DTU Compute
Department of Applied Mathematics and Computer Science

SBML Converter for ERODE

Bachelor project

SBML Converter for ERODE

Bachelor project
July 26th, 2021

By
Björn Wilting s184214

Supervisors: Andrea Vandin, Georgios Agyris and Alberto Lluch Lafuente

Cover photo: Vibeke Hempler, 2012
Published by: DTU, Department of Applied Mathematics and Computer Science, Richard Petersens Plads, Building 324, 2800 Kgs. Lyngby Denmark
www.compute.dtu.dk

Abstract

ERODE is tool to analyse various types of models using a special language format developed for this purpose. SBML is a widely used XML-based format that can represent a vast amount of different biological models. The aim of this project is to develop a converter between ERODE and SBML-qual, an SBML-extension used to represent the model types that are supported by ERODE. To develop such a converter, the project begins with an analysis of both formats and the conversion requirement. The obtained knowledge is then used to design and implement the converter. During the entire development process, the project is being tested extensively to ensure a correct conversion.

Contents

1 Introduction

Mathematical models constitute the study basis of real world systems from various scientific domains like biology, physics and computer science. The complexity of systems leads to large mathematical models with many variables and parameters which hinders our capability to manipulate them and explore their properties. ERODE (Cardellia et al. n.d.) is a tool to analyze, simulate and minimize many types of models like ordinary differential equations, chemical reaction networks, and Boolean Networks (BNs). Since ERODE uses its own language to execute these tasks, it is often necessary to translate representations of such systems into the format of ERODE.

Boolean Networks (BNs) constitute an established qualitative modelling approach for biological systems. Currently, ERODE has only limited support for BNs, wherein variables take values in the Boolean domain ({0,1}). However, support is under development for multi-valued networks (MNs), wherein variables take values in some bounded integer domain e.g. {0,1,2}. SBML (Systems Biology Markup Language) is the most common representation format for biological systems, and, particularly, SBML-qual is an extension of SBML designed for the representation of BNs and MNs.

The goal of this project is to create a converter that can translate between the formats SBML-qual and ERODE. The converter tool is developed in Java and is based on the JSBML-library (Dräger, Rodriguez, et al. 2020), a library providing framework to parse SBML files. Given that ERODE is written in Java as well, the converter is developed as a plugin, that can be integrated into ERODE and as a standalone application.

1.1 Report structure

The report is organized as follows: Section 2 presents project's objectives and the project management. Section 3 introduces the various tools and technologies used to complete the project. In section 4, a thorough analysis of both the SBML-qual and ERODE formats is given. This analysis is then used to explain how to convert between the two formats. Section 5 explains the system architecture that was designed for this conversion process. The report continues with the implementation of this system in section 6. Particularly, this section focuses on the implementation of the individual converter components and the conversion of mathematical expressions. Section 7 explains how testing was used to ensure a correct conversion, after which a demonstration of the conversion process is given with a small demo application in section 8. In section 9, a few ideas of possible extensions or optimizations to the final converter are given, before the report is wrapped up by a conclusion in section 10.

2 Project Management

Managing a project is just as important as working on the project itself, as is allows to direct the project towards a designated goal. For this project the first step was to divide it into several stages. For each stage, a milestone was defined, declaring what should be achieved during that stage. During the project, some of the original milestones eventually had to be abandoned, as it for example was discovered that the available tools were unable to achieve the desired result or that the amount of work required would extend far beyond the project's time frame. Specifically, the milestone to add support for the conversion of multi-valued networks (MNs) had to be abandoned, because ERODE's developers were unable to add support for MNs within the time-frame of this project.

The following list contains all milestones that were kept until the end of the project:

- **Familiarize with SBML and the JSBML-library.** In order to. eventually, create a converter between SBML and ERODE, it is required to understand how biological models are represented in SBML. The same goes for the JSBML-library as it is a library used to represent SBML-models in Java.
- **Create a few small JSBML demos, that can extract data from SBML files.** These demos were used as small stepping stones, towards the construction of the converter and an opportunity to experiment with the JSBML-library.
- **Create a prototype SBML converter that can generate ERODE-files from an SBML-qual input.** In this first stage, the aim was to develop a basic framework for the conversion, a proof of concept for the later stages of the converter.
- **Extend the prototype to support all kinds of BNs by adding support for all Boolean operators supported by SBML.**
- **Extend the prototype to support conversion from ERODE to SBML.**
- **Test changes and added features during development.** This is required to ensure a correct conversion between the two models representations. By testing each newly added feature and ensuring that the previously added tests still succeeds, this can be verified.
- **Extract the core functionality to a Java-library to support integration into other projects.** By decoupling the core functionality of the converter from its program shell, the library could be used as a plugin in other projects. As an example, the converter could be integrated into ERODE, to allow direct importing and exporting of SBML-files.

To achieve these milestones the project was planned, and its progress tracked, on two different levels. On project level, the milestones and a few larger tasks were planned and monitored using a Gantt-chart to visualize the overall progression. Additionally, weekly meetings with the supervisors were used to plan and track the progress of the individual milestones.

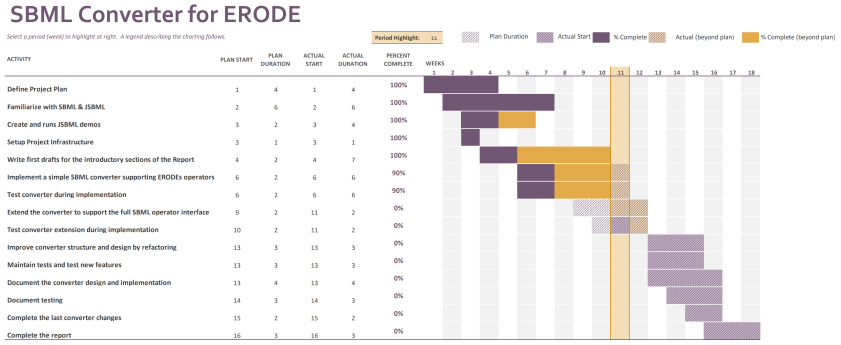


Figure 2.1: Gantt chart used to track the project's progress

The project code was managed using GitHub as a version control system and the Apache Maven (Porter, Zyl, and Lamy 2021) development tool as a code management for Java-projects (see section 3). The GitHub repository can be found at https://github.com/Unfunctioned/SBML-Converter-for-ERODE.

3 Tools and Technologies

3.1 SBML & JSBML

The Systems Biology Markup Language (SBML, Hucka et al. 2018) is a common XML-based format used to represent biological systems for computational modelling. The SBML format comes with multiple extensions that provide additional features, depending on the nature of the model. For example, variables may take values in the real, Boolean or multivalued domain. The different SBML extensions support several types of variables. This project focuses on SBML-qual, an extension used to represent both Boolean and multivalued networks.

The models represented in SBML format become available to Java using the JSBML library (Dräger, Rodriguez, et al. 2020). It is an open source pure Java API designed specifically to parse and write SBML and its extensions. When parsing an SBML-file, JSBML replicates the structure of the SBML-file using Java-objects, where each of these objects represents of the SBML-elements. Once parsing is complete, the entire structure is provided to the host program.

3.2 Apache Maven

Apache Maven (Porter, Zyl, and Lamy 2021) is a project management tool that provides a wide range of useful features for the development of Java projects. Particularly, the tool provides uniform build system, simple dependency management framework, and unit testing capabilities.

Maven projects are built using a project object model (POM) that contains all configurations, dependencies and plugins. The POM is loaded from the *pom.xml* file, and is also used to manage dependencies to external sources. Once the developer has supplied all necessary dependency configurations to Maven, it will automatically download and install the required resources. This requires that they exist in the Maven Central Repository Sonatype n.d., a public database containing millions of Java-modules. In case some resources do not exist in the repository, it is however still possible to add them manually. In addition to managing all project configurations, Maven also detects source files. This removes the need to specify, what files to use when building the project.

Maven is also a very useful tool, when it comes to testing projects, using unit test or similar, as the tool is capable of detecting implemented tests. This makes it possible to use Maven's interface to run tests and collect Maven generated test reports, instead of running tests manually.

3.3 Cucumber and the Gherkin language

Cucumber Open (SmartBear 2021) is an open source tool created to support Behaviour Driven Development (BDD), a development process that allows collaboration between programmers and non-programmers. Tests are defined used the Gherkin language, a verbose language that makes it anyone to define system tests. Once the tests are defined in *feature-files (.feature format)*, the developers can use Cucumber to implement the various steps of a test scenario. Cucumber will then, map the steps definitions from the feature file to the Java test methods executing these steps. Due to this method mapping, a step definition can be reused in other scenarios requiring the same action, making it possible to assemble scenarios in a building-block-like manner.

To define a cucumber scenario, three different types of step definitions are required. The *Given* step defines the initial state of the scenario. Once this state has been defined, a system action can be defined using the *When* keyword for the next step definition. The code mapped to the *When* statement will simulate the system action, when executing the test scenario. Finally, for a meaningful scenario it is also required to assert if the simulation ended as intended. For this, *Then* statement can be defined to define how the expected end state.

The advantage of using Cucumber instead of e.g. regular unit tests is that the building-block-like structure of cucumber scenarios allows to minimize the test code. Often used steps only need to be implemented once, because each step can be executed independently. Another advantage is that the verbose test scenarios also can be used as system documentation that does not require the ability to read code. Finally, also due to the verbose nature of the scenario definitions, the choice of words for these definitions can be selected to fit a certain type of reader. For example, if the developers are the only people reading the test documentation, some technical language may be included. On the other hand, if the intended reader is a manager or customer without the technical know-how, technical terms should generally be avoided.

```
#Some cucumber scenario examples written in Gherkin
Feature: Converting SBML files

   Scenario: Converting a valid SBML file
      Given a valid SBML file
      When the SBML file is converted to ERODE format
      Then a valid file in ERODE format has been created

   Scenario: Attempting to convert an invalid SBML file
      Given an invalid SBML file
      When the SBML file is converted to ERODE format
      Then the conversion fails
      And the converter raises an exception
```

4 Analysis

This section analyzes how Boolean Networks can be converted from SBML to ERODE format and vice versa. The topic is rather extensive, so the analysis is divided into several subsections. In section 4.1, the definitions and some examples of Boolean and multi-valued networks are given. Section 4.2 shows how these networks are represented in the SBML-qual format. Section 4.3 explains the corresponding network representation in ERODE. Finally, section 4.4 discusses the conversion between the two formats.

4.1 Boolean and Multi-valued networks

We first give a formal definitions of a Boolean Network as in Argyris et al. 2021. A Boolean network is a pair (X, F) where:

- $X = \{x_1, x_2, \ldots, x_n\}$ is a set of variables, and
- $F = \{f_{x_1}, f_{x_2}, \ldots, f_{x_n}\}$ is a set of update functions that correspond to each variable of the set X.

