Simulink Model Transformation for Backwards Version Compatibility

Bhisma Adhikari, Eric J. Rapos, Matthew Stephan
Computer Science & Software Engineering
Miami University
Oxford, OH, USA
{adhikab,rapose,stephamd}@miamioh.edu

***Abstract*—Simulink is a leading modelling language and data-flow environment for Model-Driven Engineering, prevalent in both industrial and educational contexts. Accordingly, there are many standalone publicly-available tools for analyzing and using Simulink models for various purposes. However, Simulink's model format has evolved to a new proprietary format, rendering many of these tools useless. To combat this, we devise an approach, SLX2MDL, that applies transformation rules based on Simulink syntax to transform the new SLX format models to models conforming to the legacy MDL syntax. The resulting approach enables backwards compatibility with existing tools, including previous versions of Simulink itself. Our 4-phase process includes analysis and extraction, merging and transformation of the common elements, transformation of the specialized Stateflow elements, and output production. We position this problem within the literature by comparing and contrasting similar, but insufficient, related approaches. We evaluate and validate SLX2MDL by applying it to 543 standard and publicly available models from an established and curated corpus. Our evaluation demonstrates 100% validity and correctness on these models based on functional equivalence. Further, we evaluate our approach's performance and find it consistent and scalable as model size and complexity increases.**

***Index Terms*—model transformation, Simulink, backwards compatibility, versioning, model evolution, model syntax**

I. Introduction

Similar to problems faced by legacy code, software modelling languages evolve over time. This presents challenges for tools and technologies based on deprecated language definitions. One significant example of this is the evolution of Simulink model formats. Simulink is a modelling language, which is part of the Matlab environment, that allows engineers to create executable representations of a variety of systems. A new model format was introduced that altered significantly the way Simulink models were stored in files. Both formats are still supported to some extent today. The change was essentially an underlying textual syntax shift from a plain text representation (MDL) to an archive file containing various hierarchical and model-sensitive XML-based representations (SLX). These models, having a different underlying textual syntax representation, are not backwards compatible. As a result, they cannot be opened in older versions of Simulink. Additionally, many of the supporting tools that were designed based on the MDL format do not work with the new format.

Simulink itself possesses some facilities to convert between these model formats. However, many standalone and/or external tools work with Simulink models in only their original MDL format. They are no longer capable of analyzing models created in newer versions of Simulink. In this article, we describe our design and application of rule-based model transformations to facilitate automatic standalone conversion of Simulink SLX files to corresponding Simulink MDL files. This involves model analysis, including contextual and structural concerns, and corresponding rule-based manipulation of models so they conform to the MDL syntax and are thus compatible with the dearth of existing MDL-based tools.

A. Motivation

There are several standalone tools that rely explicitly on the MDL file format. Some of these applications focus on model clone detection, for example, Simone [1], ConQAT [2], and many others [3]. Other notable examples include safety analysis [4], synthesis of fault trees [5], and abstract simulation [6]. Rather than requiring each of these independent tools to re-implement their functionality for the updated model format, it became evident to us and others that a single and open approach to transform models from one format to another, in the form of a preprocessor, would be an ideal solution with meaningful impact. Furthermore, due to financial and/or technical barriers, some users rely explicitly on older versions of MATLAB/Simulink that are incapable of opening models in the new SLX format nor able to automatically convert SLX models to the existing MDL format. This may especially be true in the software analysis domain where researchers and engineers may be more interested in the artifact analysis than their actual use within the development environment. While conversion is possible in newer versions of Simulink, it is not something readily available to all users. Thus, in the spirit of openness and accessibility to all, we pursue this research. One such example, from the MATLAB Central forum, motivates this explicitly. Regarding different file formats, a user indicates that they '...prefer the MDL file and actually in [their] work [they] need the plain text MDL file'[1]. Another user responding to that post replies, "I do not have Simulink, but from time to time I can debug simulink problems by reading the text MDL file." This is one example

[1]https://www.mathworks.com/matlabcentral/answers/60706-newbie-question-about-simulink-new-file-format-slx

from multiple on Matlab Central and other forums. For these reasons and more, it is important to research, devise, and apply an open and automatic transformation-based approach capable of transforming SLX-based models to the corresponding MDL syntax representations. It must be standalone/external to Matlab to remove barriers caused by requiring any specific version of MATLAB/Simulink.

B. Contributions

Through this work, we make the following contributions,

- Empirically derived definitions of SLX and MDL syntax for the domain-specific visual language Simulink
- Implementation and application of an automatic model transformation and supporting tool support in the form of SLX2MDL, which can transform SLX representations to MDL version representations while preserving the semantics
- An evaluation of effectiveness applying SLX2MDL on an established corpus of Simulink models

II. Background

In this section, we provide brief background on source/syntax transformation and Simulink. Given the context of this article, we presume that the readers have sufficient background in Model-Driven Engineering.

A. Source Transformation

Source transformations take many forms, but the term generally applies to the conversion of program or artifact source from one form to another following some set of rules changing the syntactical elements, the semantics, or both. One such example, and one of the most popular implementations of this type of transformation, is the TXL source transformation language[7]. TXL implements a standard source transformation, and is the most closely related to our work. However, a source transformation may take other forms and employ other techniques, such as reverse engineering/design recovery [8], software reengineering/restructuring [9], or forward engineering or metaprogramming [10]. Any source/syntax transformation tool relies on the conformance to input and output language definitions (grammars), and a mapping between them (transformation rules). Ours is another example and application of such a transformation technique in that we are transforming the syntactical elements from one format to another while being sensitive/aware of the semantics denoted by the Simulink models.

B. MATLAB Simulink

Simulink is a combination textual and data-flow graphical programming environment that leverages many of the benefits of MDD to design and implement quality software. There are a variety of domains that leverage Simulink including wireless communications, power electronics control design, control systems, signal processing, robotics, advanced driver assistance systems, image processing and computer vision, and more. Additionally, it is employed at many academic institutions in a variety of STEM (Science, Technology, Engineering, and Mathematics) disciplines. Simulink is a visual language that employs model artifacts that represent the structure and behavior of a system to facilitate execution, perform analysis, and ultimately generate code for deployment.

