

Testing UML designs ☆

Orest Pilskalns ^{a,*}, Anneliese Andrews ^b, Andrew Knight ^c, Sudipto Ghosh ^d, Robert France ^d

^a School of Engineering and Computer Science, Washington State University, Vancouver 14204 NE Salmon Creek Avenue, Vancouver Washington, USA

^b Department of Computer Science, University of Denver, 2360 South Gaylord Street, Denver Colorado, USA

^c School of Electrical Engineering and Computer Science, Washington State University, Pullman P.O. Box 642752, Pullman Washington, USA

^d Department of Computer Science, Colorado State University, 601 South Howes Street, Fort Collins CO, USA

Received 28 November 2005; received in revised form 5 September 2006; accepted 8 October 2006

Available online 30 November 2006

Abstract

Early detection and correction of faults in the software design phase can reduce total cost and time to market of a software product. In this paper we describe an approach for testing UML design models to uncover inconsistencies. Our approach uses behavioral views such as Sequence Diagrams to simulate state change in an aggregate model. The aggregate model is the artifact of merging information from behavioral and structural UML views. OCL pre-conditions, post-conditions and invariants are used as a test oracle.

© 2006 Elsevier B.V. All rights reserved.

Keywords: UML design models; Sequence diagrams; UML views; OCL pre-condition and post-condition; Class diagrams

1. Introduction

Current practice in UML design validation uses reviews and inspections [1]. While, in general, conventional review approaches work quite well, they can also be problematic. For example, ad hoc techniques [2,3] rely solely on reviewers' intuition and expertise and thus are very subjective and not necessarily systematic [4]. Checklist-based review approaches [2,3] may not scale well when the number of items to check increases and the designs get larger. Reviewers can get overwhelmed [4]. This is why Traore and Aredo [4] suggest to enhance structured reviews with model-based verification. Their approach requires definition of a formal semantics for part of the UML notation.

In addition to inspection, UML-based CASE tools provide help in identifying errors. Most existing UML-based tools provide simple static analysis capabilities that can check model consistency. However, they do not validate

behavior between views and only syntax check against OCL expressions (Object Constraint Language).

Because of the limitations of reviews and static analysis it makes sense to complement them with testing. Runeson et al. [5] have shown, that for units of code, inspections and unit testing find different kinds of faults. This might be the case for designs as well.

If you want to test UML designs, what needs to be defined? Similar to code testing, one needs the following [6,1]:

1. testing criteria
2. test generation
3. test execution
4. test result validation

Andrews et al. [6] defined testing criteria for UML designs. Our research addresses test generation, execution, and validation. Further, we are interested in testing UML diagrams from both a structural and behavioral perspective. This means testing multiple types of diagrams or views. Analyzing these views separately will not reveal the inconsistencies across the views. Some views focus on structure (such as Class Diagrams), others on behavior

☆ This research was partially supported by NSF CCR-020328.

* Corresponding author. Tel.: +1 360 546 9110; fax: +1 360 546 9438.

E-mail addresses: orest@vancouver.wsu.edu (O. Pilskalns), andrews@cs.du.edu (A. Andrews), aknight@eecs.wsu.edu (A. Knight), ghosh@cs.colostate.edu (S. Ghosh), france@cs.colostate.edu (R. France).

(such as Sequence Diagrams). Both are needed for test execution. Thus, our approach defines an aggregate model that combines both structural and behavioral information and can be used to simulate execution for testing purposes. In addition, we use OCL statements to validate test results and to identify inconsistencies.

The following example illustrates the benefit of integrating views into an aggregate model. Fig. 1 contains partial UML Views showing structure, behavior, and constraints of the open source project Batik [7]. In open source projects, inheritance is often a way of extending functionality without changing the existing code base. However, extensive inheritance chains can lead to complex designs which are often difficult to understand and even more difficult to debug [8], resulting in the need for new analysis approaches. In this example we highlight the *JPEGTranscoder* class to signify that the class was recently added to the inheritance hierarchy. In the Sequence Diagram, the *driver* instance calls *transcode* on the *JPEGTranscoder* instance. The *transcode* method is an inherited method, thus the *JPEGTranscoder* relies on a ancestor class in the class hierarchy. The *TranscoderSupport* class, is linked via

an association to the *TranscoderHints* class and constrained by a 1 to 1 multiplicity. However, the *JPEGTranscoder* class never creates a *TranscoderHints* class (the creation could be in a different sequence diagram, but we are assuming this is not the case). If the driver needs the information in the *TranscoderHints* class, this could lead to a transcoding error in the implemented system. Notice that inspection of the Sequence Diagram alone cannot reveal this defect. The Class Diagram must be consulted to find the inheritance hierarchy and the multiplicity constraint must be tested. As design models become more complex, finding inconsistencies such as this becomes more and more difficult through inspection. Therefore, we need an approach that can be automated to find inconsistencies between behavioral views, structural views, and constraints.

This paper presents an approach to test aggregate design models which contain behavioral and structural information from UML designs. The approach uses Sequence Diagrams to describe behavior. Structural information is described by Class Diagrams. We use information from both views to create an aggregate model. It is used for testing the underlying UML design artifacts.

Section 2 discusses related work and test adequacy criteria. Section 3 describes the aggregate model that integrates information from Sequence Diagrams, Class Diagrams, and OCL Expressions and gives an algorithm that extracts information for UML XMI-based models. We also define how to generate test inputs and how to execute tests. In Section 4 we apply the algorithm to a multi-view example that uses OCL expressions. In addition we conduct a case study that examines if there is a correlation between the design faults uncovered by our approach and implementation faults. Section 5 summarizes the findings and limitations of this technique and points out potential design analyses that can be based on the aggregate model.

2. Background

2.1. Related work

Work on testing using UML artifacts has tended to focus on generating requirements and criteria for code testing from UML diagrams. Binder [1] described generic code test requirements that can be derived from UML models. Offutt and Abdurazik [9] developed a technique for generating test cases for code from a restricted form of UML State Diagrams. Abdurazik and Offutt [10] also developed test criteria based on Collaboration Diagrams for static and dynamic testing of code. Building on this work, they proposed methods for statically checking code relative to a Collaboration Diagram using classifier roles, collaborating pairs, messages or stimuli and local variable definition-usage link pairs.