Particularly, $\forall i$ we have that $f_{x_i} : \mathbf{B}^n \rightarrow \mathbf{B}$ with $\mathbf{B} = \{0, 1\}$. Figure 4.1 displays a Boolean network on 3 variables i.e. $X = \{x_1, x_2, x_3\}$. We denote with **x**(t) the vector $(x_1(t), x_2(t), x_3(t))$.

$$\begin{array}{lcll} f_1(\mathbf{x}(t)) & = & x_1(t+1) & = & \neg x_3(t) \vee x_1(t) \\ f_2(\mathbf{x}(t)) & = & x_2(t+1) & = & x_1(t) \vee x_2(t) \vee \neg x_3(t) \\ f_3(\mathbf{x}(t)) & = & x_3(t+1) & = & x_2(t) \wedge \neg x_3(t) \end{array}$$

Figure 4.1: A Boolean network

A multivalued network is a pair (X, F) where X,F are a set of variables and a set of update functions respectively as in Boolean networks. In contrast with Boolean networks, the variables of multivalued networks take values in some discrete domain of the form $\{0, 1, 2, \ldots, m\}$. Hence, we have that: $\forall i\ f_{x_i} : \mathbf{D} \rightarrow \mathbf{D}_{x_i}$ with $\mathbf{D} = \prod_{i=1}^{n} \mathbf{D}_{x_i}$. Figure 4.2 displays a multivalued network:

$$x_1(t+1) = \begin{cases} 2 & (1 \leq x_1(t)) \vee (x_3(t) \geq 1 \wedge x_1(t) \geq 1) \\ 1 & x_1(t) < 1 \wedge x_3(t) \geq 1 \\ 0 & otherwise \end{cases}$$

$$x_2(t+1) = \begin{cases} 1 & x_1(t) \geq 1 \\ 0 & otherwise \end{cases}$$

$$x_3(t+1) = \begin{cases} 1 & x_2(t) \geq 1 \\ 0 & otherwise \end{cases}$$

Figure 4.2: A simple MN

In this case $\mathbf{D}_{x_1} = \{0, 1, 2\}$ whereas $\mathbf{D}_{x_2} = \mathbf{D}_{x_3} = \mathbf{B}$. If at least one variable of the set X take values in a domain $\mathbf{D} \neq \mathbf{B}$, the network is considered multivalued.

4.2 The structure of SBML-qual models

All SBML-models are organized in treelike structures, similar to how an abstract syntax tree represents the syntactic structure of a piece of code. The root of the SBML-tree is the `SBML`-element, which specifies the SBML level, the version and the extensions used to represent the model (see Listing 1 below). It also contains the `model`-element, used to represent the model. The contents of the `model`-element varies, depending on the model type and the extensions required to represent it. This project focuses on qualitative models i.e. SBML-files with the -qual extension. SBML-elements that are part of the SBML-qual extension are denoted with the `qual:`-prefix.

```
<sbml level="3" version="1"
    xmlns="http://www.sbml.org/sbml/level3/version1/core"
    xmlns:qual="http://www.sbml.org/sbml/level3/version1/..."
    qual:required="true">
        <model>
            <!-- model content-->
        </model>
</sbml>
```

Listing 4.1: Outer structure of an SBML-qual model

The primary structures used to represent SBML-qual models are the `listOfQualitativeSpecies` and the `listOfTransitions`. The former defines the model's species, the latter the model's transitions. The species are used to represent the state of the model, whereas the transitions define how the species change state.

4.2.1 Qualitative Species

Each species is defined by the following six attributes:

- The **SId** (`qual:id`) is the unique identifier of the species.
- The **Name** is an optional attribute storing the name of the species.
- The **Compartment** attribute, which indicates in which compartment the given species is located.
- The **Max Level** is another optional attribute indicating the maximum value the species can achieve in any of its states.
- The **Initial Level** an optional attribute that indicates the value of a species in its starting state.
- The **Constant**-attribute is a Boolean value indicating, whether or not the value of the species is constant.

```
<qual:qualitativeSpecies qual:id="S_1" qual:name="Species1"
    qual:compartment="comp1"
    qual:maxLevel="1" qual:initialLevel="0"
    qual:constant="false"/>
```

Listing 4.2: An example of a qualitative species in SBML-qual

4.2.2 Transitions

A transition consists of three defining structures:

- A list of `Outputs` containing IDs of the species are affected by the result of the transition.
- A list of `Inputs` containing IDs of the species that may affect the outcome of the transition.
- A list of `FunctionTerms` containing the logical expression that determine the outcome of the transition. All function terms have a *result level* that indicates, what value should be assigned to the species, in case their condition is met. There are two types of function terms in SBML-qual. The regular `functionTerm` and the `defaultTerm`. The `functionTerm` contains a logical expression, written in MathML, whereas the `defaultTerm` is an empty expression that always returns true. It acts as a last resort and assigns a default value to the species if none of the other conditions are met.

In a more legible mathematical notation, a simple multi-valued transition could look like this:

$$\text{Given the Species:} \{S_1, S_2, S_3\}$$
$$S_1 = \begin{cases} 2 & \text{if } S_2 = 1 \wedge S_3 == 2 \\ 1 & \text{if } S_3 \neq 2 \\ 0 & \text{otherwise} \end{cases}$$

In the example species S_1 is the output species, that will be assigned a new value. Species S_2 and S_3 act as input species, as they are used in the logical expressions, that determine the output value. Finally, the term *otherwise* indicates the default term, which assigns the default value. Boolean transitions can be expressed in the exact same manner as multi-valued transitions. The only difference between them is, that the Boolean transitions is limited to binary output $\{1, 0\}$ (true, false), which means that they always have a single function term in addition to the *otherwise* case. It should also be noted that function terms need to be evaluated with their result levels sorted in descending order, for a transition to be evaluated correctly.

```
<qual:transition>
  <qual:listOfInputs>
    <qual:input qual:qualitativeSpecies="S_2">
    </qual:input>
    <qual:input qual:qualitativeSpecies="S_3">
    </qual:input>
  </qual:listOfInputs>
  <qual:listOfOutputs>
    <qual:output qual:qualitativeSpecies="S_1">
    </qual:output>
  </qual:listOfOutputs>
  <qual:listOfFunctionTerms>
    <qual:defaultTerm qual:resultLevel="0">
    </qual:defaultTerm>
    <qual:functionTerm qual:resultLevel="1">
    <math xmlns="http://www.w3.org/1998/Math/MathML">
        <!--S_2 == 1 && S_3 == 2 in MathML-->
```

```
    </math>
    </qual:functionTerm>
    <qual:functionTerm qual:resultLevel="2">
      <math xmlns="http://www.w3.org/1998/Math/MathML">
      <!--S_3 != 2 in MathML-->
    </math>
    </qual:functionTerm>
  </qual:listOfFunctionTerms>
</qual:transition>
```

Listing 4.3: The transition example in SBML-qual (simplified)

For a full overview of the SBML and SBML-qual structures it is recommended to take a look at the SBML (Hucka et al. 2018) and SBML-qual (Chaouiya et al. 2015) specifications, which can be found at SBML.org (*SBML Specifications* 2019).

4.3 Representation of Boolean networks in ERODE

SBML and ERODE utilise different data structures to represent species and transitions. In ERODE, the set of values used to define a species only partially overlaps with the set of values used by SBML. However, as both formats are used to represent similar types of models, the overlapping values in the definition sets are the most important ones.

The values of an ERODE species that overlap with those of an SBML species are the *name*, *originalName* and the *initialConcentration*, although their names differ from those used in SBML. The *name* of an ERODE species and the *id* in SBML both are used to represent the variable name for that species in the model. Also, an ERODE species *originalName* attribute corresponds to the *name* attribute in SBML, since both represent the actual name of the species. The last attribute pair is the *intialConcentration* in ERODE and the *initialLevel* in SBML. Both of these attributes are used to indicate the starting value for the species.

The transition representations differ even more than than the species. Where SBML uses a list to represent its transitions, ERODE uses a HashMap. Additionally, the transitions themselves are also represented differently. In SBML a the transition data structure contains three items:

- a list of *inputs*
- a list of *outputs*
- a list of *function terms*

In ERODE, transitions only have a single output and a single function term, called the update function. These two items form a single entry in the HashMap used by ERODE to store its transitions. In both formats, the output is a reference to the species affected by the result of the transition. In ERODE, this species reference is used as a key, which the function term (update function) is mapped to. The HashMap essentially acts a dictionary that is used by ERODE to look up a species update condition. The update function structure in ERODE, also differs slightly from the function term used in SBML. In SBML function terms all have a result level indicating the what value will be assigned to the species if their condition is met. However, due to ERODE being limited to the Boolean domain, this is not required. The update function implicitly represents the value 1 (true) if its condition is met, and 0 otherwise.

Just like SBML, the species and transition are contained within an outer structure representing the model itself. In ERODE, object instances that implement the *IBooleanNetwork*-interface are used for this.

4.4 Conversion between the two formats

To successfully convert a given network model from one format to the other, it requires that both the *species* and the *transitions* of the model can be converted. This requires and analysis of which attributes are represented in both representations and which are not. The common attributes can then be converted by simply transferring their values to the other format. For the other attributes it must be analyzed, whether these attributes are required in the other format, and if so how they can be converted.

4.4.1 Converting qualitative species

As previously mentioned, the species representations of both formats have three matching pairs of data, as shown in the following list. As a note, the SBML attribute is always given first:

- The *id* and *name* attributes, which specify the variable name for the species, that is used in the model
- The *name* and *originalName* attributes, which represent the actual name of the species
- The *initialLevel* and *intitialConcentration* attributes, which specify the initial value of the species

For these pairs the only actual conversion that might be required is to change the data type used to represent them, as for example the conversion of the *initialLevel* from SBML to ERODE, as shown in figure 4.3.

Instead of being represented as in integer like in SBML, the *initialConcentration* in ERODE is represented both as a big decimal type and a string. This particular type conversion however is rather simple, as a big decimal is a special type of integer representation for arbitrary sized integers. In addition to this rather complex integer representation, the initial value of a species is also stored in the *initialConcentrationExpr*, a string representation of the initial value. Since the *initial level* of an SBML species is an optional value, the converter can simply use a default value 0 as the *initial concentration* in case the *initial level* is unset. When converting this attribute back to SBML, a simple type conversion between the two integer types is required, as the possible set of values for the ERODE representation is limited to $\{0, 1\}$.