III. Language and Format Preliminary Analysis and Observations

Our work exploits the fact that the textual syntax used by Matlab to store Simulink models was migrated from the original MDL format to a new SLX format. The SLX format was introduced in Simulink version R2012a, and was made the default format to store Simulink models beginning from Simulink version R2012b [11]. In conducting our research and realizing our goals in devising an open and automatic approach for transforming SLX-syntax models to MDL-syntax models, we first performed an analysis of the two different formats. This was necessary to empirically derive sufficient definitions of the source and target meta-models, as well as rules to enable the transformations. Our meta-model definition process involved grammar analysis, format inspection, and documentation of the model source syntax. We highlight and discuss this analysis in this section. Due to our use of the underlying textual nature of Simulink model representations, and our specific transformation approach, graphical representations of the meta-models were not needed. Instead the language definitions are represented explicitly and directly in the source code for our process. A formal validation of these meta-model definitions occurs through their use in the SLX2MDL process and its evaluation.

A. MDL Format

The MDL model format is a textual file format used to store Simulink models. This is a proprietary file format owned by Mathworks, the organization responsible for Simulink. The standard syntax to denote different file components is not available publicly. We leverage existing work devising a TXL grammar for MDL format in describing the MDL file structure [1].

In MDL format, the Simulink syntax uses key-value pairs. Each key-value pair constitutes a *default_element*. While the *key* of a *default_element* is always an unquoted string of characters, referred to as an *id* in the TXL grammar, the *value* can take one of several different forms. Based on these different forms, a *default_element* can be either a *default_single_element* or a *default_list_element*. The *value* of a *default_single_element* can be either a number, or a string, or a list of numbers. In contrast, the *value* of a *default_list_element* is a list of other *default_element*s can be within a pair of braces. We use the terms *default_list_element* and *block* interchangeably from hereon. A *default_element* can contain one or more nested *default_element*s. As a result, the overall structure of an MDL file contains the Simulink model information organized as hierarchical key-value pairs.

Listing 1, which we derived from our analysis, shows a typical hierarchical structure of an MDL file. Not all *default_element*s are shown in this listing for the sake of

brevity. The *default_element* with "Model" as its *key* lies at the top of the MDL hierarchy. This *default_element* consists of other *default_single_element*s, for example, *Name*, and *default_list_element*s for example, *Graphical Interface*. While the vast majority of Simulink models have *Model* as their topmost *default_element*, some models may have *Library* or *Subsystem* instead. As illustrated in Listing 1, it is important to note the *Stateflow* block is not nested within the *Model* or *Library* or *Subsystem*. It occurs as a sibling of the *Model* block rather than its child in terms of the MDL hierarchy. Some other blocks that occur as a sibling of the *Model* are *MatData*, and *MatResources*.

```
Model {
    Name "my simulink model"
    Version 10.0
    GraphicalInterface {}
    Array {
        Simulink.ConfigSet {
            Array {
                Simulink.SolverCC {}
                Simulink.DataIOCC {}
                Simulink.OptimizationCC {}
                Simulink.DebuggingCC {}
                Simulink.HardwareCC {}
                Simulink.ModelReferenceCC {}
                Simulink.SFSimCC {}
                Simulink.RTWCC {}
                SlCovCC.ConfigComp {}
                hdlcoderui.hdlcc {}
            }
        }
    }
    BlockDefaults {}
    AnnotationDefaults {}
    LineDefaults {}
    MaskDefaults {}
    MaskParameterDefaults {}
    BlockPaameterDefaults{}
    System {
        Block {
            System {
                Block {}
                Line {}
            }
        }
    }
}
Stateflow {
    machine{}
    chart {}
    state{}
    transition{}
    junction{}
    data{}
    event{}
    instance{}
    target{}
}
```

Listing 1. MDL File Syntax

B. SLX Format

An SLX file is an archive file that contains mostly XML files arranged in a defined hierarchical structure, with some additional non-XML files providing supplemental data. Similar to the MDL format, SLX is composed of a proprietary file format and syntax owned by Mathworks. According to Mathworks, "Saving Simulink models in the SLX format typically reduces file size and solves some problems in previous releases with loading and saving MDL files containing Korean and Chinese characters." [12].

We derived Listing 2 from our analysis. It illustrates a typical structure of an SLX archive file. The majority of the Simulink model contents are stored in the following files,

- simulink/blockdiagram.xml
- simulink/configSet0.xml
- simulink/bddefaults.xml
- simulink/graphicalInterface.xml
- simulink/stateflow.xml

```
decompressed-slx
    [Content_Types].xml
    _rels
    metadata
        coreProperties.xml
        mwcoreProperties.xml
        mwcorePropertiesExtension.xml
        mwcorePropertiesReleaseInfo.xml
        thumbnail.png
    simulink
        ScheduleCore.xml
        ScheduleEditor.xml
        _rels
            blockdiagram.xml.rels
            configSetInfo.xml.rels
        bddefaults.xml
        bdmxdata
            x.mxarray
            y.mxarray
            z.mxarray
        blockdiagram.xml
        configSet0.xml
        cofigSetInfo.xml
        graphicalInterface.xml
        modelDictionary.xml
        plugins
            AnimationPlugin.xml
            DiagnosticSuppressor.xml
            LogicAnalyzerPlugin.xml
            NotesPlugin.xml
            SLCCPlugin.xml
            WebScopes_FoundationPlugin.xml
        stateflow.xml
        windowsInfo.xml
```

Listing 2. Structure of SLX Archive File

Whereas supplemental model contents can be found in the following files,

- metadata/coreProperties.xml
- simulink/plugins/DiagnisticSuppressor.xml
- simulink/plugins/LogicAnalyzerPlugin.xml
- simulink/plugins/NotesPlugin.xml
- simulink/plugins/SLCCPlugin.xml
- simulink/plugins/WebScopes_FoundationPlugin.xml

Interestingly, in our analysis and experimentation, we found some files in the SLX folder that contain information with no direct correspondence to the MDL file format:

- _rels/.rels
- metadata/mwcoreProperties.xml
- metadata/mwcorePropertiesExtension.xml
- metadata/thumbnail.png

- simulink/_rels/blockdiagram.xml.rels
- simulink/_rels/configSetInfo.xml.rels
- simulink/modelDictionary.xml
- simulink/scheduleCore.xml
- simulink/ScheduleEditor.xml
- [Content_Types].xml

Similarly, the *simulink/bdmxdata/*.mxarray* files are present only in a few Simulink models. These are binary files and are used to store Simulink *workspace* data.

