Model analytics for ASML's data and control modeling languages

Suresh, A.

Award date:
2018

Model Analytics for ASML's Data and Control Modeling Languages

Aishwarya Suresh

Eindhoven University of Technology
5612 AZ Eindhoven, The Netherlands

Abstract

In software engineering, duplication of code is an antipattern. In model driven engineering, clones might appear as (partial) duplication of models. Clones might be considered harmful for the maintenance of software artifacts, including models. Companies such as ASML, which is the leading producer of lithography systems, use models to define the functional behavior of such systems. In multidisciplinary teams that deal with ever evolving models, the presence of clones might be a negative quality, resulting in inconsistent evolution. To this end, the tool SAMOS (Statistical Analysis of ModelS) has been used to find clones within certain data models and control models as used by ASML. The approach taken to detect clones within these models is discussed in this work along with the clones detected using this approach. We also discuss whether, and how, such clones can be or should be eliminated or not, based on consultation with domain experts.

Keywords: Model clones, ASML, data models, control models, SAMOS

Contents

1. Introduction

With the increasing application of model-driven engineering (MDE), the need to address issues pertaining to the increasing number and variety of MDE artifacts, such as domain specific languages (DSLs) and models, is on the rise. One such issue is that of duplication and cloning in those artifacts. The presence of clones might negatively affect the maintainability and evolution of software artifacts in general, as widely reported in the literature [1]. When multiple instances of the same code or model fragment exist, a change required in such a fragment (to fix a bug, for instance) would also have to be performed at all instances of this fragment. Inconsistent changes to such fragments might also lead to incorrect behavior. Therefore, eliminating such redundancy in software artifacts might result in improved maintainability. While not all cases of encountered clones can be considered negative, as some might even be inevitable, it is worthwhile to explore what types of clones exist and what their existence might imply for the system.

Model clone detection techniques have been employed by tools such as NiCad/SIMONE [2], ConQAT [3], MACH [4] and SAMOS [5] to detect clones in MATLAB/Simulink and UML models, Ecore metamodels and other types of models. The primary focus of this work is to discuss the applicability of SAMOS for the detection of clones in data and control models provided by ASML. ASML is a provider of lithography systems used to manufacture semiconductor chips. Different disciplines that work together to manufacture these lithography machines employ the use of various domain-specific modeling ecosystems [6]. A collection of models from one such ecosystem was provided for analysis, with the goal of detecting clones among these models. This goal was accomplished using the tool SAMOS. The framework of this tool was extended to incorporate the ability to process the models provided by ASML. Following this extension, the various settings available to compare model elements was used and the results found have been discussed.

The following section provides discussion of the research questions that were pursued in this work. This is followed by an introduction to the tool SAMOS along with its existing functionality. The nature of the models provided by ASML is then discussed, followed by a description of the approach taken to find clones in these models. The case studies conducted using this approach are then presented and the results of these case studies are discussed. We then present the threats to validity of our approach and discuss some related work. Finally, the possibility of future work in this area is

discussed along with some conclusions about our work.

2. Research Questions

This section presents the research questions that were pursued to detect clones in the models used at ASML.

We consider the functionality available in SAMOS to tackle the problem of clone detection. Since SAMOS already provides a means to detect clones for models adhering e.g. to the Ecore metamodels (inspired by the UML model clone classification [4], we explore how this framework can be extended to include models adhering to the domain-specific metamodels used at ASML.

ASML uses the ASOME modeling language [7] to model the behavior of its machines. ASOME allows modeling by applying the Data, Control, Algorithms (DCA) architecture, which they have developed in order to to separate the data, control and algorithm aspects such that they can be defined, verified and validated in isolation. This should result in a higher quality (less errors, more predictable behavior) system specification and implementation.

To analyze ASOME models in SAMOS, we first need to understand the elements involved in these models, based on the metamodels they adhere to. This is necessary to configure the feature extraction part, determining settings such as which model parts to extract (and in which specific way) or to ignore. Moreover, while SAMOS defines comparison schemes for the comparison of features extracted from models such as Ecore metamodels and feature models, it has yet to be examined if these comparison schemes are suitable for ASOME models.

Once clones have been detected within ASOME models, there is the matter of evaluating these clones to gauge their accuracy and relevance. The goal of the clone detection in this context is to find a way to use this information to reduce the level of cloning in the models. To this end, we examine firstly if the model fragments detected as clones are indeed clones. Secondly, we examine which of the clones that have been found can or should be removed. In conclusion, to address the problems as discussed above, the following research questions were posed:

RQ1 *"How can SAMOS be applied or extended to detect the relevant clones in the ASOME models provided by ASML?"*

The detection of clones within the ASOME models provided by ASML involved an extension to SAMOS. Thanks to this extension, we could conduct

case studies to determine what kind of clones could be found in the given ASOME models. The following research question addresses the results found using the extension of SAMOS as mentioned in *RQ1*.

RQ2 *"What clones can be found in ASOME models using the extended SAMOS?"*

RQ3 *"(How) can we use the discovered cloning information to help maintain and potentially improve the MDE ecosystems at ASML?"*

The research questions presented above have been discussed in various sections of the work. RQ1 has been addressed through sub-questions in Sections 6.1 and 6.2. RQ2 and RQ3 are discussed in Section 7.

3. Background: SAMOS Model Analytics Framework

The SAMOS (Statistic Analysis of MOdelS) framework is a tool developed for the large-scale analysis of models using a combination of information retrieval, natural language processing and statistical analysis techniques [5]. The model analytics workflow of SAMOS is as shown in Figure 1. The process starts with an input of a collection of models that adhere to a particular metamodel. SAMOS has so far been used for the analysis of e.g. Ecore metamodels [8] and feature models [9].

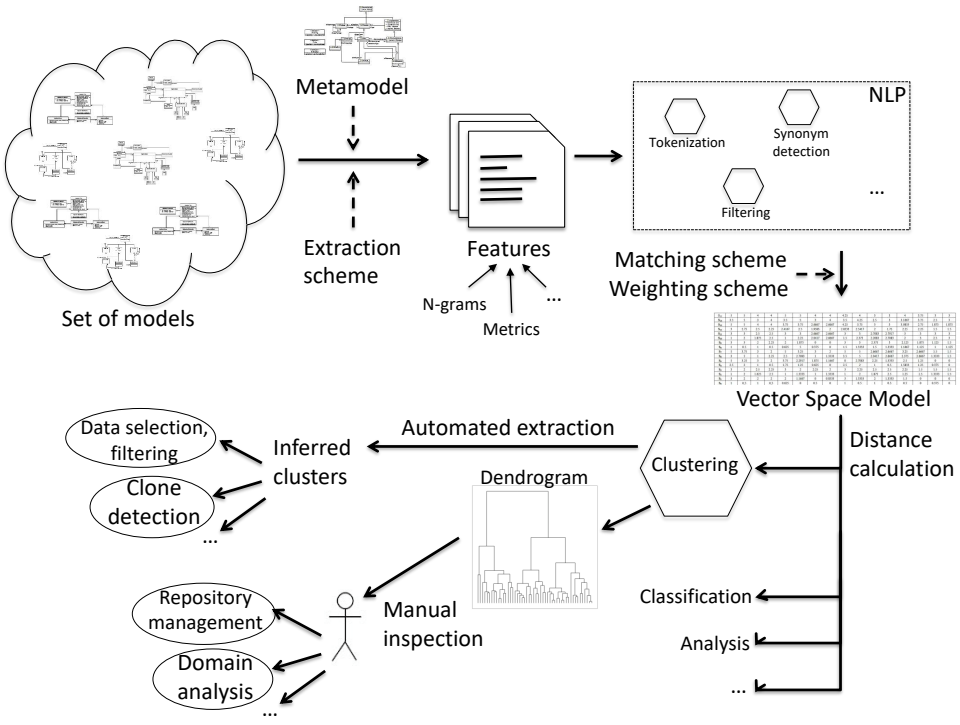

Figure 1: SAMOS Workflow

Given a collection of models[1], SAMOS first applies a metamodel specific extraction scheme to retrieve the features of these models and store them in feature files. Features are attributes of model elements that have been considered relevant for the comparison of such elements to detect clones. As seen in the figure, SAMOS represents the features in terms N grams, or metrics [5]. Once the feature files have been extracted, the following steps are independent of the type of model. The extracted features from the feature files undergo a natural language processing step to tokenize and lemmatize the values of these features for a fair language based comparison. Features are compared to construct a Vector Space Model (VSM) [10] after assigning a weighting scheme to feature attributes based on the type of feature being compared. The comparison schemes available in SAMOS allow the user to configure what aspect of the features needs to be compared, for example, the

[1]In the model analytics context, we use metamodels synonymously with models, in the sense that they are models too (adhering to the corresponding meta-metamodel).

name of the feature or the type of the model element. After choosing the comparison scheme, a VSM is constructed where each model is represented as a vector comprised of the features that occur in these models. Ultimately, the distance between the vectors represents the distance between the models they represent. The ability to calculate the distance between models allows one to cluster the set of models based on how similar they are to each other. Dendrograms provide a hierarchical clustering of models with respect to their similarities. Aside from detecting clones, inspecting such dendrograms would give one insight into the working of the ecosystem. Language focused analyses could be done to detect common concepts discussed across the various domains in a MDE ecosystem.

The workflow as detailed above can be modified to include scopes. By identifying meaningful scopes for models (such as treating classes and packages separately in a class diagram, in contrast to the whole model as a single entity), the settings in SAMOS allow for an extraction of features at the level of the defined scope. Furthermore, the comparison scheme used for the construction of the VSM can be configured to employ various kinds of comparisons, e.g. treating model elements such as names and types in certain ways, or ignoring them altogether.

4. MDE Ecosystems at ASML

The development of complex systems involves a combination of skills and techniques from various disciplines. The use of models allows one to abstract from the concrete implementation provided by different disciplines to enable the specification and operation of complex systems. Artifacts originating from such models coming from different disciplines; i.e., code generated from these models, are used to run these systems. However, shortcomings or misunderstandings between the disciplines involved at the model level can become visible on the implementation level. To avoid such shortcomings, it is essential to resolve such conflicts on the model level. To this end, Multi-Disciplinary Systems Engineering (MDSE, used synonymously with MDE in our work for simplicity) ecosystems are employed to maintain the consistency among inter-disciplinary models.