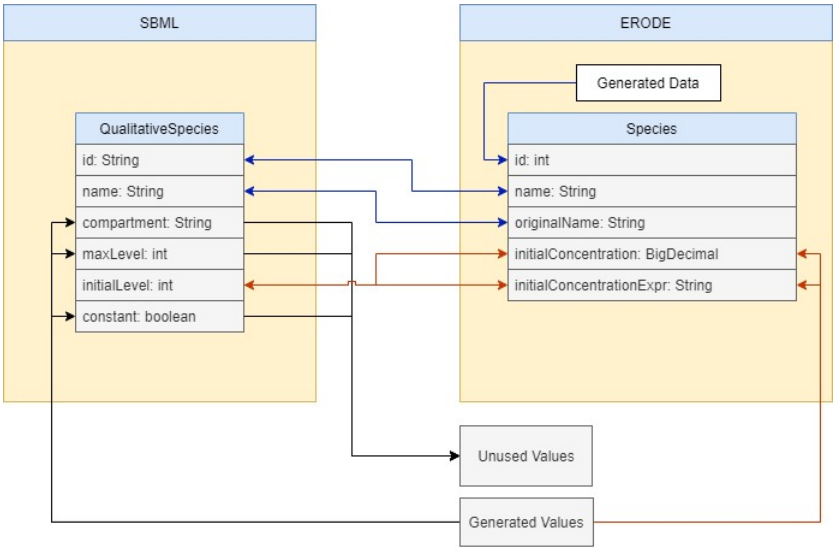


Figure 4.3: Species conversion mapping schema

When looking at figure 4.3, it can also be seen that the species representation used by SBML, which can be found on the left side of the figure, has more attributes than the ERODE representation on the right. This is due to the fact, that ERODE specifies these values implicitly. Firstly, ERODE does not support declaring species as *constants*. This can only be handled implicitly, for example by not giving those species an update function that would make a value-change possible. Secondly, the *maximum level* is already defined, since ERODE only is capable to represent models within the Boolean domain $\{0, 1\}$. Making a successful conversion of a multi-valued network impossible in ERODE's current state. Finally, the *compartment* attribute is a value simply cannot be translated to ERODE format. However, upon speaking with the ERODE's developers about this problem, it was decided that all models can be interpreted as single-compartment models, making this attribute irrelevant. Due to these reasons, these three SBML attributes can be safely discarded during the species conversion to ERODE format. Of course, this also means that these attributes require the generation of default values, when converting back to ERODE format. These default values are:

- A default compartment name, that is the same for each species, such as ”default”
- A maximum level of 1, due to all models being in the Boolean domain.
- The *constant* attribute being set to `false`, as species without transitions cannot change value, even if it is allowed.

Finally, the last attribute that needs special treatment during the conversion from SBML to ERODE is ERODE's *id*-attribute. This value is used to give each species a unique identifier within ERODE itself and therefore has no counterpart in SBML. To provide such a value, the converter can simply start a counter during the species conversion, which is incremented each time another species has been converted. Since the counter value changes to a new value after each iteration, the counter value can be used as an *id* for the ERODE species.

4.4.2 Transition conversion

Converting a transition from SBML to ERODE is much simpler than the conversion of a species, since ERODE requires a lot less data from the SBML representation. The drawback of this is that the converter needs to generate a lot more data, when converting back

to SBML. This is due to SBML being an extensive format, that can be used to represent a large variety of model types.

as shown in figure 4.4, the conversion of an SBML transition to ERODE format requires a reference to the updated species and an update condition to be supplied. In SBML these attributes are contained in the *outputs* and *function terms* of a transition. The *outputs* are a list containing references to each species updated by the given SBML transition. The *function terms* are a list of items representing the condition, when to apply which result level. Since ERODE only supports models from the Boolean domain, only the function term for the result level `1` (true) is required, since ERODE handles the *otherwise*-case implicitly.

Before a *function term* can be used to represent a ERODE transition, the logical expression it contains must be converted to ERODE format. This is required, because both formats use different implementations to represent abstract syntax trees (ASTs), a data structure used to express mathematical expressions unambiguously. Once the expression has been converted, the resulting expression (the update function) is used to create an entry in the LinkedHashMap, for each species referenced in the list of outputs. This creates a dictionary in ERODE, that can be used to look up, which species require which condition to be true, to change to the *true*-state.

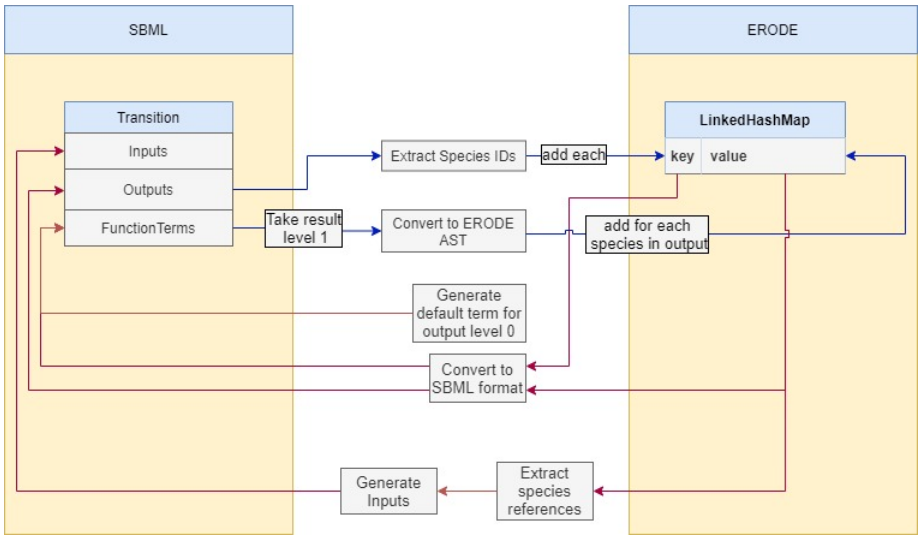


Figure 4.4: Converting transitions between SBML and ERODE

When converting to SBML format, lists for the *inputs*, *outputs* and *function terms* must be created for the generation of an SBML transition. The *inputs* store information about the species referenced in the function terms of the transition. To generate an input-instance three pieces of information are required:

- A string *id*, to specify the unique identifier for the input
- The name of the variable used to reference a given species in the function term, stored in the *qualitativeSpecies*-parameter
- A *transitionEffect* that specifies how the input is affected by during the transition. This can always be set to `none`, since ERODE does not support transition effects.

To generate these *inputs*, the converter needs to analyze the update function mapped to a species and find each variable used in the function expression. For each new variable

found, a new input needs to be generated, duplicates can be skipped. This step is shown at the bottom of figure 4.4.

The list of *outputs* stores similar information, but about the species that are modified by the transition. Just like *inputs*, each *output* requires an *id*, a reference to a *qualitativeSpecies* and the specification of a *transitionEffect*. The only difference is, that the *transitionEffect* needs to be set to `assignmentLevel`, since the values of the *output*-species are overwritten by the transition. The only piece of information that is required from ERODE to generate an *output*-instance, is the variable used for the species. The species references, which are used as keys in the *LinkedHashMap*, can be used for the *output*-generation.

Finally, to generate the list of function terms required by each transition instance, the update functions contained in the ERODE's hash map, can be used to genreate *function terms* for the result level 1 (true). This simply requires a format conversion of the AST represented in the update function. However, for the list of *function terms* to be complete, another function term needs to provided. This is necessary, because SBML requires all possible outcomes of a transition to be defined by a function term. The converter must therefore generate a *default term*, a special condtion-free function term, with a result level of 0 (false). This *default term* represents the *otherwise*-case in the formal notation of a Boolean or multi-valued network. Once all three lists have been generated, the converter can generate the converted SBML-transition.

Regarding the conversion between SBML and ERODE, all requirements for the conversion of Boolean networks have now been covered. For a deeper understanding of SBML (Hucka et al. 2018) and SBML-qual (Chaouiya et al. 2015), it is therefore recommended to take a look at their specifications. and SBML-qual specifications, which can be found at SBML.org (*SBML Specifications* 2019).

4.4.3 Conversion of logical and relational operators

Although the principle, how to convert a logical expression from SBML to ERODE format, is quite simple, the operator interfaces of both formats have not been taken into account so far. To be able to convert all valid logical expressions, it requires that all operators in one format, can be expressed using the available operators of the other format. To verify that this is indeed possible, an overview of both operator interfaces is required. The overview is shown in the following table:

Operator	SBML compatible	ERODE compatible
AND $\wedge$	✓	✓
OR $\vee$	✓	✓
XOR $\oplus$	✓	✗
NOT $\neg$	✓	✓
IMPLIES $\rightarrow$	✓	✓
XNOR (binary equality) $\leftrightarrow$	(✗)	(✗)
EQUALS $=$	✓	✗
NOT EQUALS $\neq$	✓	✗

Table 4.1: Operators usable in logical expressions for BNs and their support

When looking at table 4.1 above, it can be seen the `XOR`, `XNOR`, `EQUALS` and `NOT EQUALS` operations are not supported by ERODE. This means that these missing operators need to be expressed using a combination of the existing operator set.

4.4.4 The XNOR- and XOR-operation

The XNOR operator does not exist in either formats, yet it could be used in logical expressions. SBML does not support the XNOR operator because it corresponds to a negated XOR ($\neg\oplus$) operation. Since both operators required to expression the *XNOR*-operation already are supported by SBML, there is no need for a separate operator. ERODE, on the other hand, does not support the XOR operation. This means, it has to be constructed using the available operators.

Given two variables P and Q, the XOR-operation is defined to be true, if either P or Q are true, but not both. In propositional logic, this condition can be expressed as:

$$(P \vee Q) \wedge \neg(P \wedge Q)$$

P	Q	$(P \oplus Q)$	$(P \vee Q) \wedge \neg(P \wedge Q)$	$P \oplus Q = (P \vee Q) \wedge \neg(P \wedge Q)$
T	T	F	F	T
T	F	T	T	T
F	T	T	T	T
F	F	F	F	T

Table 4.2: Truth table verification that XOR is equivalent to $(P \vee Q) \wedge \neg(P \wedge Q)$

As it can be seen in the truth table 4.2 above, the two logical expressions are equivalent, which means that the XOR-operation can be translated to ERODE format using its supported operators.

4.4.5 Equality and inequality operators

Just like the XNOR-operation, the inequality operation can be constructed by negating the equality operator. However, neither equality operator is currently supported by ERODE. Fortunately, the equality can be expressed differently, depending on the domain it is used in.

In the Boolean domain, the XNOR-operation also represents binary equality: it is true if both sides are true, or both sides are false. Since XOR is the opposite of XNOR, the XOR-operator must be equivalent to the inequality-operator. So the big question is, why not just translate these operators to their corresponding logical equivalents?

To answer this question, it is necessary to take ERODE's current state into account. The program is under development and it is intended to implement support for multi-valued networks. In other words, ERODE will eventually support models outside the Boolean domain. Since this converter should be able to convert SBML-qual models into ERODE format, it would be wise to design the converter such that the changes required to adapt to new version of ERODE are minimal.

Since ERODE is restricted to the Boolean domain and these relational operators need to be supported; A good midway would be to implement them in the Boolean domain, while making them as independent from the other operators as possible. This means finding an expression, that has a minimal overlap with the XOR- and XNOR-operators.