Labiche and Briand [11] described the TOTEM (Testing Object oriented systems with the Unified Modeling Language) system test methodology. System test requirements

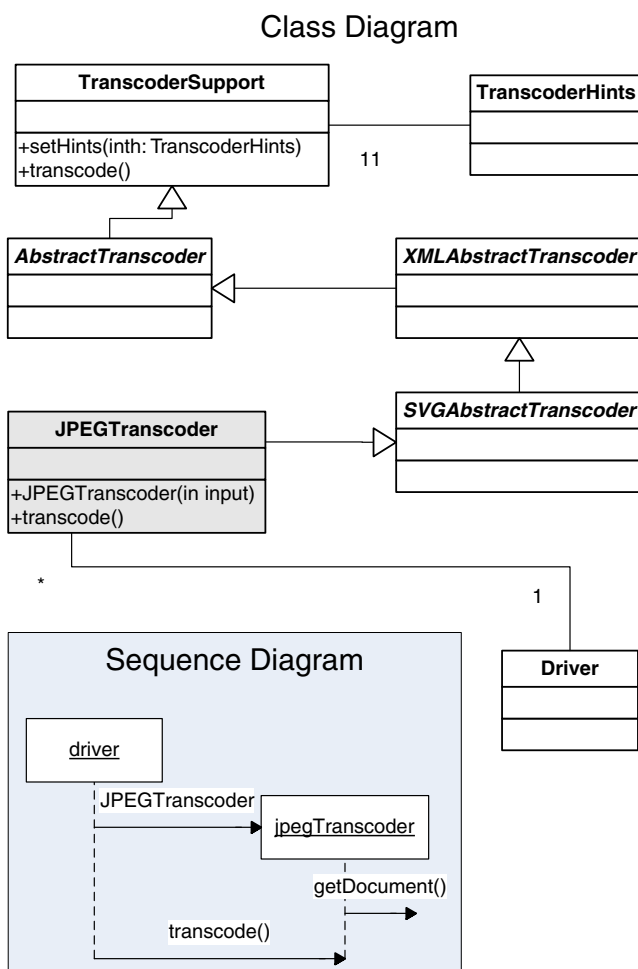


Fig. 1. UML Views of Open Source Project.

are derived from UML analysis artifacts such as use cases, their corresponding Sequence and Collaboration Diagrams, Class Diagrams and the use of OCL across all these artifacts. The test requirements are then transformed into code-level test cases, test oracles and test drivers using more detailed design information. Briand et al. [12] outline a code testing strategy by specifying paths through a flattened statechart. The paths are used to specify test cases and the test case inputs. The test case inputs are generated from an analysis of OCL contract and transition guards.

Scheetz et al. [13] describe an approach to generating system (black box) test cases from UML Class Diagrams. In the approach, a restricted form of UML Class Diagrams is used to represent the conceptual architecture of the system under test. The Class Diagrams used consist only of classes, associations (including aggregation), and specialization structures. From the Class Diagrams test objectives are derived that cover single classes, groups of classes, and their associations as depicted in the Class Diagrams.

Gogolla et al. [14] described an approach of using snapshots of a model to validate OCL artifacts. In the approach, UML Class Diagrams are used to generate object states. These states are then checked against the OCL constraints using their validation tool USE. Their work is concerned with checking for consistency between Class Diagrams and OCL models. Our work differs from [14] because it uses the behavioral information of Sequence Diagrams combined with the structural elements of the Class Diagram, allowing us to validate OCL expressions against combined UML views. The Sequence Diagram information in our approach allows us to construct state information that reflects behavioral scenarios in the design. In addition, our approach employs test adequacy criteria which are critical in assessing test coverage.

Several techniques have been proposed to execute UML models. An operational semantics for the UML is required to execute UML designs with test inputs. Riehle et al. [15] propose a virtual machine that has the UML as its instruction set and memory management facilities of an existing Java Virtual Machine as the memory model. The advantage of using a virtual machine is that UML models can be executed without being transformed into another form. However, there are currently no publicly available virtual machines that cover the UML diagrams we are targeting in our work, and thus we had to investigate other approaches to “executing” design models.

Mellor and Balcer [16] use domain-specific model compilers to produce executable UML models. UML designs may also be executed by executing the code that is generated from them. Executing the code is same as executing the model when both the model and the code contain the same information. Harel and Gery [17] describe code generation from UML models consisting of class diagrams and statecharts. Engels et al. [18] present a set of rules to generate code from class diagrams and collaboration diagrams. Industrial tools such as Together [19] generate code skeletons from both class diagrams and sequence diagrams.

The FUJABA tool [20] represents designs using class diagrams and a notation called *Story Diagram* which combines statecharts and collaboration diagrams. FUJABA translates story diagrams to skeletal Java code.

The work described in this paper, unlike previous work, is concerned with testing of UML models, and, thus, is concerned with applying test criteria for testing UML models and generating tests that will be applied to UML models. Our work is concerned with pre-implementation consistency between views (currently Class and Sequence Diagrams) including OCL expressions. Models are evaluated for consistency by using a “symbolic execution” approach that relies on testing techniques to generate input values. We have not yet addressed timing issues (added in UML 2.0), but plan to address this in the future. This paper defines the construction of an aggregate model through a series of mapping from Class Diagrams and Sequence Diagrams to the aggregate model. It also presents a definition of a test process for use on the integrated model. In addition, treatment of OCL is described in detail. Lastly, we provide a fault model that classifies faults depending on when in the test and analysis process they are found. A sizeable example has been included that demonstrates the approach.

2.2. UML testing criteria

Andrews et al. [6] defined a family of test adequacy criteria for Class Diagrams and Collaboration Diagrams. The test criteria presented in Table 1 are adapted from the paper [6]. The Class Diagram Criteria are the same as in [6]. The Sequence Diagram Criteria are adapted from those for Collaboration Diagrams. Most notably, Sequence Diagrams Criteria do not require numbering of messages and thus we eliminated the criteria related to message numbering. Each test criterion applies to a single UML diagram type, in this case Class Diagrams and Sequence Diagrams.

The first set of test adequacy criteria apply to structural models and are defined for Class Diagrams. The association-end multiplicity criterion evaluates the design with respect to class relationships. An association-end multiplicity criterion requires that a set of representative (values that are possible in the system) multiplicity tuples be created during a test. The representative multiplicity tuples can be obtained by employing a form of category partition testing [1]. Using this method, the multiplicity domain is partitioned into equivalence classes, and one value from each class is selected for the set of representative multiplicities.

The generalization criterion requires evaluation of the class hierarchical structure. It defines the set of specialization types that must be created from a Class Diagram’s super class when testing. Tests that satisfy this criterion create at least one type of each specialization in the Class Diagram.

The class attribute test criteria focuses on the internal structure of each class. Each class may contain attributes,

Table 1

Test adequacy criteria definitions adapted from the paper [6]

Class diagram criteria

Association-end multiplicity criterion

Given a test set T and a System Model SM , T must cause each representative multiplicity-pair in SM to be created.

Generalization criterion

Given a test set T and a System Model SM , T must cause every specialization in a generalization to be created.

Class attribute criterion

Given a test set T , a System Model SM , and a class C , T must cause a set of representative attribute value combinations in each instance of class C to be created.

Sequence diagram criteria

Condition coverage criterion

Given a test set T and Sequence Diagram SD , T must cause each condition to evaluate to both TRUE and FALSE.

Full predicate coverage criterion

Given a test set T and Sequence Diagram SD , T must cause each clause in every condition in SD to take the values of TRUE and FALSE while all other clauses in the predicate (condition) have values such that the value of the predicate will always be the same as the clause being tested.

All message paths criterion

Given a test set T and Sequence Diagram SD , T must cause each possible message path in SD to be taken at least once.

which can be partitioned. All valid combinations of partitioned values can be used to create the test sets.