ASML is developing such MDE ecosystems by formalizing the knowledge of several disciplines into one or more domain specific languages [11]. The separation of concerns among the different disciplines helps with handling the complexity of these concerns. Clear and unambiguous communication

between different disciplines is facilitated to enable not only the functioning of the complex system, but also its ability to keep up with the evolving performance requirements. Furthermore, the design flow is optimized, resulting in a faster delivery of software products to the market.

In such an ecosystem, concepts and knowledge of the several involved disciplines are formalized into of one or more domain specific languages (DSLs). Each MDE ecosystem has its own well defined application domain. Examples of developed MDE ecosystems at ASML are:

- *ASOME*, from ASML's Software application domain. It enables functional engineers from different disciplines to define data structures and algorithms, and allows software engineers to define supervisory controllers and data repositories [7];
- *CARM2G*, from ASML's Process Control application domain. It enables mechatronic design engineers to define the application in terms of process (motion) controllers (coupled with defacto standard Matlab/Simulink), providing a means for electronic engineers to define the platform containing sensors, actuators, the multi-processor, multi-core computation platform and the communication network, and means for software engineers to develop an optimal mapping of the application on to the platform, see [12, 13];
- *Wafer handler (WLSAT)*, from ASML's Manufacturing Logistics application domain. It provides a formal modeling approach for compositional specification of both functionality and timing of manufacturing systems. The performance of the controller can be analyzed and optimized by taking into account the timing characteristics. Since formal semantics are given in terms of a (max, +) state space, various existing performance analysis techniques can be reused [14], [15], [16].

4.1. ASOME Models

The ASOME MDE ecosystem is a software development environment that supports the DCA pattern, which separates Data, Control and Algorithms. A motivation to employ this architecture pattern is to avoid changes in the behavior of a system based on a change in data. In situations where a single change in measured data leads to a change in a chain of actions, this DCA pattern proves to be advantageous. The result of employing such a pattern is providing an environment where data can be stored and manipulated

separately from the environment where the behavior can be defined. Moreover, algorithms can independently be designed to perform operations using the defined data. Using techniques of MDE, ASOME provides metamodels to create data and control models independently of each other. Algorithm metamodels allow for the specification of algorithm models that perform computation on the data during different steps of the control flow.

In the context of DCA and ASOME, data is one of the aspects. (Similar, we also talk about the 'control', 'algorithm' and 'system' aspect.) Within this data aspect, several kinds of systems, interfaces and realizations can be recognized. Domain Interfaces and Repository services are just a few examples. Concepts such as Data Shifters, Data Access Interfaces, Domain Data Services etc. also exist. ASOME models that contain elements can be categorized according to these aspects. For the work in this paper, our investigation was limited to the area of Domain Interfaces with respect to data. This area is referred to when data models are discussed.

Different data elements of an ASML component are represented using one or more data models, the elements of which adhere to a collection of metamodels. Data models, as seen in figure 2 contain:

- *Domain Interfaces* Any kind of interface in ASOME can express a dependency on another interface. Interfaces allow dependencies between models to allow the use of model elements among different models. Within these domain interfaces, several model elements reside including enumerations, entities, primitive type definitions and value objects.

- *Attributes and Associations* that have a name and *type* information. Attributes and associations have multiplicities to represent how many instances of these elements can exist at run-time. Associations additionally have a source multiplicity to denote how many instances of the source of the association can be involved in the association for each target. For attributes and associations that could have multiple instances, the order of such instances might be relevant. To represent such optional ordering of instances, an attribute *order* is used to represent if this collection of instances is ordered or not.

- *Entities* which are model elements that contain value objects, attributes and associations to other entities (within the same model or from different models). Entities additionally allow a user to define properties

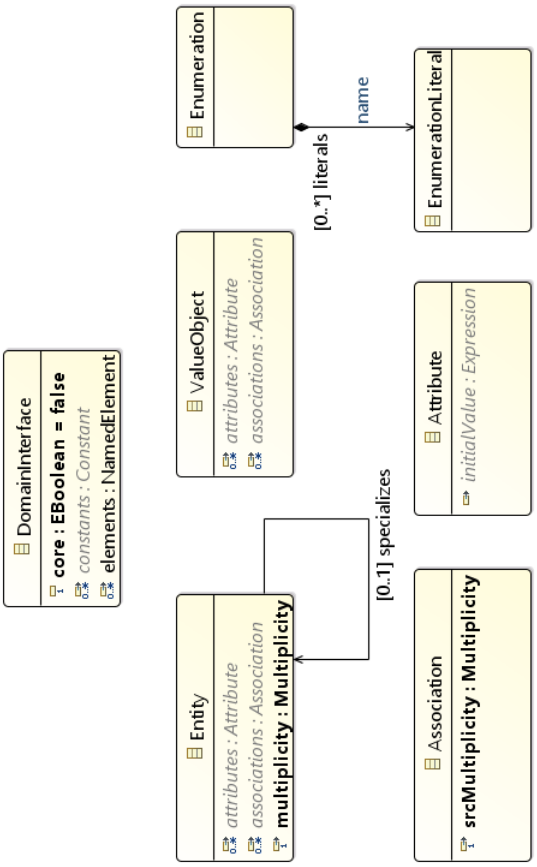

Figure 2: Basic elements in a data model

such as deletability, mutability, and persistency.

- *Value Objects* contain attributes and associations to other entities or value objects. The concept of value objects has been introduced to be able to avoid repetition. Value objects facilitate the modeling of data that does not need to be persisted. This relates to the DCA pattern where the Algorithm component might need some intermediate artifacts for some computations.

- *Enumerations* which contain a collection of constants called *Enumeration Literals*.

Control models allow a user to model the behavior of different components of the system at hand. This is done using state machines. From the models received, it was observed that control models were of three different types - composite, interface and design[2]. We have categorized control models accordingly for the purposes of this work. The construction of complex systems in ASOME control models is done using instances of some smaller systems. This concept is often called the application of 'a system of systems' pattern. Composite models contain a decomposition defining what system instances it is made up of along with how they are connected through ports and interfaces. An interface model provides a protocol for a state machine along with a definition of how the system and its interfaces can be defined. A design model uses this protocol to define a concrete realization of this system. The components of interest within these models, and as seen in figure 3 are:

- *State Machines* defined in the protocols of interface models or realizations of design models. A state machine consists of states, transition states and variables used within these states.

- *States* used to represent different states of the system being modeled using control models. Every state machine consists of a number of states; one of them is indicated as the initial state.

[2]composite is strictly a part of the 'system' aspect and not the 'control' aspect. As with data models, the concept of a control model as such does not exist within ASOME. For the purposes of this paper, we group three kinds of systems or interfaces and call them control models

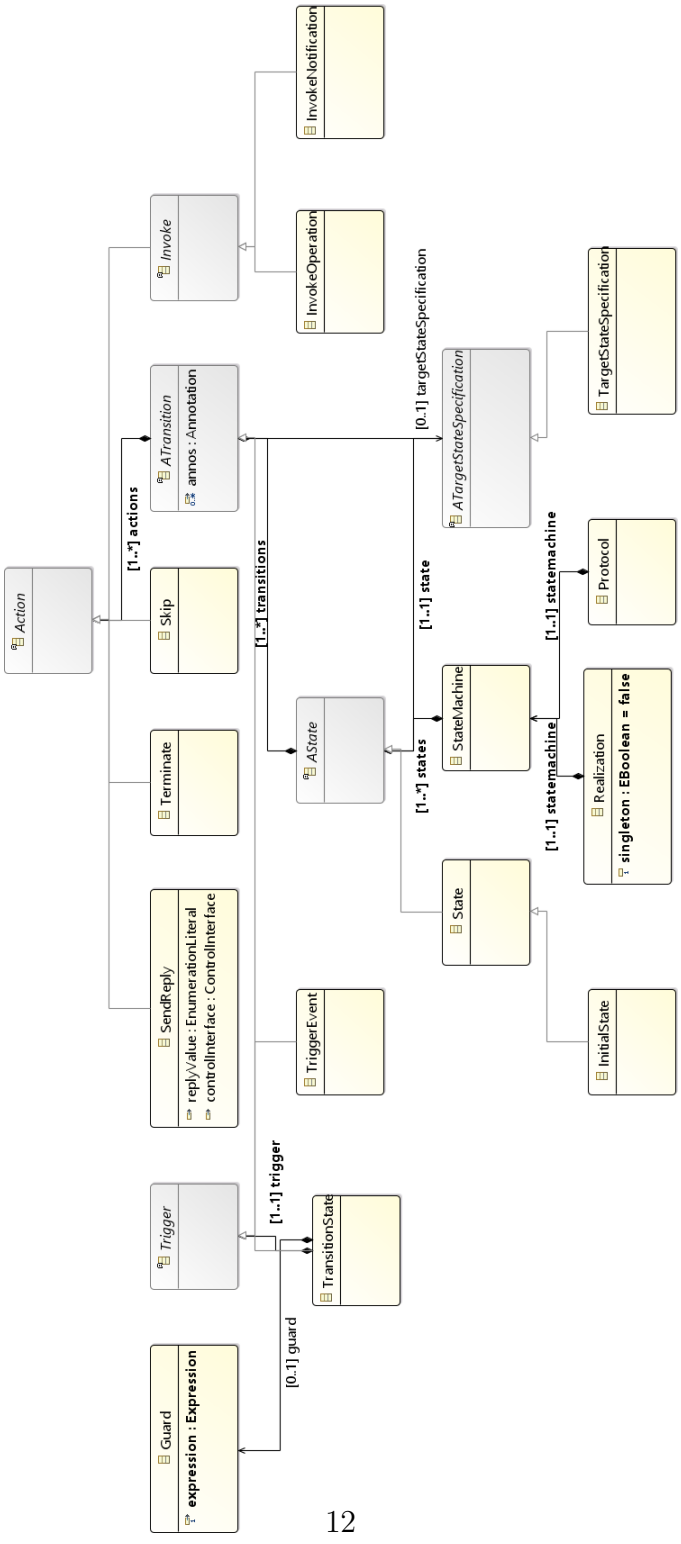

Figure 3: Basic elements of a control model

- *Transition States* contained within the state to model the behavior of the system based on different triggers. Each transition state is associated with such a trigger, followed by one or more actions and optionally a guard and a target state specification. Guards are specified using the expressions defined in the *Expression* metamodel.

- *Actions* which define the activity that follows when the event defined in the trigger occurs. A sequence of one or more actions is defined in each transition state. These actions could include sending a reply to an interface of a model, terminating the control flow, invoking an operation or a notification, etc.