This can be realized using the `implies`-operator. This is due to the fact, that logical equality also corresponds to the *if and only if*-conditional ($\leftrightarrow$) in propositional logic. The conditional can be expressed using the `implies`-operator, thereby making it highly independent from the other operators.

$$A \leftrightarrow B = (A \rightarrow B) \wedge (B \rightarrow A)$$

To prove that $(A \rightarrow B) \wedge (B \rightarrow A)$ only is true, when A and B are equal, consider the possible outcomes of an implication:

A	B	$A \rightarrow B$
T	T	T
T	F	F
F	T	T
F	F	T

Table 4.3: Outcomes of an implication given A and B

As it can be seen in table 4.3, the only way for an implication to be false, is when the first argument is true and the latter false. This is also, why $(A \rightarrow B) \wedge (B \rightarrow A)$ cannot be true, when the two arguments are different, as shown below.

$$A \rightarrow B \qquad = F \qquad (4.1)$$

$$B \rightarrow A \qquad = T \qquad (4.2)$$

$$(A \rightarrow B) \wedge (B \rightarrow A) \qquad = T \wedge F \qquad \text{from (4.1) and (4.2)} \qquad (4.3)$$

$$T \wedge F \qquad = F \qquad \text{from (4.3)} \qquad (4.4)$$

$$(A \rightarrow B) \wedge (B \rightarrow A) \qquad = F \qquad \text{from (4.4)} \qquad (4.5)$$

Since it has been shown, that equality can be expressed as $(A \rightarrow B) \wedge (B \rightarrow A)$ in the Boolean domain, it is also evident that inequality would be its negation, This means, that the required relational operators can be converted from SBML to ERODE without having great similarity to the `XOR`- and `XNOR`-operators.

It has now been shown that all supported by SBML can be expressed using a combination of ERODE's operators. Unfortunately, these expression conversions cause a significant loss in readability, due to their lengthiness. The expression length problem is especially severe, when it comes to the *equality* and *inequality* operations, since these are used very frequently in the SBML format. Their high usage is due to the fact that many networks are represented using a multi-valued notation (integer based notation) restricting the model to use comparison operations, for operations on variables and constants. This also extends to networks that are within the Boolean domain, as all Boolean models are a subset of all the possible multi-valued models.

To solve the problem of lengthy and unreadable expressions, it would require to update ERODE and extend its operator interface. With this update, the original expression could be maintained and the conversion would have no impact on its readability. Once the developers were presented with this problem, they did eventually decide to extend the operator interface to support the same operator set as SBML.

5 Design

Now that both data formats have been analyzed and conversion schemes have been proposed, a system can be designed to support this conversion. This section will explain the system architecture and its design process. Initially in section 5.1, a framework system is designed that deals with the practical requirements and limitations of the conversion and acts as a container for the conversion modules. Such requirements and limitations include considering how to exchange data with the converter. Section 5.2 gives an overview of the design goals that are rooted in aesthetics and propose design approaches how to achieve them. Such design goal include ease-of-use (simplicity) and maintainability (code management). Section 5.3 presents how the system's component structure divides the conversion process into multiple steps. Section 5.4 shows how a certain independence between those components can be achieved. Finally, the base structure of the converter components is explained in section 5.5.

5.1 System requirements and limitations

The most basic requirement of the converter, is that it must be able to convert SBML-qual to ERODE-format and vice versa. To do so, the converter must be capable of basic input/output of files (file-I/O), as a standalone program. Once the converter has read its input file, it must be able to extract the required data from the file. For SBML this can be done using the JSBML-library, a Java-API to read and write SBML files. In case of the ERODE format, the ERODE-library is capable of file-I/O, but only to the extent of writing. This means, that the standalone converter can only generate ERODE-files from an SBML input, but not vice versa. The compiler used to interpret the ERODE format (.ode) is also not publicly available, which is why there are no means to parse ERODE-files directly in the converter. However, the converter must still be able to converter ERODE's data representation into SBML format, since it still could be used inside ERODE to export to SBML. To enable the converter to convert from ERODE to SBML, the reading and writing of files, simply needs to be separated from the conversion process itself. In other words, by converting between the data representations used by ERODE and JSBML. This will essentially outsource all file-I/O to the JSBML and ERODE plugins, and turn the converter into a bridge between the two formats.

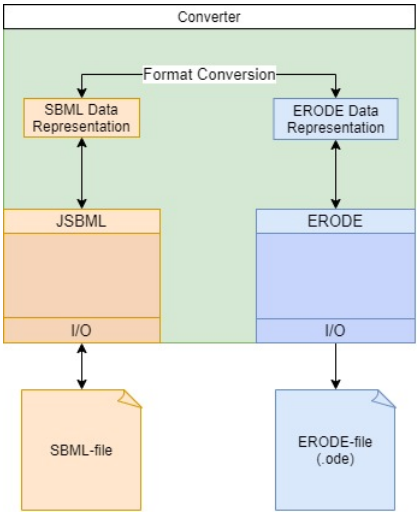


Figure 5.1: An overview of the conversion system

As shown in figure 5.1, the converter will be given an SBML file to convert. The converter will then take the data representation of the file and convert it to the other data representation, which then is written to a file. When integrated into ERODE, the converter will be given directly from the host program, and generate an SBML file representing the given model as output.

5.2 Development goals and design approach

Now that the basic system has been designed based on its requirements, it is time to think about how such a system can be realized in practice. That means deciding:

- How file-I/O should work in this system.
- How the data conversion component should be designed and structured.
- How modifiable the system should be.

For the standalone converter, the first question is relatively simple to answer. Since both the JSBML- and the ERODE-plugin are contained within the converter, the converter itself must have a file-I/O interface of its own. This interface must be able to take in an SBML-file as an input and return a new ERODE-file as output.

The answers to the second and third question are intertwined, as the design of the conversion component will define how dependent the converter will be on the current state of both formats. Since the ERODE-tool still is in development and therefore subject to change, the converter should have a design that easily can be adapted to support changes in both formats. This requires the converter to be designed in a way that is both easy to maintain, whilst being highly modifiable.

This can be achieved using the SOLID-principle, which consists of the basic rules to create highly maintainable and extendable applications using object oriented programming. The principle itself is a collection of five principles:

- **S**: The *single responsibility principle* states that software components should be kept small and dedicated towards a single specific task. This will keep code segments shorter, as well as easier to understand and change.

- **O**: The *open-closed principle* states that existing software components should be open to extension but closed for modification. This means, that any existing functionality of a component should remain unchanged, as any changes may impact the whole system. In this the case of the converter, it may not always be possible to ensure that the design always follows this principle. Since the conversion formats still are being developed, it may become necessary modify existing functionality of the converter to adapt to changes in both formats.

- **L**: The *Liskov substitution principle* states that components that are derived from other components should be able to replace the component they were derived from without causing unexpected results.

- **I**: The *interface segregation principle* state that interfaces should never force components to support interface-methods that should not be supported by the component. Instead such an interface should be broken down into smaller ones that can be implemented selectively.

- **D**: The *dependency inversion principle* states that components should depend on interfaces instead of direct dependencies. This increases their independence and allows components that implement the same interface to be used interchangeably.

It should be remarked that these definitions are based on the SOLID definitions given in Apex Design Patterns by *Jitendra Zaa* and *Anshul Verma*

5.3 Converter layering

The analysis of SBML-qual in section 4.2 revealed the complex multilayered structure of the SBML-format. Due to the complexity, the conversion to and from SBML is therefore quite an extensive task. However, this task can be broken down in to smaller sub-tasks by separating the layers and converting them in different converter components. This layer separation also reduces the responsibilities of each converter component, since each component only operates on a single layer. This enforces the *single responsibility principle*.

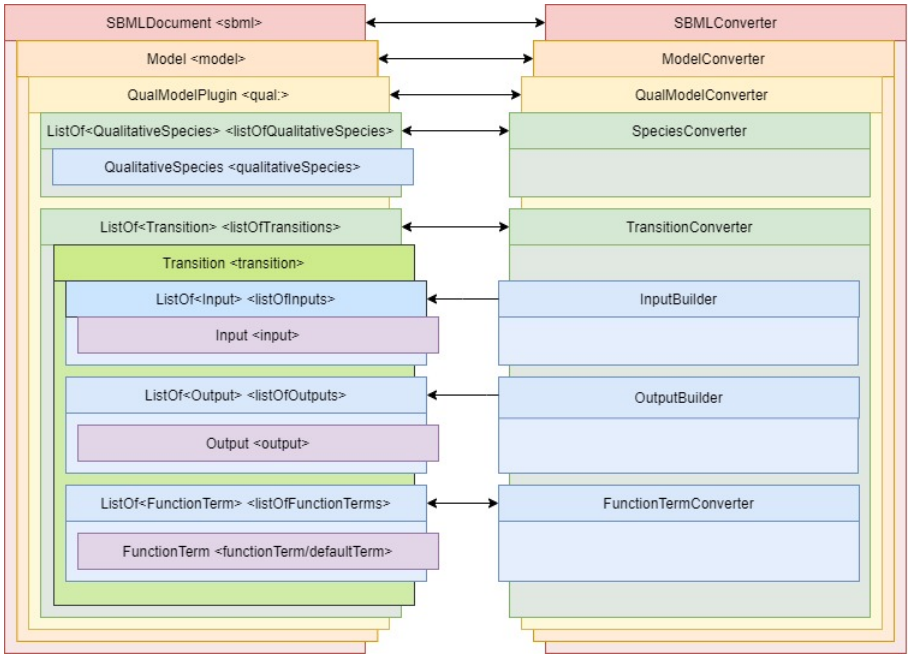


Figure 5.2: Mapping of SBML-qual layer to corresponding converter component

Figure 5.2 shows the SBML-qual structure to the left and the converter's component structure to the right. Components that are enclosed by an outer component, are contained within their outer component. The coloring indicates which components are located within the same layer. The arrows are used to indicate which of the converter's components are responsible for the conversion of specific layer in the SBML-qual structure. As an example, the *ModelConverter* is responsible for the conversion of the *Model*-layer in SBML-qual.

The arrows also indicate, which conversion the various converter components are responsible for. Arrows pointing at the converter indicate that the component is responsible for the conversion of SBML to ERODE format. Arrows pointing at the SBML-qual structure indicate the opposite, the conversion from ERODE to SBML-qual.

Components in the SBML-qual structure that are not connected to a converter component via an arrow, are converted as a part of their enclosing layer. This also counts for SBML-layers that only are pointed at by a converter component. For example, each *input* (<input>) is converted as a part of the list of inputs (<listOfInputs>) it is contained by. The list of inputs is generated by the *InputBuilder*, when converting to SBML-format, but it is converted as a part of each transition, when converting to ERODE's format.

The reason, why some of these layers are converted along with their outer layers is that the outer layer for one or both conversion directions, simply does not contain enough data of its own. For example, list are used to contain multiple item of the same type, but apart from these items they contain no relevant data of their own.

Now that the overall converter structure has been designed, it is important to consider how the converter components interact with each other during the conversion. This requires a step by step analysis of the conversion process.