IV. SLX2MDL Approach and Implementation

Having completed our initial analysis and experimentation with the different formats, we decided to frame the problem as a model transformation that manipulates the underlying textual syntax while preserving the Simulink model semantics. Our solution considers and codifies a specification based on metamodels, and uses that to transform input instances to output instances. We overview our overall process in Figure 1.

Our process consists of several steps to create a resulting MDL file from an input SLX file. The basic flow of phases consists of analysis and extraction, specialized and separate transformation of *Stateflow* and non-*Stateflow* contents, and merging, as we show in Figure 2. We chose Python to implement our transformation algorithm because of its greater suitability and growing popularity for exploratory research projects [13]. We discuss alternatives in Section VII.

A. Extraction

SLX2MDL begins with the input SLX file. It is essentially a compressed archive file, which SLX2MDL decompresses and extracts all its files. SLX2MDL automatically produces an SLX folder containing these files, most of which are XML, organized in a well-defined SLX hierarchical structure. This structure is based on our observations and deductions from our earlier analysis. The extraction works on either individual model files or collections of models for batch processing of model sets.

B. Merging & Transformation of Non-Stateflow Content

In the SLX format, Simulink model data is distributed across mostly XML files, with a small number of binary and other files. In contrast, as we discovered during our preliminary analysis, the MDL format represents Simulink models in a single text file. Moreover, the model information in MDL format is contained entirely within a top-level *Model*, *Library*, or *Subsystem* block, with the exception of a special *Stateflow* block, which occurs outside these 3 blocks. Stateflow is a language within Simulink that allows for state diagrams and flow-charts. Therefore, before transforming each XML tag into its corresponding MDL representation, SLX2MDL merges together the file contents of all relevant XML files into one single XML file, except for *simulink/stateflow.xml*, which it processes separately. SLX2MDL arranges the contents of these XML files in the order in which they appear in the corresponding MDL format. SLX2MDL ensures the resulting merged file remains a valid XML file with the appropriate object type as its top-level tag during this intermediate stage.

Following SLX2MDL's merging of contents into a single XML file, *merged.xml*, SLX2MDL transforms each element to its corresponding MDL element individually, as each element type requires specific transformation rules. SLX2MDL represents each XML tag that it discovers from the source SLX model definition by its own internal class, with a set of features:

- a constructor for creating an instance of the class from an XML string. The input XML string must conform to XML metamodel.
- a public method *strmdl()*, which gives the MDL string representation (a *default_element*) of that XML element. The output string from this method must conform to the definition of *default_element* defined in the MDL metamodel. Thus, these methods from all classes that model XML tags define the SLX-to-MDL transformation rules collectively.

```
class Model(XmlElement):
    def __init__(self, strval, parent_xml):
        ...
        self.ps = []
        self.graphicalInterfaces = []
        ...
        for x in self.inner_xmls:
            if x.tag == 'P':
                self.ps.append(P.from_XmlElement(x))

            if x.tag == 'GraphicalInterface':
                self.graphicalInterfaces.\
                append(GraphicalInterface.\
                from_XmlElement(x))
            ...

    @classmethod
    def from_XmlElement(cls, xml_element):
        return Model(xml_element.strval,
                     xml_element.parent_xml)

    @property
    def strmdl(self):

        str_ = 'Model{\n'

        for x in self.ps:
            str_ += f'{x.strmdl}\n'

        for x in self.graphiclaInterfaces:
            str_ += f'{x.strmdl}\n'
        ...
        str_ += '}\n\n'
        return str_
```

Listing 3. Class definition of $< Model >$ tag

We show an example class definition for the *Model* class in Listing 3. This class models the $< Model >$ tag from the *simulink/blockdiagram.xml* file. After merging all relevant XML files into *merged.xml* file, SLX2MDL's transformation pipeline for non-stateflow content creates an instance of the *Model* class by passing the content of *merged.xml* as the input XML string to its constructor. In doing so, all nested XML tags contained inside the $< Model >$ tag; that is $< P >$ tags, $< GraphicalInterface >$ tags, and all other nested