The second set of test adequacy criteria apply to behavioral models [6] and is defined for Sequence Diagrams. The defined criteria include conditional coverage, full predicate coverage, and all-paths coverage. Conditional coverage requires that each conditional statement, in the Sequence Diagram, be assigned true and false values. Full Predicate [21] coverage requires that each clause in a condition be separately tested by allowing true and false values to be assigned to each clause, which is similar to the Modified Condition Decision Coverage (MCDC) [22]. All-paths coverage requires that every path in a Sequence Diagram be traversed.

2.3. XMI

The XML Metadata Interchange (XMI) specification defines a language that allows the exchange of objects defined using the UML (Unified Modeling Language). XMI documents are intended to be stored in a file system or streamed across the Internet from a database or repository. The XMI specification proposal covers the transfer of UML models and Meta Object Facility (MOF) meta-models. The XMI identifies standard XML Document Type Definitions (DTDs) to allow the exchange of UML and MOF information. XMI also enables the automatic generation of XML DTDs for each meta-information model.

Currently there is no agreed-upon industry standard, which leads to a variety of proprietary formats, each specif-

ic to a vendor tool. The XMI specification was designed for information interchange not for executing tests. Our approach maps UML models, specified with the XMI, to our own model, the Testable Aggregate Model (TAM). This model contains behavioral, structural, and constraint information that can be used for executing tests.

3. The testing approach

We construct a Testable Aggregate Model (TAM) from information in standard UML models (usually contained in XMI files). We call our model testable since it allows us to generate and execute tests. The model is also an aggregation of several views since it is composed of Sequence Diagram, Class Diagram, and OCL information. The TAM can be thought of as another representation of UML Models that is more amenable to testing.

We did make some assumptions to simplify the problem. We did not address timing features, such as time or duration constraints, added to Sequence Diagrams in UML 2.0. We did not treat asynchronous messages as a special case. Standard UML notation often omits return calls on synchronous messages, because they are implied. However, when we translate the UML models to our TAM we must include these implied returns. Thus every synchronous message results in two vertices in the TAM. In addition, to simplifications, we employed a third party OCL tool, USE [23], for parsing and validating OCL expressions. The USE API allowed us to validate non-trivial and non-primitive OCL expressions.

The testing approach is based around constructing a TAM and using a TAM to generate and validate test cases. Fig. 2 contains a diagram outlining the approach. The testing approach consists of the following steps.

1. Build a Testable Aggregate Model (TAM) using a UML model.
 - (a) Construct a Directed Graph (DG) from each Sequence Diagram.
 - (b) Construct Class and Constraint Tuple (CCT) from Class Diagram and OCL expressions.
 - (c) Combine DG and CCT into a TAM.
2. Determine input model and generating test cases.
 - (a) Determine which attributes need partitioning.
 - (b) Partition attributes with domain analysis [1] to generate test cases
3. Execute the tests.
 - (a) for each test:
 - i record potential faults
 - ii validate test results

In step 3, faults may be found when the execution trace is validated against the OCL information. It is here that we find errors that cannot be automatically found using standard UML development tools. Standard UML development tools only statically check consistency. The testing process is designed to allow for automation. Our tool, ADAPTUML, reads in Class Diagrams, Sequence Diagrams,

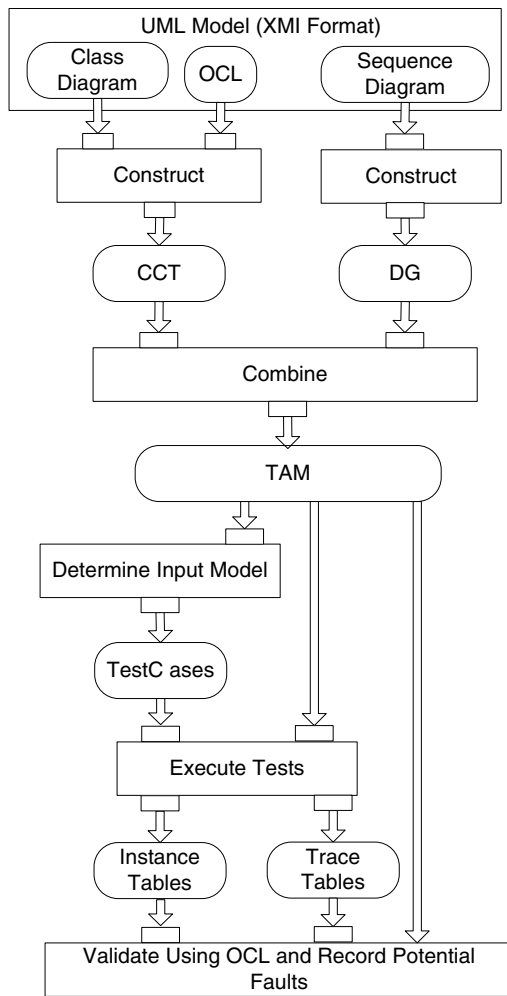


Fig. 2. The Approach.

and OCL statements in the XMI format. The tool requires the user to select ranges of values for attributes and arguments for partition testing purposes. The tool uses the TAM as an internal representation of the UML design in order to facilitate testing. ADAPTUML uses the USE tool to validate OCL expression. Each step of the approach is described in the following subsections.

3.1. Building a Testable Aggregate Model (TAM)

3.1.1. From sequence diagram to the directed graph (DG)

The testing approach prompted the creation of our directed graph (DG). The DG is constructed using the behavioral information in Sequence Diagrams. The purpose of the DG is similar to the purpose of the spanning tree created from a state transition diagram when trying to generate round-trip test sequences [1]: it facilitates test sequence generation.

A Sequence Diagram consists of classifier (object or class) boxes (for example, in Fig. 3 there are four such rectangular nodes marked *main*, *dealer*, and two *hand* nodes). Extending vertically from each classifier node is a lifeline

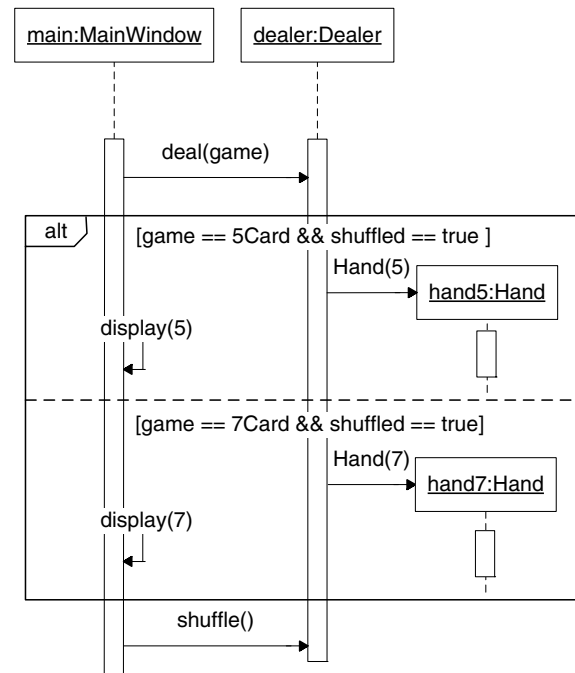


Fig. 3. A UML Sequence Diagram.