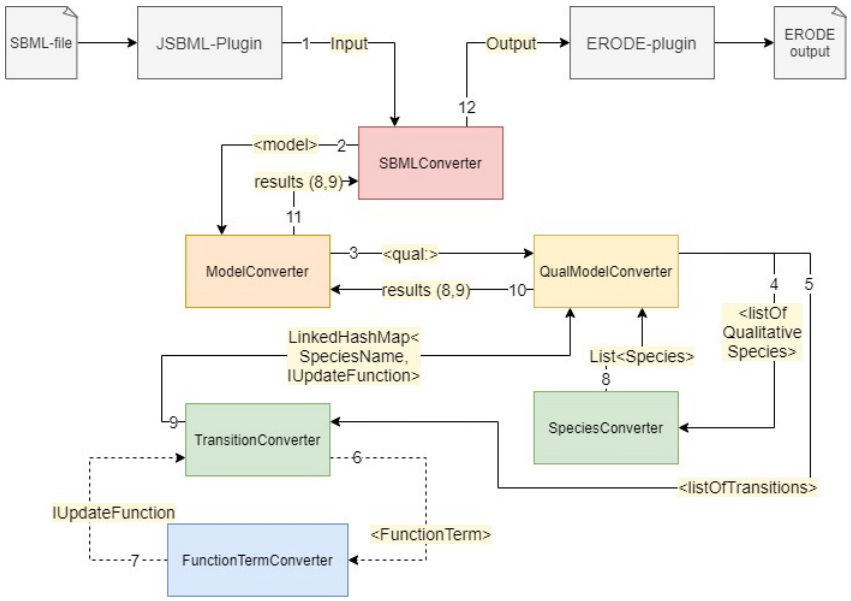


Figure 5.3: SBML to ERODE conversion process

As shown in the steps 1-3 of figure 5.3, the outer converter components simply remove the outer-most layer of the SBML-qual input they are given and pass on the remaining data to the next component. This is the only requirement for the first two components since the outer SBML-layers contain only meta-data about the model, which is not required by ERODE. Thereafter, the `QualModelConverter` splits the data representation as shown in steps 4 and 5, since species and transitions are independent structures that do not require data about the other during their conversion. Because species do not contain sub-layers, the `SpeciesConverter` can convert each species and return a list of all species in ERODE format to the *QualModel converter* (steps 4 and 8).

To convert transition and additional conversion step, the conversion of *function terms*, is required. This is handled by the *FunctionTermConverter* in steps 6 and 7, which receives a *function term* from the *TransitionConverter* and returns the function expression within as an *IUpdateFunction*. The *IUpdateFunction* is the corresponding structure in ERODE. The steps 6 and 7 are highlighted using dotted arrows, as they have to be repeated for each single transition. This means that these steps will run on a loop until all transitions have been converted. In steps 8 and 9, the species and transitions are then returned in ERODE format, which then are routed back to the conversion entry/exit point in the `SBMLConverter`. As a final step, the `SBMLConverter` will then package the converted ERODE structures into an ERODE model, which is passed on to the ERODE plugin to generate the file.

Apart from a few minor details, the reverse conversion, from ERODE to SBML, largely corresponds to the execution process in figure 5.3, but in reverse. The main differences are the the *input* and *output* builders from figure 5.2 need to be included to generate SBML transitions and that the ERODE model is supplied by the ERODE host program instead of a file.

5.4 Component separation

When looking back at, how each of these converter components exchange data, it quickly becomes visible that all components request data from components in a lower layer (sub-components), whereas they always respond to a component in a higher layer (parent).

Since that is the case, all components can be separated by introducing interfaces that define the contract for their communication. This enforces the *dependency inversion principle* and increases the maintainability of the converter even further, by design.

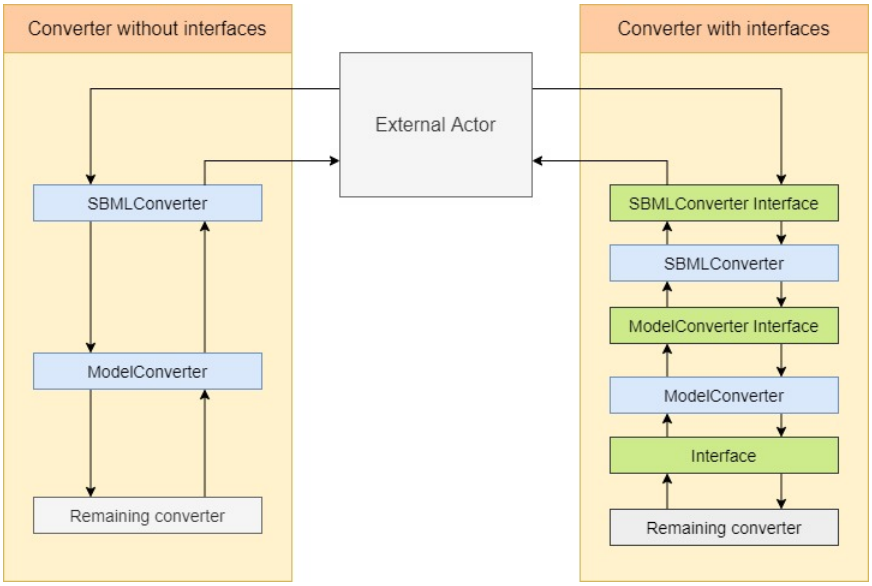


Figure 5.4: Converter structure with and without interface

When comparing the two converters shown in figure 5.4 it can be seen that the converter without interfaces consists of components that depend on each other directly, this means that there is one specific type of component that is required to ensure the conversion can succeed. The other converter on the other hand relies on interfaces, which act as contracts of interaction. Since it relies on interfaces instead of a direct dependency, components are implemented completely independent of each other. Implementations are hidden by the interface allowing components that implement the same interface to be used interchangeably.

5.5 Component structure

Another way of increasing maintainability by design, is to divide each converter component into sub-components. Since each component currently is responsible for both conversion directions, they can be divided into separate pieces internally. From the outside there still is only one component to interact with, but internally the two ways of conversion are handled by different sub-components. To facilitate this, each component would require some kind of manager that can identify, which conversion process is required and initializes it accordingly.

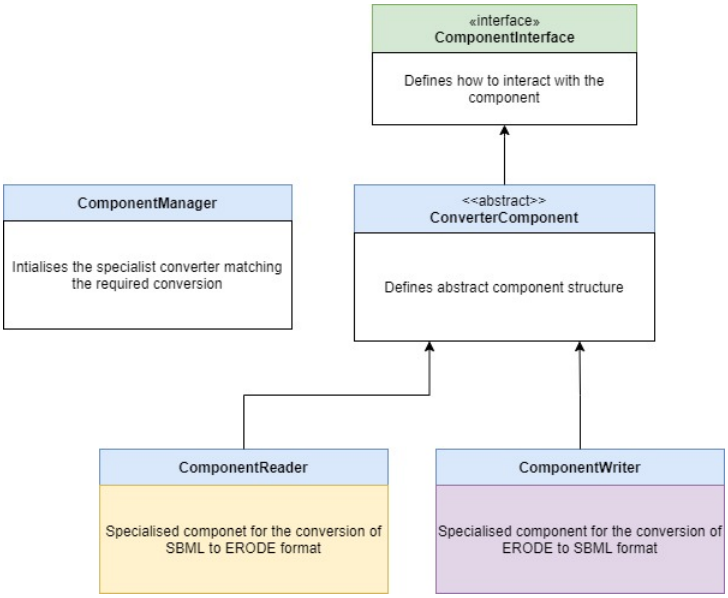


Figure 5.5: General component structure

As it can be seen in figure 5.5, each component is controlled by a manager that initializes the required conversion unit. The possible interactions with the conversion unit from outside the component is defined by the *ComponentInterface*. The interface is implemented by the *ConverterComponent*, an abstract class defining the general converter unit's object-structure. The *ConverterComponent* is extended by the *reader*- and *writer*-units, which define the specialized conversions of both formats.

All conversion unit *readers*, take care of *reading SBML*, converting it to ERODE format, whereas *writers* take ERODE format as input, *writing* it to SBML. This naming convention is used to indicate the conversion direction for each of these components without needing to use either SBML or ERODE as a part of the class' name. It should be added that this general structure may vary slightly from component to component, as some components may require additional functionality, which according to the SOLID-principle would need to be added as separate entities.

6 Implementation

Now that the analysis of the conversion and the converter design is complete, it is time to consider its implementation. A substantial part of the converter system can be implemented by simply creating a java structure corresponding to the conceptual structure given in the design section. For example, a converter component could be realized as a package containing classes and interfaces, which define its sub-components. The tasks performed by each of these components could then be implemented as methods in the corresponding classes, according to the conversion rules specified in the analysis. However, when it comes to implementations there are often multiple ways to achieve the same goals and sometimes practical limitations that can prevent you from using a certain approach.

In this section, some of the most prominent implementation choices will be showcased to give an understanding how the converter was realised and how some practical challenges have been solved. Section 6.1 will explain, how the component design given in section 5.5 is implemented by showcasing the *ModelConverter* component. In section 6.2 it will then be explained how to convert function terms between the two formats.

6.1 Component construction

From the component design shown in figure 5.5, it can be seen that the *manager* and the *interface* are the only independent structures of a component, since all other components inherit from the interface. These two structures are also the only ones, that are visible to the outside.

The reason for this is, that the other structures contain contain information about the components implementation, which according to the SOLID-principles should remain hidden to the outside. As such, the *manager* also needs to act as an entry point for outside data, since this data is required for the converter initialisation. Due to the component's structure, it is also the only possible way to handle this, since the *interface* cannot be accessed before the converter itself has been initialised, leaving no entity but the *manager* accessible from the outside.

6.1.1 Showcase ModelConverter

When looking at the contents of the *model*-package of the converter, it can be seen that it consists of the five entities specified in the component design.

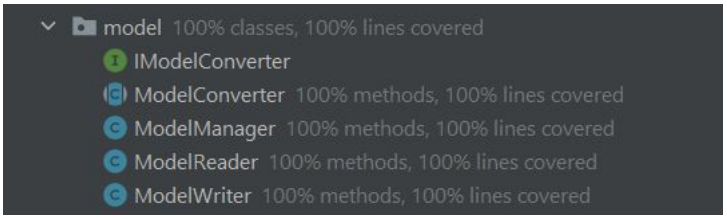


Figure 6.1: The contents of the model-package

The *IModelConverter* is the interface that specifies the methods that can be used to interact with the component.

```java
public interface IModelConverter {

    Model getModel();

    String getName();

    List<ISpecies> getErodeSpecies();

    LinkedHashMap<String, IUpdateFunction> getErodeUpdateFunctions();
}
```

Figure 6.2: The *IModelConverter*-interface

As it can be seen in figure 6.2, the interface consists of four methods that can retrieve data from the component. The `getModel()`-method returns a `Model`-object, which is the SBML-structure containing the data of an SBML-model. The `getName()`-method is used to retrieve the name of the model from the given network. Since the representations of both formats store the name of the network, it was added as a separate method, to allow format independent access to that value. The last two methods in the interface, the `getErodeSpecies` and the `getErodeUpdateFunctions`, are used to retrieve the data structures for species and transitions (update functions) in ERODE format.