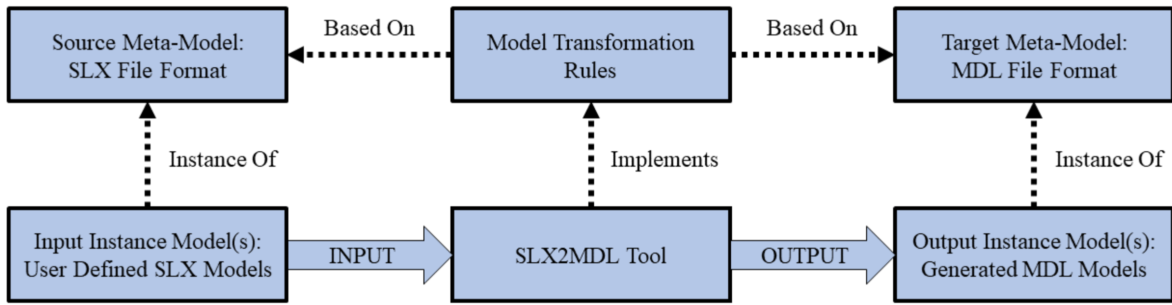

Fig. 1. SLX2MDL Transformation Overview

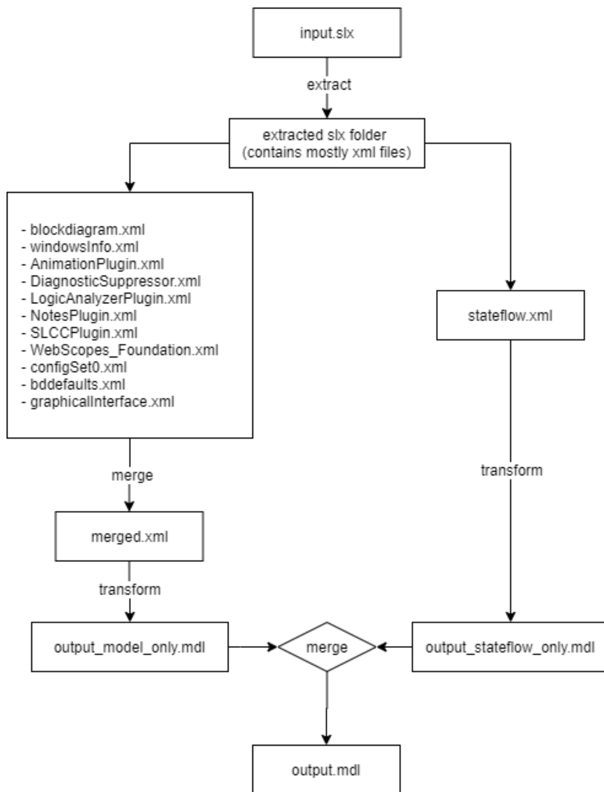

Fig. 2. SLX2MDL Implementation Flow

tags; are parsed by SLX2MDL. It then creates corresponding class instances as the attributes of the *Model* class instance. SLX2MDL performs this recursively for each nested XML tag until the content of the XML tag is a non-XML string, which is the recursive base case. The *strmdl()* method for each of these classes returns an MDL representation of the corresponding class instance by wrapping the MDL string representations (*default_elements*) of the nested XML tags in a new *default_element*. In short, calling the *strmdl()* method on an instance of class *Model* works recursively and returns the MDL representation of the entire *Model* block, which SLX2MDL then writes to the file *output_model_only.mdl*, as we illustrate in Figure 2.

An important consideration is that we specifically defined separate class definitions for each possible XML tag instead of defining a single generic class to model all XML tags collectively. We do this intentionally to make our SLX2MDL implementation more manageable and maintainable by addressing the nuances involved with the transformation of different XML tags separately. This modularization allows for the evolution of our language definitions to support yet-to-be discovered model elements that were not present at the time of our rule definition.

C. Stateflow Transformation

From our analysis, we determined that *Stateflow* content requires separate and particular handling for an effective transformation due to its unique nature compared to the other model elements. Specifically, the transformation of *Model* (or *Library* or *Subsystem*) elements versus that of *Stateflow* are different in nature and complexity as we describe below.

Firstly, main *Model* (or *Library* or *Subsystem*) transformations do not require any "flattening" of the XML hierarchy. The tree structure of the elements in the XML format remains almost the same even after transforming it to MDL format. In contrast, *Stateflow*'s transformation necessitates flattening the XML hierarchy such that "major" blocks; that is, *machine*, *chart*, *state*, *transition*, *junction*, *data*, *event*, *instance*, and *target*; appear immediately inside the *Stateflow* block rather than being nested within other blocks as in the *simulink/blockdiagram.xml* file. Secondly, the transformation of *Stateflow* requires generation of identifiers for different elements and keeping track of those identifiers in order to generate the "derived" attributes, such as *linkNode* and *treeNode*, correctly. We call these attributes "derived" as they do not occur explicitly in the XML model representation. These identifier-based attributes establish the *parent-child* and *siblings* relationships in the MDL representation of the *Stateflow* model. This was our conclusion in our analysis and experimentation, and is also corroborated by the works of Chen [14], [15] and Dominguez [16] focusing on clone detection and Stateflow to SMV translation, respectively. Such relationships in the SLX format are maintained implicitly within the XML tree hierarchy. Identifier generation needs to satisfy some uniqueness

constraints. Any error in this step can result in a corrupted *Stateflow* block. Thirdly, unlike the transformation of non-*Stateflow* content, the order of blocks matters for *Stateflow* transformations. For example, in a *Stateflow* block, an *instance* block must not appear before a referenced *machine* block. Lastly, the XML tags that appear in *simulink/stateflow.xml* are specific to *Stateflow* only. None of these tags, with the exception of the $< P >$ tag, appear in other XML files, nor do any of the tags from other files appear in *simulink/stateflow.xml*.

```
class Stateflow(StflXmlElement):
    def __init__(self, strval, parent_xml):
        ...
        self.ps = []
        self.machines = []
        ...
        for x in self.inner_xmls:
            if x.tag == 'P':
                self.ps.append(
                    P.from_StflXmlElement(x))

            if x.tag == 'machine':
                self.machines.append(
                    Machine.from_StflXmlElement(x))
            ...

    @classmethod
    def from_StflXmlElement(cls, stfl_xml_element):
        return Stateflow(stfl_xml_element.strval,
                         stfl_xml_element.parent_xml)

    @property
    def strmdl(self):
        str_ = 'Stateflow {\n\n'

        for x in self.attrs:
            if not x.name \
                in self._id_based_mdl_attrs:
                str_ += f'{x.name} "{x.value}"\n'

        for x in self.ps:
            if not x.name_attr.value \
                in self._id_based_mdl_attrs:
                str_ += f'{x.strmdl}\n'

        for x in self.machines:
            str_ += f'{x.strmdl}\n'
        ...
        str_ += '}'
        return str_
```