While ASOME also facilitates the specification of Algorithm models, these were not considered for the purpose of finding clones in this work. The following section details how SAMOS can be used to extract information from a collection of data and control ASOME models in order to find clones.

5. Model Clones: Concept and Classification

Before detailing the process of clone detection, it is essential to consider what defines a clone. Model clone detection is a relatively newer topic of exploration as compared to code clone detection [17]. While there are clear definitions of what constitutes a clone for code, such a definition is not as clear for models. The first step to approach the problem of clone detection for ASML models using SAMOS, was to define what model clones are. A model fragment (a part of a model) is considered to be a clone of another fragment if the fragments are considered to be highly similar to each other. Therefore, the idea of model clones boils down to *groups of model fragments that are highly similar to each other* in the general sense.

Another aspect of model clone detection is the categorization of the types of clones that can be detected. For the purposes of this work, the classification used in [5] has been used, and is as follows:

- *Type A.* Duplicate model fragments except secondary notation (layout, formatting), internal identifiers, including *cosmetic changes* in names such as case.

- *Type B.* Duplicate model fragments with a *small percentage* of changes to names, types, attributes with a few addition or removal of parts.

- *Type C.* Duplicate model fragments with a *substantial percentage* of changes or additions or removal of names, types, attributes and parts.

For the ASOME data models, the names of elements are considered relevant (elements with similar names are candidates for elimination). The classification of clone types takes changes in the name of model fragments into account. However, for the ASOME control models, since the behavior of these models is to be analyzed and the structure of the models represents behavior, the classification of clones takes into account the addition or removal of components that modify the structure of the model (in the sense of finding *structural* clones). This is partly in line with the clone category of *renamed clones*, as investigated in the model clone detection literature (e.g. in [2] for Simulink model clones).

6. Using and Extending SAMOS for ASOME Models

SAMOS is natively capable of analyzing certain types of models, such as Ecore metamodels. However, it needs to be configured to the domain-specific ASOME models. The current section addresses:

[RQ1] *"How can SAMOS be applied or extended to detect the relevant clones in the ASOME models provided by ASML?"*

The workflow of SAMOS, as represented in Figure 1 involves the extraction of relevant features from the models. This extraction scheme is metamodel specific and therefore, an extension to SAMOS is first required, to incorporate a feature extraction based on the ASOME metamodels. To answer this question, we first consider the following subquestion:

RQ1-1. *How can we incorporate a way to process ASOME models in SAMOS?*

As addressed in Section 3, SAMOS already uses a customizable workflow for extracting and comparing model elements, e.g. for clone detection. The first step to do this is the metamodel dependent extraction of features; separate extractors exist for different types of models. However, once the extraction has been completed, the comparison of extracted features is done in a model-independent way. Therefore, to begin the clone detection for ASML's ASOME models, a custom feature extraction needs to be implemented for each type of model, addressed in the following subsection.

6.1. Feature Extraction

The current section discusses the following aspect of the extension to SAMOS to incorporate ASOME models:

RQ1-2. *What kind of information that is relevant for clones needs to be extracted from the models? How can this information be represented in SAMOS?*

The first step to analyzing the set of models to detect clones is to determine the information that is relevant for comparing model elements. In the feature extraction phase as seen in Figure 1, first, the collection of metamodels which jointly define what the Data and Control models adhere to were inspected. Along with input from a domain expert, we gained insight into the features for each model element that could be considered relevant for clone detection. For instance, the model element *Model* contains the feature *name* that was considered relevant for extraction. An *Attribute* on the other hand contains the features *name* and *type*. Given that the ASOME models to be analyzed were Data and Control models, a separate extraction scheme exists for each type of model.

Running the comparison part of SAMOS on the extracted features results in a model to model comparison based on all the elements contained in each model using their features. This extraction scheme however does not allow for the detection of clones within each model as it compares each model as a whole with every other model. We therefore used the scoping facility of SAMOS to restrict the entities to be extracted and compared. The scopes involved for Data and Control models are detailed below.

Data Models. Figure 2 is a basic representation of the elements contained in the data models. The scopes defined for the extraction of data models are as follows:

- *Model:* Setting the scope to this level allows results in the extraction of one feature file per model, containing all the elements present in the model along with their relevant features. Data models contain domain interfaces along with model elements such as imports and data ports. Such elements were deemed less important for the comparison of models by domain experts and hence lead to the following reduced scope for comparison.

- *Domain Interface:* The scope domain interface extracts a set of features for each domain interface contained in the model. This set of features contains all the model elements, along with their features, present in the domain interface.

- *Structured Type and Enumerations:* Reducing the level of comparison to this scope allowed one to compare each entity, value object and enumeration within a model with each other. Feature files per model fragment at this level contain the model elements attributes and associations, along with their features.

- *LevelAA:* Using this scope resulted in the extraction of one feature file per attribute or association consisting of the features of this attribute or association. These features include the name of the attribute or association along with their multiplicities. Additionally, a property *Asome Type* represents the type of the attribute or association. This *Asome Type* could be an Entity, Enumeration, Value Object or Primitive Type defined in the same model or in a different model. The detection of clones at this level was used to find micro clones within the models [18].

Control Models. Figure 3 represents the basic elements of ASOME control models.

As for data models, the level of granularity of comparison can also be altered for control models.

- *Model:* This level allows a comparison of models in their entirety.

- *Protocol:* Narrowing down the scope to the level protocol allows for the comparison of elements contained within a protocol or realization. A control interface, defined in an interface model, uses a state machine to specify the allowed behavior (protocol) along that interface. The state machine is contained in (or by) the control interface. A control realization, defined in a design model, needs to provide a specification that adheres to the control interfaces it provides and requires. This specification is also done by a state machine. A control realization also owns a state machine.

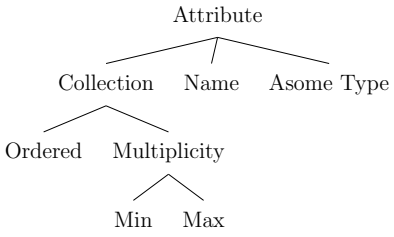

Figure 4: Attribute structure

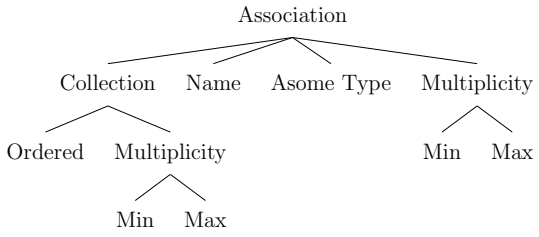

Figure 5: Association structure

6.1.1. Domain Specific Concerns for Extraction

The feature extraction with the scopes detailed above allow for the comparison of models based on the extracted features. However the as is representation of model elements might lead to inaccurate results. The following problems have been identified and the solutions to these problems have been presented.

Collections and Multiplicities. The feature extraction process following the containment relation between the model elements was problematic in the case of *multiplicities* and *collections* in data models. For attributes and associations, two model elements, *collections* and *multiplicities* exist to denote the number of instances of the attribute. Collections contain a flag *ordered* which can be set to true if the collection of instances of the attribute or association are ordered. Collections also contain multiplicities which have two properties min and max to denote the upper and lower bound of the number of instances of the aforementioned attributes or associations. This can be seen in Figures 4 and 5. As seen in Figure 5, associations contain an additional *multiplicity* (outside the *collection* element) with min and max properties to denote the multiplicity of the source of the association. For associations, the multiplicity of the target of the association is represented through the *collection* element. The Asome Type then denotes the target of the association.

Such an extraction of features leads to a redundant representation of features which in turn leads to inaccurate similarities between models. For attributes or associations that have single instances, a line for the Collection element with the property *ordered* set to false is extracted. Additionally, there is also a line denoting the Multiplicity element with properties *min* and *max* set to 1. However, when entities or value objects contain multi-

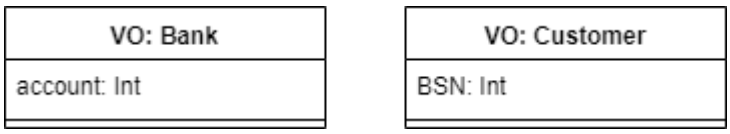

Figure 6: Example problem: collections and multiplicities

ple attributes and associations with single instances, it was observed that the result of the clone detection was inaccurate. Especially for the scope *Structured Type and Enumeration*, it was observed that two entities or value objects that did not have many similar attributes were still in the same cluster. This was due to the presence of separate features for the collection and multiplicity elements.

Figure 6 shows two value objects with one attribute each. A domain expert might consider these two value objects completely different as they do not have the same names or the same attributes. However, the feature files for these two value objects would look as follows:

1. *type:*ValueObject, *name:*Bank
   *type:*Attribute, *name:*account, *Asome Type:*Int
   *type:*Collection, *ordered:*false
   *type:*Multiplicity, *min:*1, *max:*1

2. *type:*ValueObject, *name:*Customer
   *type:*Attribute, *name:*BSN, *Asome Type:*Int
   *type:*Collection, *ordered:*false
   *type:*Multiplicity, *min:*1, *max:*1

While the two value objects are different from each other, comparing them would result in a similarity of 50% due to the model elements collection and multiplicity.

To solve this problem of collections and multiplicities resulting in inaccurate clone detection, the extraction of the features in the model was modified for attributes and associations. Now, instead of extracting features for collections and multiplicities individually, these properties are appended to the attribute or association. Figure 7 denotes how the extraction has been modified for associations. The properties Name, Asome Type, Ordered, Min,

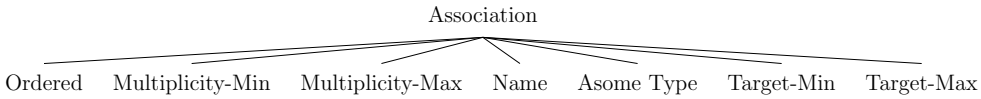

Figure 7: Modified association structure

Max, Target_Min and Target_Max have now been added to the association, eliminating the need for the separate extraction of the collection and multiplicity elements. Similarly, the properties Name, Asome Type, Ordered, Min and Max have been appended to attributes to solve the problem of collections and multiplicities. The presented solution allowed for the representation of features in files that were not cluttered with lines representing collection and multiplicity values that showed two files as more similar than they were. The result of this modification is a more meaningful comparison of model elements resulting in a more accurate clone detection process.

Unigram structure. For the scopes detailed for data models, the features were extracted with a *unigram* structure. This representation extracts a line of features for each model element contained in the fragment chosen using the scope. This representation of model features does not include any structural information about the elements, i.e., does not include any relationship between the model elements.

Annotations. With MDE systems, maintaining traceability between models and eventually derived or generated artifacts, such as code, is important. ASOME uses annotations in control models to provide this traceability between their systems. In control models, for transition states within a state, such annotations are introduced. During the extraction of features from models, annotations are also extracted. However, the behavior of the model does not depend on these annotations and therefore, including these annotations impedes the accuracy of the clone detection process. Therefore, the extraction of model features excluded the extraction of such annotations. *However, as these annotations have been used widely, and consistently, the ASOME language could be modified to incorporate these annotations as an essential part of the models.*

One depth tree structure. Unlike data models, for control models, the structure of the model needs to be taken into account for clone detection. This stems from the fact that the structure for these models represents its behavior. To include this structural information of the model, a tree based extractor was constructed for control models. For this representation, each node consists of the properties of the model element, similar to each line of the features extracted in the *unigram* comparison.

Following this, between these model element nodes, a node exists that contains information about the relationship between these nodes. The nodes are extracted following the containment relationship of the model. However, associations between these nodes are also separately modeled (using information from the control metamodels). The result of the extraction is a tree whose root node starts at the scope that has been defined.

However, the comparison of models on the full tree structure was too time consuming. To overcome this problem, we reduce the depth of the trees being compared.

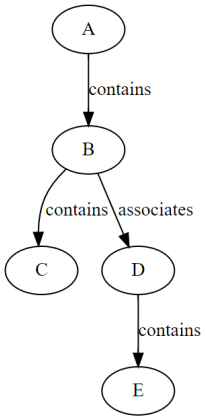

Figure 8: Full tree representation

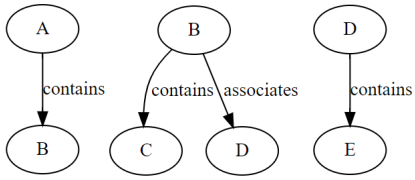

Figure 9: One depth tree representation

Figure 8 represents a full tree. A modification to the tree extraction allowed one to extract corresponding one depth trees, as shown in figure 9. In such a representation, for each node, the nodes related through the containment relationship and associations were modeled as children. For example, a node containing the features of a state modeled as a parent node would have all the transition states contained in the state as children with an intermediate node representing that the relationship between the state

and the transition states is containment. A separate node then exits for each transition state (modeled as a parent node) with all the contained actions as children and an intermediate node to denote the containment relationship between these elements.

Generic comparison techniques exist for models, such as a graph comparison of model structure. However, through the feature extraction described above, using domain knowledge improves the power of the approach as compared to a generic one.

6.2. Feature Comparison and VSM Calculation

A different aspect of RQ1 is considered in this section:

RQ1-3. *How can the extracted information be compared to detect patterns?*

Using the features that have been extracted as described in the section above, SAMOS then performs the feature comparison step. This is done by constructing a Vector Space Model (VSM). All the features extracted correspond to a column in this VSM. Every row in the VSM corresponds to the model fragment being compared, as defined by the scope, i.e., if the scope is *Model*, a row exists for each model. In this row, a number representing the occurrence of each feature in the model is calculated for the column representing the feature. The resulting VSM is a vector for each model fragment (based on the selected scope) representing the occurrence of all the features within this model fragment. This construction of the VSM allows for some settings for the comparison of model elements.

The comparison of two features in SAMOS involves a combination of comparisons between the names of the features, their types and the remaining attributes they contain. The comparison of names has been the focus of SAMOS. Many techniques have been combined to facilitate name comparisons. Natural language processing techniques such as tokenization, lemmatization, synonym detection, etc. have been employed to compare names as accurately as possible.

SAMOS allows for the comparison of features by ignoring either the name or the type information of the features. The setting *Ignore Type* takes into account all the attributes of a feature excluding the type. The type information for data models is a combination of both the model element type as well as the *Asome Type*, which could be an Entity or Value Object belonging to the same or different model or a primitive type. The setting *Strict Type*

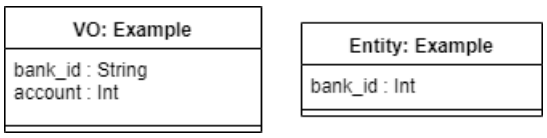

Figure 10: Example model elements for comparison

allows for the comparison of all the attributes of a feature including the type. However, this places a constraint on the comparisons in that two model elements of different types with similar attributes would not count as similar in this setting. To include the type information for comparison, the *Relaxed Type* setting has been introduced by attaching a weight of 0.5 to mismatched elements and a weight of 1 to matching elements. To facilitate the comparison of features without considering the names of these features, the *No Name* setting has been introduced. For data models, names are a relevant part of the comparison. However, for control models, where structure is more important, maybe the names of the elements would not be relevant for the comparison. Along with the scoping settings for data and control models, SAMOS, therefore, also allows for the type comparison settings *Ignore Type*, *Strict Type*, *Relaxed Type* in combination with two name comparison settings, namely *Name* and *No Name*. To illustrate how the VSM is influenced by the different comparison settings, consider the following example. We compare two model elements on the *Structured Type and Enumeration* scope. These model elements may not necessarily belong to the same model. Figure 10 is a basic example of an entity and a value object (possibly from different models), being compared to build a vector space model. The feature files for these model elements are as follows:

1. The feature file for the entity would look as follows:
   *type:*Entity, *name:*Example
   *type:*Attribute, *name:*bank_id, *Asome Type:*Int

2. The feature file for the value object would look as follows:
   *type:*ValueObject, *name:*Example
   *type:*Attribute, *name:*bank_id, *Asome Type:*String
   *type:*Attribute, *name:*account, *Asome Type:*Int

The vector space model for these two model elements would be constructed based on the comparison scheme involving a combination of settings for name and type comparison as described above. Figure 11 shows the

Model Fragment	Type Setting	<Entity, Example>		<bank_id, Int>		<ValueObject, Example>		<bank_id, String>		<account,Int>	
		Name	No Name	Name	No Name	Name	No Name	Name	No Name	Name	No Name
Entity	Strict	1	1	1	1	0	0	0	0	0	1
	Relaxed	1	1	1	1	0.5	0.5	0.5	0.5	0	1
	Ignore	1	1	1	1	1	1	1	1	0	1
		Name	No Name	Name	No Name	Name	No Name	Name	No Name	Name	No Name
ValueObject	Strict	0	0	0	1	1	1	1	1	1	1
	Relaxed	0.5	0.5	0.5	1	1	1	1	1	1	1
	Ignore	1	1	1	1	1	1	1	1	1	1

Figure 11: Example VSM to illustrate comparison settings

values in the vector space model for all the combinations of the comparison schemes. The vector space model illustrated in Figure 11 is a linear one. It is called so because each element is compared with every other element in the other models. SAMOS adapts this VSM to create a quadratic VSM which represents the sum of comparisons of each feature in question with every other feature. For example, using the *No Name* and *Relaxed Type* setting, for the row representing the Value Object, and the column representing <account, Int>, the value would be a sum of:

- 1 : for the attribute account of type int in the value object.
- 0.25: for the feature ValueObject (no name comparison ignores the names, relaxed type comparison multiplies the mismatch between the type and Asome Type with 0.5 each).
- 0.5: for the feature Attribute with the name *bank_id* (no name setting ignores the name and the mismatch between the Asome Types assigns a penalty of 0.5).

While a value of 1 is assigned in the linear VSM, the quadratic VSM assigns a value of 1.75 for the row Value Object and column <account,Int> under the *Relaxed Type* and *No Name* setting.

However, not all comparison combinations are relevant for the detection of clones for different models. For instance, the combination *Ignore Type* along with the *No Name* setting might result in all elements with just name and type information but no attributes to be considered equivalent. Therefore, while all these comparison settings are available, for the case studies conducted, only a selected combination of settings is applied.

6.3. Distance Measurement, Clustering and Other Analyses

Following the construction of the VSM, SAMOS performs a vector-based distance measurement to compute the distance between each pair of model

elements being compared. This is done using a Bray-Curtis distance measurement [5]. Following this, based on the distance measurement, clustering is applied to identify clusters of clones. This has been done using the *Density-Based Spacial Clustering of Applications with Noise* (dbscan) algorithm in the dbscan package in R. The choice of this algorithm is justified because of its ability to find clusters in various shapes and to detect noise, and because of its suitability for large datasets.

7. Case Studies with ASOME Models

The current section discusses the case studies on clone detection for the ASOME models at ASML, addressing the following research questions:

RQ2 *"What clones can be found in ASOME models using the extended SAMOS"*
RQ3 *"(How) can we use the discovered cloning information to help maintain and potentially improve the MDE ecosystems at ASML?"*

7.1. Clone Detection in ASOME Data Models

This section discusses the results of the case studies performed using the different settings of SAMOS on the ASOME data models.

7.1.1. Dataset and SAMOS Settings

The dataset consists of 28 models, containing one domain interface each. Within these domain interfaces, 291 structured type and enumerations model fragments were found. Finally, a total of 574 elements were found for the comparison on the *LevelAA* scope.

For the comparison of data models, the scopes *Model* and *Domain Interface* did not yield significant results at higher similarity thresholds. This means that the different models or domain interface were not sufficiently similar enough to each other in terms of their contents. However, similar model fragments were found on the lower scopes, *Structure Type and Enumerations* and *LevelAA*.

For the comparison of data models, the following settings were used:

- *Scopes:* Structured Type and Enumerations, LevelAA.
- *Structure:* Unigram. For model elements at the chosen scopes, the structure of these elements do not have as much relevance as the features of the elements contained within these model elements [4].

- *Name Setting:* Named comparison of elements. For data models, the names of the elements are considered relevant for the comparison of model fragments. Names attach meaning to the elements and elements having similar names might be candidates for potential elimination or refactoring.

- *Type Setting:*

  - *Structured Types and Enumerations:* Relaxed type comparison of elements.

    Data models contain elements with *Asome Type* information (relating to entities or value objects within the same model or from a different model; or to primitive types defined in the ASOME language). This type information might again be relevant for comparison, however, the comparison value for non matching types is increased from a 0 provided by the strict type setting to the 0.5 provided by this relaxed type setting. This implies that two elements that do not have matching type information are not immediately regarded as unequal.

  - *LevelAA:* Strict type comparison of elements. While attributes present in entities or value objects may have the same name, they are not exactly the same unless they are also similar in terms of their Asome Types. The detection of attributes that are exactly the same, distributed over various entities or value objects, might signify the need to refactor the models to lift this attribute to a higher level of abstraction [18].

On the given set of data models, using the settings above, the following results were found.

7.1.2. Results and Discussion

This section discusses, per scope, the results obtained through the chosen settings. The discussion is structured as follows: first, the model fragments considered to be clones are discussed; second, the proposition for reducing the level of cloning is presented and finally, the opinion of a domain expert on this proposition is presented.

Scope: Structured Type and Enumerations

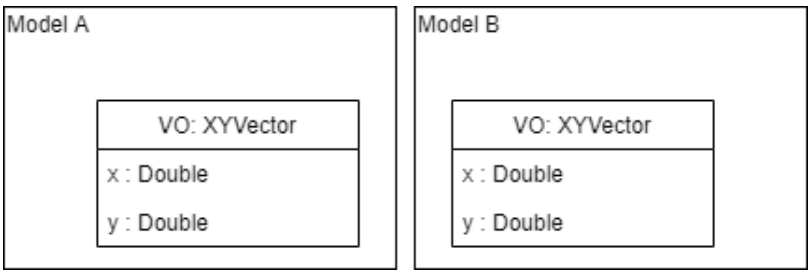

Figure 12: Type A value object clones

Type A Clones. One cluster was found for this category consisting of two Value Objects that were similar terms of their contents as well as their names. These value objects are represented in Figure 12. As can be seen, both the value objects have the same name, XYVector and the same attributes x and y of the same type Double.

Model A was a *core* data model, whose elements were used through imports in several other data models. The duplicated instance of this value object could be eliminated in model B through the reuse of the same value object from the core data model A.
According to the domain expert:

"This is definitely an example where we should look whether commonality should be factored out. Part of the reason why these clones have been detected is explained by the state of the project. The project responsible for delivering the common model, also known as core model, is lagging behind a bit. This has led to duplication of core concepts in other models. These clones or duplicates have been detected by SAMOS.

There might be reasons not to refactor some of these clones. For instance, if one of the clones is expected to develop into a different direction then the other. In that case, also a more different name might be expected though."

Type B Clones. Four clusters were found of this type containing two models in each cluster. Figure 13 is an indication of the types of clusters that were found as Type B clones.

As seen in Figure 13, the elements found in the clusters for Type B clones consisted of model elements with different names but contained attributes and associations with the same names, types and other attributes. The two entities also have partially similar names. They additionally contain the

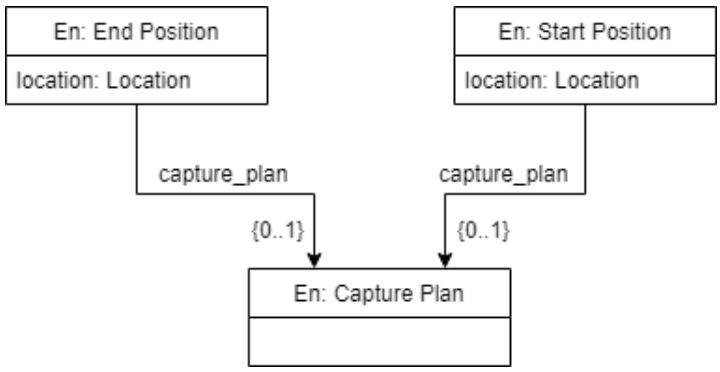

Figure 13: Type B example 1

same attribute with the same Asome Type. Moreover, they both contain an association to a third entity with the same association name and target. The {0..1} at the target of the association represents that the multiplicity of the target entity during instantiation is between 0 and 1.

A technique to eliminate this redundancy in attributes is to introduce inheritance. A parent Entity *Position* containing the attribute location and the association to the *Capture Plan* entity could be introduced with the entities *End* and *Start* inheriting these properties from the parent.
According to the domain expert:

"The elements that are modeled here, result in the generation of repositories. In that, we decided that each repository is a singleton. In the example above, that leads to two repositories holding positions, one holding End_Positions and one holding Start_Positions. If we would migrate that into one repository, then an additional attribute per Position would be needed indicating whether it represents a start or an end position. Given these design constraints, it would be doubtful whether that would be an improvement. So with the current concepts, this proposal would likely not be accepted. Generalizing this example, I dont think that we will see many of these kind of merges. This results from the fact that we decided to treat the generated repositories as singletons."

Type C Clones. Figures 14 and 15 are two examples of some of the 23 Type C clone clusters that were detected. Figure 14 shows two value objects with partially similar names. While these value objects have some similar attributes with the same Asome Type, they each have an additional attribute

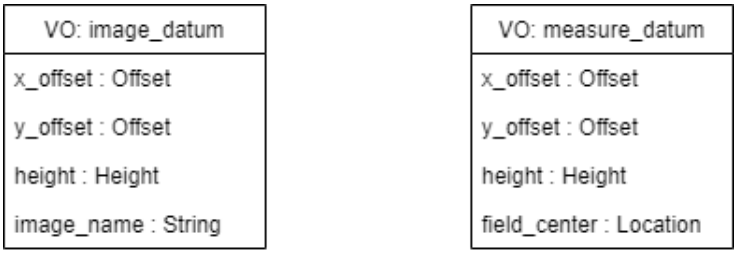

Figure 14: Type C example 1

Figure 15: Type C example 2

that is not present in the other. Figure 15 is also an example of the Type C clones that were detected where one value object contains an attribute that the other does not, while two other attributes are the same.

For both these instances of Type C clones, some elimination of redundancy can be done by creating one value object containing the common attributes and having these models use instances of that value object.
According to the domain expert:

"These kind of discoveries are beneficial and in essence are variants of the type A clones shown in figure 12."

Scope: LevelAA

Type A Clones. Fifty three clusters were found containing a few elements with similar names and the exact same values for all other properties. However, the most interesting result proved to be the association *task*. One cluster containing 9 elements was found, all representing this association named *task*. The target of this association is an entity *Task* which belongs to a core data model (a special type of data model considered crucial for other models). This pattern along with the fact that these associations were all named the same is an indication of consistency and good design, as confirmed by a

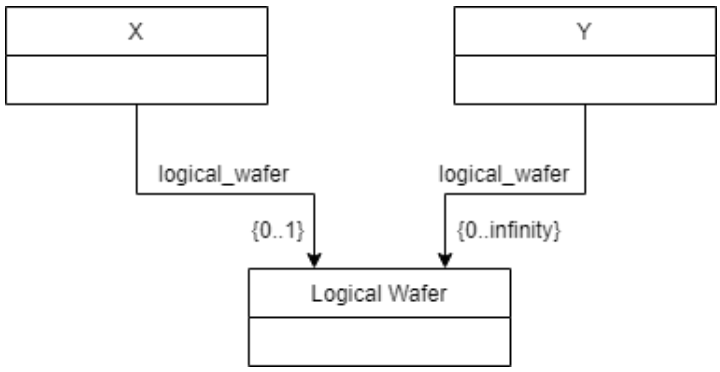

Figure 16: Type B - LevelAA example

domain expert. This shows that not all clones are harmful and in this case, the clones are an indication of good design.

For attributes that are Type A clones, refactoring by lifting these attributes to a common parent entity or a value object that can be reused eliminates these clones. Duplicate associations in some cases cannot be eliminated. Entities from different models associated to an entity, for instance, cannot be refactored to reduce the level of cloning.
According to the domain expert:

"Im not sure that it has been a wise idea to support an inheritance like concept in our data language. There are too many complicated consequences that came with the introduction of inheritance. So the proposal to solve something with inheritance will be treated with a lot of suspicion and care. Im not so sure that we would like to act on the level of individual attributes and/or associations."

Type B Clones. Sixty five clusters were found containing elements that were considered type B clones. An inspection of such elements revealed the differences between the attributes and associations in the cluster, as indicated in Figure 16. For clones of these types, the name and Asome Type of the elements are the same with one property different. In the figure, the names and targets of the association are the same. However, they differ in terms of the upper bound of the multiplicity of the target entity.

Clones of these types, however are not candidates for elimination or refactoring.

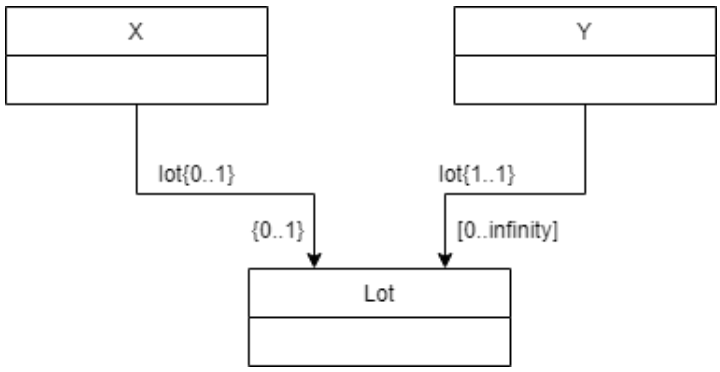

Figure 17: Type C - LevelAA example

Type C Clones. Eighty one type C clone clusters were detected. Figure 17 presents an example of associations that are considered type C clones. The differences between these associations are subtle, given that these are micro clones. There is a substantial change in the number of properties that differ between the two associations. While their names and targets are the same, the number of associations from X and Y to the entity Lot is different. Moreover, the upper bounds of the multiplicities of the target entity are different. The values of *ordered* for these multiplicities are also different, denoted by the difference in the brackets used.

As with the type B clones that were found, these clones are also not candidates for refactoring.

Overall Discussion. Inspecting attributes or associations that were found similar using the available settings, provided an insight into the natural language processing aspect of SAMOS. Attributes with names such as n1 and n2 were considered similar. The comparison of names containing a numerical part ignores this numerical aspect. Moreover, attributes with names that were slightly different were also grouped in the same cluster. Some examples of this are: changed, *un*changed and chang*ing*. In such a situation, lemmatization of words in the Natural Language Processing (NLP) part is a disadvantage. While this NLP does provide a more thorough comparison of names, in the case of *LevelAA* clones containing only attributes and associations, these *LevelAA* elements are grouped together even though their names might signify different properties.

7.2. Clone Detection in ASOME Control Models

This section first discusses the case studies performed on control models as well as the results of these case studies.

7.2.1. Dataset and SAMOS Settings

The approach taken to detect clones within control models is different compared to data models. This is due to the importance of structure, representing behavior, in these models. However, a graph representation of these models would result in an expensive computation for the comparison of models [19]. Therefore, the first step to this approach is to narrow down on the number of elements for comparison using the unigram representation as with data models. On the elements that have been narrowed down, a tree extraction and comparison is performed.

The data set of control models provided contained 691 models, 531 protocols and realizations. A pre-processing step excluded 10 protocols and realizations because the sizes of these protocols and realizations were very large compared to the other models. Excluding these for the comparison was justified considering it was less likely to find similar models to these ones based on their size. Moreover, these models would slow down the comparison significantly while constructing the VSM. The following settings were chosen for the comparison of control models.

- *Scope:* The chosen scope of comparison was the *protocol* scope. On this level of comparison, one can compare models based on their behavior, as defined using the state machines residing in these protocols or realizations.
- *Structure:* For the first round of comparisons, the *unigram* setting was used to find clusters of similar model elements. 50 such clusters of models were found. The second round of comparison was done for models in each cluster. For this round, a *one depth tree* structure was used to compare models using structural information.
- *Name Setting:* A *no name* setting was used for the two rounds of comparison of control models. This was done so we could find models that were structurally equivalent disregarding names.

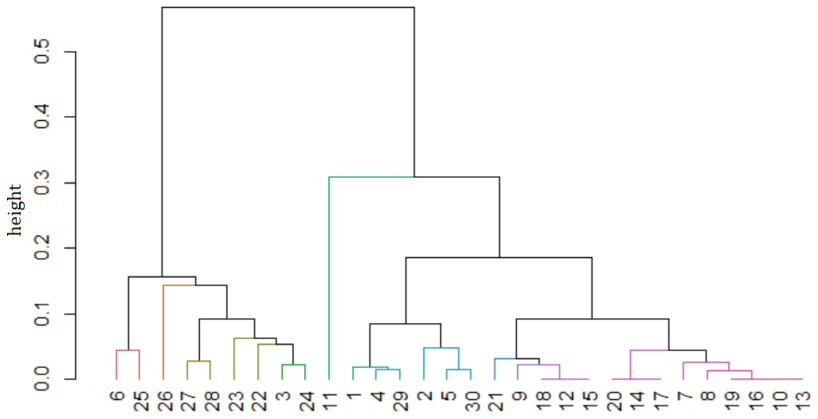

Figure 18: Hierarchical clustering - Cluster 1

- *Type Setting:* A *strict type* setting was used for both rounds of comparisons for control models. This setting was chosen because for control models, type information included was the type of model element being compared, as opposed to the *Asome Type* for data models. In a no name comparison, to find structurally similar models, this setting allows one to detect models as similar based on the number of similar model elements in the model.

7.2.2. Results and Discussion

This section discusses the types of clones found in the set of provided control models.

Example Cluster 1:

Figure 18 represents the hierarchical clustering of elements contained in the cluster after the initial round. This hierarchical clustering of elements is used to find models that can be considered clones.

The models inspected in this cluster were quite large. These models contained a single state with a variation in the number and type of transition states. This behavior is similar to that of an *accepting state*. A combination of patterns found in the models is shown in Figure 20. The state X contains a number of transition states. The patterns of the different types of transition states found in the models are represented by TS1 through TS6.

Figure 19 is an example of a visualization of a few of the transition states in the single state models found in this cluster. The figure shows a single state

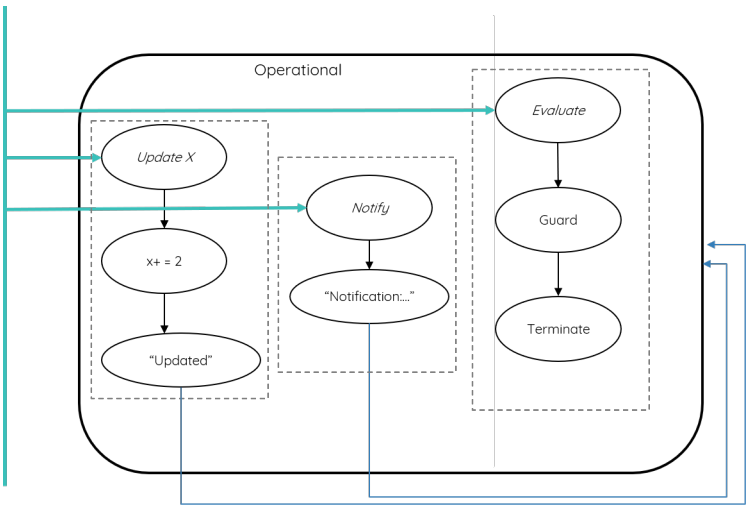

Figure 19: Example visualization of some transition states in cluster 1

Operational which defines behavior using three transition states. A trigger exists for each transition state. The triggers here are *Update X*, *Notify* and *Evaluate*. Depending on the trigger that has been received, the corresponding transition state is executed. For example, the *Update X* trigger is followed by the action of a State Variable Update where the variable x is updated. Following this, the value "Updated" is sent as a reply. Once the reply is sent, the transition state specifies the same state *Operational* as a target state.

The discussion of the different types of clones that were found can now be done based on the number of occurrences of each type of transition state in the models.

Type A Clones. The elements shown in Figure 18 that are represented at the same height and are part of the same hierarchical cluster can be considered type A clones. Examples of such clones are models 20, 14 and 17, and models 18, 12 and 15. An inspection of a collection of models of this type showed that these models had a single state X with multiple transition states. The behavior of the transition states is as shown in figure 20. An inspection of models 19, 16, 10 and 13 showed that they each had 18 occurrences of the pattern TS1; 1 occurrence each of patterns TS2, TS3 and TS4; and 8 occurrences of the pattern TS5. TS6 however, did not occur in these models.

Figure 20: Representation of combined behavior of cluster 1 models

Type B Clones. Type B clones are models with a small percentage of difference between them. The hierarchical clustering shown in figure 18 was used to identify elements that are not exactly the same but could be considered similar to each other. Model 8 for instance, is not exactly the same as, but is similar to the cluster of models 19, 16, 10 and 13. To examine how these models are different from each other, models 8 and 10 were investigated. As with the discussion of type A clones, the behavior of the models is discussed in terms of the number of occurrences of transition states TS1 through TS6 in these models, as shown in Figure 20. Models 8 and 10 were found to have the same number of occurrences of every transition state except TS6. They each had 18 occurrences of TS1, 1 each of TS2, TS3 and TS4 and 8 occurrences of TS5. However, while model 10 had no occurrences of TS6, model 8 had 2. It is because of these two additional transition states in model 8 that these models could be considered type B clones.

Type C Clones. Type C clones are models with a substantial percentage of change between the model elements.The clustering shown in figure 18 is used to identify models that could be considered as type C clones. Models 27 and 24 were chosen as candidates as they can be considered sufficiently different

Transition State	Model 24	Model 27
TS1	5	6
TS2	1	1
TS3	1	1
TS4	1	2
TS5	7	4
TS6	2	0

Figure 21: Number of occurrences of transition states in models 24 and 27

enough from each other through the clustering. Figure 20 can be used to judge how different these models are from each other based on the number of occurrences of the six transition state patterns. Figure 21 shows the number of times each transition state pattern was found in the models.

As seen in the figure, the number of occurrences are slightly different for four out of six transition state patterns. Therefore, these models are instances of type C clones.

The overall discussion in this section discusses the implications of these findings in terms of how the level of cloning can be reduced.

Example Cluster 2:

Figure 22 represents the hierarchical clustering of the elements in a cluster found using the unigram setting in round one. The three types of clones in this cluster are distinguished below.

Type A Clones. The elements in this cluster excluding models three and four in this cluster can be considered type A clones. These models had the same structure, as shown in figure 23. The models were all protocols, defining state machines with this structure. The action of sending a reply is associated with a control interface defined in the model. in each of these models, it was observed that the value of the reply sent to the control interfaces in all these models was *void*.

Type B Clones. The models excluding model 3 and model 4 could be considered similar to model 4, while not exactly the same, as shown in figure 22.

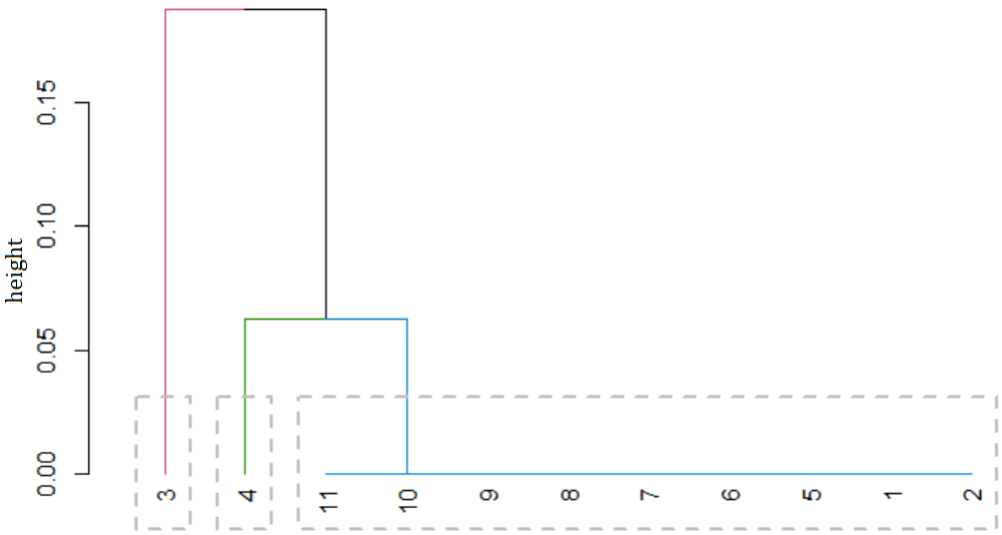

Figure 22: Hierarchical clustering - Cluster 2

Figure 23: Representation of behavior of cluster 2 models

The behavior of these models is depicted in figure 23. Upon investigating these models, it was noted that the difference between the other models and model 4 is in the action *Send Reply*. While the other models sent an empty reply to the control interface, model 4 replied to the control interface with a value. Since this is a small percentage of change between these models, model 4 and the models excluding model 3 can be considered type B clones.

Type C Clones. Model 3, as seen in figure 22 can be considered significantly different from the models in this cluster, excluding model 4. The behavior of these models is depicted in figure 23. The differences between these models is that model 3 was a realization while the other models were protocols. In addition to this, model 3 also sent a value back to the control interface in the *Send Reply* action, like model 4.

Overall Discussion. The example clusters discussed above represent the types of clusters detected after performing a comparison on the extracted one depth trees representing control models on the 50 unigram clusters. Some clusters that were investigated, however, only contained type A clones because all the models found were similar to the other models in that cluster.

While eliminating clones was straightforward for cases in data models, this is not as easy for control models. The presence of duplicates in terms of a sequence of actions might be inevitable if that is the intended behavior of the models. This presents the case for the idea that not all clones can be considered harmful, and some are in fact, intended.

However, many occurrences of some transition state patterns have been found in the models. The transition state pattern TS1 as seen in the example cluster 1 shown in figure 20 was found 18 times each in two inspected models. For such transition states, maybe the metamodel could allow for an easier representation of such a pattern to make it easier for a user to implement this sequence of actions.
According to the domain expert:

"Detecting such patterns of control behavior definitely can be used to investigate whether the user could benefit from a more comfortable syntax. Then an evaluation is needed that needs to take into account:

1. *Whether the new syntax requires more time to learn by the user.*

2. *Whether the simplification really simplifies a lot (see below).*

For instance, in the example above, even for TS1, the user will need to specify the trigger and target state somehow. In case of a non-void reply, also the reply value will need to be specified. So, TS1 cannot be replaced by one simple keyword.It will always need 2 or three additional inputs from the user. In this case, we will not likely simplify this pattern. However, the way of thinking to inspect whether we can support the user with simplifying the language is interesting. It will always be a trade off between introducing more language concepts vs. writing (slightly) bigger models."

Another suggestion for control models is to investigate the unigram clusters to find the different types of patterns found within the control models. Following this, checking what models do not adhere to these patterns might reveal outliers to investigate, to find unexpected behavior.
According to a domain expert:

"I see the line of reasoning and it brings me to the idea of applying machine learning to the collection of models and let the learning algorithm classify the models. Then, investigating the outliers indeed might give some information about models that are erroneous. However, these outliers could also be models describing one single aspect of the system, which would justify the single instance of a pattern. However, I would expect that these models would also have been identified by other, less costly, means (like verification, validation, review etc.)"

8. Discussion

The extension and application of SAMOS on ASOME models resulted in a collection of model fragments that could be considered clones. These clones have been classified based on different categories. For each classification of clones found for the selected scope and the chosen settings, a proposition for the elimination of these has been provided.

For data models, the propositions included modifying the models themselves in terms of design. Some of these propositions have been validated by a domain expert, favoring the elimination of the amount of cloning in some model fragments. However, as mentioned by the domain expert, not all the discovered clones can be candidates for elimination. Some clones are in fact are intended to be so. An example of this are the model fragments shown in Figure 13. The domain expert clarifies that this design is a result of the

decision to maintain singleton repositories and the elimination of these clones would hinder developers from achieving that goal. An instance where clones were actually an indication of consistent design is that of the association *task* found at the *LevelAA* scope. All these associations pointed to the same entity *Task* defined in a core data model. By targeting all these duplicated associations to the same entity found in a core data model, the need to define the concept *Task* is eliminated from each model. In conclusion, while some of the detected clone instances can be eliminated by modifying the design of the model and increasing the amount of reuse, not all clones are harmful and some are even instances of good design.

For control models, modifying the models themselves would not be a desirable way to eliminate clones. This is because the instances of clones detected were occurrences of the same pattern of behavior. However, a change introduced to the metamodel could provide a developer with some *syntactic sugar*. This would allow a user to specify the same behavior with reduced effort. Yet the ease with which (in terms of time and effort) a user can learn such syntax needs to be considered. Moreover, the simplification of some patterns might not add much value, as mentioned by the domain expert for the example pattern TS1 in figure 20. Generalizing this pattern as a language concept would eliminate some clone instances but would still require additional input from users. From the given set of control models, a number of clusters containing models with similar behavior was found. An analysis of such patterns could be done using machine learning techniques to classify models as having expected or unexpected behavior.

In conclusion, for control models, the level of cloning cannot be eliminated by modifying the models themselves but by adding language concepts to the metamodel. In some cases these concepts might not be desirable because they might not add much value. However, in some cases, adding such concepts could ease the effort needed by a developer to create control models.

For the clones that have been found, certain assumptions or simplifications have been made which might result in a threat to the validity of the approach. These threats to validity are discussed in the following section.

8.1. Threats to validity

Thanks to our extension in this work, SAMOS is adapted for detecting clones in ASOME data and control models. The comparison of data models has been done ignoring the structure of the models on the scopes of *Structure Type and Enumerations* and *LevelAA*. While it can be argued that for these

scopes, the structure of the elements is less important, case studies on the level of *Model* and *Domain Interface* could focus on the structure of such models and domain interfaces. Here, comparisons could be made to find similar patterns in the models, perhaps using a pattern catalogue.

The detection of clones in control models has been done on the *Protocol* scope using, initially, the unigram setting (comparing model elements without including structural information), followed by a comparison including the structural information through one depth trees. The use of one depth trees allowed us to reduce the computational time for comparison while still including structural information of the models. The disadvantage of this approach, however, is that there is a loss of context while using this approach. The lower level elements of the models, the actions for instance, contained in the transition states which are in turn contained in the states, are compared independently of the state they belong to. This problem is slightly solved by appending most of the information required within the action node, avoiding for instance sub-trees representing expressions used in guards. While this solution improves the accuracy of the comparison, increasing the depth of comparison would further improve the accuracy of the results. Moreover, a slight change in the metamodel that increases the depth of the lower level elements by adding a level of containment, would reduce the accuracy of the results. There is a trade-off between the accuracy of the comparison and the running time for this comparison, the most accurate with the most expensive running time for comparison being a full graph one comparing state machines.

SAMOS offers the functionality to compare trees either including or leaving out the order of the nodes of the tree. For data models, since tree comparisons were not performed, this was not a problem. For control models, however, currently, an unordered comparison of tree nodes is performed. While the order of the states in the protocol or realization and the transition states within these states is irrelevant for the comparison, the order of the actions within these transition states might be relevant. Therefore, a combination of comparisons where the order of some elements is considered while the order of other elements is ignored, would improve the accuracy of the results obtained.

The comparison of elements for control models using the *No Name* name setting is similar to the *blind renaming* approach taken in [2]. In such an approach, the identifiers of all the model elements are blindly renamed to the same name, therefore ignoring the relevance of names for the comparison.

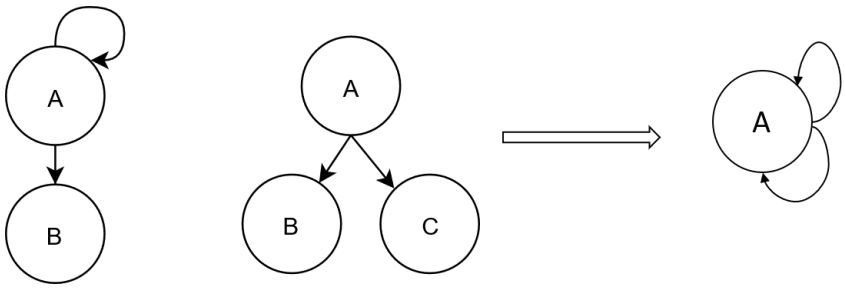

Figure 24: Counter example for blind renaming, where SAMOS (erroneously) cannot distinguish between the two cases.

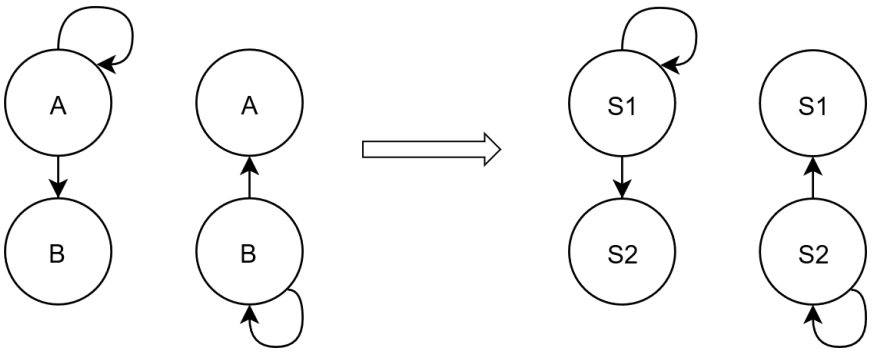

Figure 25: Counter example where consistent renaming would be inaccurate.

This approach allows us to find model elements that have similar structure but different values for elements such as guards or triggers or target state specification. While this improves the recall of the results found, the behavior of the two states as shown in figure 24 cannot be distinguished. The behavior of the model where state A has two transition states that specify state A and state B as the target state is similar to one specifying states B and C as target states. This is because while the structure is mainly captured by the extracted trees, some structural value is also attached to the names of elements, especially target state specifications. While consistent renaming of model elements might solve this problem, this approach was not taken because the order in which these states are renamed could result in inaccurate results of comparison. Figure 25 represents a case against the consistent renaming of model elements. If state A is defined first, followed by state B, even though state B in the figure on the right has similar behavior to state A in the figure on the left, consistent renaming would miss this.

9. Related Work

The following section discusses existing literature in the area of model pattern or clone detection and techniques used previously to find clones in models.

9.1. Model Pattern Detection

Model pattern detection is a prominent research area, related to the tasks we are interested in for our research. However, the word *pattern* has been mostly considered synonymous to *design* patterns or *anti* - patterns in the literature [20]. One approach uses pattern detection as a means to comprehend the existing design of a system to further improve this design [21]. This approach involves a representation of the system at hand, as well as the design pattern to be detected, in terms of graphs. Ultimately, the similarity between the two graphs is computed using a graph similarity algorithm. The paper claims to find (design) patterns within the system even when the pattern has been slightly modified. This approach, however, involves building a collection or catalogue of expected patterns as graphs. Since there were no expectations of the kind of patterns that needed to be detected in our case, we focused on finding e.g. model clones in an unsupervised manner as discussed in the section below.

9.2. Model Clone Detection

While code clones have been previously explored in abundance and hence can be associated with some standard definition and classifications [22, 23], relatively less work has been done in the field of model clone detection, resulting in the lack of such clear definitions. Model clones have been defined as *"unexpected overlaps and duplicates in models"* [24]. Störrle discusses the notion of model clones in depth, as *a pair of model fragments with a high degree of similarity between each other* [4]. Model fragments are further defined as model elements closed under the containment relationship (the presence of this relationship between elements implies that the child in the relationship cannot exist independently of its parent).