This interface is then implemented by the abstract class *ModelConverter* shown in figure 6.3, which specifies the the set of fields both converter units have in common and it provides constructors to initialise them. Since it is an abstract class, it cannot be initialised itself, but the constructors can still be utilized by its derived classes.

```java
abstract class ModelConverter implements IModelConverter {
    protected static final String EXTENSION_NAME = "qual";

    protected Model model;

    protected String name;

    protected IQualModelConverter qualModelConverter;


    public ModelConverter(@NotNull Model model) {
            this.model = model;
            this.name = model.getId();
    }

    public ModelConverter(@NotNull IBooleanNetwork booleanNetwork) {
        this.name = booleanNetwork.getName();

    }

    @Override
    public Model getModel() {
        return this.model;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public List<ISpecies> getErodeSpecies() {
        return this.qualModelConverter.getErodeSpecies();
    }

    @Override
    public LinkedHashMap<String, IUpdateFunction> getErodeUpdateFunctions() {
        return this.qualModelConverter.getUpdateFunctions();
    }
}
```

Figure 6.3: An overview of the abstract *ModelConverter*-class

The *ModelConverter* enforces the *dependency inversion principle* as the only converter reference is via the *IQualModelConverter*-interface. Since nothing but the interface is known to the *ModelConverter*, the internal logic of the *QualModelConverter* remains hidden from it.

The *ModelConverter* base class is then extended by the *ModelReader* and the *ModelWriter*, which define the two ways of conversion. Both classes contain a constructor and a set of private methods designed to deal with the specifics of their conversion.

```
class ModelReader extends ModelConverter {

    public ModelReader(Model model) {
        super(model);
        this.qualModelConverter = QualModelManager.create(this.tryGetQualModel());
    }

    private QualModelPlugin tryGetQualModel() {
        try {
            QualModelPlugin qualModelPlugin =
                (QualModelPlugin) model.getExtension(EXTENSION_NAME);
            if (qualModelPlugin == null)
                throw new NullPointerException();
            else
                return qualModelPlugin;
        } catch (Exception e) {
            throw new IllegalArgumentException("Invalid input, the SBML-model " +
                "is not an SBML-qual model");
        }
    }
}
```

Figure 6.4: The *ModelReader-class*

```
class ModelWriter extends ModelConverter {
    private static final SBMLConfiguration CONFIG =
        SBMLConfiguration.getConfiguration();

    public ModelWriter(IBooleanNetwork booleanNetwork) {
        super(booleanNetwork);
        this.model = new Model(CONFIG.getLevel(),
            CONFIG.getVersion());
        this.model.setName(this.name);
        this.qualModelConverter = QualModelManager.create(
            booleanNetwork, model);
        this.packageSBML();
    }

    private void packageSBML() {
        QualModelPlugin qualModelPlugin =
            qualModelConverter.getSbmlQualModel();
        model.addExtension(EXTENSION_NAME,qualModelPlugin);
    }
}
```

Figure 6.5: The *ModelWriter*-class

When looking at the figures 6.3 through 6.5, one might notice that none of these classes have the *public* modifier. This enforces that their visibility is limited to the package, they are contained in. Since each component is implemented in a different package, all non-public classes are therefore completely invisible to classes in other packages.

Lastly, the *ModelManager* acts as an entry point for the converter. It implements two `create()`-methods, which take care of initialising the two different converters. Since each `create()`-method, takes an input, the input is used to define the type of converter to be returned. As shown in figure 6.6, the first `create()`-method takes a `Model`-object as input, which is an object-type used by JSBML to represent SBML-models. Since this method is dealing with SBML-input, the converter capable of converting to ERODE is required. The second `create()`-method handles the reverse case, where ERODE-input is given, an a converter that can convert to SBML is initialised.

```java
public class ModelManager {
    public static IModelConverter create(
        @NotNull Model model) {

        return new ModelReader(model);
    }

    public static IModelConverter create(
        @NotNull IBooleanNetwork booleanNetwork) {

        return new ModelWriter(booleanNetwork);
    }
}
```

Figure 6.6: The *ModelManager*-class

6.2 Function term conversion

As shown in the analysis, the function term structure in SBML-qual defines the condition for a given transition result. As such, it contains the *result level* and the *logical expression* forming the condition. The logical expression is given in the form of an abstract syntax tree (AST), a structure used to represent the operation precedence in machines to avoid ambiguity.

Since both formats represent the transition conditions using ASTs, the primary task is to replicate the exact same tree-structure in the other format. The challenging part of this conversion is that different expression will have different ASTs structures representing them. This means, that the converter must be able to analyze each element in the AST and select the correct conversion method for that element to ensure a correct conversion. To solve this task, an algorithm is required that can traverse the AST, analyze each element and select the correct conversion method for it. Additionally a converter structure is required to execute the conversion of these elements.

In order to convert an given AST to another format, the conversion algorithm must be able to recognize the different elements that can occur in it. In general, every AST is a graph consisting of vertices (nodes) and edges (links). The vertices represent operators, variables and constants, whereas the edges indicate how these nodes are connected. Based on the amount of outgoing connections of a node it is possible to divide nodes into three groups:

- *Binary* nodes representing binary operations with two outgoing connections

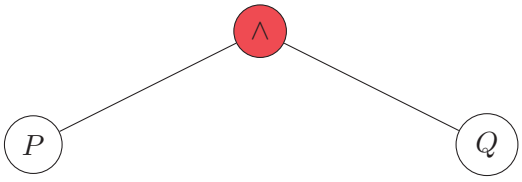


Figure 6.7: Binary node (red) binding variables

- *Unary* nodes representing unary operations that only have a single outgoing connection

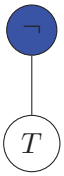


Figure 6.8: Unary node (blue) with a single constant child node

- *Leaf*-nodes representing a variable or a constant, which have no outgoing connections

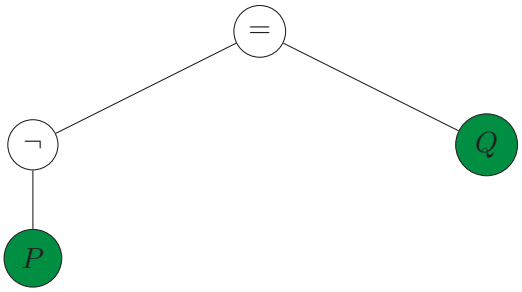


Figure 6.9: Leaves (green) in an AST

The components required to provide the conversion functionality can be structured in a similar fashion as the other converter components using the SOLID-principle. An abstract *node converter* is introduced hiding the three different converter types to convert the three node types, which then again are split in two to separate the conversion directions.

With the conversion infrastructure in place, it is now time to look for an algorithm that can handle the tree traversal during the conversion. In the case of ASTs, there are two approaches of traversal, the *bottom-up* approach, where the tree analysis begins at the leaves working its way up towards the root and the *top-down* approach, starting at the root, ending at the leaves.

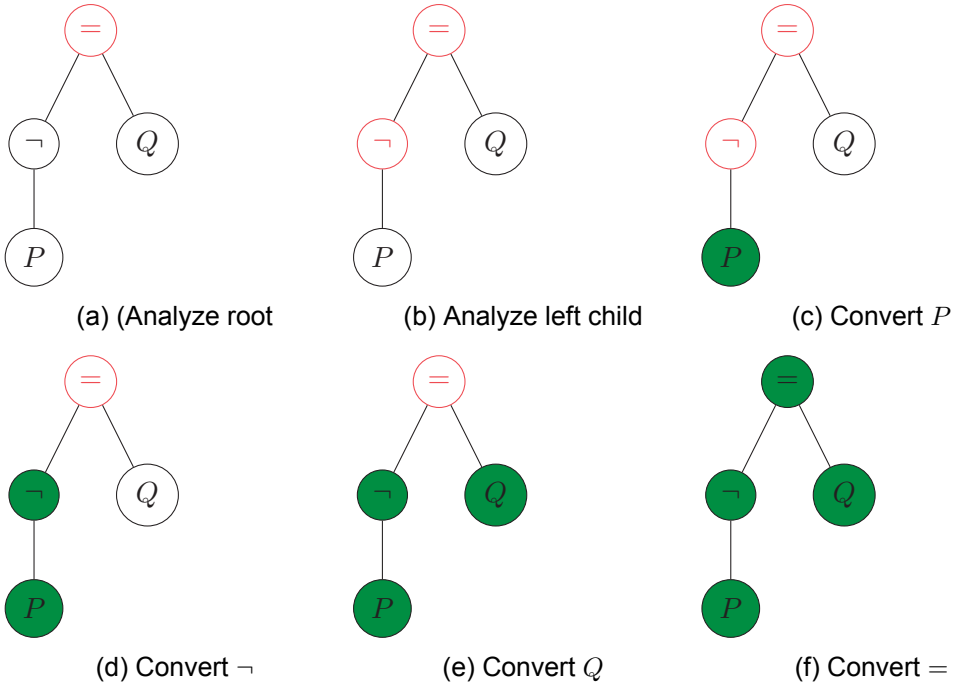
From a performance point of view, algorithms using the bottom-up approach are considered to have a better performance as they only need to traverse each node once. In top-down approaches the algorithms need to back-trace their steps to a previous node, as branches in the tree may split forcing them to prioritise one node over the other.

In order to obtain the performance benefit fro the bottom-up approach it is required that the leaves of the tree can be accessed directly, and that children (lower-level nodes) have a reference to their parent (higher-level nodes). Both in SBML-qual and ERODE the only directly accessible node is the root of the AST, taking away the performance benefit of bottom-up approaches. Since all node of the AST are accessible through the root in these formats, the data structure could be transformed into one where bottom-up approaches can be used effectively, but the additional transformation would nullify the performance benefit.

Due to the fact, that there is no performance loss and no additional structural transformation required. It is viable and also much simpler to use a top-down approach. A suitable algorithm for such a tree traversal is a *DFS* (depth-first-search). The algorithm will traverse down the branches of the tree, analyzing each node in the search for leaves, as they have no children and can be converted instantly. If a node is not a leaf, it will advance

to its leftmost child, that has not been converted and analyze it. Once it finds a leaf it will convert it and backtrack converting each node, that now has all data required for its conversion. Once it finds a node that requires an additional child to be converted it will then recursively search for a leaf that can be converted and repeat the backtracking once it has been converted. Since all nodes in the tree require some constant amount of work to be converted and all nodes in the tree must be converted, the algorithm is tightly bounded by $\Theta(n)$, where n is the amount of nodes in the tree. The fact that this algorithm is tightly bounded by the amount of nodes in the graph, means that both the worst-case and the best-case scenarios for the execution of the algorithm have a linear time complexity.

6.2.1 Showcase depth-first-search



(a) (Analyze root (b) Analyze left child (c) Convert P

(d) Convert $\neg$ (e) Convert Q (f) Convert $=$

7 Testing

During this project two testing approaches have been used to verify that the converter is capable of a correct conversion. On one hand, simple input/output testing was used to test the demos and the converter during the early stages of the project. This form of testing helped to investigate how to use the JSBML and ERODE libraries and figuring out how to realize the most basic conversion steps in code.

