
            }
            UPDDestinationAttributes dst2_attr {
                    port: 50002
                    idNoId: true
            }
        }
}
```