Quite a few approaches advocate representing and analysing models with respect to their underlying graphs, for clone detection purposes. One such approach involves representing Simulink models in the form of a labeled model graph [3]. In such graph-based methods, the task of finding clones in

the models boils down to finding similar sub-graphs within the constructed model graph. To do this, all maximal *clone pairs* are found within the graph (with a specification as to what constitutes a clone pair in their case). The approach of finding these maximal clone pairs is NP-complete and to reduce the running time, [3] the approach is modified to construct a similarity function for two nodes as a measure of their structural similarity. Finally, the detected clone pairs are aggregated using a clustering algorithm to find the resulting *clone classes* in the model. The disadvantage of this approach however, is that *approximate* clones are not captured.

The work presented in [25] compares block based models by assigning weights to relevant attributes for comparison such as names, functions of the block and interfaces. A similarity measure is defined to assign a value for the comparison and this value is stored for every pair of blocks being compared. This approach is taken to find variability in models in the automotive domain. Variations were introduced to a base model to add or remove functionality. By inspecting the similarity values, one could find models similar to a selected base model. SAMOS also uses the idea of computing similarity using a vector space model to represent the occurrence of features in each model.

In a different approach, Chen et al. detect clones in Stateflow models [2]. Stateflow models are an extension to Simulink that provides a modeling environment to represent state machines. This approach aims to detect structural clones in these Stateflow models. The linear representation in which the models are stored, one where the description of substates is not nested within the description of superstates, is converted to a hierarchical representation, in accordance with the logically nested nature of the Stateflow models. To accomplish this representation, each object is"folded" into its parent object to consolidate all referenced elements from the parent object into one self contained unit, and each such unit is compared to every other unit. Additionally, the level of granularity of model elements being compared can be defined. This means that the models can be compared on the level of Stateflow *charts*, representing entire machines, or on the level of *states*, representing the states in all the charts. This text based representation as well as the changing of the granularity are techniques also employed by SAMOS.

Störrle provides a contradictory notion however, that for some UML models, the graph structure may not necessarily be the most important aspect of the models to consider for clone detection [4]. Section 4.2 discusses that for some UML models, most of the information worth considering resides in the nodes as opposed to the links between these nodes. Therefore, the

approach taken in this paper defines the similarity of model fragments as the similarity of the nodes in such model fragments instead of the similarity of the graph structure of these model fragments. To construct this measure of similarity, the approach involves using heuristics based on the names of the elements being compared. Such an approach is justified when considering that *"most elements that matter to modelers are named"* [4]. This approach works for models where structure does not represent much in terms of model behavior. However, when the behavior of the models is represented in terms of structure, this approach cannot be used.

10. Conclusion and Future Work

The sections above present an overview of the tool SAMOS, along with the extensions developed to detect clones within ASOME data and control models. The relevant attributes for comparison of model elements was extracted, followed by a comparison of the extracted features. The case studies discuss the types of clones found within the various data and control models. While some clones could be eliminated using refactoring techniques, not all clones are harmful and are in some cases, unavoidable. The threats to validity of the approach have also been discussed.

The settings available in SAMOS allow for many kinds of comparisons of models. While not all of them have been pursued for the case studies in this work, future work could explore results of the clone detection process with the now easily configurable settings.

One such instance of this could be the detection of patterns (design or anti patterns) in data models. While the case studies did not focus on this aspect of data models, structural information of these models on the *Model* and *Domain Interface* scopes could be explored to find patterns within or among these models. As with the approach taken in [21], a catalogue of design or anti patterns could be created to find such patterns within data models.

Comparing the different elements of data and control models in the case studies conducted did not place an importance on the types of model elements considered similar. For example, a domain expert might find it more important in a data model if two entities were similar to each other than if an entity was similar to a value object. Such a situation would call for a weighted comparison of model elements placing different weights for different elements being compared.

While case studies were conducted for control models that successfully detected clones within models, future work in this area has much potential. The comparison of models using structure was done in terms of one depth trees for the conducted case studies. This approach was chosen because tree comparison for comparing full trees with the Protocol or Realization at the node, was expensive in terms of computational time. The validation of the results obtained using this approach was positive, however, the accuracy of the comparison improves with the depth of the trees being compared because the depth introduces more structural information. Therefore, future work could involve altering the depth of the trees being compared to see how the accuracy of the results is affected.

SAMOS has the functionality available for tree comparison that either takes into account the order of the nodes being compared or compares trees irrespective of the order of nodes. For the comparison of control models, the order of the states or the transition states of the models did not affect the similarity of the models. However, the order in which the action occur in each transition state does affect the behavior of the models. While currently an unordered comparison is being performed, future work could use a combination of ordered comparisons for actions in a transition state and unordered comparisons for other model elements to further improve the accuracy of the results.

SAMOS currently has the ability to extract features from four scopes for data models and two scopes for control models. The comparison of model elements for control models using structure might be less expensive while comparing model elements at lower scopes, such as state or transition state. Observing clones at the level of transition states might provide an insight into what sequences of actions occur frequently and for these sequences, the metamodel could include a more compact way to specify them. Moreover, for the types of patterns found in the control model clusters after the unigram comparison, one could create a pattern catalogue and find what models do not adhere to these patterns. This could be an indication of unexpected behavior in the models.

References

[1] C. J. Kapser, M. W. Godfrey, Supporting the analysis of clones in software systems, Journal of Software Maintenance and Evolution: Research and Practice 18 (2006) 61–82.

[2] J. Chen, T. Dean, M. Alalfi, Clone detection in matlab stateflow models, Software Qual J 24 (2016).

[3] F. Deissenboeck, B. Hummel, E. Jrgens, B. Schtz, S. Wagner, J. Girard, S. Teuchert, Clone detection in automotive model-based development, in: 2008 ACM/IEEE 30th International Conference on Software Engineering, pp. 603–612.

[4] H. Störrle, Towards clone detection in uml domain models, in: Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, ECSA '10, ACM, New York, NY, USA, 2010, pp. 285–293.

[5] Ö. Babur, L. Cleophas, Using n-grams for the automated clustering of structural models, in: B. Steffen, C. Baier, M. van den Brand, J. Eder, M. Hinchey, T. Margaria (Eds.), SOFSEM 2017: Theory and Practice of Computer Science, Springer International Publishing, Cham, 2017, pp. 510–524.

[6] R. Schiffelers, Empowering high tech systems engineering using mdse ecosystems, 2017. ICMT Keynote, STAF 2017 [Accessed: 29-11-2018].

[7] W. Alberts, ASML's MDE Going Sirius, `https://www.slideshare.net/Obeo_corp/siriuscon2016-asmls-mde-going-sirius`, 2016. Accessed: 2018-11-12.

[8] Ö. Babur, Clone detection for ecore metamodels using n-grams, in: The 6th International Conference on Model-Driven Engineering and Software Development, pp. 411–419.

[9] Ö. Babur, L. Cleophas, M. van den Brand, Model analytics for feature models: case studies for S.P.L.O.T. repository, in: Proc. of MODELS 2018 Workshops, co-located with ACM/IEEE 21st Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018., pp. 787–792.

[10] J. N. Singh, S. K. Dwivedi, Analysis of vector space model in information retrieval, in: Proc. of IJCA National Conference on Communication Technologies & its Impact on Next Generation Computing, volume 2, pp. 14–18.

[11] R. Schiffelers, Y. Luo, J. Mengerink, M. van den Brand, Towards automated analysis of model-driven artifacts in industry, in: 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, SCITEPRESS-Science and Technology Publications, Lda., pp. 743–751.

[12] R. R. H. Schiffelers, W. Alberts, J. P. M. Voeten, Model-based specification, analysis and synthesis of servo controllers for lithoscanners, in: Proceedings of the 6th International Workshop on Multi-Paradigm Modeling, MPM '12, ACM, New York, NY, USA, 2012, pp. 55–60.

[13] S. Adyanthaya, Robust multiprocessor scheduling of industrial-scale mechatronic control systems, Ph.D. thesis, Eindhoven: Technische Universiteit Eindhoven, 2016.

[14] B. van der Sanden, M. Reniers, M. Geilen, T. Basten, J. Jacobs, J. Voeten, R. Schiffelers, Modular model-based supervisory controller design for wafer logistics in lithography machines, in: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 416–425.

[15] L. van der Sanden, Performance analysis and optimization of supervisory controllers, Ph.D. thesis, Department of Electrical Engineering, 2018. Proefschrift.

[16] J. Nogueira Bastos, L. van der Sanden, O. Donk, J. Voeten, S. Stuijk, R. Schiffelers, H. Corporaal, Identifying bottlenecks in manufacturing systems using stochastic criticality analysis, in: FDL 2017 - Proceedings of the 2017 Forum on Specification and Design Languages, volume 2017-September, IEEE Computer Society, United States, 2018, pp. 1–8.

[17] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, A. Stevenson, Models are code too: Near-miss clone detection for simulink models, in: 28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012, pp. 295–304.

[18] M. Mondai, C. K. Roy, K. A. Schneider, Micro-clones in evolving software, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp. 50–60.

[19] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, B. Schaetz, Model clone detection in practice, in: Proceedings of the 4th International Workshop on Software Clones, IWSC '10, ACM, New York, NY, USA, 2010, pp. 57–64.

[20] M. Stephan, J. R. Cordy, Identifying instances of model design patterns and antipatterns using model clone detection, in: Proceedings of the Seventh International Workshop on Modeling in Software Engineering, MiSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 48–53.

[21] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis, Design pattern detection using similarity scoring, IEEE Transactions on Software Engineering 32 (2006) 896–909.

[22] C. K. Roy, J. R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Science of Computer Programming 74 (2009) 470 – 495.

[23] R. Koschke, Survey of research on software clones, in: Dagstuhl Seminar Proceedings, Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[24] D. Rattan, R. Bhatia, M. Singh, Model clone detection based on tree comparison, in: 2012 Annual IEEE India Conference (INDICON), pp. 1041–1046.

[25] S. Holthusen, D. Wille, C. Legat, S. Beddig, I. Schaefer, B. Vogel-Heuser, Family model mining for function block diagrams in automation software, in: Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2, SPLC '14, ACM, New York, NY, USA, 2014, pp. 36–43.