that indicates the duration of existence. Connecting between the lifelines are message lines that show the ordered communication between classifiers (e.g. *deal(game)*, *shuffle()*, etc in Fig. 3). Fig. 3 also contains a *combined fragment* (the box marked “alt”) which allows the developer to describe the control flow of messages with conditions. A *combined fragment* is used to visually display the conditional flow in a Sequence Diagram by grouping sets of messages together in a box with a conditional guard or guarded loop (e.g. contains min and max values). The UML 2.0 defines 11 different interaction operators for *combined fragments*; this work focuses upon three commonly used operators. The *option* and *alternative* operators define conditional constructs that allow optional execution of their operands (a set of messages). The *option* operator contains a boolean guard statement that must be satisfied in order for the operand to be executed. The *alternative* operator contains a set of operands, each with their own guard constraint. The *option* operator is similar to an ‘if’ statement. The *alternative* operator resembles an ‘if else’ or a ‘switch’ statement. The *loop* operator defines an iterative structure that allows for the repetition of messages. The loop fragment may contain a boolean guard condition, as well as a minimum and maximum number of iterations.

The DG is created by mapping classifier and sequence message calls from a Sequence Diagram to vertices and edges in a directed graph. To aid in understanding the mapping between UML Diagram artifacts and our model, we included in Fig. 4 a class diagram that serves as a meta-model. The mapping between Sequence Diagram and DG preserves the sequence relationships in the Sequence Diagram. The mapping consists in (1) associating messages

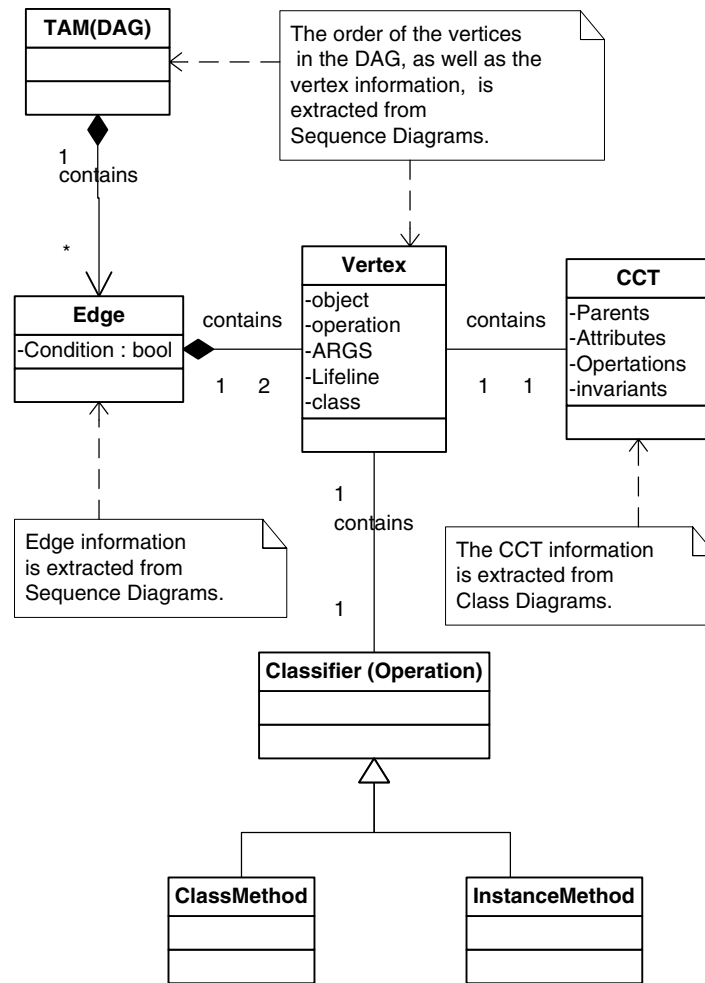


Fig. 4. Class Diagram of the Approach (Meta-Model).

in the Sequence Diagram with their originating objects and (2) traversing the Sequence Diagram for the purpose of mapping successive message executions to edges of the DG.

The DG is defined hierarchically, such that a *combined fragment* in a Sequence Diagram is represented as a DG. Thus a DG may contain other DGs. This occurs when a Sequence Diagram contains *combined fragments*. We will refer to DG's contained within other DG's as sub-DGs. The DG is a tuple $G = \langle V, E, s \rangle$ where V is a set of vertices, E is the set of edges, and s is the starting vertex. A vertex in a DG represents either: (1) a message in a sequence diagram, or (2) a sub-DG which represents a *combined fragment*. Edges represent the ordering between either messages in a Sequence Diagram or sub-DGs.

Sub-DGs may be nested. Each vertex in the DG represents either a message or set of messages (a sub-DG) from the Sequence Diagram. Each sub-DG, $v_{sub} = \langle [boolean, min, max], \langle V, E, s \rangle \rangle$, is composed of a DG and a guarded boolean expression. The boolean expression indicates the condition that must be satisfied for the sub-DG to be traversed, and the min and max values indicate how many times it should be traversed. Note that the loop arguments do not have to be specified in the case that the

combined fragment is not iterative. A vertex contains the calling object and the method call. The vertex is classified as *new*, *exists*, or *deleted* to specify its status. A *new* vertex represents a constructor or a create action, and a *deleted* vertex represents a destructor or a destroy action. The state of a vertex labeled *exists* indicates the vertex exists, thus is really a side affect of the representation. A vertex is either a message vertex or a grouping of vertices (sub-DG). Each message vertex, v , is defined by the tuple $v = \langle o, m, lifeline, \{ARGS\}, c \rangle$, where o is an object calling m , m is the message, $lifeline$ classifies an object as *new*, *exists*, or *deleted*, $ARGS$ is a set of argument tuples, and c the class name of the instance o . The $ARGS$ tuple is composed of $\langle type, name, value \rangle$, where the *type* is the argument type, the *name* is the argument name, and the *value* is any assigned value. Any of the $ARGS$ tuple elements may be null if they are not specified in the Sequence Diagram. Overloading methods does not present a problem, because the class name associated with the instance can be obtained from the UML model.

The edges of a DG are used to model the flow of control through the Sequence Diagram. While edges are not directly contained in the XMI format, they may be

obtained by observing the message order. Analyzing a Sequence Diagram to determine the message order is little different than analyzing source code to determine its flow of control. This task has been used before to automate the generation of test cases [24]. If m_i and m_j are two messages in the Sequence Diagram, let v_i and v_j be corresponding vertices in the DG. An edge is added from v_i to v_j if it is possible to execute v_j directly after v_i .

Fig. 5 shows a graphical representation of the DG corresponding to the Sequence Diagram of Fig. 3. The first message in Fig. 3, *deal(game)*, is from the *main* object to the *dealer* object. This results in the vertex,

$v_1 = \langle \text{main}, \text{deal}(), \text{exists}, \{ \langle \text{Game}, \text{game}, \text{null} \rangle \}, \text{MainWindow} \rangle$.

