Addressing the shortcomings of commercial-of-the-shelf model-to-model transformations with open-source tools; from SysML to AUTOSAR

Horacio Hoyos Rodríguez
hoyosroh@mcmaster.ca
McMaster University
Hamilton, Ontario, Canada

Faezeh Siavashi
siavashf@mcmaster.ca
McMaster University
Hamilton, Ontario, Canada

Monika Jaskolka
monika.jaskolka@@stellantis.com
Stellantis Canada
Windsor, Ontario, Canada

Vera Pantelic
pantelv@mcmaster.ca
McMaster University
Hamilton, Ontario, Canada

Mark Lawford
lawford@mcmaster.ca
McMaster University
Hamilton, Ontario, Canada

Richard Paige
paigeri@mcmaster.ca
McMaster University
Hamilton, Ontario, Canada

ABSTRACT

Model-Based Systems Engineering (MBSE) is a widely adopted approach to managing the complexity of modern cyber physical systems, including automotive systems. In the domain of automotive engineering, it is common for engineers to use a variety of languages, at various levels of abstraction, to provide diverse and concrete perspectives on a system. However, a significant incompatibility challenge arises due to weak or nonexistent integration among these languages. In some cases, these challenges can be addressed by using commercial off the shelf (COTS) model-to-model (M2M) transformation tools. However, in certain cases these tools have semantic and technical limitations that hinder the development process, produce sub-optimal results, and generate trace information in a proprietary format. In this paper, we present how the same transformation can be implemented using an open-source tool. First, we discuss the technical limitations and present how the open-source tool provides better development support. Then, we present the results of running both implementations for a set of test models and show that the open-source implementation provides more detailed output models and produces more fine-grained traceability data. By using the open-source implementation, we reduce the development effort, produce output that is better suited for purpose and generate trace information that can be easily consumed in other tools.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Model-driven software engineering**.

KEYWORDS

Model-Based Systems Engineering, Epsilon Transformation Language, AUTOSAR, SysML

ACM Reference Format:

Horacio Hoyos Rodríguez, Faezeh Siavashi, Monika Jaskolka, Vera Pantelic, Mark Lawford, and Richard Paige. 2024. Addressing the shortcomings of commercial-of-the-shelf model-to-model transformations with open-source tools; from SysML to AUTOSAR. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (MODELS 24).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MODELS 24, September 22–27, 2024, Linz, Austria

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Model-Based Systems Engineering (MBSE) has been widely used in industrial environments such as the aviation and automotive, across various development phases. MBSE utilizes modeling principles throughout system development activities and relies on the use of model-based tools for creating, transforming and validating the artifacts used at different phases of the industrial process, such as requirements elicitation and specification, design, and verification and validation (V&V) [14].

Large companies usually rely on multiple COTS tools within their MBSE toolchains. These tools can be incompatible at the technical level (e.g., proprietary model format) or at the formalism level (e.g., SysML vs. AUTOSAR) [4]. In previous work[12], we presented a solution for bridging the formalism level of SysML vs. AUTOSAR, while also bridging the gap at the tool level (IBM Rhapsody vs. PREEvision) through a SysML to AUTOSAR transformation. Our solution was based on a COTS M2M transformation tool called M2M_IE that is integrated with IBM Rhapsody. In that paper we reported on the limitations of the M2M_IE tool, pertaining to both the technical aspects of the implementation and the semantics of the transformation engine. The main consequences of those limitations were slow implementation times, sub-optimal output models, and coarse-grained trace information. It is precisely these impediments that drive the motivation for this paper.

In this paper, we present an open-source alternative to the M2M_IE SysML to AUTOSAR transformation, based on the Epsilon Transformation Language (ETL) [9]. The main contributions of the open-source alternative are:

- An abstraction of the SysML and AUTOSAR model access using the Epsilon Model Connectivity layer, in order to address the API limitations.
- An implementation of the SysML to AUTOSAR M2M transformation using ETL, which provides improved semantics.
- A custom extension of the Trace*a* DLS [3] to improve traceability information.

We also report on the integrated development environment (IDE) tool facilities provided by ETL and other tools used during development, to highlight how they reduce development time and bugs while facilitating maintenance and future improvements.

The paper proceeds as follows.

2 FROM SYSML TO AUTOSAR (CLASSIC)

Our implementation of the SysML to AUTOSAR M2M transformation will follow the same functional requirements [12] of the M2M_IE implementation. In order to compare both implementations and highlight their differences, we present some challenges of the SysML to AUTOSAR transformation from the perspective of the M2M transformation semantics.

2.1 Model to Model Transformations

A **transformation** is the automatic generation of a target model from a source model, according to a transformation definition [7]. A **transformation definition** is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A **transformation rule** is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.

2.1.1 Transformation Languages. Transformation languages support defining rules with different cardinality, mainly $1:1$, $1:n$, or $1:m..n$. In $1:1$ rules, one source language construct is transformed into one target language construct. In $1:n$ rules, one source language construct is transformed into multiple target language constructs; the number of target constructs is known when the rule is defined. In $1:m..n$ rules, one source language construct is transformed into multiple target language constructs; the exact number of target constructs can only be determined at runtime and usually depends on some properties of the source model. Transformation languages can be declarative or imperative. In imperative languages, the user defines the order of execution of the transformation rules. On the other hand, in declarative languages, the transformation engine decides the execution order of the transformation rules.

Typically, transformation definitions are written using a model transformation language and these languages are executed by a model transformation engine. The SysML to AUTOSAR transformation is exogenous. An exogenous transformation is defined as a process where source model(s) are converted to target model(s), and each model adheres to a distinct language. Exogenous transformations serve various purposes, including tasks like model synthesis and reverse engineering. The SysML to AUTOSAR transformation is a model synthesis that bridges the gap between two abstraction levels.

2.1.2 Traceability. Traceability plays a significant role in systems and software development, supporting project management, software evolution, and verification and validation. In M2M transformations, traceability support is typically not part of the language but is provided by the transformation engine during execution. In some cases, the traceability information is stored in a model, that conforms to a traceability metamodel [6]. Using a metamodel allows the trace to capture, apart from the links between source and target elements, information about the transformation artifacts (i.e., transformation specification and rules) and quality aspects that can be used to interpret the relevance and integrity of traces [3].

To support traceability, the transformation engine must offer mechanisms for maintaining an explicit link between the source and target models [10]. Additionally, the granularity of traceability is usually directly tied to the cardinality of rules. In other words, the trace model only includes references to model elements explicitly specified in the constructs of the rules. Some transformation engines automatically persist the trace model to a predefined format while others allow the users to post-process the trace model and choose the persistence format.

2.2 SysML to AUTOSAR Transformation Semantics

The AUTOSAR language is at a lower level of abstraction than SysML, which results in AUTOSAR models using more elements in order to provide greater fine-grained system details. From a transformation perspective, this means that some of the rules must be $1:n$, or $1:m..n$. The need for $1:m..n$ rules is the result of the semantics of AUTOSAR communication modes [12]. In particular, in SysML a `Port` that provides an `Interface` does not care how many ports require the same interface. However, in AUTOSAR, a `RPortPrototype` acting as a server port (requires an interface) needs a unique `ClientComSpec` for each `PPortPrototype` acting as a client (requiring an interface). The number of ports can only be determined at runtime.