Listing 4. Class definition of $< Stateflow >$ tag

Except for these differences, the transformation of *Stateflow* proceeds similarly to our transformation of non-*Stateflow* content. SLX2MDL models all XML tags it discovers in the *simulink/stateflow.xml* file in separate classes. As an example, Listing 4 shows the class definition that models the $< Stateflow >$ tag found in *simulink/stateflow.xml* file. In order to "flatten" the XML hierarchy, we define the *strmdl()* method of some classes such that the hierarchical contents appear outside the closing brace, "}", of its XML parent block. We illustrate this in Listing 5 where we present the definition of the *strmdl()* method of class *Machine*. This class models the $< Machine >$ tag found in *simulink/stateflow.xml* file. In the MDL representation, SLX2MDL must move the *children* of *Machine* out of the *Machine* block such that they appear immediately inside the *Stateflow* block. Therefore, in this method definition, SLX2MDL writes them outside the closing brace of the *Machine* block. It writes the output of *Stateflow*'s transformation to the file *output_stateflow_only.mdl*, as per the process we illustrated in Figure 2.

```
@property
def strmdl(self):
    str_ = 'machine {\n'
    str_ += f'id {self.idmdl}\n'
    str_ += f'name "dummy_name"\n'

    if self.firstTarget:
        str_ += \
        f'firstTarget {self.firstTarget.idmdl}\n'

    for x in self.attrs:
        if not x.name in self._id_based_mdl_attrs:
            str_ += f'{x.name} "{x.value}"\n'

    for x in self.ps:
        if not x.name_attr.value in self.\
        _id_based_mdl_attrs:
            str_ += f'{x.strmdl}\n'

    for x in self.debugs:
        str_ += f'{x.strmdl}\n'

    str_ += '}\n\n'

    if self.children:
        str_ += f'{self.children.strmdl}\n'

    return str_
```

Listing 5. Method *strmdl* definition of $< Machine >$ tag

D. Producing Final Output via Merging

The final step in our transformation pipeline is to merge the intermediate files (*output_stateflow_only.mdl* and *output_model_only.mdl*) into a single result file representing the same Simulink model(s) but using the MDL syntax for the underlying text format. SLX2MDL concatenates the contents from these two files to produce the final output, which it then writes to the file *output.mdl*, as we showcase in the bottom right of Figure 2. We stress that not all Simulink models have *Stateflow* elements. For such models, *output_stateflow_only.mdl* is not applicable. The final step in the process is the removal of all intermediate files from the directory, leaving only the input SLX file, and the resulting MDL file which, by default, is renamed from *output.mdl* to have the same base name as the input model. For example, *Engine.slx* would produce an output model named *Engine.mdl*.

V. Application and Evaluation

To evaluate the effectiveness of our application of transformation to facilitate backwards compatibility in the form of SLX2MDL, we conducted an experiment converting SLX models to their corresponding models adhering to the MDL syntax. Specifically, we conducted a systematic evaluation applying SLX2MDL on a curated, public, and independently verified corpus of Simulink models [17]. This section describes the models we selected, the design of our evaluation experiment, and our results. SLX2MDL and all the instructions and

materials pertaining to our evaluation are available publicly for reproduction and replication purposes [18].

A. Materials - Model Selection

Chowdhury et al. have curated and published a large set of Simulink models intended to assist empirical research and tool development [17]. Their set is available online to all. We chose this set because of its large size, its models are from a variety of application domains (automotive, avionics, electronics, energy, robotics, and other), and its public availability allows our experiments to be replicated and reproduced by independent researchers. Furthermore, this set of models has been analyzed and validated independently for use in empirical research [19]. They do so in consultation with industry collaborators, concluding the models are sufficiently large and "mature" for research purposes and are diverse enough for good replication.

The corpus contained a total of 1166 Simulink models at the time of evaluation. Only 946 of these models are downloadable directly from the repository, with the rest available via links to download directly from Mathworks due to licensing requirements. Of these 1166 Simulink models, 547 are in SLX format, with the rest already in MDL format and, thus, not suitable for our experiment. We removed four of the SLX files from the evaluation dataset for various reasons: two of the models failed to load in Matlab Simulink, and the other two failed to upgrade to our baseline version of R2019b (see Section V-B1). As a result, we used 543 SLX-formatted Simulink models as the input for the evaluation of SLX2MDL.

B. Method - Experimental Design

When considering what constitutes a correct transformation result, we decided on two major criteria that must be met for each transformation: a valid model file conforming to the MDL syntax, and functional/semantic equivalence between the SLX-syntax and MDL-syntax model. In order to evaluate and validate our application of SLX2MDL for model transformation using the corpus of models, we performed a 5-step evaluation experiment, which we describe in this subsection.

1) Standardization of SLX Model Versions: The 543 SLX files of the evaluation dataset were created in various versions of Matlab Simulink, meaning they are encoded with different SLX versions. Even within the SLX model format, there exist differences between versions. Since we designed SLX2MDL based on recent features, it works best for SLX versions corresponding to Matlab Simulink R2017b or newer. Thus, in order to have a uniform version for evaluation and our subsequent claims, we first converted all input models to a recent version (R2019b) using Simulink's built-in conversion functionality. Additionally, by upgrading all input SLX files to one single recent version, we are able to better ensure reproducibility of our evaluation experiments.

2) Simulink-Based Conversion for Baseline Comparison: For each model stored in SLX format, we first used Simulink to convert the SLX model to a corresponding MDL format using its internal proprietary conversion algorithms available only in recent versions of Matlab. We do this to obtain our baseline ground truth in the form of an example "valid" and equivalent MDL file for later comparison.

3) SLX2MDL Conversion: After obtaining the baseline, we independently converted each model using SLX2MDL to generate an MDL model file. We completed this process using a batch processing feature of SLX2MDL that we added for faster processing of a large group of models.

4) Verifying Model Validity in Simulink: For each of the models we converted using SLX2MDL, we opened them manually within Simulink to demonstrate that the resulting model conforming to the MDL syntax is valid, well-formed, and capable of opening without issue. We performed a detailed and methodical (point-by-point) inspection of each and every model to ensure that no errors were present. If the model opened successfully with no errors present, we noted that the model was the result of a valid transformation.