The *Dealer* object then calls either for a *Hand* with five cards or a *Hand* with 7 cards, resulting in two sub-DGs, s_1 and s_2 in the DG (see Fig. 5). Since s_1 and s_2 immediately follow v_1 in the Sequence Diagram of Fig. 3, there are

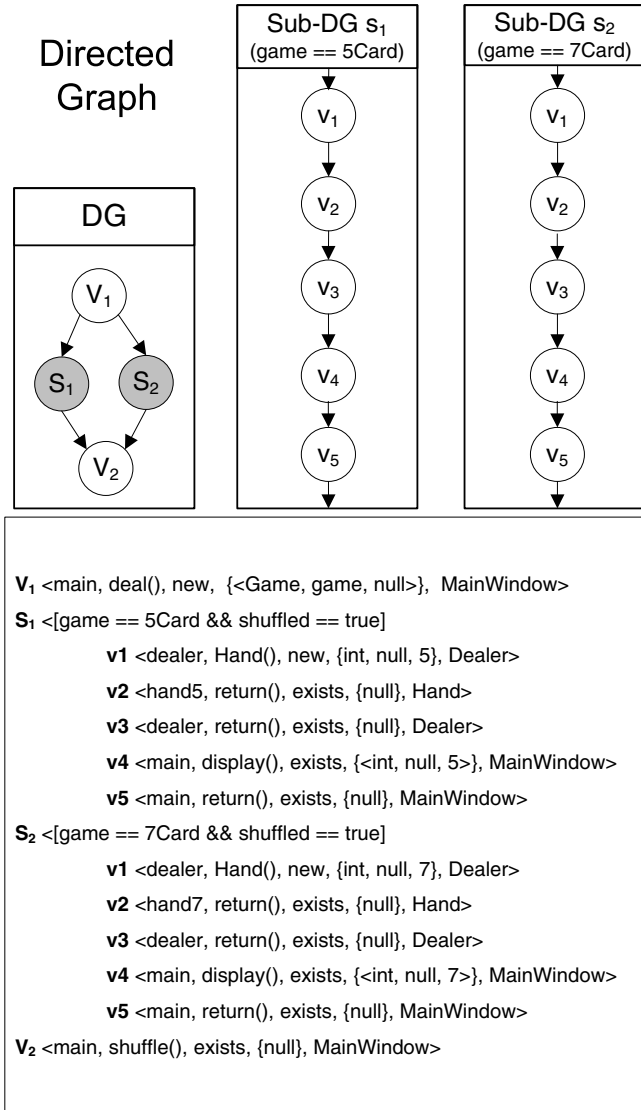


Fig. 5. A Directed Graph (DG).

two edges (v_1, s_1) and (v_1, s_2) in the top layer. When the Sequence Diagram has been fully traversed, the edges and sub-DGs of Fig. 5 result.

3.1.2. Constrained Class Tuples (CCT)

The next step in building an integrated model is to extract structural and constraint information from an XMI representation. We extract class information needed to build an instance of the class. We use the Class Diagram generalization hierarchy to more easily construct object instances for testing. An instance needs attribute and operation information from its parent class or classes. Associations and multiplicity information are transformed into OCL invariant expressions. In addition, we use OCL invariants, pre-conditions, and post-conditions that constrain attributes, classes, and operations in the Class Diagram. Table 2 lists OCL expressions used in our approach and their corresponding Class Diagram elements.

Our tuples consist of a class name, attributes from the class and super classes (if applicable), operations for the class and super classes (if applicable), and OCL information for both the attributes and operations. A Constrained Class Tuple of a class (c) has the form:

$$CCT(c) = \{ \{ \langle \text{ParentCCT} \rangle \}, \{ \langle \text{Attribute} \rangle \}, \{ \langle \text{Operation} \rangle \}, \{ [\text{invariant}] \} \} \quad (1)$$

where, c is the class name, $\{ \langle \text{ParentCCT} \rangle \}$ is a set consisting of parent class CCTs for each parent class of c (e.g., if the parents of c are c_1 and c_2 , then $\{ \langle \text{ParentCCT} \rangle \} = CCT(c_1), CCT(c_2)$), $\{ \langle \text{Attribute} \rangle \}$ is a set of attribute tuples with constraints, $\{ \langle \text{Operation} \rangle \}$ is a set of operation tuples with constraints, and invariant is a set of constraints at the class level (e.g. number of instances). The *Attribute* tuple is defined as follows: *Attribute* = $\langle \text{attribute name}, \text{attribute type}, \text{visibility}, \text{invariant}, \langle CCT \rangle \rangle$. The *Operation* tuple is defined as follows: *Operation* = $\langle \text{name}, \text{return type}, \text{visibility}, \text{pre-condition}, \text{post-condition}, \langle \text{Parameters} \rangle \rangle$. The parameters tuple contains two elements: the parameter name and type. Associations from the Class Diagram and their multiplicities are transformed into OCL invariants and stored in the CCT. When the system is tested the associations (represented as constraints) are validated. Class Diagram Associations are lines that connect classes to show relationships and multiplicities between classes. Thus, each association line has an *end* connected to a class. Notice that one association will result

Table 2

A list of OCL expressions that may constrain elements in Class Diagrams

OCL expression	Class Diagram elements
Invariant on primitives	Primitive attributes
Invariant on classes	Associations and classes (attributes that are classes)
Pre-conditions	Operations
Post-conditions	Operations

in two invariants each stored in the CCT associated with the class belonging to an association *end*. For example, in Fig. 6 class Dealer and PokerGame have a 1 to 1 association. This could be represented using the constraints:

contextPlayerinv : *PokerGame.allInstances* – > size = 1

contextPokerGameinv : *Player.allInstances* – > size = 1

Any or all of the CCT elements can have null values denoted by a null place holder for that element. The *Parent CCT* has the same structure as the CCT, thus part of the CCT is recursively defined. Each element of the CCT can be extracted from the XMI specification. One CCT is created for each class used in the DG. Classes in the Class Diagram that are not used in the DG directly or indirectly are ignored. If *A* is the set of classes referenced in a Sequence Diagram and *B* is the set of classes in a Class Diagram, then we are only interested in classes in *B* that are also found in *A*, i.e. $(A \cap B)$. If $A \setminus B \neq \emptyset$, then there are classes in the Sequence Diagram that are not defined in the Class Diagram; this is a fault. We use the CCT to instantiate necessary objects during testing and to check if constraints are met.

3.1.3. Completing the Testable Aggregate Model

The final step in building an integrated model is to combine TAM and CCT information. This is done in two steps:

1. Replace each class name, *c*, in each TAM vertex *v* with the corresponding CCT(*c*)

2. Check if CCT operation tuple elements correctly correspond with TAM operation names and *ARGS* sets, if not, flag as a fault.