In order to understand the semantics of the SysML to AUTOSAR transformation and some of the challenges it presents, we examine three rules of the existing M2M_IE. We picked a $1:1$, $1:n$, and a $1:m..n$ rule. We chose a graphical representation (See Fig. 2) to describe the rules, in order to avoid discussing transformation language details. The diagram notation resembles a UML class diagram, where language constructs are depicted as boxes, with their types specified in a top compartment. While both SysML and AUTOSAR constructs may contain multiple attributes, for simplicity we only show the `name` attribute. Arrows depict relationships between constructs. Three types of relations are used: association, containment, and mapping. Associations and containment are only valid between constructs of the same language. Mappings are applicable exclusively between constructs of different languages.

2.2.1 Project to AUTOSAR ($1:1$ Mapping). Figure 1 presents the mapping from SysML `Project` to AUTOSAR `AUTOSAR` (the root construct of an AUTOSAR model is the AUTOSAR construct) pairing. This is a $1:1$ pairing where each SysML `Project` maps to one AUTOSAR `AUTOSAR`.

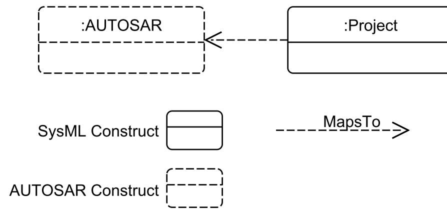

Fig. 1: Mapping SysML `Project` to AUTOSAR `AUTOSAR`.

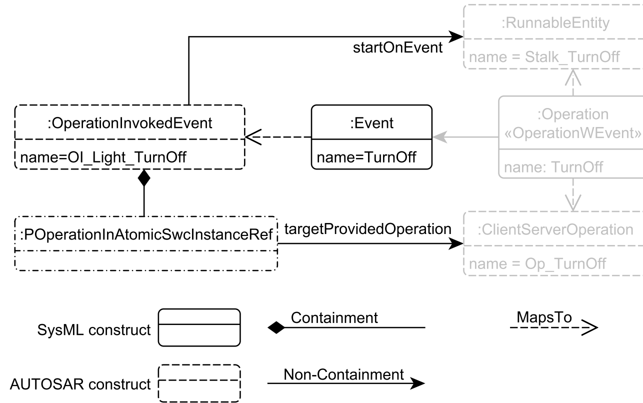

Fig. 2: Mapping SysML `Event` to AUTOSAR `OperationInvokedEvent` (in the context of an `OperationWEvent`).

2.2.2 Event to OperationInvokedEvent ($1:n$ Mapping). The `OperationWEvent` stereotype was defined to capture the event that triggers the operation invocation in SysML [12]. Conceptually, a SysML `Operation` has a reference to an `Event`. Fig. 2 presents the mapping from SysML `Event` to AUTOSAR, showing that two target constructs are required: `OperationInvokedEvent` and `POperationInAtomicSwcInstanceRef`. In Fig. 2 we also present elements that are required (in gray) to correctly set all the properties and references of the new target constructs. This means that this mapping would need access to the trace information in order to access the AUTOSAR `RunnableEntity` and `ClientServerOperation` elements that where created from the SysML `Operation` that references the `Event`.

2.2.3 OperationWEvent to Client–Server Elements ($1:m..n$ Mapping). The complexity of this mapping comes from the AUTOSAR semantics for ClientServer communication. In particular, a different set of elements are needed at the client and the server end. Further, the exact number of the elements needed depends on the number of ports that use (require/provide) the interface that contains the operation.

Fig. 3 shows the overall mapping for `OperationWEvent` including multiple mappings to complete the required AUTOSAR structures for client and server. First, each `OperationWEvent` must be transformed into an AUTOSAR `ClientServerOperation`; this is a $1:1$ mapping. On the server side, we have the SysML `Port`(s) that provides the interface. The `OperationWEvent` must be transformed into an AUTOSAR `ServerComSpec` for each SysML `Port` that provides the interface; this is a $1:m..n$ mapping. On the client side, we have the SysML `Port`(s) that require the interface. The `OperationWEvent` must be transformed into an AUTOSAR `ClientComSpec` for each SysML `Port` that provides the interface to define the communication attributes required by the port; this is a $1:m..n$ mapping. Additionally, the `OperationWEvent` must be transformed into an AUTOSAR `RunnableEntity`, `SynchronousServerCallPoint` and `ROperationInAtomicSwcIntanceRef` for each SysML `Port` that provides the interface in order for the client to be able to call the operation on the server; this is a $1:m..n$ mapping. This mapping also needs access to the trace information in order to access the AUTOSAR `PortPrototypes` and `ClientServerInterface` elements that where created from the SysML `Ports` and `Interface`.

3 THE M2M_IE TRANSFORMATION LANGUAGE

IBM® Engineering Systems Design Rhapsody® (commonly known as Rational Rhapsody) is an environment for modeling and design tasks. It supports various modeling languages such as UML, SysML, UAF, and provides AUTOSAR import and export capabilities. The AUTOSAR import/export is provided via the M2M_IE plugin. This section presents the characteristics of the language and discusses the limitations we encountered while developing the transformation.

3.1 The language semantics

The M2M_IE plugin provides a rule-based m2m, declarative, transformation language, in which rules are specified within a tabular format known as a *RuleSet*. Each rule is described in a separate row and columns are employed to capture the rule's properties such as source/target constructs and processing functions. Source constructs can either be specified by their Rhapsody Metaclass or Stereotype. Target constructs can be specified by their EClass (from a proprietary implementation of the AUTOSAR metamodel). Each rule can define a condition that the source elements must satisfy to be transformed. Processing functions are used to describe how the target elements are organized hierarchy and to set their attributes and relations. Conditions and processing functions must be written in JavaScript. Finally, the priority level can be used to override the declarative execution and define a specific rule execution order. All rules with a defined priotity level will be executed imperatively in the specified order.

Most notably, the tabular nature of rules means that M2M_IE only supports $1:1$ rules. Additionally, a semantic limitation imposed by the transformation engine is that a source model element can only be transformed once. That is, although multiple rules can have the same source Metaclass, a source model element will only be transformed by the first rule that applies (condition is matched). The transformation trace is accessible to the processing functions and can be persisted after execution.