5) Functional Simulink Model Comparison: The next aspect of evaluation was to ensure that the resulting model was functionally (semantically) equivalent to the one produced using the proprietary Simulink conversion. To do this, we leveraged the built-in comparison tool provided by Simulink, which is capable of identifying any differences between two models.

Based on our preliminary analysis of the two syntaxes, there were a number of differences that we considered acceptable/allowable differences for the purpose of evaluation. These include graphical changes and other non-semantic differences, which is consistent with other work in model comparison [3], as well as any variations in Model Configuration Parameters. For the latter, this is based on the fact that our preliminary analysis and experimentation determined that there was no discernible equivalence between the original SLX and the Simulink output MDL file with respect to these parameters. As a result, we were unable to codify transformation rules for these model aspects. We further deem this as acceptable given that these parameters relate to simulation configuration rather than the actual model content and structure itself, thus excluding these types of differences does not impact our definitions of correct model transformation. To summarize, for each comparison, if the MDL-syntax conforming model generated by SLX2MDL was equivalent to the one generated by Simulink, omitting the above exceptions, we recorded the transformation as correct.

6) Performance Evaluation: As an added level of evaluation, we examined the performance of our transformations to ensure that the transformation was not only feasible, but also that as models increase in size and complexity, the time required to convert between SLX and MDL does not increase inordinately. To that end, we decided to measure the execution time of each transformation and record it for each model in the evaluation set. Further for each model, we determined its size, defined as the number of lines in the resulting MDL file. While size is not solely based on the number of source lines, this provides a consistent metric to with which to measure against performance.

TABLE I
EXPERIMENTAL RESULTS OF MODEL TRANSFORMATION APPLICATION

Repository	Input SLX files	Valid	Equivalent
GitHub	335	335 (100%)	307 (91.6%)
Matlab Central	171	171 (100%)	132 (77.2%)
Source Forge	17	17 (100%)	17 (100%)
Other	20	20 (100%)	15 (75%)
TOTAL	**543**	**543 (100%)**	**471 (86.7%)**

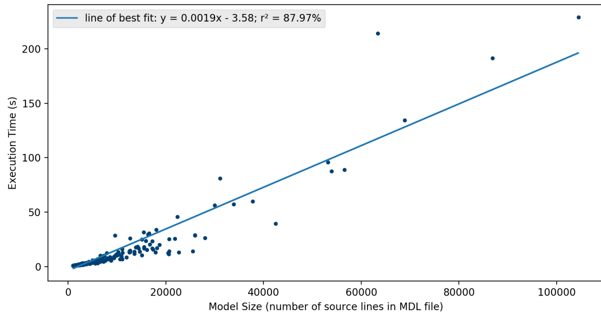

Fig. 3. SLX2MDL Execution Times as a Function of Model Size

VI. RESULTS & DISCUSSION

A. Results

For each model in the input set we recorded a result for our two criteria: model validity and functional equivalence. Additionally, we also recorded the time the transformation took and the number of lines in the resulting MDL file to observe and evaluate the transformation performance. We display the results of our evaluation as totals in Table I. Through our evaluation, we determined that 543/543 input SLX example models (100%) produced valid output MDL models, capable of opening in Simulink. Further, we found 471/543 of the same evaluation models (86.7%) to be functionally equivalent to those produced using the Simulink conversion algorithms. In the discussion, we consider the remaining non-equivalent models.

In terms of performance for the evaluation model set, SLX2MDL requires an average execution time of 6.29 seconds per model. On average, a Simulink model in the evaluation dataset contains 5163 lines of text in MDL format. Perhaps more interestingly, Figure 3 plots the execution time for SLX2MDL to perform a transformation against the size of the model, measured by the number of source lines in the resulting MDL file. We added a line of best fit to approximate the execution time as a function of model size with the following equation, $exec_time(s) = 0.0019 * num_mdl_src_lines - 3.58$. This line presents an r^2 value of 0.8797, indicating strongly that the execution time scales linearly as model size increases. This indicates that as the model size increases, the execution time increases linearly, boasting an O(n) complexity.

B. Discussion

Although SLX2MDL is not able to produce corresponding MDL blocks for binary files (data files and image files) in the SLX format, the converted MDL files are still valid Simulink models that load without issue in Simulink based on our point-by-point inspection.

All the models we found not to be functionally equivalent are a result of missing information from binary files that cannot be obtained directly from the SLX format due to their proprietary encoding. This is acceptable, however, as the data contained in these binary files relates only to the storage or workspace data and image information, none of which have an impact on the function or structure of the model itself. Given this, if we add this context/exception to our definition of equivalence, all the models are functionally/semantically equivalent. Doing so brings the total of functionally equivalent models generated by SLX2MDL up to 543/543 (100%).

Our Python-based implementation is reasonably quick. It takes SLX2MDL an average of 6.29 seconds to transform a Simulink model from various domains, and the conversion time grows approximately in a linear fashion. A typical Simulink model contains several thousand lines of text in the MDL format. We believe this is acceptable because the application of SLX2MDL will likely be done once on an SLX model.

1) Limitations & Threats: The first limitation is model files storing content in a binary format. Some Simulink models contain some information stored as binary files in the SLX format. These files are mostly used to store workspace-data, and are stored as .mxarray files located at *simulink/bdmx-data/*.mxarray*. In the absence of the exact format on how information is stored in these files, we cannot read them, and the resulting MDL file produced by our tool lacks the corresponding information. We addressed why this is acceptable earlier. Some Simulink models contain PNG image files, which are located at *simulink/resources/*.png* in the SLX format. In a standard MDL format (produced by Simulink itself), such image information is written in the *MatResources* block as ASCII characters. However, in lieu of Simulink's proprietary algorithm that converts such binary image file data into a sequence of ASCII characters, our tool cannot create the corresponding *MatResources* block to represent such image files. Hence, the converted model lacks any image-file information present in the original SLX file. However, the converted model still loads in Simulink without the workspace-data and image information.