Step 1 produces the vertex:

$$v = \langle o, m, lifeline, \{ARGS\}, CCT(c) \rangle \quad (2)$$

This representation combines the behavioral, structural, and constraint information into a TAM vertex. The TAM, together with the input model, forms the basis for test generation and execution.

3.2. The input model and test generation

The purpose of the input model is to create partitions for variables to be used in test case generation. The term variables refer to attributes, whose values determine a path when traversing the TAM. Pre and post-condition constraints are not used in the input model because they are used for validation purposes. Previous work in partition testing has focused on determining test cases for code [25]. We use the principles of Binder's Combinatorial Models [1] to define partitions for variables. This approach is also similar to the technique of revealing subdomains and path partitions by Weyuker [26]. When testing designs, there is less information available than when testing code. Specifically we can only test values based on variable partitions that can be derived from design information. For our approach this means that partitions for variables can

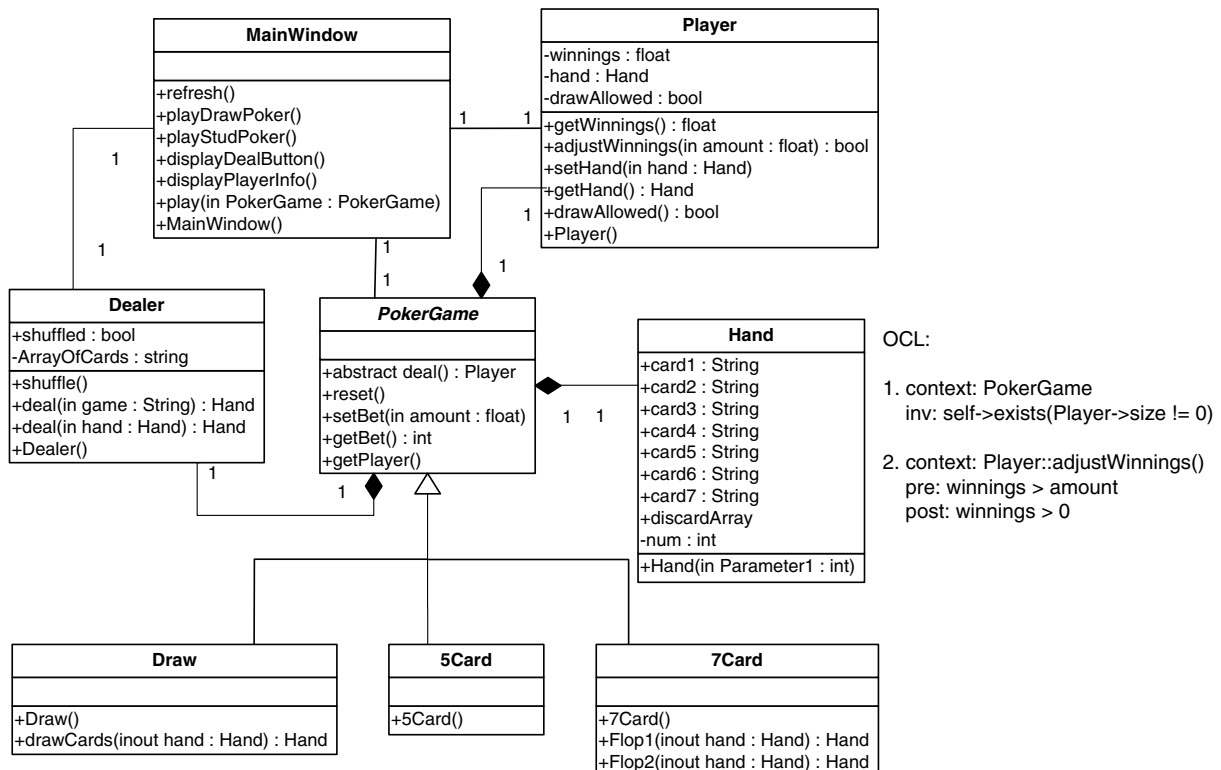


Fig. 6. A UML Class Diagram for a video poker game.

only be constructed from conditional statements in the design. The TAM contains the conditional information. Since the design cannot distinguish execution behavior for variables that are not contained in conditions in the TAM, we only need to define partitions for variables that occur in at least one condition.

Different combinations of variable values define and characterize how an object interacts with other objects [1]. The conditional expressions originate from Sequence Diagram's alternative structures and loop structures and decide path traversal in the TAM. If a variable is located in a conditional expression it is selected for analysis. Defining the input model consists of the following steps:

1. Identify the set of variables that occur in conditions. These variables define the dimensions of our input domain.
2. Determine the range based on type of variable. Use one of the combinational model techniques in [1] to determine partitions and combinations of partitions.
3. Select test values based on the combinational techniques used in step 2.

Our tool uses non-binary domain analysis [27] as its combinatorial model. A domain is the set of all inputs for a variable. A subdomain is a partition of the domain defined by boundary conditions. Non-binary domain analysis uses decision tables which are divided into distinct subdomains. Currently we need the tester to input the subdomains defined by the boundary conditions. One test strategy is to pick one *ON* point and one *OFF* point per boundary. An *ON* point lies on a boundary, and conversely, an *OFF* point lies outside the boundary. Table 3 uses the two conditions from Fig. 5. The table shows the test inputs generated by applying non-binary domain analysis to the variables *game* and *shuffled*. We will assume that a test case that satisfies the conditions is associated with a primary path (PP). A secondary path (SP) is associated with unsatisfied conditions. Note that the test cases T1 and T3 are the same, as well as T4 and T6. Since

the variables are Boolean, the *ON* and *Typical* will always be the same, resulting in duplicate test cases. In practice this can be avoided, by dropping duplicates as they occur.

3.3. Test execution

Test execution consists in traversing the TAM using the test cases generated in Section 3.2. Our approach addresses the problem of inconsistency between the Class Diagram, Sequence Diagram, and OCL constraints. To address this problem we need to assess how a sequence of events (Sequence Diagrams) changes the system. The test cases are used to assign values to variables in order to facilitate path traversal. Thus our definition of execution is a simulation (not code execution). The structure of the CCT makes it possible to traverse all related objects by visiting each parent and attribute element in a CCT. To record the changes to the system, state and trace information is recorded for each path. The recorded information is used for validation against OCL constraints. Each instantiated class is tracked through an instance table. An instance table records the object and class names and keeps count of the number of instances for that class. For example, this information can now be used to check if size constraints are violated in OCL invariants. When a new object is created its class name is added to the instance table if not already present. For each new instance of a class, the counter in the instance table is incremented. For each deletion of a class, the counter in the instance table is decremented. In addition to the instance table, a trace table records each message, each object, and every attribute assigned values. Test execution builds both the instance table and trace table. Each test case T_j contains attribute values for a series of conditions c_1 through c_k in the TAM. The following steps are used for each test case T for execution:

For each test case T_j execution consists of the following steps:

1. Begin with the start vertex s of the TAM
2. For every vertex:

Table 3
Domain analysis of the poker game

Conditions	Boundary	Test cases							
		T1	T2	T3	T4	T5	T6	T7	T8
$(game = 5Card)$	ON	5Card	4Card	5Card	7Card	5Card	7Card	5Card	
	OFF								
	Typical								
$(game = 7Card)$	ON					5Card	7Card		7Card
	OFF								
	Typical								
$(shuffled = true)$	ON	T	T	T	T	T	T	F	F
	OFF								
	Typical								
Expected results		PP	SP	PP	PP	SP	PP	SP	SP

- (a) Record message in trace table and (if present) check if the values satisfy the OCL pre-conditions and post-conditions using the USE tool.
 - (b) Record object name in trace table and (if present) check if OCL invariant is satisfied. An option in the tool allows for checking if an object has parents and then checks if the OCL invariants of the parents are satisfied (assumes Liskov principle).
 - (c) Record object count in instance table. If the life-line indicates the object is new, add the object and all objects contained in the associated CCT to the instance table. Adding all related objects requires traversing each parent and attribute element in the CCT from left to right (objects are uniquely identified and are checked for prior existence). Update the counter for all objects.
 - (d) For each conditional vertex v_i , assign and record attribute values to v_i from T_j , where T_{ji} is the i th attribute set in the test case T_j . Each attribute is recorded by appending its unique object name or class name (if class variable) using dot notation.
3. Select next vertex in TAM (based upon the path and condition):
 - (a) If last vertex in path, end test case.
 - (b) If number of successor vertices = 1, select only child.
 - (c) If number of successor vertices ≥ 2 (conditional vertex), select vertex that contains a condition that is satisfied by i th attribute set T_{ji}
- OCL expressions are checked (using USE) by systematically examining the trace and instance tables for values that violate the invariant, pre-condition, or post-condition.

3.4. Fault analysis

When applying the UML evaluation approach, faults and inconsistencies can be revealed throughout the process. Like other tools, our process can reveal a set of faults related to inconsistency via static analysis. However, the advantage to our approach lies in the application of dynamic testing techniques applied to the aggregate model. When test cases are executed two different types of faults can be revealed. The first type of fault can be classified as a *path fault*, which is found by traversing the TAM. When traversing the TAM with *ON* and *OFF* points, paths may not be traversable or the path may not exist (this often occurs in code when exceptional cases are not addressed, e.g. you assume the user will enter correctly formatted information). These types of faults often occur because the designer did not address all paths associated with a condition, e.g. a Sequence Diagram contains a path for $x \leq 5$, but not for $x > 5$. This may be intentional (a trivial case that will be addressed in the implementation phase) or it may be an oversight on the part of the designer. In any case, notification of path faults can aid in both the design and implementation phases. In addition, if a traversal of a path is

not possible because an operation is private, abstract, or not available, then a path fault is revealed.

The second set of execution faults, known as *OCL faults*, occurs when states recorded in the execution trace or instance table violate OCL expressions. As test executions are recorded, variables may take on values that violate OCL constraints. In addition, the instance table is a record of active objects in the system. The OCL is often used to constrain sets of objects. Thus when OCL specifies constraints on sets of objects, the instance table is consulted to check if the OCL constraint is violated. The Poker Game contains an example of the *OCL fault*. There is a constraint (see Fig. 6) that states that a *Player* must exist if there is an instance of the *PokerGame* class. However, in the Sequence Diagram, the *Player* is not created. The constraint can only be automatically verified if we have structural information available when executing a sequence of events. A violation of this constraint would also require instance information.

3.5. Practical considerations

This approach works for incomplete models. Often designs do not specify every sequence of events in the system. For example, if a Sequence Diagram for a poker game does not contain a shuffling operation, the dealer may never enter certain states. Our approach only examines states that are reachable via transitions in a Sequence Diagram and is not concerned with all reachable states and missing states. Therefore, we are looking at a small subset of all possible states that are generated from taking the Cartesian product of class attributes. It is the subset states that are recorded in trace and instance tables and are used for consistency checking against the OCL information. Since we allow for incomplete models and partial analysis, this approach is very flexible.

The consistency checking compares each state in a system (e.g. trace and instance table information) against each associated constraint. An associated constraint is defined as any constraint that contains a variable also found in the state. States are bounded (in the TAM) by the number of method calls responsible for state changes. Thus the number of states in the system is equivalent to the number of entries in the trace table, N . Searching for constraints based on variables names can be made into a constant time operation using a hashing function. Checking if the values associated with the variables satisfies a constraint is dependent on the constraint.

One aspect of our tool allows users to view TAMS. In practice, we found that visualizing the faults using a graph viewer was often helpful in correcting the fault, however, we have not empirically investigated this aspect (one of our future research goals). Viewing TAMS for large designs can be difficult if there are large amounts of message sequences with no branching (decisions). Our tool can compress TAMS by combining all sequential vertices into a composite vertex representing a linear sequence. For

example, in Fig. 5 the sub directed graphs can be hidden resulting in only the top level graph. In the example in Section 4, we implement these compressions to make the TAM easier to view and understand.

In addition, for large designs we can group related vertices together to create a higher level of abstraction. For example, there could be several sub-DGs connected to each other. One could then model the collection of sub-DGs as a top level DG. As needed, we could “zoom” into a vertex in the top level DG to see more detail. This is similar to embedded Sequence Diagrams (combined fragments) in the UML.

4. An example

In this example, we apply our UML analysis approach to a more extensive UML design and describe each step of the testing approach. In addition, we conducted a case study in which the design is implemented and then tested to reveal faults. The implementation faults are compared to the faults found in the design in order to see if design faults result in similar implementation faults. Thus the UML Diagrams contain faults, listed in Table 4 that naturally occurred in the design process. Table 4 contains the location of the fault in the Sequence Diagram, but not that all of the faults involve inconsistencies, between more than one diagram.

The design specifies Class and Sequence Diagrams as well as OCL expressions for a Poly-Imager software application. The imaging application, given a polygon description file (lists of vertices and colors), will create an image file containing several overlaid polygons. We employed the use of files instead of streams too make the project simple. We purposely decided to constrain the number of polygons (20 definitions) that could be in memory, because we wanted to use the application on handhelds, which may limited resources. Each polygon is two-dimensional, contains between 3 and 6 vertices, and can be filled with either a Gouraud shading or solid color. Gouraud shading is a method to smoothly shade a flat polygon to look curved or provide an appearance of lighting. The Solid Polygon provides a simple alternative, in which the entire Polygon is just one shade. Each input file contains the type of polygon (gouraud, or solid), a polygon size, and a list of coordinate and color values.

4.1. Case study

To see if our approach detects faults in designs that eventually propagate into implementation faults we asked

students to implement the design described in this section. The description of the design and the design documents were taken directly from this example, and thus included a brief description of the program and the UML documents. The case study was conducted in a classroom environment. The subjects were 10 students in a Senior Software Engineering course at Washington State University. The project was part of a take home test in their Senior Software Engineering Class. They were asked to implement the design using Java. They were not given an in-class time constraint, but allowed to develop the software over the weekend. All students (but one) created running implementations of the design, albeit with some faults.