The M2M_IE engine is opinionated about two aspects: element names and containment. For names, it will automatically populate the AUTOSAR element's *shortName* (if present) value from the

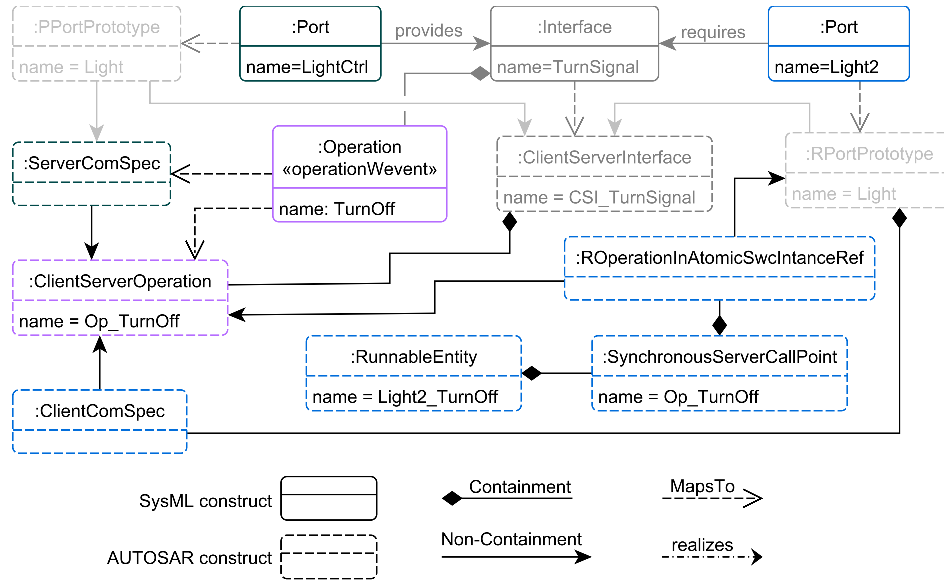

Fig. 3: Mapping SysML `OperationWEvent` to AUTOSAR constructs required for Client and Server communications.

SysML element's name (if present). For containment, target elements will be automatically be added to the containment hierarchy of the target model. Consider that source element S_c is contained in source element S_p, for example a `Block` within a `Package`. Then, the target element T_c (created from S_c) will be added to the containment hierarchy of T_p (created from S_p). The exact reference used to add the target child is automatically selected by the M2M_IE engine.

3.2 The language limitations

From a developer's perspective the biggest limitation we faced was the lack of support tools for editing the condition and processing functions. Currently, these functions have to be edited in a simple text editor that lacks features such as syntax-highlighting, static analysis and input content-assist. Since the engine lacks support for debugging, syntax and API access errors took longer to fix.

Another more subtle issue is that the API to access SysML elements uses one-base indexing, while the AUTOSAR uses zero-based indexing. We found that this resulted in hard to catch errors. Similarly, there are two separate functions to retrieve elements from the trace: *mapMDW2RhpElements* and *mapRhp2MDWElements*. The similarity in names and the lack of static analysis results in hard to catch errors.

With respect to source constructs, altough M2M_IE allows the use of `Project` as a source metaclass, not all packages in the project are transformed. As a result, before the transformation execution the user has to select the package that should be transformed. An effect of this restriction is that elements that belong to imported packages, such as profiles, are not considered as source elements. The reason being that imported pacakges are located at the root of the project as opposed to under the package selected by the user. One notable effect of this limitation is that DataTypes from the SysML Profile are not transformed.

Listing 1: IsStaticAttribute Function (JS)

```
1 function IsStaticAttribute(attribute) {
2   return attribute.getIsStatic() == 1;
3 }
```

A consequence of the $1:1$ mapping restrictions is that the $1:n$ and $1:m..n$ transformations required by the SysML to AUTOSAR transformation can't be specified completely in the RuleSet. The workaround is to create AUTOSAR elements in the processing functions. The downside of this approach is that the trace is not aware of the additional elements being created. As a result, the trace produced by the M2M_IE transformation is coarse-grained and does not correctly capture all the AUTOSAR elements created for each SysML element.

Finally, we would also like to mention that during development we faced an implementation bug in the M2M_IE plugin. The bug related to the setting of the direction of operation arguments. In the M2M_IE AUTOSAR implementation, the direction is defined using an enumeration. Although we tried several approaches, we were not able to use the different enumeration values with the processing functions. As a result, all arguments used the default value (`IN`).

3.3 A rule example

An example rule in the M2M_IE transformation, that describies how SysML (Static) `Attributes` are transfromed to AUTOSAR `VariableDataPrototypes` is defined as follows. **Metaclass:** `Attribute`, **Target EClass:** `VariableDataPrototype`, **Condition Function:** *IsStaticAttribute* and **Post-process Function:** *SetParentAddInitValue*. The *IsStaticAttribute* function (see Listing 1) accepts the `Attribute` source element as argument and the implementation checks if the attribute is static.

Listing 2: SetParentAddInitValue Function (JS)

```
function SetParentAddInitValue(mdwVarDataProt) {
  var rhpAttr = mapMDW2RhpElements.get(mdwVarDataProt);
  var rhpBlk = rhpAttr.getOwner();
  var mdwAppSwComp = mapRhp2MDWElements.get(rhpBlk);
  var mdwSWIntBeh = mdwAppSwComp.getInternalBehavior();
  mdwSWIntBeh.get(0).getExplicitInterRunnableVariable().add(mdwVarDataProt);
  var mdwNumValSpec = model.create("NumericalValueSpecification");
  mdwNumValSpec.value = 10;
  mdwVarDataProt.initValue = mdwNumValSpec;
}
```

The *SetParentAddInitValue* (see Listing 2) function accepts the `VariableDataPrototype` target element as an argument. In line 2, we use the *mapMDW2RhpElements* function to get the `Attribute` source element. In line 3 we capture the `Attribute`'s owner, a SysML `Block` and in line 4 we use the *mapRhp2MDWElements* function to get the `ApplicationSwComponentType` target element (created by the rule that transforms Blocks to ApplicationSwComponentTypes). In lines 5-6, we find the `ApplicationSwComponentType` 's *internalBehavior* (`SwcInternalBehavior`) and add the `VariableDataPrototype` to its list of *explicitInterRunnableVariables*. In lines 7-9, we create and assign an initial value (*initValue*) to the target `VariableDataPrototype`. As mentioned previously, the `NumericalValueSpecification` element created in line 7 will not be present in the transformation trace.

4 THE ETL IMPLEMENTATION

When setting out to provide the alternative open-source implemenation of the SysML to AUTOSAR transformation, we had three objectives in mind:

- Use a transformation language that supports $1:1$, $1:n$, and $1:m..n$ rules.
- Use a transformation language that provides better development tools.
- Use a transformation language that allowed us to use a unified API to access the SysML and AUTOSAR models.

The Epsilon Transformation Language (ETL) satisfies the three objectives. The Epsilon Framework includes an ETL editor that provides syntax and error highlighting, as well as code templates and graphical tools for configuring, running, debugging and profiling ETL programs[1]. The Epsilon Model Connectivity (EMC) layer provides abstraction facilities over concrete modelling technologies which enables Epsilon programs to read/write a wide range of heterogeneous models in a uniform manner. ETL supports the specification of $1:1$, $1:n$, and $1:m..n$ rules. Finally, although the ETL transformation trace is not persisted automatically by the execution engine, it can be easilly accessed for post-processing and persistence.

Next, we give an overview of the work on the EMC and trace, but we skip the details are they are not the focus of this paper. Following, we go into the details of the ETL transformation script and use a set of three different rules to make a 1-to-1 comparison with the M2M_IE implementation. Finally, we highlight some of the differences in the generated models.