Once the first conversion steps were implemented and familiarity with those libraries were obtained, the development process began turning more and more into a test-driven (TDD)/behaviour-driven (BDD) development process. Using the cucumber testing tool, test were defined to assess the program behaviour, which then were implemented. Every time a new feature was implemented all test were run again, failed test were analyzed and problems were fixed, before moving on the next feature.

In addition to the tests, a code coverage tool was used to analyze, which lines of code in the converter were executed during the tests. Code segments that were not converted by the tests were then analyzed in order to ascertain the reason why they did not run. Mostly, this was due to certain scenarios not being tested, in some rare cases it was simply due to the fact that there was no scenario being able to reach those lines. Depending on the case, either new test scenarios were then added to verify missing code segments, or code segments simply were removed, due to not being needed.

As a final means of testing, some end-to-end testing has been performed, by converting an real-world SBML-model to ERODE and back. The conversion result was then compared to the original, to see if they still represented the same model. Eventually, the converter had been tested and improved to the point, where such a conversion succeeded. However, the two files still differed, as some of the additional data, irrelevant to ERODE, was lost during the conversions.

When looking at, what kind of tests were implemented, it can be seen that the majority are success tests. This is due to the fact, that the most important feature of the converter is that it should be able to convert valid models correctly. Since both formats for these models are obtain through external library plugins, only a limited amount of failure testing was required, as for example JSBML already takes care of validating SBML-models. Of course, the converter cannot do completely without failure tests, as it only can convert valid SBML-qual models. It was therefore, necessary to implement some error handling and failure testing to assert that only valid SBML-qual models were converted. In case of non-SBML-qual models, the converter simply reports an error and the conversion fails.

8 Demo

This section demonstrates the conversion of Boolean network from SBML to ERODE and vice versa. Particularly, section 8.1 introduces the network and briefly describes its SBML representation. Section 8.2 gives a short overview of the demo-program.

8.1 The example network

For this demonstration an example network has been created to cover the full extent of the converter's operator interface. The Boolean network is defined as follows:

- Given the set of species $S = S_1, S_2, S_3$
- Given the set of update functions $F = f_1, f_2, f_3$

The update functions F are defined as:

$$\begin{array}{lcll} f_1(S(t)) & = & S_1(t+1) & = & S_2(t) \oplus S_3(t) \\ f_2(S(t)) & = & S_2(t+1) & = & S_1(t) \wedge \neg S_3(t) \\ f_3(S(t)) & = & S_3(t+1) & = & S_1(t) \rightarrow S_2(t) \end{array}$$

Figure 8.1: The update function definitions

This network definition was then used to create the *DemoNetwork.sbml*-file. Due to the length of the file, only a few excerpts from it are included in this report. The full length-file can be found in the repository on Github.

```
<qual:listOfQualitativeSpecies xmlns:qual=
"http://www.sbml.org/sbml/level3/version1/qual/version1">
    <qual:qualitativeSpecies qual:compartment="default"
        qual:constant="false"
        qual:id="S_1" qual:initialLevel="0"
        qual:maxLevel="1"/>
    <qual:qualitativeSpecies qual:compartment="default"
        qual:constant="false"
        qual:id="S_2" qual:initialLevel="0"
        qual:maxLevel="1"/>
    <qual:qualitativeSpecies qual:compartment="default"
        qual:constant="false"
        qual:id="S_3" qual:initialLevel="0"
        qual:maxLevel="1"/>
</qual:listOfQualitativeSpecies>
```

Listing 8.1: The species definitions of the network

As shown in the SBML-species definitions, all species share the same compartment and are non-constant. Due to being in the Boolean domain $0, 1$, their maximum level has been set to 1 and they all have been assigned an initial level of 0.

```
<qual:transition>
  <qual:listOfInputs>
     <qual:input qual:id="Input0"
     qual:qualitativeSpecies="S_2"
     qual:transitionEffect="none"/>

     <qual:input qual:id="Input1"
     qual:qualitativeSpecies="S_3"
     qual:transitionEffect="none"/>
  </qual:listOfInputs>

  <qual:listOfOutputs>
     <qual:output qual:id="Output0"
     qual:qualitativeSpecies="S_1"
     qual:transitionEffect="assignmentLevel"/>
  </qual:listOfOutputs>

  <qual:listOfFunctionTerms>
     <qual:functionTerm qual:resultLevel="1">
        <math xmlns="http://www.w3.org/1998/Math/MathML">
           <apply>
              <xor/>
                 <apply>
                    <eq/>
                    <ci> S_2 </ci>
                    <cn type="integer"> 1 </cn>
                  </apply>
                  <apply>
                     <eq/>
                     <ci> S_3 </ci>
                     <cn type="integer"> 1 </cn>
                  </apply>
               </apply>
            </math>
         </qual:functionTerm>
         <qual:defaultTerm qual:resultLevel="0"/>
      </qual:listOfFunctionTerms>
   </qual:transition>
```

Listing 8.2: The transition F_1 in SBML format

As seen in the figure above, the list of *outputs* contains a reference to the species S_1, since it is updated by the transition. The list of *inputs* contains species S_2 and S_3, as they are referenced in the update condition, and the function term contains the AST of the update condition. It should be noted, the SBML-representation of the BN used in this example is represented using multi-valued notation, as it is common practice to do so in SBML.

8.2 An overview of the demo program

As it can be seen in figure 8.2, the *SBMLExporter* program, performs can be divided into three segments. In the first segment, the program will use the path to the SBML-file to read file and parse it to SBML's data representation. The parsed data representation is then available as an *SBMLDocument*-object.

```
public class SBMLExporter {
    /**
     * Small demo converting an .sbml file to .ode format
     * and back
     * @param args
     */
    public static void main(String[] args) throws IOException, XMLStreamException {
        //Replace the string below with the full path to the file
        String path = "Path to file";
        SBMLDocument sbmlDocument = (SBMLDocument) SBMLManager.read(path);

        //Convert to ERODE format
        ISBMLConverter converter = SBMLManager.create(sbmlDocument);
        GUIBooleanNetworkImporter guiBooleanNetworkImporter = converter.getGuiBnImporter();
        GUIBooleanNetworkImporter.printToBNERODEFIle(
            guiBooleanNetworkImporter.getBooleanNetwork(),
            guiBooleanNetworkImporter.getInitialPartition(),
            "DemoNetwork.ode",
            null, true, null, null, false);

        //Convert back to SBML Format
        converter = SBMLManager.create(guiBooleanNetworkImporter.getBooleanNetwork());
        sbmlDocument = converter.getSbmlDocument();

        print(sbmlDocument);
    }

    private static void print(SBMLDocument sbmlDocument) {
        String writePath = System.getProperty("user.dir");
        System.out.println("Working Directory = " + writePath);
        try {
            File sbmlFile = new File("DemoSBMLFile.sbml");
            if(sbmlFile.createNewFile()) {
                System.out.println("Created file: " +
                    sbmlFile.getName() +
                    "at path: " +
                    sbmlFile.getPath()
                );
            }
            System.out.println("Writing to file...");
            SBMLWriter.write(sbmlDocument, sbmlFile, "SBMLConverter", "1.0");
            System.out.println("Finished");
        } catch (IOException | XMLStreamException e) {
            System.out.println("An error occurred");
            e.printStackTrace();
        }
    }
}
```

Figure 8.2: The demo program

The second segment uses, the *SBMLDocument* to execute the conversion to ERODE format. The conversion is execute by initialising the *SBMLConverter* using the *SBMLManager*'s `create()`-method. Once the conversion is complete, the *GUIBooleanNetworkImporter* can be retrieved from the converter, an object containing the network model in ERODE format. ERODE's model representation is then printed to the *DemoNetwork.ode*-file using ERODE's `printToBNERODEFIle()`-method. This results in the following network in
ERODE-format:

```
begin Boolean network DemoNetwork
 begin init
  S_1
  S_2
  S_3
 end init
begin update functions
  S_1 = (S_2 XOR S_3)
  S_2 = ((S_1 = true)&(S_3 != true))
  S_3 = (S_1 -> (S_2 = true))
end update functions

end Boolean network
```

Listing 8.3: The ERODE formatted network

In the last segment of the program shown in figure 8.2, the converts the ERODE representation back to SBML-qual. Just like in the conversion to ERODE format, a new converter is initialized using the *SBMLManager*'s `create()`-method. This time however, it is given ERODE's data representation instead. Once the conversion is finished, the SBML representation is retrieved by calling the *getSbmlDocument()*-method. The file is then printed to the *DemoSBMLFile*. When comparing the original input file with the newly generated file, using a file comparison tool, it can be seen that the contents of the file are identical.

9 Future Steps

Considering the current state of the converter and the functionality it provides, there are a few things that could be added to the converter with further development. Suggestions for how some of these features can be implemented, can be found in the appendix.

For one, the current converter is unable to support multi-valued networks. This however, requires ERODE to be extend to support them first. It should also be taken into account that such an extension can turn into very big task, since there is a larger set of network-operations that can be performed on networks in the multi-valued domain. For example, the multi-valued representation of SBML also support the consumption of transition inputs, and production of outputs. This means, that inputs to a transition can be reduced during a transition and outputs increased, instead of overwritten.

Apart from that, there are also a series of small optimisation and extensions that can be done, that do not require ERODE to be updated. For example, algorithms that minimize the network representation could be added. This could include an update function analysis that joins output species in the same transition, given they have the same update conditions. Another option could also be to feature algorithms that can reduce a function terms logical expression to its most compact and concise form.

A final option for further development could also be to optimize program performance by introducing concurrency. In its current state, the program runs on a single thread, since performance was of no great import. However, there are many places in the program where the execution branches of in different directions, where each of these new branches could be executed concurrently.

10 Summary

Overall, this report discusses the conversion between the SBML-qual and the ERODE format. A converter, capable of converting back and forth between these formats, was successfully implemented. The converter is based upon principles that render the implementation maintainable. In other words, one can easily extend and modify the converter if further development is needed.

To achieve this goal, several milestones, representing each stage of the project, were defined. During the first stage, the two formats haven been analyzed to become familiar with them and obtain an overview of the conversion requirements. In the second and third stage the knowledge about the conversion requirements was then used to design and implement the converter application. Initially, a few small demo-applications were create to develop a framework for the final application. This framework was then extended into the final converter by adding new functionality iteratively. During each iteration each new piece of functionality was tested thoroughly to ensure that the formats are converted correctly. An important part of the converter design was also to structure it in a way that allows an integration into other host programs.

The development process was guided by a project plan to track the overall progress of the project. Additionally, weekly meetings with the project supervisors. Regarding the project management, a suitable approach has been chosen. The beginning of the project included a variety of demos and experiments building the foundation for the final product. The weekly meetings helped tackle problems early and find the best way to advance in each development stage.

References

Cardellia, Luca et al. (n.d.). *ERODE Website*. URL: https://www.erode.eu/.

Dräger, Andreas, Nicolas Rodriguez, et al. (Apr. 2020). *sbmlteam/jsbml: JSBML is a community-driven project to create a free, open-source, pure Java™ library for reading, writing, and manipulating SBML files (the Systems Biology Markup Language) and data streams. It is an alternative to the mixed Java/native code-based interface provided in libSBML.* URL: https://github.com/sbmlteam/jsbml.