The second limitation stems from our definition of the SLX syntax. Given that the full syntax/format is not published by Mathworks, our definition is based on our analysis and experimentation of models, derived empirically. Thus, our current implementation cannot handle instances when the input SLX contains XML files that contain tags that we did not account for, nor ones that are added Simulink evolves. However, this is not a major obstacle nor a concern as it requires only a minor update in the definition. If SLX2MDL encounters any such previously unseen tags, it raises an exception which is logged in a file. This file contains the information necessary to allow the language definition to be updated with relative ease. Because we publish the source/project and evaluation of

SLX2MDL publicly, anyone has the ability to do this [18].

A threat to the validity of our evaluation is our allowance of variations in the model configuration parameters in considering a model functionally equivalent. While this was a reasonable determination in our interpretation due to the unknown nature of how these values are converted internally by Simulink, it could still present an issue for the deployment of SLX2MDL broadly. However, given the target applications of model analysis, clone detection, and other similar approaches that tend to focus more on models' structural aspects, which is captured by SLX2MDL, the omission of parametric changes is not an issue for our evaluation.

VII. Related Work

Our initial research aimed to leverage the successes of the source transformation community. We first considered the TXL Source Transformation Language[7] as a means of converting from SLX to MDL, which is one of, if not the leading, source transformation language [20]. TXL grammar files exist for both XML and MDL files, so there was no need to define a language grammar, only the transformation rules. However, since the SLX file format consists of many XML files, the solution to this problem necessitated an advanced contextualization of these files to create a single input XML file. This was the first in a series of roadblocks we faced early on, especially since our analysis demonstrated that SLX model source file contextualization was not always completed by Simulink in a consistent manner. The second significant issue arose with the realization that the existing MDL grammar file for TXL has not been updated in several years and thus did not include new language features that exist in the newer Simulink models. This issue is further exacerbated by the complexities of Stateflow models. These updates would have required significant effort to update the formal grammar, which is something that we would need to do in a standalone implementation regardless. The final issue we faced with a TXL based approach is that the transformation is not always as simple as a consistent application of replacement rules. There is a need to track object identifiers across multiple files, and sometimes these identifiers are based implicitly on file location rather than explicitly identified. This was something that can be fairly challenging in TXL, especially in contrast to our eventual solution. Due to these considerations, TXL became less feasible as we delved deeper into the research and work. Thus, we abandoned TXL in favor of a more imperative programming approach.

It is important to discuss model transformation research in the context of related work. The most related and popular example of model transformation is the Atlas Transformation Language (ATL) [21]. ATL uses a similar approach to TXL but is more focused on model elements rather than the textual definitions provided by grammars. We dismissed ATL for a number of reasons, including some of the same reasons we opted not to use TXL. While ATL performs model to model transformations, it is not explicitly intended for underlying model format transformations. ATL requires rigidity not possible given the proprietary nature of the Simulink model formats. By using our own flexible language definitions focused on Simulink models, which we defined empirically, we were able to tailor our transformation accordingly.

Model transformation in Simulink models has been the focus of previous research. In one example Denil et. al. [22] applied rule-based model transformations to alter Simulink models based on user-defined rules. Yang and Vyatkin[23] also implemented model transformations between MATLAB Simulink and Function Blocks. While both of these approaches implement model transformations involving Simulink, they differ from our work in that their goals either apply transformations within the Simulink model itself, not to the source or format, or transform to some format outside of Simulink. As such, the SLX2MDL approach provides a novel transformation by converting between Simulink model formats without changing model contents.

There is work that considers models as graphs, most of which focuses on UML models or traditional visual programming [24]. For example, the Graph Rewriting and Transformation Language (GREAT) uses UML models as a means of representing the graph grammars of the input and output [25]. VIATRA similarly provides a visual automated model transformation system for UML models [26]. Schurr et al. summarizes approaches of graph transformation applied to visual languages [27]. These approaches and the ones that follow are focused almost exclusively on modifying the graphs/models (semantics) specifically, changing/generating their editors and environments, and performing analysis. The most similar works to ours are approaches that perform analysis on model/visual language syntax to transform it to another form, for example, predicates, metamodels (graphs), or behavior trees [28], [29], [30]. Ours differs in that we are concerned explicitly with the syntax of the underlying textual representations of the graphs/models and transforming it to another textual format.

Ultimately, our choice to implement an independent transformation approach for SLX2MDL rather than an existing model transformation environment was based on the customizability afforded through our approach, and the ease in which it can be adapted to new language features as the SLX-model format evolves. Further, our performance results indicate this approach scales well with model size.

VIII. Conclusion

SLX2MDL is a standalone, open, and publicly available model transformation approach capable of applying consistent transformations of Simulink SLX-syntax models to produce equivalent MDL-syntax conforming models. This work is necessary in an open and publicly available fashion for a number of reasons including providing the ability to interact with newer models in older Simulink versions as well as their use in external tools that rely on the MDL format, such as text-based clone detectors like Simone.

Through our 4-phase rule-based transformation involving several iterations of extraction, transformation, and merging, we are able to realize a model transformation approach based on our empirically derived definitions of SLX and MDL syntax. Using an established set of Simulink examples, we evaluated SLX2MDL's effectiveness to produce not only valid, but functionally equivalent, output models conforming to the MDL syntax. Through our evaluation using the internal and proprietary conversion provided by Simulink as the ground truth, SLX2MDL was able to produce a valid model 100% of the time. Further, when determining the correctness of the resulting models, it is able to achieve 100% correct transformations, given our extended definition of functional equivalence.

SLX2MDL's performance was not a detriment to its contributions, nor does the increased complexity of input models have a significant impact on the performance. Through this validation we determined an average execution time of 6.29 seconds per input model and a worst-case complexity of O(n).