[1]The Epsilon Framework, https://eclipse.dev/epsilon/, last accessed 3-Mar-2024

Listing 3: The ETL concrete syntax

```
rule <name>
  transform <sourceParameterName>:<sourceParameterType>
  to <targetParameterName>:<targetParameterType>
    (,<targetParameterName>:<targetParameterType>)*
    (extends <ruleName> (, <ruleName>*)? {

    (guard (:expression)|({statementBlock}))?

    statement+
}
```

4.1 Writing transformations with the Epsilon Transformation Language

"Epsilon is a family of scripting languages and tools for automating common model-based software engineering tasks such as code generation, model-to-model transformation, model validation and model visualization"[2]. The Epsilon Transformation language (ETL) supports a wide range of modeling languages/technologies. ETL is rule-based, with rules defined in an ETL script. The ETL natively supports $1:1$ and $1:n$ rules; $1:n..m$ can be defined by leveraging the feature that rule are not limited to create types of the target model. The details of $1:n..m$ are discussed in Sec. 4.7.

Listing 3 presents the ETL concrete syntax. The *rule* keyword is followed by the rule name. The source and target constructs are defined after the *transform* and `to` keywords. Notice that there can be many `to` constructs in order to support $1:n$ rules. A transformation rule can also define a number of other transformation rules it *extends*. The *guard* is used to define a condition on the source construcs that must be satisfied for the rule to execute. Finally, multiple statements can be used to set the target constructs attributes and references. Statements in ETL are written in the Epsilon Object Language (EOL) [8].

4.2 Rhapsody and AUTOSAR EMC

In the M2M_IE implementation, the result of the transformation is an AUTOSAR model (in ARXML, the AUTOSAR XML serialization format), that can then be imported into PREEvision. On one hand, it provides flexibility on what AUTOSAR tool to use. On the other hand, given that the AUTOSAR tool is known, it adds another step to the workflow. Ideally, the SysML-to-AUTOSAR transformation, similarly as to how it reads the SysML model directly from Rhapsody, should be able to write the AUTOSAR model directly into PREEviison. However, at the moment, the PREEvision tool does not provide an API that can be used to read/write AUTOSAR models directly. As a result, we follow the same approach as the M2M_IE implementation and create an ARXML file as an output.

For the SysML model, Rhapsody does provide an API[3] that allows other tools to interact with SysML models. Using this API, we developed and EMC driver that can read/write Rhapsody's SysML models. The two main features of our implementation are 0-based collection indexing and Stereotype promotion to type. The former, standardizes collection access via 0-based indexing. The latter is

[2]https://eclipse.dev/epsilon/, last accessed Jan. 25 2024

[3]Rhapsody API, https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/design-rhapsody/9.0.2?topic=api-java-version-rhapsody, last accessed 3-Mar-2024

used to provide seamless support with M2M_IE, where the source construct can be either a SysML type or a Stereotype.

For the AUTOSAR model, we implemented the AUTOSAR metamodel using the Eclipse Modelling Framework (EMF) [13]. The two main reasons for this where that Epsilon provides an EMC driver for EMF, the ECore language is well suited to express the AUTOSAR metamodel and EMF allowed us to provide a custom serialization that respects the ARXML serialization rules [1]. Additionally, we can also generate the AUTOSAR Schema [2] from the metamodel, allowing easy distribution of models (ARMXL) with their required schema.

4.3 The Trace Model

In an m2m transformation, the trace should include as a minimum the links between source and target elements, and the transformation rules that created those links. However, for the trace information to be of value for supporting the engineering process in activities such as change impact or certification, it should include additional metadata such as temporal information, trace origin and confidence[3]. For this reason, we chose to extend and adapt the Trace*a* metamodel proposed by Batot et al. [2021].

Our adaptations where done to increase the details captured about the transformation script and the model elements. The reasoning is that since transformation scripts and models can be stored across different systems in the organization, we should be able to locate them when inspecitng the trace. For this purpose, both the Artefact and ModelFragment (element) types, were augmented with an *URI* attribute to capture a unique identifier of the TracingElement. For rules, it is the path to the ETL script concatenated with the rule name. For ModelFragments, since both Rhapsody and AUTOSAR support the notion of a unique ID, the URI can be the model URI plus the element id.

After the ETL transformation has executed, we translate the ETL trace information into a model that conforms to the extended Trace*a* metamodel. A part from the model artefacts and the trace links, we augment the trace model with the code artefacts, agent information (the ETL engine) and confidence. Thus, the trace model is not only more fine grained (due to the nature of ETL), but also contains more metadata than the M2M_IE trace.

4.4 ETL development benefits

Before discussing the implementation details of ETL, we discuss some of the ETL features that improve the development and increase the maintainability of the transformation script.

4.4.1 Property navigation and Collection access. As mentioned previously, the differences in array index based access was a pain-point for development. By wrapping the SysML and AUTOSAR APIs with the EMC, we standardize the array access to 0-index based. But the ultimate benefit of using ETL is that most collection access is done via the EOL first-order operators in order to apply filter, map and collect functions to the collections.

Another important aspect of using ETL is that property access is wrapped by the EMC. Thus, property navigation in ETL is done via attribute name, and the EMC is responsible for calling the required method, e.g. getter to read a property. For example, Listing 4 presents a snippet of how the value of the short name of an AUTOSAR element can be retrieved in M2M_IE. For this, two getters are called: `getShortName()` and `getValue()`. Listing 5 shows the same statement written in ETL. This makes the ETL code less verbose and less suceptible to bugs where the parenthesis where ommited.

Listing 4: M2M_IE property access

```
1 mdwEl.getShortName().getValue()
```

Listing 5: ETL property access

```
1 mdwEl.shortName.value
```

4.4.2 Model navigation. When setting attributes and relations of elements created in a transformation rule, it is common to need to navigate the source and target models. To facilitate the navigation between the source and target models, both M2M_IE and ETL provide access to the trace model during execution. As mentioned previously, in M2M_IE, access to the trace model is enabled via two functions: *mapMDW2RhpElements* and *mapRhp2MDWElements*. The former can be used to get the target element from a source element, and the latter to get a source element from a target element. We found that the similarity in the names makes it easy to introduce bugs related to using the incorrect function when retrieving elements. The *mapMDW2RhpElements* is required because the post-processing function, as presented in Listing 2), has only one argument which is a target element. Thus, the only way to find source elements is via the *mapMDW2RhpElements* function.

Conversely, in ETL, a transformation rule has access to both the source and target elements. As a result, only one function, target from source, is required. In ETL, this is provided via the *equivalents* function. Having only one function makes it more difficult for developers to inject bugs. Further, the *equivalents* function accepts an optional list of rule names, to invoke and return only the equivalents created by specific rules. This reduces the amount of filtering required when a source element can be transformed by multiple rules.

4.5 1 : 1 rules

Section 2.2.1 presented the mapping for the 1 : 1 rule of SysML `Project` to AUTOSAR `AUTOSAR`. Since only the project will be transformed, no condition is required. Moreover, since the target AUTOSAR element is the root, there is no need for a context function. Similarly, as there are no attributes other than the “name” attribute, no post-processing function is required. Thus, all that is required for this rule is the row definition in the RuleSet table: **Metaclass:** `Project`, **Target EClass:** `AUTOSAR`. The *IsStaticAttribute* function (see Listing 1) accepts the `Attribute` source element as argument and the implementation checks if the attribute is static.

The ETL implementation of the `Project` to `AUTOSAR` mapping is presented in Listing 6. However, since ETL is not opinionated about containment, we need build the target containment hierarchy. Thus, in line 4 we use the *equivalents* operation to retrieve all AUTOSAR `Packages` created by other rules and add them to the list of *arPacakge* of the AUTOSAR element. Although additional code is required in ETL, we have complete control on what references to use for containment.

Listing 6: Project to Autosar ETL rule

```
rule ProjectToAutosar
  transform sPrj:SysML!Project
  to aAutosar:AUTOSAR!AUTOSAR {
  aAutosar.arPackage.addAll(sPrj.packages.equivalent());
}
```

4.6 1 : *n* rules

Section 2.2.2 presented the mapping for the 1 : *n* rule of SysML Event to AUTOSAR OperationInvokedEvent. Since we are only interested in Events that are defined as events of OperationWEvent operations, we need a condition function. The default naming and containment hierarchies are enough for this rule. However, we need to create the additional POperationInAtomicSwcInstanceRef, and set its *targetProvidedOperation* and *startOnEvent* relations. Thus, the rule definition in the RuleSet table is: **Metaclass:** Event, **Target EClass:** OperationInvokedEvent, **Condition Function:** *IsEventforOperationWithEvent* and **Post-process Function:** *SetOperationInstanceRef*.

The Condition function (in JavaScript) is presented in Listing 7. The function's parameter is the SysML Event. On line 2, we use the *findOpWEvntForEvent* helper function to find the Operation, with an OperationWEvent stereotype that uses the Event as its *event*. That function iterates over all blocks, over all ports, over all required/provided interfaces and over all operations in the interfaces. The reason for looping over ports as opposed to just interfaces, is that the function is reused in other rules where the port information is also important. If an Operation that matches the condition is found, then the *operation* attribute of the returned JavaScript object is not null. The result of that test is the return value of the condition function, line 3.

Listing 7: Event Condition

```
function ceIsEventforOperationWithEvent(event) {
 var objOperationAndPort = findOpWEvntForEvent(event, false);
 return isNotNull(objOperationAndPort.operation);
}
```

The post-processing function is presented in Listing 8. The function's parameter is the AUTOSAR OperationInvokedEvent. The three nested loops, lines 7, 11 and 16 are used to find the AUTOSAR ApplicationSwComponentType where the OperationInvokedEvent parameter should be added (line 29). The *createPOpRef* (line 19) is responsible for creating the POperationInAtomicSwcInstanceRef. If that method returns false, it means the inner loop *mdwElement* (line 17) is not the correct ApplicationSwComponentType. If it returns true, it also means that the POperationInAtomicSwcInstanceRef was created. In line 30, we use a helper function to find the operation that implements the OperationWEvent (operations from interfaces are implemented in blocks). From the operation implementation we can retrieve the target AUTOSAR RunnableEntity (line 31), which we assing to the OperationInvokedEvent.

Listing 8: Event Post-processing

```
function ppeSetOperationInstanceRef(mdwOpInvkEvent) {
  var rhpEvent = mapMDW2RhpElements.get(mdwOpInvkEvent);
  var rhpPkg = rhpEvent.getOwner();
  var mdwArPkg = mapRhp2MDWElements.get(rhpPkg);
  var mdwPkges = mdwArPkg.getArPackage();
  for (var j=0 ; j< mdwPkges.size() ; j++) {
    var mdwArSwTypes = mdwPkges.get(j);
    if (mdwArSwTypes.getShortName().getValue() == "SoftwareTypes") {
      var mdwArSwTypesPkgs = mdwArSwTypes.getArPackage();
      for (var k=0 ; k< mdwArSwTypesPkgs.size(); k++) {
        var mdwArCompTypes = mdwArSwTypesPkgs.get(k);
        if (mdwArCompTypes.getShortName().getValue() ==
              "ComponentTypes"){
          var mdwElements = mdwArCompTypes.getElement();
          var mdwAppSwComp;
          for (var i=0 ; i<mdwElements.size(); i++) {
            var mdwElement = mdwElements.get(i);
            if(mdwElement.eClass().getName()==
                  "ApplicationSwComponentType") {
              if(createPOpRef(mdwElement, rhpEvent, mdwOpInvkEvent)){
                mdwAppSwComp = mdwElement;
                break;
              }
            }
          }
          if (isNull(mdwAppSwComp)) {
            return;
          }
          var mdwSwInternalBeh =
                mdwAppSwComp.getInternalBehavior().get(0);
          mdwSwInternalBeh.getEvent().add(mdwOpInvkEvent);
          var rhpOp = findOpWEvntImplForEvent(rhpEvent, false);
          var mdwRunnableEntity = mapRhp2MDWElements.get(rhpOp);
          mdwOpInvkEvent.setStartOnEvent(mdwRunnableEntity);
          return;
        }
      }
    }
  }
}
```

The ETL implementation of the SysML Event to AUTOSAR OperationInvokedEvent is presented in Listing 9. The first thing to notice is that both the OperationInvokedEvent and POperationInAtomicSwcInstanceRef are listed in the *to* constructs (lines 3–4). The *guard* uses the same logic as the M2M_IE implementation, which is possible because we translated the *findOpWEvntForEvent* function to EOL. In line 10 we use the *findOpWEvntForEvent* function once more to get the relevant SysML Operation and Port. Lines 11–12 are uses to set the POperationInAtomicSwcInstanceRef references to the equivalent target elements. Note that instead of using the *equiavlents* operation, we use the ::= operator (EOL special assignment operator) that is syntactic sugar for calling the *equivalents* operation. Lines 13–15 are used to set the OperationInvokedEvent attributes and references. Note that in ETL we must explicitly set the AUTOSAR element's *name* attribute.

Of importance is that in the ETL implementation there is no need for the nested loops. The reason for this is that, since in ETL we have control over the creation of the containment structure, it is more convenient to add the OperationInvokedEvent to the ApplicationSwComponentType in the rule that creates the later. Listing 10 presents a snippet demonstrating how this is done. In line 3, we are adding all events to the SoftwareComponent's internal behavior (*aIntBhvr*). For the events, we go over all the events of the Package (line 4) and filter the ones used in the block (line 5). By applying the *equivalents* operation (line 6) we get all the AUTOSAR elements created by the transformation. Since the *EventToOperationInvokedEvent* is 1 : 2, we need to filter the results to select the OperationInvokedEvents (line 7).

Listing 9: Event to OperationInvokedEvent ETL rule

```
rule EventToOperationInvokedEvent
  transform se: SysML!Event
  to aOpInvEvnt: AUTOSAR!OperationInvokedEvent,
     aPOpRef : AUTOSAR!POperationInAtomicSwcInstanceRef {

  guard {
    var opAndPort = findOpWEvntForEvent(se, false);
    return opAndPort.op.isDefined();
  }
  var opAndPort = findOpWEvntForEvent(se, false);
  aPOpRef.targetProvidedOperation ::= opAndPort.op;
  aPOpRef.contextPPort ::= opAndPort.port;
  aOpInvEvnt.`operation` = aPOpRef;
  aOpInvEvnt.startOnEvent ::= opAndPort.port.owner.operations
      .selectOne(op | op.name == opAndPort.op.name);
  aOpInvEvnt.shortName = se.name;
}
```

Listing 10: Adding OperationInvokedEvents to the internal behavior

```
...
  // Events
  aIntBhvr.event.addAll(sSwCmp.owner.events
      .select(e | usesEvent(sSwCmp, e))
      .equivalent()
      .select(eq | eq.isTypeOf(AUTOSAR!OperationInvokedEvent)));
...
```

4.7 $1:m...n$ rules

Section 2.2.3 presented the mapping for the $1:m..n$ rule of SysML `OperationWEvent` to Client–Server Elements. Given that a separate structure is needed for the server and the client sides, ideally this mapping could be implemented in two or three separate rules. However, since M2M_IE has the restriction that an element can be transformed at most once, it is impossible to do so. For this reason, the rule in the M2M_IE RuleSet only captures the common elements, mainly the `OperationWEvent` to `ClientServerOperation`. For the other elements, the post-processing function was used to identify the intended use (i.e. client or server) and create the required elements accordingly. This rule has a condition function, which checks that the operation belongs to an interface, as presented in Listing 11. The reason for this check is that operations can also belong to Blocks.