Porter, Brett, Jason van Zyl, and Olivier Lamy (July 2021). *Welcome to Apache Maven*. URL: https://maven.apache.org/.

Hucka, Michael et al. (2018). "The systems biology markup language (SBML): language specification for level 3 version 2 core". In: *Journal of integrative bioinformatics* 15.1.

Sonatype (n.d.). *maven central repository search*. URL: https://search.maven.org/.

SmartBear (2021). *Tools & techniques that elevate teams to greatness*. https://cucumber.io/. URL: https://cucumber.io/.

Argyris, Georgios et al. (2021). "Reducing Boolean Networks with Backward Boolean Equivalence". In: *arXiv preprint arXiv:2106.15476*.

Chaouiya, C. et al. (2015). "SBML Level 3 package: Qualitative Models, Version 1, Release 1". In: *Journal of Integrative Bioinformatics* 12.2, pp. 691–730. DOI: 10.1515/jib-2015-270.

SBML Specifications (Apr. 2019). URL: http://sbml.org/Documents/Specifications.

A Appendix

A.1 Getting started with JSBML

The most important thing to know about JSBML is that this library mostly is a literal implementation of the SBML-structures defined in the SBML specifications Hucka et al. 2018 Chaouiya et al. 2015. The JSBML-classes represent the structures defined in SBML, providing getter- and setter methods to populate their fields.

In figure A.1 the steps required from reading an SBML-file to the extraction of an SBML-qual model's species are shown. After specifying the path to the SBML-file to be read, JSBML's *SBMLReader* is used to read the provided file. This returns an object of type *SBase*, an abstract class used for all SBML-object types, that stores meta-data about the object, such a the SBML-level and version used to represent it. To gain access to the SBML-file data representation, it need to be cast to the *SBMLDocument*-type used to represent SBML-files. The *Model* it contains can the be retrieved by calling the `getModel()`-method.

Model represented by one of the SBML-extensions are not represented by the *Model*-class itself, but instead by a separate model-class. In case of SBML-qual, this is the *QualModelPlugin-class*, which can be retrieve from the main *Model* by calling the `getExtension()`-method using "qual" as an key to the extension map. In the last step of the figure, the species are then retrieved using the appropriate getter-method.

Retrieving SBML-transitions an the structures contained within, works in the same manner as with species and the other SBML-structures, which is why these are not included as a demo in this report. However, the *JSBMLReadingDemo*-class in the *sbml.demos* package in the repository contains a full demo of the code used to read the various SBML structures required by the converter.

```
String path = "Insert path to sbml file here";

//Read the SBML file and retrieve the data representation
SBase sbmlEntity = SBMLReader.read(new File(path));
SBMLDocument sbmlDocument = (SBMLDocument) sbmlEntity;

//Retrieve the model contained in the document
Model model = sbmlDocument.getModel();

//Retrieve the SBML-qual extension
QualModelPlugin qualModel = (QualModelPlugin) model.getExtension("qual");

//Retrieve the model's species
ListOf<QualitativeSpecies> species = qualModel.getListOfQualitativeSpecies();
```

Figure A.1: Species retrieval using JSBML

When creating object to represent SBML-structures is very important to use a consistent SBML-level and version for all object in a model representation, as JSBML will complain and throw exceptions otherwise. While almost all SBML-structures have default constructors that allow their creation without specifying level and version, it is a good idea to specify them anyway, as unset version also can cause several problems. In the converter's implementation this problem has been solved by adding a *configuration*-package, which is used to set SBML-level and version for the entire converter.

In figure A.2, it is shown how to create a small (and empty) SBML-model with an SBML-qual extension. Using either *set*- or *add*-methods, the various structures are added to their parents. When dealing with instances of the *ListOf*<>-structure it is important to the parent of those lists to be the *Model*, they are contained in. If this is not done, JSBML will not recognize these lists as part of the model and throw an exception.

```
11  public class JSBMLWritingDemo {
12
13      private static final int LEVEL = 3;
14      private static final int VERSION = 1;
15
16      public static void main(String[] args) throws IOException, XMLStreamException {
17
18          //Demo: Adding an extension to an existing SBML Document
19
20          //Initialisations
21          SBMLDocument sbmlDocument = new SBMLDocument(LEVEL, VERSION);
22          Model model = new Model(LEVEL, VERSION);
23          QualModelPlugin qualModel = new QualModelPlugin(model);
24
25          //Create model content
26          ListOf<QualitativeSpecies> species = new ListOf<>(LEVEL, VERSION);
27          ListOf<Transition> transitions = new ListOf<>(LEVEL, VERSION);
28
29          //Add model to the SBML representation
30          sbmlDocument.setModel(model);
31
32          //Add SBML-qual extension
33          model.addExtension("qual", qualModel);
34
35          //Add model content to the extesion model
36          species.setParent(qualModel.getModel());
37          qualModel.setListOfQualitativeSpecies(species);
38
39          transitions.setParent(qualModel.getModel());
40          qualModel.setListOfTransitions(transitions);
41
42      }
43  }
```

Figure A.2: SBML model generation using JSBML

When dealing with the JSBML-functionality used in the converter, the last important part is to know how to create the logical expressions in the *FunctionTerm*-structures used by SBML. For this, the *ASTNode*-class is used, which represents the MathML-elements of an SBML-file.

A.3 shows the most important ways to construct *ASTNodes* in this converter. Depending on what input is given to the constructor, the *ASTNode* will represent a different element. As shown in the figure, variables can be create by providing the variable name in form of a string. Integer constants on the other hand are created by providing an integer value. To create other types, such as Boolean constants and operator, the *ASTNode*-class also provides the *Type*-enum, which defines all operators and special constants *ASTNodes* can represent. The other figure (A.3) shows how to initialise and retrieve data from an AST consisting of *ASTNode*-objects.

```java
public class ASTNodeCreationDemo {

    public static void main(String[] args) {
        ASTNode variable = new ASTNode("variableName");

        ASTNode integerConstant = new ASTNode(1);

        ASTNode booleanConstant = new ASTNode(ASTNode.Type.CONSTANT_FALSE);

        ASTNode operator = new ASTNode(ASTNode.Type.LOGICAL_AND);
    }
}

```

Figure A.3: Examples of ASTNodes representing elements of an AST

```java
public class ASTNodeLinking {

    public static void main(String[] args) {
        //Creating an AST
        ASTNode root = new ASTNode(ASTNode.Type.LOGICAL_AND);
        ASTNode leftChild = new ASTNode(ASTNode.Type.CONSTANT_FALSE);
        ASTNode rightChild = new ASTNode(ASTNode.Type.CONSTANT_TRUE);

        root.addChild(leftChild);
        root.addChild(rightChild);


        //To retrieve children of the AST root
        List<ASTNode> children;
        if(root.getAllowsChildren()) {
            int childCount = root.getChildCount();
            children = root.getChildren();

            //To retrieve a specific child
            ASTNode child = (ASTNode) root.getChildAt(0);
        }
    }
}

```

Figure A.4: Example of an AST initialisation and data retrieval from it

A.2 Extending the converter to support MNs

To add support for multi-valued models in the converter only requires minor changes to the existing functionality. In total only four different methods need to be extended or modified. Before explaining the required changes, it needs to be mentioned that the classes used to represent integer-values in ERODE, **must implement** the *IUpdateFunction*-interface for this modification to work. Should this not be the case, the node conversion module in the *sbml.conversion.nodes* package would need to be replaced by a different module that also implements the *INodeConverter*-interface.

To extend the converter using the existing modules, the first method that needs to be modified, is the `create()`-method in the *NodeManager*-class shown in figure A.5. This method analyzes the given update function element from ERODE's AST to analyze its type. Depending on its type, it then calls the corresponding manager to continue the converter initialisation. Since the missing update function type is a type representing integers, this is part of the *ValueASTConverter*'s responsibilities.

The simplest way to modifiy is method is to move the
`return ValueASTConverter...` statement to the default case, replacing the current exception. Once that is done the *REFERENCE*,*TRUE* and *FALSE* cases can be removed.

Since the exception in the default case currently only serves the purpose of catching the new update function type(s), once ERODE is extended it will no longer be needed.

```
public static INodeConverter create(IUpdateFunction updateFunction) {
    Class<?> classType = updateFunction.getClass();
    String className = classType.getSimpleName();
    switch (className) {
        case Strings.BINARY_EXPRESSION:
            return BinaryASTConverter.create((BooleanUpdateFunctionExpr)updateFunction);
        case Strings.NEGATION:
            return UnaryASTConverter.create((NotBooleanUpdateFunction)updateFunction);
        case Strings.REFERENCE:
        case Strings.TRUE:
        case Strings.FALSE:
            return ValueASTConverter.create(updateFunction);
        default:
            throw new IllegalArgumentException("Unknown update function type");
    }
}
```

Figure A.5: The `create()`-method used when converting to SBML

The next method to be changed, is the *convert()*-method in the *ValueWriter*-class. This method simply requires an additional case, to take the new update function type into account. The *currentNode* for the new case should be set to the result of the *element.constant()*-call.

```
@Override
protected void convert() {
    Class<?> classType = updateFunction.getClass();
    if(classType.equals(ReferenceToNodeUpdateFunction.class))
        this.currentNode = element.reference(updateFunction);
    else
        this.currentNode = element.booleanConstant(updateFunction);
}
```

Figure A.6: The `convert()`-method of the *ValueWriter*-class

Once these steps have been performed, the actual conversion step for the integer representations need to be implemented in the *SBMLElement*-class. This class currently contains an unused method, called `constant()`, taking an *IUpdateFunction*-instance as input and returns an *ASTNode*, using the *ASTNodeBuilder*. The current method body, can be replaced entirely, since this method will take integer representations as input and not Boolean constants. The method body would then simply require some way of reading the *IUpdateFunction*'s integer value and pass it to the *ASTNodeBuilder*'s `integer()`-method, which creates integer constants.

```
@Override
public ASTNode constant(IUpdateFunction node) {
    Class<?> classType = node.getClass();
    if(classType.equals(TrueUpdateFunction.class))
        return builder.integer(1);
    else
        return builder.integer(0);
}
```

Figure A.7: The current `constant()`-method in the *SBMLElement*-class

Finally, the `constant()`-method, implemented by the *ERODEElement*, also requires adjustment, since it currently translates SBML-integers to ERODE-booleans. The new version should return ERODE's integer-representation instead.

```java
@Override
public IUpdateFunction constant(ASTNode node) {
    switch (node.getInteger()) {
        case 1:
            return new TrueUpdateFunction();
        case 0:
            return new FalseUpdateFunction();
        default:
            throw new IllegalArgumentException("Given node value is not a boolean value");
    }
}
```

Figure A.8: The current `constant()`-method in the *ERODEElement*-class

Technical
University of
Denmark

Richard Petersens Plads, Building 324
2800 Kgs. Lyngby
Tlf. 4525 1700

www.compute.dtu.dk