A. Future Work

A natural extension of this work is to attempt to overcome the limitations imposed by SLX2MDL's inability to process the contents of mxarray files, thus omitting some data from the transformations. While this omission is acceptable for the sake of structural transformation, ideally SLX2MDL ought to handle all forms of input. Further, the ability to also correctly map the configuration parameters from input to converted output models is desirable in the long term. Both of these potential extensions rely on the official definition of Simulink's proprietary algorithms and file structures. However, it may eventually be possible to reverse engineer or simulate these through further analysis. However, as we discussed, these two omissions do not have significant impacts on the results of SLX2MDL's transformations as the resulting transformed models are otherwise identical to those produced by Simulink.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1849632.

References

[1] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson, "Models are code too: Near-miss clone detection for simulink models," in *International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 295–304.

[2] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert, "Clone detection in automotive model-based development," in *ICSE*. IEEE, 2008, pp. 603–612.

[3] M. Stephan and J. R. Cordy, "A Survey of Model Comparison Approaches and Applications," in *International Conference on Model-Driven Engineering and Software Development (Modelsward)*. SCITEPRESS, 2013, pp. 265–277.

[4] A. Joshi and M. P. Heimdahl, "Model-based safety analysis of simulink models using scade design verifier," in *SAFECOMP*. Springer, 2005, pp. 122–135.

[5] Y. Papadopoulos and M. Maruhn, "Model-based synthesis of fault trees from matlab-simulink models," in *International Conference on Dependable Systems and Networks*. IEEE, 2001, pp. 77–82.

[6] A. Chapoutot and M. Martel, "Abstract simulation: a static analysis of simulink models," in *2009 International Conference on Embedded Software and Systems*. IEEE, 2009, pp. 83–92.

[7] J. R. Cordy, "The txl source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[8] T. J. Biggerstaff, "Design recovery for maintenance and reuse," *Computer*, vol. 22, no. 7, pp. 36–49, 1989.

[9] R. S. Arnold, "Software restructuring," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 607–617, 1989.

[10] J. R. Cordy and M. Shukla, *Practical metaprogramming*. Queen's University. Department of Computing and Information Science, 1992.

[11] D. B. Staple, "Information about simulink mdl and slx formats?" May 2014. [Online]. Available: https://stackoverflow.com/questions/23408186/information-about-simulink-mdl-and-slx-formats

[12] MathWorks, Inc., "Save the model," 2021. [Online]. Available: https://www.mathworks.com/help/simulink/ug/saving-a-model.html

[13] R. T. Narayanan, "Novice programmer to new-age application developer: What makes python their first choice?" in *ICCCNT*. IEEE, 2019, pp. 1–7.

[14] C. Chen, J. Sun, Y. Liu, J. S. Dong, and M. Zheng, "Formal modeling and validation of stateflow diagrams," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 6, pp. 653–671, 2012.

[15] J. Chen, T. R. Dean, and M. H. Alalfi, "Clone detection in matlab stateflow models," *SQJ*, vol. 24, no. 4, pp. 917–946, 2016.

[16] A. L. Dominguez, "mdl2smv: A tool for translating automotive feature models in matlab's stateflow to smv," https://cs.uwaterloo.ca/∼aljuarez/mdl2smv.html, 2012, accessed: 07-26-2020.

[17] S. A. Chowdhury, L. S. Varghese, S. Mohian, T. T. Johnson, and C. Csallner, "A curated corpus of simulink models for model-based empirical studies," in *International Workshop on Software Engineering for Smart Cyber-Physical Systems*. IEEE, 2018, pp. 45–48.

[18] B. Adhikari, E. J. Rapos, and M. Stephan, "mdstepha/slx2mdl: Initial version of slx2mdl," Jul. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5117126

[19] A. Boll, F. Brokhausen, T. Amorim, T. Kehrer, and A. Vogelsang, "Characteristics, potentials, and limitations of open source simulink projects for empirical research," *Software and Systems Modeling*, vol. tbd, no. tbd, p. 20pp, 2021, in press.

[20] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider, "Software engineering by source transformation-experience with txl," in *International Workshop on Source Code Analysis and Manipulation*. IEEE, 2001, pp. 168–178.

[21] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," *Science of computer programming*, vol. 72, no. 1-2, pp. 31–39, 2008.

[22] J. Denil, P. J. Mosterman, and H. Vangheluwe, "Rule-based model transformation for, and in simulink," in *Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative*, 2014, pp. 1–8.

[23] C. Yang and V. Vyatkin, "Model transformation between matlab simulink and function blocks," in *International Conference on Industrial Informatics*. IEEE, 2010, pp. 1130–1135.

[24] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE software*, vol. 20, no. 5, pp. 42–45, 2003.

[25] A. Agrawal, G. Karsai, and F. Shi, "A uml-based graph transformation approach for implementing domain-specific model transformations," *Software and Systems Modeling*, pp. 1–19, 2003.

[26] D. Varró, G. Varró, and A. Pataricza, "Designing the automatic transformation of visual languages," *Science of Computer Programming*, vol. 44, no. 2, pp. 205–227, 2002.

[27] A. Schúrr, "Application of graph transformation to visual languages," *Handbook of graph grammars and computing by graph transformation*, vol. 2, p. 105, 1999.

[28] J. W. Janneck and R. Esser, "A predicate-based approach to defining visual language syntax," in *Symposia on Human-Centric Computing Languages and Environments*, 2001, pp. 40–47.

[29] P. Bottoni, D. Frediani, and P. Quattrocchi, "A transformation-based metamodel approach to the definition of syntax and semantics of diagrammatic languages," in *Visual Languages for Interactive Computing: Definitions and Formalizations*. IGI Global, 2008, pp. 51–73.

[30] L. Grunske, K. Winter, and N. Yatapanage, "Defining the abstract syntax of visual languages with advanced graph grammars—a case study based on behavior trees," *Journal of Visual Languages & Computing*, vol. 19, no. 3, pp. 343–379, 2008.