In order to test the implementations we used boundary values from the input domain. The code tests were easy to create, for example the *undefined r value* fault can be demonstrated by simply creating an input file using the exact same values we used for testing our design. We analyzed each fault and checked to see if the fault contained the same attributes or operations that were found to contain faults in the design. Table 6 contains a summary of our results. Column one contains an anonymous project # (for privacy of the student), column two contains the total number of faults we found when testing the software, column 3 contains the list of faults that where directly

Table 5
OCL Table (some pre/post conditions not specified by designer)

Color	
r	inv: $0 < r$ and $r < 255$
g	inv: $0 < g$ and $g < 255$
b	inv: $0 < b$ and $b < 255$
Color(r,g,b)	pre: $0 < r$ and $r < 255$ and $0 < g$ and $g < 255$ and $0 < b$ and $b < 255$
Position	
x	inv: $1 < x$ and $x < 300$
y	inv: $1 < y$ and $y < 300$
Position(x,y)	pre: $1 < x$ and $x < 300$ pre: $1 < y$ and $y < 300$
VertexList	
getSize()	post: $result = v.size()$
getVertex(i)	post: $0 < i$ and $i < self.vertex - > size()$
getVertex(i)	post: $result = self.vertex - > elementAt(i)$
ScanLine	
ScanLine(s,e,c,dcdx)	pre: $1 < s$ and $s < 300$ and $1 < e$ and $e < 300$ pre: $1 < x$ and $x < 300$
XPosColor(x)	
FrameBuffer	
writeLine(sl,y)	pre: $1 < y$ and $y < 300$
Polygon	
Polygon()	inv: $Polygon.allInstances() \rightarrow size() < 10$
Gouraud	
Gouraud()	inv: $Gouraud.allInstances() \rightarrow size() < 10$
Solid	
Solid()	inv: $Solid.allInstances() \rightarrow size() < 10$
Image	
InputFile	inv: $self.vertexList.getSize() > 0$

Table 4
List of design faults

Fault type	Location	Description of fault
Static	Fig. 9	Uses Method in SD that does not exist in CD
Static	Fig. 11	Uses Private Method
Dynamic	Fig. 10	<i>undefined r value</i> used to create color object
Dynamic	Fig. 10	<i>x, y values off by one</i>
Dynamic	Fig. 9	<i>polygon constraint</i> violated by loop bounds

Table 6
Design faults present in the implemented software

Project Id.	Total # of code faults	Design fault(s) in code
1	4	Polygon constraint error
2	3	Polygon constraint error
3	3	Polygon constraint error
4	5	Undefined r value Polygon constraint error
5	5	Undefined r value Polygon constraint error
6	6	Off by one Undefined r value Polygon constraint error
7	4	Polygon constraint error
8	2	Polygon constraint error
9	4	Polygon constraint error

related to faults in the design. We found that only one project had an *off by one* fault present. We assume students overcame the design fault by using vertex values that included 1 and 0 when testing their software. We found that 3 out of 9 projects had null pointer exceptions when the color was not defined properly. This was related to the *undefined r value* design fault. These null pointer exceptions could most likely have been avoided if the UML design had correctly assigned a default value to the color object. All 9 projects did not correctly constrain the number of polygons created. Since this fault did not crash the software, we assume that the students ignored the OCL constraint. In addition to the faults that were related to design problems we found several null pointer exceptions which we could not attribute to design problems. The projects had several other problems with null pointer exceptions, due to using inputs that were not properly initialized. In addition, the horizontal edge removal process was not properly addressed in the design. Eight of the 10 projects contained faults concerning null pointer exceptions due to edge removal. These faults were due to the design being incomplete. This cannot be automatically detected by our approach.

4.2. Building the Testable Aggregate Model

All design diagrams were created using UML 2.0 [28]. Fig. 7 shows the Class Diagram for Poly-Imager, which includes all of the classes, relationships, and multiplicities. The Class Diagram has 15 classes each with a median of 2 attributes, and 3 operations per class. There are 20 different associations and 3 instances of inheritance. The *Image* class acts as the main driver. The *Image* class uses the *Solid* and *Gouraud* classes to create images. These two classes extend an abstract *Polygon* class, filling in details specific to that class. Inside the *Polygon* sub-classes, *VertexList* is used to represent each input file as a list of vertices, and the *Interpolator* uses the information to fill the *FrameBuffer*. The *FrameBuffer* is the internal canvas upon which each polygon is drawn. It contains the information which is written to an image file. There is a collection of helper classes such

as *Position*, *Color*, *Vertex*, and *Edge* that keep track of necessary data.

Table 5 shows a partial listing of the OCL constraints for Poly-Imager. For any color in the program, the values for red, green, and blue can vary between 0 and 255. The image size is 300×300 , so the *x* and *y* values for a position can vary between 1 and 300. The program is not allowed to request a vertex number greater than the number of vertices; there is a similar, but unlisted constraint for the edge table. Finally, we are only allowed to have up to 10 *Polygon* instances at one time. This constraint extends to *Gouraud* and *Solid* polygons as well.

The class information from the Class Diagram is extracted into a Constrained Class Tuple. Only classes used in the Sequence Diagram are transformed into CCTs. Fig. 8 shows and labels the CCT for the *Polygon* class.

The Sequence Diagram in Fig. 9 describes the overall execution of the Poly-Imager. This “top level” Sequence Diagram contains several references to other Sequence Diagrams as well as combined fragments. The Driver class creates an instance of the *Image* by using the constructor to pass in Polygon Definition Files. The *VertexList* constructor, Fig. 10, reads in a Polygon Definition file and creates a list of vertices that represent a Polygon. Notice that there can be 20 definitions, consisting of 10 *Solid* and 10 *Gouraud* polygons. Fig. 10 shows the conversion of data in the file into actual vertex objects. *VertexList* first reads in the *Type* of Polygon, and then the number of sides. After this, the constructor acts as a giant loop which reads through the input file, and stores the coordinate and color information in a Vertex for later use. In the loop, the size (the polygon’s number of sides) varies from 3 to 6, providing min and max values.

The *VertexList* object specifies the creation of either a *Solid* or a *Gouraud* polygon by using the *getType()* method. The creation of the *Gouraud* polygon is referenced and shown in a separate Sequence Diagram (see Fig. 11) due to its complexity. Inside of the *Gouraud* Sequence Diagram is the *Gouraud* class which sets up an *Interpolator*, which in turn creates the *EdgeTable*. The *EdgeTable* transforms vertices into edges, which are used to perform mathematical interpolation operations. The loop fragment in the *Gouraud* diagram provides the method calls necessary for transforming *Vertices* into *Edges*. At the end of the Diagram functions are called to for sorting and removing unnecessary horizontal edges. Control passes back to the calling object in the main Sequence Diagram.

At this point, the Sequence Diagram has created either a *Solid* or *Gouraud* polygon object which will interpolate the information from the *VertexList* and turn it into an image of a polygon. Before an image can be created the polygon needs to be filled. This intensive computation consists of two nested loops of 300 iterations. