Overview of slicing and feedback techniques for efficient verification of UML/OCL class diagrams

Shaikh, Asadullah; Wiil, Uffe Kock

Published in:
IEEE Access

DOI:
10.1109/ACCESS.2018.2797695

Publication date:
2018

Document version
Publisher's PDF, also known as Version of record

Document license
Unspecified

Citation for published version (APA):
Overview of Slicing and Feedback Techniques for Efficient Verification of UML/OCL Class Diagrams

ASADULLAH SHAIKH\textsuperscript{1,2} AND UFFE KOCK WIIL\textsuperscript{1}
\textsuperscript{1}The Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Odense 5230, Denmark
\textsuperscript{2}Department of Information Systems, College of Computer Science and Information Systems, Najran University, Najran 61441, Saudi Arabia
Corresponding author: Asadullah Shaikh (asshaikh@nu.edu.sa)

ABSTRACT Model-driven engineering is a methodology for software development that focuses on the use of models in the software development process. These models can be transformed into code, saving developers both time and effort. One of the most widely used models for transformation is the unified modeling language (UML) class diagram, along with its object constraint language (OCL) constraints. Before transforming UML/OCL models into code, it is essential to find defects in the model, as model transformations and code generation may spread errors to other notations where they are more difficult and time-consuming to trace and detect. Formal verification of models is a time-consuming process and there are several formal verification tools that can check the accuracy of UML/OCL models, but their high computational complexity limits their scalability. In this paper, we present an overview of disjoint and non-disjoint slicing techniques that can break UML/OCL class diagrams into independent submodels in order to reduce the complexity. These submodels can then be verified separately through any verification tool or engine. Furthermore, an overview of a novel feedback technique is also proposed, which highlights any unsatisfiable submodels with their integrity constraints from the complex hierarchy of a UML/OCL class diagram.

INDEX TERMS Model verification, efficient verification, slicing technique, disjoint slicing technique, and non-disjoint slicing technique.

I. INTRODUCTION

Model-driven engineering (MDE) is a well-known software development methodology that emphasises the use of models in the software development process. In the later stages of software development, these models are transformed into code in order to accelerate the process and save the time and effort of the software engineers [18].

During the past decade, several software development methodologies have been developed to make effective and useful software systems that can be used in business, industry and academia. The development of software systems is purely based on the requirements of the customer. Previously, these requirements were collected in textual formats and software developers were not fully aware whether these requirements were achievable until or unless they analysed them manually, i.e. by developing a business model. In order to cope with these problems, modelling techniques were introduced such as the business motivation model and the business process model, which can model the business from a strategic point of view with respect to its day-to-day activity. These methods help us to understand the current operations of the business and then develop a model of the business because the notations are clear and explicit. Unified Modeling Language (UML) was introduced to analyse the modules of requirement and object-oriented analysis with the help of use cases and actors. Furthermore, the sequence diagram was introduced to provide a better understanding of the functional requirements of software systems [11].

These software models can be transformed from high-level language to low-level machine instructions using compilers. However, defining the level of abstraction for transformation is a challenging task. Model driven development (MDD) is a software development methodology that focuses on the use of the models instead of textual descriptions. The primary objective of MDD is to program at a higher level of abstraction, which allows the software developer to specify less and create more. The use of models at a higher level adds greater flexibility; for example, it is much easier to make a change at a graphical level than at the source code level.
cost-effective solutions for the software development life cycle process. For example, using models instead of code in the software development process saves the company significant resources and also reduces the risk of failure, since the requirements can be checked before the development starts. Other advantages of MDD include the fact that it simplifies development, fewer skills are required to work on large applications, and it is open source\(^1\) [28].

One major challenge with MDD is that if the programmers are used to coding everything manually, then MDD can be a little rigid because of its major advantage in allowing programming at a higher level of abstraction. Another challenge with MDD is version control. While programming using MDD techniques, it is important to maintain versions of graphical models since the previously developed models may not be accessed with the newer products and vice versa [17].

Finding model defects in the paradigm of MDD at earlier stages before transforming the model into code always represents a significant concern. It is more difficult to find model defects at later stages, such as after transformation. One of the widely used models for transformation is the UML class diagram, along with its Object Constraint Language (OCL) constraints.

For satisfiability, two kinds of different satisfiability are considered for verification: strong and weak satisfiability [2], [9], [32], [33]. A class diagram can be considered as weakly satisfiable if it is possible to generate a legal instance which is non-empty. However, strong satisfiability provides a more restrictive condition as the legal instance has at least one object from each class and a link from each association.

In terms of satisfiability, formal verification techniques can check the correctness of the UML/OCL class diagram, but their high computational complexity limits their scalability. This means that all of the tools/formal verification methods that support the verification of UML/OCL class diagrams have this drawback. Another problem with these tools is that the correctness of the UML/OCL class diagrams is formalism dependent. Therefore, only if the formalism is sufficiently efficient can the property be verified.

The execution time for the verification of UML/OCL class diagrams for each formalism depends on the number of classes, instances, associations, and OCL constraints, i.e., the size of the model. In the case of complex class diagrams, the current tools/formalisms available in the literature cannot verify the models within a given time period (we set the time to 7200 seconds).

To resolve the issue of the verification of large/complex UML/OCL class diagrams, we propose a couple of model slicing approaches called UML/OCL Slicing Technique (UOST) (1) slicing of disjoint set of submodels (2) slicing of non-disjoint set of submodels. The slicing procedure breaks a model \(p\) into independent partitions \(p_1, p_2, p_3, \ldots p_n\) submodels, where \(p\) is satisfiable if all \(p_1, p_2, p_3, \ldots p_n\) submodels are satisfiable.

The disjoint slicing procedure is very conservative in several ways; for example, it only considers disjoint slices. Sometimes it is not possible to slice a UML/OCL model with disjoint slicing procedure due to the navigation of OCL constraints since the model could be fully constrained, i.e., a common class appears in each constraint. In this situation, the sliced model will be the same as the input model. Therefore, disjoint slicing will not provide any benefit to the verification process.

However, the non-disjoint slicing procedure is more aggressive in that it can still slice the model and preserve the satisfiability in the case of one or more common classes appearing in several OCL constraints. As an example, consider a UML/OCL model \(M = \{cl_1, cl_2, cl_3\}\) that restricts the set of OCL constraints \(C = \{c_1, c_2\}\) where \(cl\) represents the class and \(c\) are the OCL constraints in the model. Constraint \(c_1\) restricts the classes \(cl_1\) and \(cl_2\), and \(c_2\) restricts the classes \(cl_2\) and \(cl_3\). We will receive only one submodel \(s_1 = \{cl_1, cl_2, cl_3\}\), if we apply disjoint slicing since the same class appears in both constraints. However, two submodels \(s_1 = \{cl_1, cl_2\}\) and \(s_2 = \{cl_2, cl_3\}\) will be received in non-disjoint slicing. The developers can select either disjoint or non-disjoint slicing depending on the size and the nature of the model. We have provided detailed examples of disjoint slicing and non-disjoint slicing in Section VIII.

Furthermore, we propose a feedback technique for unsatisfiable UML/OCL class diagrams. The technique uses the slicing algorithm that identifies unsatisfiable submodels with their integrity constraints from the complex hierarchy of the complete UML/OCL class diagram. Therefore, developers need only to concentrate on the correction of particular offending part(s) rather than the entire model. This approach saves the developers both time and effort.

**Hypotheses:** The following hypotheses can be derived from the above discussion:

- H1. The verification time can be reduced by applying the slicing technique.
- H2. Slicing will enable the verification of more complex UML/OCL class diagrams.
- H3. Implementation of the slicing technique will be possible on existing verification tools without considering their formalism.

**II. RESEARCH METHODOLOGY**

We have used qualitative and design research methods. A questionnaire-based survey containing 13 questions was also conducted, being shared with 80 researchers and software designers who are domain experts. We received responses from 46 participants. The primary purpose of the survey was to confirm the research direction by identifying the use and the problems with the scalability of complex UML/OCL models in academia and industry. Several results have been found. Subsection II-C presents the survey report and Appendix 1 provides complete results.

---

\(^1\)For further reading on MDE and MDD refer to [11], [16], and [28].
A. QUANTITATIVE APPROACH

In the quantitative approach, we focused on the methodology, which involves improving the efficiency of the verification process, described as follows:

- Identify related tools in literature.
- Examine the specific verification tools available in the literature.
- Propose benchmarking to compare the efficiency of different tools.
- Analyse the formalisms used by available verification tools.

B. DESIGN RESEARCH

Design Research: Another name for design research is Improvement Research. It is mostly used for problem solving in several domains. For our problem, we need to solve the scalability issue for the verification process. In this paper, the steps of design research are followed as shown in Figure 1.

i/ Knowledge Flows: In our domain, the knowledge flow is the improvement of the efficiency of verification process and the limitations of current verification methods of UML/OCL models so that they scale well in the case of large UML design. Therefore, this innovation is required in the field of model verification.

ii/ Process Steps: The following steps are involved:

- **Awareness of Problem:** After a thorough literature study in the field of formal verification, we identified that there is a problem in the verification and validation of complex UML/OCL models, i.e. the scalability of verification. During the initial research, multiple tools and methods were tested, analysed and examined to explore the problem.

- **Suggestion:** In this step, we give a solution, i.e. slicing complex UML/OCL models into a set of independent submodels. After evaluation of the problem, a question was raised about the improvement of scalability of the verification process, i.e. the property being verified may affect only a part of the model and may not require all of the information available in the model. It may also be possible to identify the subset of the models that holds the property in need of verification. Through the above technique, the validity of the property will not be affected by slicing. If the property is satisfiable within the partition, then it will also be satisfiable in the original model. Additionally, it is also possible to abstract the model in such a way that the properties that are independent can be taken apart from the model, so that the dependent properties will not be affected and can be verified later.

- **Development:** The development of the framework was divided into six work packages (WPs), each of which was assigned a task. Figure 2 shows the connections among WPs. WP 1 represents the identification wherein we examined different verification tools and diagnosed the problem, i.e. exponential worst-case runtime. WP 2 explores the creation of a disjoint slicing technique, while WP 3 is based on the development of a slicing technique using a real-world case study. WP 4 describes the development of a non-disjoint slicing technique by integrating WP 1, WP 2 and WP3. In WP 5, the feedback technique is proposed for an unsatisfiable model, while WP 6 is based on the implementation of the disjoint slicing, non-disjoint slicing, and feedback techniques.

In order to improve the efficiency of verification for complex UML/OCL models, we need to design a framework/technique that can verify complex models more quickly. For this, we have defined several WPs, each of which consists of a task related to the proposed technique:

- **WP1:** To achieve the exact goal we need to examine and use different verification tools which are currently available. The motivation behind the evaluation of these tools is to gain a better understanding of the current methods and formalism used in the verification process. We conducted a short survey to...
TABLE 1. Process steps with outputs.

<table>
<thead>
<tr>
<th>Process Steps</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Awareness of problem and suggestion</td>
<td>Proposal - Problem understanding</td>
</tr>
<tr>
<td>Development</td>
<td>Artifact - Tool and verification method</td>
</tr>
<tr>
<td>Evaluation</td>
<td>Performance measure - Time and memory measures</td>
</tr>
<tr>
<td>Conclusion</td>
<td>Results - Verification of large UML/OCL class diagrams</td>
</tr>
</tbody>
</table>

evaluate the UML/OCL tools in depth and identified their weaknesses and strengths. This detailed survey is based on available problems in current verification tools, i.e., usability issues, UML support, OCL support, etc. Multiple tools were examined and analysed by testing different examples of UML models annotated with OCL constraints. We gave several inputs in order to fully understand the formalism used in each tool. Moreover, several limitations of the tools were also identified in order to track the exact point of the problem. Through these experiments multiple concepts were clarified, such as the verification method, support for UML/OCL by each tool, user interface, installation requirements, usability requirements, etc.

Results: These results are published in [39].

WP2: We classified and compared the different verification and validation tools to gain insight into verification processes and methods. The automatic abstraction of the model is one of the important tasks to be addressed in order to track the efficiency of the model verification technique. In the automatic abstraction of the model, we have developed a technique that abstracts all irrelevant components of the model; we named it the UML/OCL slicing technique (UOST). Automatic partitioning is the primary concern of the developed technique. We slice the model into submodels with respect to OCL constraints.

Results: The results are published in [35].

WP3: After the development of the partitioning technique, the next step was to apply the developed technique to real-world case studies in order to prove that the slicing procedure is neither tool-dependent nor formalism-dependent and can be applied to the verification of real-world problems.

Results: The results are published in [38].

WP4: To integrate WP1, WP2, and WP3 we need to investigate the method or technique to improve the scalability in verification procedure for models which are fully constrained and hence difficult to slice. In this work package, we have developed a technique that can still partition a model even though the model is fully constrained.

Results: The results are published in [40].

WP5: The slice of a model into independent submodels can give constructive feedback in the case of a model being unsatisfiable, and it would be possible to identify the submodel(s) in which the error occurred. Software designers can therefore give their attention to the incorrect submodels.

Results: The results are published in [37].

WP6: In this work package, the slicing and feedback technique was programmed in a tool called UMLtoCSP (UOST). This tool is uploaded as an open source tool over the web [45].

Results: The results were published in [36].

• Evaluation: After development, the next stage is the evaluation of the slicing technique. The quality of a verification tool can be determined through computation of the amount of CPU time and memory that is required to verify a specific instance of the model. In some cases where verification tools are unable to respond because of the high memory and CPU time consumption, evaluation can be measured through:
  – The amount of CPU time taken by verification tools (before and after slicing).
  – Memory required by verification tools (before and after slicing).

We have achieved a drastic acceleration, i.e., 99% and 43% for a real-world case study.

• Conclusion: In this phase we compare the achieved results before and after the slicing is applied to several examples. We manually programmed the slicing technique in an external tool; the results are shown. The developed slicing techniques for UML/OCL class diagrams are efficient and, therefore, it is possible to verify complex class diagrams which were previously not verifiable. Hence, the scalability process is improved with the implementation of new slicing procedures.

iii/ Output: Output is the result which is moreover concerned with the process steps. Each process step has a different output. Table 1 shows the outputs coherent to knowledge flow and process steps.

The inspiration for work is UMLtoCSP [9], which is a tool for the formal verification of class diagrams. However, the tool is inefficient in terms of scalability. An extensive description of the verification of class diagrams using constraint programming can be found in [10]. We implemented disjoint slicing, non-disjoint, and feedback technique algorithms in UMLtoCSP and renamed the tool UMLtoCSP (UOST): A Tool for the Efficient Verification of UML/OCL Class Diagrams Through Model Slicing. The following sections are comprised of general introductions to UMLtoCSP, UML, and OCL. Section III gives a brief introduction to UMLtoCSP, followed by sections about the languages UML
A description of the disjoint and non-disjoint slicing procedures is discussed in Section VII, while in Section VIII the slicing procedure is applied to an example Production System and the results are computed. Section IX provides an overview of UMLtoCSP (UOST), followed by a discussion of tool usage in Section X. Next, Section XI summarises the contributions, and finally, Section XII presents the conclusions and identifies directions for future work.

C. SURVEY REPORT

We sent questionnaires to 80 researchers and software engineers around the world and received responses from 46 respondents in Norway, Belgium, USA, Sweden, Finland, Canada, Pakistan, India, Bahrain and several other countries. All of the respondents are domain experts currently working in the field of model verification and validation. A few conclusions were drawn and are presented as follows:

- Almost 90% of people within industry and academia use UML for their software design.
- Most people are aware of OCL and have a basic knowledge of the language.
- 56% of respondents are familiar with the UML/OCL model and use this as their development methodology.
- Half of the respondents are using transformation techniques and generate the code from the model.
- Almost half of the respondents have employed transformation of UML/OCL models.
- 27% of respondents found bugs in their transformed UML/OCL model.
- Few respondents had verified the UML/OCL model before transformation, which could be the main reason for finding bugs.
- 23% of researchers and software engineers had attempted to verify complex UML/OCL models.
- Of these, 18% had faced problems in scalability. The problems identified include: (1) OCL issues; (2) high computational time; (3) software issues; (4) formalism issues.
- Almost 70% of respondents are aware of UML/OCL verification and validation (V&V) tools.
- Widely used V&V tools are: (1) Alloy; (2) UMLtoCSP; (3) USE; and (4) HOL-OCL.
- 41% of participants believe that verification is important before model transformation.
- 31% of researchers are aware of scalability problems.

In conclusion, we found that usage of UML/OCL models exists within academia and industry. Also, it has been noted that the main reason for finding bugs is that the developers do not verify their models before transformation and as a result there are several bugs and errors. Very few people are aware of the scalability issues and this is the reason that most of the respondents avoid verification due to its worst case runtime and software and language issues.

III. UMLTOCSP TOOL

UMLtoCSP [44] is a formal verification tool for UML/OCL class diagrams. It accepts a class diagram with OCL constraints and automatically checks the accuracy of the properties. UMLtoCSP can check the correctness of properties such as strong satisfiability or weak satisfiability; or the lack of redundant constraints [9]. The underlying formalism behind UMLtoCSP is Constraint Logic Programming [10] which uses ECLiPSe [42] constraint solver as the verification engine.

Figure 3 shows a typical example of a UML/OCL class diagram (Production System), having five classes: Conveyor, Belt, Machine, Operator, and Piece. If we want to verify the Production System class diagram with UMLtoCSP, we need first to give an input to the tool, which is a class diagram (.xmi file) and OCL constraint (.ocl file). UMLtoCSP loads UML/OCL class diagram into the memory as shown in Figure 4. Secondly, the user selects the property that needs to be verified. Figure 5 shows the options for property selection. Thirdly, UMLtoCSP transforms the model into a Constraint Satisfaction Problem (CSP) in order to find the valid values for constraint verification, as shown in Figure 6. Finally, it generates the object diagram if the class diagram is satisfiable, as shown in Figure 7. In the case of unsatisfiability, the tool generates the output with a message “No satisfying instance can be found within the specified search space [9]” as shown in Figure 8. However, UMLtoCSP is unable to identify the exact failed property, which is a drawback of this tool.

IV. UNIFIED MODELING LANGUAGE

UML is a modelling language that is largely used for software analysis, design, and implementation.

In this paper, we have only considered a specific static model (UML class diagrams with OCL constraints). The class diagram is considered to be the backbone of every object-oriented design. It describes a high level static structure of the software system and is widely used for transformation. The typical structure of a class diagram is based on four sections, i.e. class name, attributes, operations, and the relationships between the classes. It has three major types of relationships: (1) Association; (2) Aggregation; and (3) Composition. On the class level there are further relationships, i.e. generalization and realization. The class diagram also has general relationships, which are dependency and multiplicity. Using Figure 3 as an example class diagram, where Conveyor is the class name (upper part of the box), capacity is one of the attributes of the class Conveyor (middle part of the box), whose data type is integer and getFullCapacity() is the operation of the class (lower part of the box). The class Conveyor and Machine are connected with each other through an association relationship and the name of the relationship between class Conveyor and Machine is output. The multiplicity between class Conveyor and Machine is 0..* (both sides) where the lower bound multiplicity is 0 and upper bound multiplicity is *. An association
FIGURE 4. Class diagram Production System is loaded into memory.

may have a role name which can be placed at the end of a line; for example, the Conveyor class role name is Conveyor. The official documentation of UML version 2.5 [31] contains further details regarding the class diagrams.

V. THE OBJECT CONSTRAINT LANGUAGE

The Object Constraint Language specification was invented by the Object Management Group (OMG) [29] and forms a part of UML standards [30]. It is a formal language used to describe the rules/textual expressions of UML models [31]. The UML class diagrams do not contain all of the necessary aspects of the system; therefore, OCL is widely used to put the rules on UML class diagram. OCL is a language of side-effect free expressions that provides navigation among UML models. Side-effect free expressions mean that the evaluation of the expression cannot change the state of the execution of the prescribed system. The OCL expressions may use attributes, associations and operations from the UML model and can be part of the ‘invariant’. The invariant is a constraint that presents a Boolean OCL expression that can only be true or false. Each single constraint is bound to a specific type, i.e. class, association class, interface in the UML model.

Figure 3 shows an example class diagram having five classes, four attributes, and four associations. The class diagram restricts three invariants, i.e. MinCapacity, ValueOfNelem, and MinBeltLength. We will limit our discussion to a particular invariant, which is:

context Conveyor inv MinCapacity: self.capacity > 0

This OCL constraint uses only one class, which is Conveyor, and accesses the attribute capacity. The nature of the constraint shows that the minimum capacity of the Conveyor should be greater than zero. In this constraint, we are using a context keyword that begins the
context of an invariant. The keyword \texttt{inv} is used to declare the constraint. An optional name after \texttt{inv} can be written; for example, the name of the constraint is \texttt{MinCapacity}. The keyword \texttt{self} refers to the instance of a class specified in context, in this case \texttt{Conveyor}. Some OCL expressions are written with \texttt{forall} and \texttt{allInstances} operations. The operation \texttt{forall} is used to specify the Boolean expression resulting in true or false; however, \texttt{allInstances} gives a set of all instances of a particular class, including all instances of its subclasses.

The official definition of OCL 2.4 [29] contains a more detailed description of the language.

VI. RELATED WORK

There are two criteria in which slicing techniques can be divided: (1) the entity being sliced; and (2) the purpose of the slicing procedure. The entity represents a program, a UML model and ontology, while the purpose represents the process of slicing; for example, synthesis, analysis, optimisation, visualisation and comprehension. Hence, all slicing techniques can be represented in two steps: first, the element of interest that should be added in the slice; second, elements that should be appended into the slice. The concepts of "element" and "element of interest" are entirely determined by what is being sliced and why. Afterwards, the entity is verified and transformed into code through verification and validation tools. In this section, we present existing work on
program slicing, model slicing, architectural slicing, verification and validation of UML models with formal verification tools. Most of the work in this area is done for program slicing, UML model slicing, and architectural slicing. The specific goal for these slicing techniques is to partition big programs or models into several independent submodels to reuse the required part. However, the slicing criteria of the UML/OCL model in terms of satisfiability are hardly found in the literature.

A. PROGRAM SLICING

The concept was first introduced by Weiser [47]. Program slicing works only on code level [43]. With the help of variables, the program slicing technique computes the desired statements while ignoring the rest of the program. The implementation of program slicing includes program analysis, optimisation, verification and comprehension. Another interesting work related to our problem was proposed by Blouin et al. [3], who proposed the Kompren language to model and produce model slicers for any domain-specific modelling languages (DSML), which can be used for several purposes. However, this work is also limited to domain specific language (DSL).

A recent study on program slicing was presented by Mastroeni and Zanardini [27], in which the authors proposed the notion of abstract program slicing. The proposed approach is applied to number and reference values, and depends on abstract dependencies between program elements. Ward and Zedan [46] provide detailed analysis and assessment of the primary attempts to build semantics for slicing. They also proposed operational semantics that correctly tackles slicing for non-terminating and nondeterministic programs.

B. MODEL SLICING AND TRANSFORMATIONS

Recent work on model slicing was proposed by Lano and Rahimi [24]. The technique slices the UML models by considering the properties of a subset of the elements. On the basis of these elements, smaller models are produced which are more effective and comprehensive. In their most recent work [25] Lano and Rahimi also discussed the slicing technique of UML models which is based on the identification of properties and the behaviour of a subset of the elements. Although this work is limited to slicing using model transformation, it reduces the complexity of the model by identifying the elements of interest, which can therefore be used for model transformation. Further work on slicing focuses on domain-specific modelling languages (DSMLs) [1], [19], [23]. Context-free slicing concerns the static and structural characteristics of a UML model. It considers the static and structural aspects of the model and discards unnecessary elements.

Further work on slicing has been proposed for specification languages such as Z specification [48] and Object-Z specification [6], which is based on the concept of data dependence used to compute slicing programs. Slicing for reducing specification concerns the reduction of complex program specification before verification [5]. This approach is used as a slicing technique for analysis in the field of formal methods and temporal logic. It computes the program structure using specification that is less complex than its real space. Pumares et al. [34] invented formal description that uses initial information of a model merges with model slicing and constraint analysis. The technique is also able to create a client domain model, and arrange the constraints according to the server independency.

There is an ongoing need for an intelligent algorithm for feature model slicing. In this regard, Krieter et al. [21] proposed a new algorithm for feature-model slicing dependent on CNF minimisation and logical resolution.

C. ARCHITECTURAL SLICING

The concept of architectural slicing is used to slice complex system architectures to reduce the complexity of the problem. Afterwards, the sliced architectures can be used for higher level specification [14], [20], [49]. The discussed literature is based on slicing only and there are no previous approaches that consider OCL constraints, while our implemented slicing techniques also deal with the verifiability of UML/OCL class diagrams.

D. VERIFICATION AND VALIDATION OF UML MODELS

In terms of verification, Buettner et al. [7] proposed an automatic translation technique for declarative, rule-based ATL transformations which can provide encoding of ATL to OCL in order to identify the properties that can be transformed. The source also discussed the verifiability for OCL metamodels which can be translated to ATL transformations. Kuhlmann and Gogolla [22] presented a transformation of relational logic from the concepts of class diagram and OCL. The source proposed natural transformation using the elements of relational logic of UML and OCL features. This approach provides the benefit of interpreting results from UML and OCL.

E. VERIFICATION AND VALIDATION TOOLS

There are several verification tools that can verify UML models, especially state machines [13], [26], [41]. These tools use different formalism such as the SPIN model checker and Abstract State Machine Model (ASM) checker. However, few tools are available that support the verification of UML class diagrams with respect to OCL constraints [4], [9], [15]. The OCL verification tools use CSP, Alloy and Higher Order Logic (HOL) formalisms and share the common drawback of exponential worst-case runtime. The basic difference between current tools and our tool is that we perform slicing first, which abstracts the irrelevant components of the model by analysing the navigation of OCL constraints and then translating the model into CSP. Therefore, it is easier to find valid instances for CSP even when the model is complex.

Recently, Gogolla and Hilken [12] invented a technique that uses a model instance finder for model validation and
verification. This approach is also able to detect faults in UML and OCL models. The authors have used the USE model validator to show their experiments. Furthermore, the approach can be used from the initial stages of development up to the testing phases.

VII. SLICING TECHNIQUES

A. DISJOINT SLICING TECHNIQUE

The algorithm of disjoint slicing works by slicing the UML/OCL class diagram into a set of disjoint submodels. Each submodel of a UML/OCL class diagram is a subset of an original model and is verified independently. It is important that all submodels should be satisfiable in the case of strong satisfiability. Alternatively, at least one slice should be satisfiable in the case of weak satisfiability.

The basic approach of this entire procedure is to ensure that when we slice the model, the output provided by the verification engine with slicing must be the same as the output given by a verification engine without slicing. In order to ensure the correct output we first check the satisfiability of each slice separately and then merge the results. As each slice consists of much fewer elements than those in the entire model, it is therefore easy to detect whether any slice is unsatisfiable; if so, it follows that the entire model will also be unsatisfiable.

Before we discuss the overview of the slicing technique, it is important to identify the reasons for which a class diagram can be unsatisfiable. There are a few reasons that can make a class diagram unsatisfiable. First, the model may provide inconsistent conditions of objects of a given type. Second, it is possible that there are no valid values found according to the given condition in an OCL expression. Finally, unsatisfiability may occur due to a combination of both factors. Therefore, to remove the potential for unsatisfiability, it is important that any unsatisfiable constraint should be removed from the model, and if there are constraints that can be verified from the same model elements, they should appear together.

The slicing algorithm is based on the following concepts:

B. GRAPH BASED REPRESENTATION

We used graph-based representation that captures the dependencies of the elements within the UML/OCL class diagram. A dependency graph is an undirected graph in which each vertex represents a class of the model. We used a flow graph to define the relationships, whereby vertices of the flow graph represent classes of the class diagram.

C. TRIVIALLY SATISFIABLE CONSTRAINTS

A new concept of trivially satisfiable constraints is introduced that helps to resolve the scalability issue by introducing a few patterns to identify the trivially satisfiable constraints. The constraints that can fit into trivially satisfiable criteria can safely be removed from the problem since they will not affect the satisfiability.

D. CLUSTER OF CONSTRAINTS

The concept of cluster of constraint is used in the slicing procedure. It computes the cluster of constraints that restricts the same model elements in order to avoid the repetition of submodels.

E. SLICING

We compute the partitions/slices for each cluster of constraints and merge identical submodels, if any exist. Each slice is independent and is a subset of the original model.

F. TIGHTLY/LOOSELY COUPLED CLASSES

We determine tightly and loosely coupled classes and add into partitions. Tightly coupled classes have an association with lower bound of \( \geq 1 \). Alternatively, loosely coupled classes represent the association with a lower bound of 0 (e.g. 0..3). Loosely coupled classes can be added into the slice if they restrict any constraint.

G. REMOVE ATTRIBUTES

A UML/OCL class diagram may have several attributes. However, it is not necessary to keep all of the attributes in a slice. Some attributes may restrict the constraints and therefore all unrestricted attributes can be removed from the model since they will not affect the satisfiability. The removal of unrestricted attributes will reduce the complexity of the UML/OCL class diagram.

H. FEEDBACK TECHNIQUE

A feedback technique algorithm is implemented to find unsatisfiable submodels or properties. With the help of the feedback technique, it is possible to detect a particular offending part that makes a model unsatisfiable among the complex hierarchy of the UML/OCL class diagram. The working of the feedback algorithm is quite simple. We apply slicing on the class diagram, which breaks the model into several independent submodels. Afterwards, each submodel is passed to the verification engine separately and the verification engine generates the object diagram. At this stage, if any property is violated, the object diagram will not be generated and the feedback technique will detect that submodel(s), along with its corresponding invariants, and display this as output.

I. NON-DISJOINT SLICING TECHNIQUE

The procedure of non-disjoint slicing is quite similar to disjoint slicing. The major difference between these two slicing techniques is the consideration of non-disjoint slices and disjoint slices. This slicing technique is more aggressive and can still partition the model if the class diagram is fully constrained. However, the slicing algorithm ensures that the verification of independent submodels with its integrity constraints never causes unauthenticated unsatisfiability, i.e. unsatisfiability will only occur if any OCL constraint is unsatisfiable. Non-disjoint slicing is more efficient than disjoint slicing since it computes the slices without clustering of constraints and we have one slice for each constraint.
The disjoint slicing technique will not be able to partition a model if a common class is used by several constraints since in disjoint slicing we only consider disjoint slices. Therefore, in the worst case, the clustering technique may result in the whole UML model and consequently no improvements in the verification time.

The concept of non-disjoint slicing is introduced, whereby independent slices are computed for each constraint. If there are constraints that can be verified from the same model elements, they will be merged together. The slicing algorithm ensures that if any slice is unsatisfiable, then the entire class diagram will be unsatisfiable. The feedback technique for non-disjoint slicing is able to show the exact unsatisfiable constraint(s) rather than unsatisfiable submodel(s).

VIII. AN EXAMPLE OF DISJOINT AND NON-DISJOINT SLICING TECHNIQUE

This section is based on the basic concept of slicing by considering an example UML/OCL class diagram. Figure 9 represents a class diagram (Production System) which is the
same as the example given in Figure 3 except for the addition of new classes that do not affect the property being verified. We will verify this class diagram using both tools (former UMLtoCSP and new UMLtoCSP (UOST)) and show the improvements in the efficiency of the verification process. This class diagram contains 16 classes, 16 links (5 associations and 11 generalisations), 5 attributes, and 6 OCL invariants.

The first step of the slicing procedure is to make a flow graph of the given example. Figure 10 shows the flow graph where in the visual styles of arcs depict different types of dependencies. The loosely and tightly coupled associations are shown in the flow graph. A dashed labelled arc represents an association with a lower bound of 0 multiplicity and a solid labelled arc shows a tightly coupled association with lower bound greater than or equal to 1 multiplicity. Loosely coupled arcs can be removed from the problem since they do not affect satisfiability. However, it would not be possible to remove tightly coupled arcs due to the restriction of the constraints.

The second step is the removal of trivially satisfiable constraints. We have defined a few patterns in order to detect trivially satisfiable patterns. There are two main patterns for trivially satisfiable constraints, i.e. key constraint and derived value constraint. The given condition for the key constraint is that the attribute must be unique and must not be constrained anywhere else. However, in a derived value constraint, the attributes depend on the values of each other. Table 2 shows the details of patterns with their respective conditions.

It is fair to say that in the start, each constraint is trivially satisfiable until or unless the attribute of that particular constraint is not used in any other constraint within the model. Considering our running example Production System, we have six OCL constraints, i.e. MinCapacity, ValueOfNelems, MinBeltLength, BeltSize, ConveyorSize, and UniqueMachineID. Initially, it seems that invariant MinCapacity, MinBeltLength, and UniqueMachineID are the key constraints and ValueOfNelems is a derived value constraint. However, there are two more constraints, BeltSize and ConveyorSize which are textual constraints, and therefore, these textual constraints can neither be key constraints nor derived value constraints. Because of these textual constraints, it is not further possible for MinCapacity and MinBeltLength to be in the pattern of key constraint and ValueOfNelems as a derived value constraint. Hence, UniqueMachineID is the only key constraint that can be abstracted from the class diagram.

The third step is the computation of the cluster of constraints. The concept of constraint support is used to identify the clusters. The constraint support defines the classes restricted by an individual constraint. This is a fully automatic procedure and is performed before slicing the class diagram. Therefore, if there is a constraint that can be verified from the same model elements, it will be removed from the problem and the constraint of that submodel will be merged with its identical submodel. Table 3 shows the details of the constraint support for each invariant. We can identify three clusters in Production System: invariants MinCapacity, ValueOfNelems, and ConveyorSize (support: Conveyor, Piece); invariants MinBeltLength and BeltSize (support: Belt); and invariant UniqueMachineID (support: Machine). The cluster UniqueMachineID (support: Machine) will be removed from the problem due to the detection of a trivially satisfiable pattern.
The fourth step is the slicing of the class diagram with the addition of tightly coupled classes. The slicing procedure is applied over the Production System and we will receive the same number of submodels as the cluster of constraints, i.e., two submodels. Submodel 1 consists of three invariants: MinCapacity, ValueOfNelems, and ConveyorSize (support: Conveyor, Piece). Since the classes Conveyor and Piece have no tightly coupled classes, i.e., association with lower bound multiplicity of $\geq 1$, this submodel will therefore be limited to these classes only. However, submodel 2 consists of two invariants: BeltSize and MinBeltLength (support: Belt). The class Belt is tightly associated with class Conveyor and Piece; therefore, the support of the final submodel 2 will be class Belt, Conveyor, and Piece. Table 4 represents the tightly associated classes with each cluster and Figure 11 shows the final slices passed to the verification tool UMLtoCSP (UOST) for strong satisfiability, while the remaining classes, associations, and attributes can be removed. In non-disjoint slicing, we will get two submodels; however, in the case of disjoint slicing, only one submodel will be formed since there are intersecting classes between Submodels 1 and 2, i.e., Conveyor and Piece. During this entire automatic procedure, the unrestricted/unused attributes will also be removed from the problem. Before we applied the slicing, the example (Production System) had 16 classes, 16 links (5 associations and 11 generalisations), 5 attributes, 6 OCL invariants, and it was not verifiable with UMLtoCSP even within 2 hours’ time. With the help of this slicing procedure, we abstracted 13 classes, 14 links (3 associations and 11 generalisations), 2 attributes and 1 OCL constraint. After slicing, we verified the example (Production System) in 0.422 seconds, which is a marked progress. Therefore, this discussion supports hypothesis 1 (H1).

The final step is to generate feedback while there are any constraints whose interaction is unsatisfiable. Considering our running example, the interaction of each constraint of entire class diagram is satisfiable. However, the addition of the following constraint can make the model unsatisfiable: context Machine inv MachineSize::allInstances() size() = 2

The feedback algorithm will automatically detect the above constraint and alert the user that the interaction of one or more constraint is unsatisfiable. The user can correct that particular incorrect part and verify the class diagram again. The computation of the feedback is very fast and the violated properties can be detected within a few seconds.

A. EXPERIMENTAL RESULTS

Tables 5 and 6 show the experimental results of Production System for non-disjoint slicing and disjoint slicing using an
TABLE 6. Verification time for disjoint slicing.

<table>
<thead>
<tr>
<th>Example</th>
<th>OVT</th>
<th>Slices</th>
<th>Attr</th>
<th>ST</th>
<th>SVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production System</td>
<td>Time-out</td>
<td>1</td>
<td>2</td>
<td>0.00s</td>
<td>0.412s</td>
</tr>
</tbody>
</table>

Intel Core 2 Duo Processor 2.3Ghz with 3Gb of RAM, where OVT is the original verification time by the former tool UMLtoCSP. Slices is the number of submodels in which the model is distributed. Attr is the attributes that we manage to abstract, ST is the time required to perform UML/OCL slicing analysis, and SVT is the total verification time after slicing. The time is measured in seconds and we set a timeout limit of 2 hours. The experimental results clearly indicate that the class diagrams which were previously not even verifiable are now verifiable in a few seconds using the slicing procedure. This discussion supports hypothesis 2 (H2).

Moreover, we have shown several experimental results in [40] achieved in a tool called Alloy by applying the model slicing approach. We showed the results in the Alloy Analyzer to prove that our slicing algorithm/technique is neither tool-dependent nor formalism-dependent. It can be implemented in any verification-driven UML/OCL tool. This supports hypothesis 3 (H3).

The input of UMLtoCSP (UOST) is the same as for the old tool; however, the transformation and the output are different, as they are in the form of slices. An overview of UMLtoCSP (UOST) is given in the following section.

IX. UMLTOCSP (UOST)

UMLtoCSP (UOST) is a formal verification tool for complex UML/OCL class diagrams using a model slicing procedure [36]. The term complex means that the class diagrams which were previously unverifiable are now verifiable using the slicing procedure. The tool has been developed to solve the major problem of scalability. We have programmed two different model slicing approaches: (1) disjoint slicing; and (2) non-disjoint slicing in UMLtoCSP (UOST). Initially, the tool uses the input of a UML/OCL class diagram and breaks the model into a disjoint or non-disjoint set of submodels; the original model is satisfiable if all submodels are satisfiable. The computation of submodels is fully automatic and invisible to users. Secondly, all computed submodels are translated into CSP and finally the constraint solver ECLiPSe checks each submodel to determine whether the solution exists in CSP. In the case of the class diagram being verifiable, the object diagram will be generated with valid objects according to the given conditions in OCL expression. Alternatively, the output will be provided with constructive feedback, which will highlight the unsatisfiable submodel(s) with its corresponding invariants in disjoint slicing, and therefore, the user needs to identify the unsatisfiable property him/herself. However, in the case of non-disjoint slicing, the tool is able to show the specific unsatisfiable invariant due to the initial check for any unsatisfiable invariant. If one is found, we run the slicing twice to compute the exact invariant and therefore the verification time will be double that of disjoint slicing. UMLtoCSP (UOST) has the following unique features:

A. STRONG SATISFIABILITY
if it is possible to generate at least one object from each class and a relationship from each association, then the class diagram is considered to be strongly satisfiable.

B. WEAK SATISFIABILITY
the class diagram is considered to be weakly satisfiable if we can generate at least one object from any class.

C. REMOVE ATTRIBUTES
unrestricted/unused attributes can be discarded from the class diagram.

D. DISJOINT SLICING
the slicing of a class diagram into independent disjoint submodels.

E. NON-DISJOINT SLICING
the slicing of a class diagram into non-disjoint submodels.

F. SHOW SPECIFIC INVARIANTS
The feedback technique is implemented behind this option. This feature of the tool has different output for disjoint and non-disjoint slicing, which is based on two different logics. This option can be selected together with strong satisfiability or weak satisfiability. In the case of disjoint slicing, the unsatisfiable submodel will be shown. However, in non-disjoint slicing the particular failed invariant will be displayed.

X. UMLTOCSP (UOST) USAGE

We have implemented our slicing and feedback techniques in UMLtoCSP (UOST) and in this section we will give a brief overview of tool usage and also show the input/output of the running example, i.e. Production System. The input procedure of UMLtoCSP (UOST) [36] is the same as for the former tool UMLtoCSP [9]; however, the transformations and output are in the form of slices. In the start, the user provides the UML class diagram in xmi format and the OCL constraints file to UMLtoCSP (UOST) as input. The tool will automatically load a UML/OCL class diagram into memory, which is invisible to the user. Figure 12 shows the running example Production System loaded into memory. The next stage is the selection of the property being verified before...
the transformation of the model into CSP. Figure 13 shows the property window in which the user needs to select the slicing criteria. Afterwards, the class diagram will be transformed into CSP and the tool will display the exact number of transformations (ecl files) as submodels. We will receive two submodels in non-disjoint slicing (strong satisfiability) and one submodel in disjoint slicing (strong satisfiability). Figure 14 shows the transformation of submodels for non-disjoint slicing. If the property is verifiable, two submodels will be generated in the case of non-disjoint slicing (Figure 15) and one submodel in the case of disjoint slicing (Figure 16).

The feedback can be generated if there are any unsatisfiable constraints. Figure 17 highlights the failed property, so that it can be corrected without checking each constraint individually.
XI. SUMMARY

In the context of MDD, the requirement for correct specifications is important because the software development process is reliant on model transformations. If the original model is wrong, it may lead to the failure of the entire software system. Unfortunately, the verification of software models is a difficult and time-consuming task, especially when complex UML/OCL models are involved.

At present, there are verification and validation methods that work quite well with small UML models and contain multiple solutions through tools using different methods and formalisms. Therefore, the verification of less complex models is supported by most tools and methods depending on the size and structure of the model. All of the literature and resources hitherto involve verification methods that have problems with scalability for large UML models. And if we try to apply the existing method to larger models, the tool takes a long time to verify the model properties.

Apart from these serious problems with current verification tools, the verification engines heavily depend on their internal formalisms. Therefore, any optimisation in the verification tool will be limited to the tool/formalism itself and other verification engines will not gain the benefit.

We have proposed two slicing techniques (disjoint and non-disjoint) in order to solve the problems with the efficiency of the verification process for complex UML/OCL models. The proposed slicing is free from any formalism and can be implemented in any verification engine that supports the verification of UML/OCL models. The working procedure of slicing is quite simple: first, it takes a UML/OCL model as an input; second, it breaks the model into indecently submodels by analysing the navigation of OCL constraints; third, it verifies each submodel individually; and finally, it generates the object diagram if the model is satisfiable.

The slicing procedure ensures that if the original model is unsatisfiable then some submodel is also unsatisfiable. Two different slicing procedures have been proposed; (1) disjoint slicing and (2) non-disjoint slices. The disjoint slicing technique only considers the disjoint sets of submodels. There is a possibility that the UML class diagram may be fully constrained due to the existence of common class(es) and therefore, it would not be possible for the disjoint slicing procedure to slice the model. In this case, the sliced model will be exactly same as the original and the verification may not have benefited; therefore, the non-disjoint slicing procedure has been proposed to overcome this limitation. The non-disjoint slicing procedure considers non-disjoint slices and is able to break the model even if the model is fully constrained, depending on the navigation of OCL constraints. The only limitation with these slicing procedures is that if the elements of the model are tightly connected with each other then it would not be possible to slice the model. However, the slicing procedure can remove trivially satisfiable constraints and unrestricted attributes, which will reduce the complexity of the model in the worst case scenario so that the verification engine will get some benefit.

Furthermore, we have proposed a feedback technique that can highlight the unsatisfiable part(s) of the class diagram from the complex hierarchy. This technique has the benefit that if the model is complex and unsatisfiable, the feedback technique using the slicing procedure will slice the model and detect the unsatisfiable part with its corresponding OCL constraints. Afterwards, the entire focus can be placed on the correction of the particular unsatisfiable submodel, while ignoring the rest of the complex hierarchy.

XII. CONCLUSION

The problem with scalability is highlighted with examples in which high worst-case computational complexity in verification engines is addressed. We also presented a survey with a benchmark based on analysis of current formal verification tools. A disjoint slicing technique is proposed to reduce the complexity of UML/OCL models so that the complex models can be verified efficiently. With the help of slicing, it has been
proven that complex models, which were previously unverifiable, can be verified using the slicing procedure. Up to 99% increased time efficiency is achieved for the verification process. The slicing procedure has also been applied to real-world case studies where up to 43% acceleration has been achieved. It is further possible that the class diagram cannot be sliced with the disjoint slicing technique since it only considers the disjoint set of submodels. Therefore, a non-disjoint slicing procedure is proposed which can still partition the model, if the model holds disjoint slices. A feedback technique is proposed to help the user if the model is unsatisfiable. It detects unsatisfiable constraints with its corresponding submodel(s) from the complex hierarchy of UML/OCL model which can save developers both time and effort.

There are a few possible directions for future work. The first is to provide more constructive feedback by suggesting possible corrections in OCL invariants. At the moment, our feedback technique is limited to the detection of an unsatisfiable property; however, we do not suggest the changes that could make the unsatisfiable invariant satisfiable. A second direction of future work is the creation of a formalism/verification engine with built-in slicing techniques. With the help of our proposed slicing procedures, the formalism would be efficient to verify complex models. However, the creation of a formalism is quite a challenging task and shares the common drawback that a formalism/verification engine will be limited to a particular tool and cannot be implemented in any other tool or formalism.

**APPENDIX 1**

**A SURVEY: THE USE OF MODEL DRIVEN DEVELOPMENT IN INDUSTRY AND ACADEMIA**

Welcome to this survey on the use of Model Driven Development (MDD) in industry and academia. The purpose of this survey is to access and analyse the use of MDD in current practice. Thank you very much for completing this survey.

**ABOUT YOU**

How old are you? I am ________ years old.

Organization Name: ____________________

Designation (current position): ______________

Email: ____________________

1) Do you use Unified Modeling Language (UML) class diagram as a high level design?
   • Yes
   • No

2) Do you have preliminary knowledge about Object Constraint Language (OCL)?
   • Yes
   • No

3) Have you ever used UML/OCL Model as your development methodology?
   • Yes
   • No

4) Have you ever transformed any model into code?
   • Yes
   • No

5) Have you ever transformed a UML/OCL model into code?
   • Yes
   • No
   • If yes please illustrate the type of transformation ______________

6) Did you find bugs in the code after model transformation?
   • Yes
   • No

7) Did you verify and validate your UML/OCL class diagram before transformation?
   • Yes
   • No

8) Have you verified and validated UML/OCL complex class diagram, i.e. more than 15 classes and OCL constraints?
   • Yes
   • No

9) Have you faced any kind of problems during the verification process of a complex UML/OCL class diagram?
   • Yes
   • No
   • If yes please illustrate the type of transformation ______________

10) Do you know about UML/OCL models verification and validation tools?
    • Yes
    • No

11) Which of the following UML/OCL tools do you use for verification and validation?
    • HOL-OCL
    • MOVA
    • ALLOY
    • UMLtoCSP
    • UML2Alloy
    • USE
    • Other: ______________

12) Do you think it is important to verify the UML/OCL model before transformation?
    • Not at all
    • Less Important
    • Important
    • Very Important

13) Are you aware of the scalability problem in UML/OCL model verification?
    • Yes
    • No
A. SURVEY REPORT

We sent the above questionnaire to 80 researchers and software engineers all around the world and received responses from 46 respondents in Norway, Belgium, USA, Sweden, Finland, Canada, Pakistan, India, Bahrain and several other countries. All of the respondents are domain experts and currently working in the field of model verification and validation. A few conclusions have been drawn, and are as follows:

- Almost 90% people from industry and academia use UML in their design.
- Most of these people are aware of OCL and have a basic knowledge of the language.
- 56% of respondents are familiar with the UML/OCL model and also use it as their development methodology.
- Half of the respondents use transformation techniques and generate the code from the model.
- Almost half of the participants have performed the transformation of a UML/OCL model.
- 27% people found bugs in their transformed UML/OCL model.
A. Shaikh, U. K. Will: Overview of Slicing and Feedback Techniques for Efficient Verification of UML/OCL Class Diagrams

• A few respondents verified the UML/OCL model before transformation and this could be the main reason for finding bugs.
• 23% researchers and software engineers have tried to verify complex UML/OCL models.
• 18% out of 23% faced problems in scalability. A few noticed problems with: (1) OCL issues; (2) high computational time; (3) software issues; (4) formalism issues.

REFERENCES