Listing 11: Operation Condition

```
function ceOwnerIsInterface(operation){
  return operation.getOwner().getUserDefinedMetaClass()
      .equals("Interface");
}
```

The creation of the extra elements is done in the post-processing functions of another rule. The reason for this is that we can determine if the port is behaving as a client/server depending on whether it requires/provides, respectively, the interface that owns the operation. This separation was a design decision. At the time, it is simpler to navigate from the port to the provided/required interface, rather than using nested loops to find all ports that provide/implement the operation that owns the interface. Since all the elements created by the post-processing function will not be traceable, it made no difference in which particular post-processing rule they were created. Thus, the client and server elements are created in the post-processing functions of the `Port-to-PPortPrototype` and `Port-to-RPortPrototype` rules.

Listing 12: RPortPrototype post-processing

```
function ppeAddPPortStructure(mdwRPortPrt) {
  var rhpPort = mapMDW2RhpElements.get(mdwRPortPrt);
  var rhpReqInts = rhpPort.getRequiredInterfaces().toList();
  var rhpProInts = rhpPort.getProvidedInterfaces().toList();
  if (!rhpProInts.isEmpty() && rhpReqInts.isEmpty()) {
    if (ceIsClientServerInterface(rhpProInts[0])) {
      var rhpPrvIntr = rhpProInts[0];
      var mdwPrvIntr = mapRhp2MDWElements.get(rhpPrvIntr);
      if (isNull(mdwPrvIntr)) {
        return;
      }
      //Set the Port Prototype Interface
      mdwRPortPrt.setProvidedInterface(mdwPrvIntr);
      var rhpIntItems = rhpPrvIntr.getInterfaceItems();
      for (var i =1; i<= rhpIntItems.getCount(); i++) {
        var rhpOp = rhpIntItems.getItem(i);
        var mdwCSOperation = mapRhp2MDWElements.get(rhpOp);
        var mdwServerComSpec = model.create("ServerComSpec");
        mdwRPortPrt.getProvidedComSpec().add(mdwServerComSpec);
        mdwServerComSpec.setOperation(mdwCSOperation);
      }
    }
  }
  ...
```

For space considerations we will only present the creation of the server side elements. Listing 12 presents a snippet of the post-processing function for `RPortPrototypes`. The two conditional blocks in lines 5 and 6 determine if the port provides an interface identified as Client-Server. If so, in lines 7-8 we use the trace to get the `ClientServerInterface` created form the provided `Interface`.

In lines 9 to 14, we search for the AUTOSAR `ClientServer-Interface` to use with the port, and if present, we assign it to the `RPortPrototype` *providedInterface* reference. In lines 15-20, for each operation in the interface, we create a new `ServerComSpec`, add it to the `RPortPrototype`'s *providedComSpec* list and sets its *operation* to the `ClientServerOperation` created from the operation. Adding the implementation of the both client-server and sender-receiver makes the post-processing functions for `PPortPrototype` and `RPortPrototype` highly intricate.

The ETL implementation of the SysML `OperationWEvent` to Client–Server Elements follows a different approach that on M2M_IE. First, since elements in ETL can be transformed by multiple rules, we are able to separate the mapping into three separate rules (as explained next). Second, we use the ability of ETL to use output types that are not part of the target constructs in order to handle the $1:m..n$ requirement.

The first rule will handle the `OperationWEvent` to `ClientSer-verOperation`, and its presented in Listing 13. The guard uses the same logic as the M2M_IE implementation, checking that the operation is owned by an interface. Additionally, we need to set the operation name (line 7) and add all arguments transformed by other rules (line 8).

Listing 13: OperationWEvent To ClientServerOperation

```
rule OperationWEventToClientServerOperation
  transform sOp:SysML!Operation
  to aClntSrvOp:AUTOSAR!ClientServerOperation {

  guard: sOp.owner.isTypeOf(SysML!Interface) and
        isOperationWEvent(sOp)

  aClntSrvOp.shortName = sOp.name;
  aClntSrvOp.argument.addAll(sOp.arguments.equivalent());
}
```

Listing 14: OperationWEvent To ServerRunnableEntities

```
rule OperationWEventToServerRunnableEntities
  transform sOp:SysML!Operation
  to runbls: Sequence {

    guard : sOp.owner.isTypeOf(SysML!Interface)
          and isOperationWEvent(sOp)
          and isClientServerInterface(sOp.owner)
          and sOp.owner.owner.classes
            .select(c | c.isTypeOf(SysML!SoftwareComponent))
            .ports.flatten().exists(p | p.providedInterfaces
              .exists(i | i == sOp.owner))

    var aClntSrvOp = sOp.equivalents()
        .selectOne(eq | eq.isTypeOf(AUTOSAR!ClientServerOperation));
    for (p in sOp.owner.owner.classes
        .select(c | c.isTypeOf(SysML!SoftwareComponent))
        .ports.flatten()
        .select(p | p.providedInterfaces
          .exists(i | i == sOp.owner
            and isClientServerInterface(i)))) {
      var aSrvComSpec = new AUTOSAR!ServerComSpec;
      aSrvComSpec.`operation` = aClntSrvOp;
      p.equivalent().providedComSpec.add(aSrvComSpec);
      runbls.add(aSrvComSpec);
    }
  }
```

Listing 14 presents the details of the `OperationWEvent To ServerRunnableEntities` that creates the elements required by the server side. In order to provide the $1 : m..n$ mapping, note that the *to* type of the rule is a `Sequence`. Effectively, any elements created in the rule statements can be added to this sequence. In this case, the guard adds a new condition that checks if the interface is provided by any port. The for loop in lines 15-24 iterates over all ports that provide the interface that owns the operation. For each port, a `ServerComSpec` is created and added to the list of *providedComSpec* of the `RPortPrototype` created form the port.

Finally, the `OperationWEvent To ClientRunnableEntities` rule, presented in Listing 15, creates the elements for the client side. The guard is simliar to the server rule, but in this case we check for required interfaces. There is a for loop (line 15) that iterates over all ports that require the interface that owns the operation. In this case, there are four new elements created for each port. In order to group all the created elements we use a `Tuple` (lines 25–30). This will facilitate finding specific elements when using the *equivalents* function from other rules. For example, Listing 16 presents a snippet of the rule that creates `ApplicationSwComponentType` and their `SwcInternalBehavior`. In this case, we need to add all

Listing 15: OperationWEvent To ClientRunnableEntities

```
rule OperationWEventToClientRunnableEntities
  transform sOp:SysML!Operation
  to runbls: Sequence {

  guard : sOp.owner.isTypeOf(SysML!Interface)
      and isOperationWEvent(sOp)
      and isClientServerInterface(sOp.owner)
      and sOp.owner.owner.classes
        .select(c | c.isTypeOf(SysML!SoftwareComponent))
        .ports.flatten()
        .exists(p | p.requiredInterfaces.exists(i | i == sOp.owner))

  var aClntSrvOp = sOp.equivalents()
      .selectOne(eq | eq.isTypeOf(AUTOSAR!ClientServerOperation));
  for (...) {
    var aROpAtmcSwcInstRef = new
          ATOSAR!ROperationInAtomicSwcInstanceRef;
    ...
    var aSynchSrvrCallPnt = new AUTOSAR!SynchronousServerCallPoint;
    ...
    var aRnblEnt = new AUTOSAR!RunnableEntity;
    ...
    var aClntComSpec = new AUTOSAR!ClientComSpec;
    aClntComSpec.`operation` = aClntSrvOp;
    p.equivalent().requiredComSpec.add(aClntComSpec);
    var result = new Tuple(
      sPort = p,
      opREf = aROpAtmcSwcInstRef,
      srvrCallPnt = aSynchSrvrCallPnt,
      rnbl = aRnblEnt,
      comSpec = aClntComSpec);
    runbls.add(result);
  }
}
```

Listing 16: Accessing $1 : m..n$ equivalent elements

```
...
// Sender-Receiver creates Runnables for operations
aIntBhvr.runnable.addAll(
  sSwCmp.ports.providedInterfaces.flatten()
    .includingAll(sSwCmp.ports.requiredInterfaces.flatten())
    .select(i | isSenderReceiverInterface(i))
    .interfaceItems.flatten().equivalent()
    .select(eq | eq.isTypeOf(Tuple)
         and sSwCmp.ports.includes(eq.sPort))
    .collect(eq | eq.rnbl));
...
```

the `RunnableEntity` elements created for each operation. In line 8 we find the `Tuple`s and in line 10 we access the *rnbl* element of the tuple, which is the `RunnableEntity` (line 28 of Listing 15).

4.8 Takeaways

The main takeaway from the implementations is that there were no scenarios that were impossible to implement in either M2M_IE or ETL. Both technologies offer the capabilities that are needed to generate transformation rules. We believe that this stems from the fact that both approaches leverage languages that allow complex algorithms to be implemented. During implementation of the ETL rules, having control of the structural hierarchy helped guide the

Listing 17: A snippet of the TurningSignal ARXML produced by ETL

```
...
<CLIENT-SERVER-OPERATION>
  <SHORT-NAME>SignalStatus</SHORT-NAME>
    <ARGUMENTS>
      <ARGUMENT-DATA-PROTOTYPE>
        <SHORT-NAME>TurnSignalStatus_Arg</SHORT-NAME>
        <TYPE-TREF
              DEST="APPLICATION-PRIMITIVE-DATA-TYPE">/Architecture/
              DataTypes/ApplicationDataTypes/Status</TYPE-TREF>
          <DIRECTION>OUT</DIRECTION>
      </ARGUMENT-DATA-PROTOTYPE>
    </ARGUMENTS>
  </CLIENT-SERVER-OPERATION>
...
```

implementation in a top-down approach, resulting in reduced complexity. Further, ETL allowed to divide some of the more complex rules in M2M_IE into smaller, more manageable rules. Having a single function to access the trace model resulted in less bugs due to misspelled names. Finally, the support for $1:n$ and $1:m..n$ rules results in a more fine-grained trace.

5 EVALUATION

In order to verify the ETL implementation, we ran the transformation on the turn-signal (client-server) SysML model used in Siavashi et al. [2023]. We also used both approaches on a windshield-wiper system that uses a sender-receiver communication mode.

5.1 ARXML file comparison

As XML is quite verbose, listing a full comparison of the generated files is not viable. Rather, we will discuss the file contents relevant to the key differences in the tools' capabilities. All the ARXML files generated by both tools comply with the AUTOSAR schema.

Listings 17 presents a snippet of the ARXML generated by ETL, in particular for a `ClientServerOperation`. The listings highlight the limitation of the M2M_IE approach to correctly set the operation's arguments direction. In the ETL output the `DIRECTION` ((listing 17) line 1042) of the *TurnSignalStatus_Arg* argument is correctly set (`OUT` direction). In contrast, in the M2M_IE output the direction is not provided, i.e. the default value will be used.

In both tools, a package in the model has to be selected as the transformation starting point. However, as opposed to M2M_IE, the ETL tool can transform elements that are outside the selected package. For example, ETL can transform `DataTypes` from imported profiles when used by another element in the model. Listing 18, shows that the primitive Boolean (from SysML) has been added to the ARXML. These DataTypes are not present in the M2M_IE ARXML.

5.1.1 PREEvision validation. In Siavashi et al. [2023] we discussed how the PREEvision tool was used to validate the AUTOSAR models, within some pre-defined error acceptance. Table 1 presents the summary of the warnings and errors reported by the PREEvision consistency checker. The results show that the ETL approach produces models with fewer warnings and errors than the M2M_IE

Listing 18: A snippet of the ARXML produced by ETL showing primitive datatypes.

```
...
<AR-PACKAGE>
  <SHORT-NAME>ApplicationDataTypes</SHORT-NAME>
  <ELEMENTS>
    <APPLICATION-PRIMITIVE-DATA-TYPE>
      <SHORT-NAME>Boolean</SHORT-NAME>
      <CATEGORY>STRING</CATEGORY>
      <SW-DATA-DEF-PROPS>
        <SW-DATA-DEF-PROPS-VARIANTS>
          <SW-DATA-DEF-PROPS-CONDITIONAL>
            <SW-TEXT-PROPS>
              <ARRAY-SIZE-SEMANTICS>FIXED-SIZE
              </ARRAY-SIZE-SEMANTICS>
              <SW-MAX-TEXT-SIZE>10</SW-MAX-TEXT-SIZE>
            </SW-TEXT-PROPS>
...
```

Table 1: PREEVision Validation Comparison

System	Tool	Warnings	Errors
TurnSignal	M2M_IE	13	4
	ETL	10	0
WindshieldWiper	M2M_IE	11	3
	ETL	2	0

approach. In fact, by using ETL we were able to remove all errors from the ARXML output for both models.

5.1.2 Transformation Trace. In M2M_IE, the trace information is captured in a table, where each row represents a trace tuple. The M2M_IE only captures the rule name, the source element and the target element. The table can only be visualized within Rhapsody (stored in proprietary format). While visualizing the table, the reference to SysML model elements can be navigated to the source element(s), similar functionality is not applicable to the target elements. Target elements are only stored as string values of the target element type and name. The major drawback of this approach is that not all AUTOSAR elements have name, in which case the trace information would be unsound. Additionally, if the trace needs to be queried at a later stage, e.g., during certification, it will be very hard to locate the AUTOSAR elements. In total, for the Windshield Wiper and Turn Signal systems, M2M_IE created 56 tuples.

For ETL we are able to generate a trace with not only more fine grained information for the model elements, but that also includes information about the transformation engine and the transformation script. A screenshot of the Trace*a* model is presented in Fig 4. The trace model has 217 tuples (LeafTraceLinks) in total and the tree structure in the right shows the extra metadata captured, like the date and the agent that generated the trace (in this case ETL). Note that the source and target elements have an 'id' that can be used to locate them. The other benefit of using Trace*a*, is that the trace model can hold trace information from other design activities/stages making it the single source of information for change management and certification, among others.

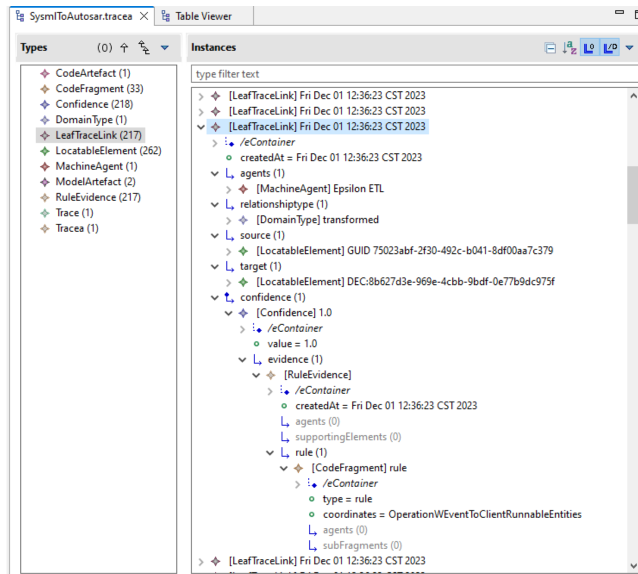

Fig. 4: Tracea Trace Links

6 RELATED WORK

Open-source tools have been successfully used in industry to offer alternatives or complement COTS tools. CaMCOA [5] is a software architecture specifically designed for Rolls-Royce's Controls and Monitoring systems. The CaMCOA workbench relies on OSS modeling frameworks, domain-specific language frameworks and model management tools. Further, some of these tools allows CaMCOA to be used in conjunction with COTS modeling and verification tools. In the cross-tool domain, support for other COTS tools has been previously added to the Epsilon EMC. Support for the PTC Integrity modeler was described by Zolotas et al. [2020], whilst Sanchez et al. [2021] described how the connection to Simulink was provided. In both cases the authors acknowledge that the EMC offers lower performance (than the tools native access), but that the use of EMC and the Epsilon languages provide usability benefits. As far as we know, there are no other publicly available references that share the experience of migrating MBSE COTS solutions to OSS alternatives.

7 CONCLUSION AND FUTURE WORK

This paper addresses the limitations of a COTS model transformation tool using an open-source alternative. We showed how abstracting the access to the SysML and AUTOSAR models can reduce bugs related to the context switch between APIs. We demonstrated that by using a language that supports the $1:1$, $1:n$ and $1:m..n$ mappings required by the transformation specification, the transformation trace can include more fine-grained information.

We also showed that allowing a source element to be transformed by multiple rules, it is possible to split complex rules into smaller rules, reducing the code complexity. Although not fully discussed, the ETL development environment helped us reduce the implementation time and injected bugs. The environment will also improve maintainability. Finally, the ETL implementation allowed us to eliminate all validation erros in the ARXML models.

As part of the future work, we plan to extend the transformation to support the reverse transformation (AUTOSAR to SysML) as an initial step to allow model synchronization between system and software architectures.

REFERENCES

[1] AUTOSAR. 2020. *ARXML Serialization Rules.* Technical Report. AUTOSAR Standards 779. AUTomotive Open System ARchitecture partnership.

[2] AUTOSAR. 2022. AUTOSAR XML SchemaProduction Rules.

[3] Edouard R. Batot, Jordi Cabot, and Sébastien Gérard. 2021. (Not) Yet Another Metamodel For Traceability. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 787–796. https://doi.org/10.1109/MODELS-C53483.2021.00125

[4] Manfred Broy, Martin Feilkas, Markus Herrmannsdoerfer, Stefano Merenda, and Daniel Ratiu. 2010. Seamless model-based development: From isolated tools to integrated model engineering environments. *Proc. IEEE* 98, 4 (2010), 526–545.

[5] Justin Cooper, Alfonso De la Vega, Richard Paige, Dimitris Kolovos, Michael Bennett, Caroline Brown, Beatriz Sanchez Piña, and Horacio Hoyos Rodriguez. 2021. Model-Based Development of Engine Control Systems: Experiences and Lessons Learnt. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. ACM/IEEE, -, 308–319. https://doi.org/10.1109/MODELS50736.2021.00038

[6] Ismenia Galvao and Arda Göknil. 2007. Survey of Traceability Approaches in Model-Driven Engineering. In *Proceedings of the Eleventh IEEE International EDOC Enterprise Computing Conference (Proceedings IEEE International Enterprise Distributed Object Computing Conference (EDOC), 11)*. IEEE, United States, 313–324. https://doi.org/10.1109/EDOC.2007.4384003 11th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2007, EDOC ; Conference date: 15-10-2007 Through 19-10-2007.

[7] Anneke G. Kleppe, Jos Warmer, and Wim Bast. 2003. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., USA.

[8] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. 2006. The Epsilon Object Language (EOL). In *Proceedings of the Second European Conference on Model Driven Architecture: Foundations and Applications* (Bilbao, Spain) *(ECMDA-FA'06)*. Springer-Verlag, Berlin, Heidelberg, 128–142. https://doi.org/10.1007/11787044_11

[9] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. 2008. The Epsilon Transformation Language. In *Theory and Practice of Model Transformations*, Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 46–60.

[10] Tom Mens and Pieter Van Gorp. 2006. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science* 152 (2006), 125–142. https://doi.org/10.1016/j.entcs.2005.10.021 Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[11] Beatriz A. Sanchez, Athanasios Zolotas, Horacio Hoyos Rodriguez, Dimitris Kolovos, Richard F. Paige, Justin C. Cooper, and Jason Hampson. 2021. Runtime translation of OCL-like statements on Simulink models: Expanding domains and optimising queries. *Softw. Syst. Model.* 20, 6 (dec 2021), 1889–1918. https://doi.org/10.1007/s10270-021-00910-0

[12] Faezeh Siavashi, Horacio Hoyos Rodriguez, Vera Pantelic, Mark Lawford, Richard F. Paige, Monika Jaskolka, Guanrui Hou, and Alessandro Verde. 2023. Bridging the Gap Between System Architecture and Software Design using Model Transformation. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 51–56. https://doi.org/10.1109/ISSREW60843.2023.00046

[13] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework* (2 ed.). Addison-Wesley, Upper Saddle River, NJ.

[14] A. Wayne Wymore. 1993. Model-based systems engineering : an introduction to the mathematical theory of discrete systems and to the tricotyledon theory of system design. https://api.semanticscholar.org/CorpusID:108125826

[15] Athanasios Zolotas, Horacio Rodriguez, Stuart Hutchesson, Beatriz Pina, Alan Grigg, Mole li, Dimitrios Kolovos, and Richard Paige. 2020. Bridging Proprietary Modelling and Open-Source Model Management Tools: The Case of PTC Integrity Modeller and Epsilon. *Software and Systems Modeling* 19 (01 2020). https://doi.org/10.1007/s10270-019-00732-1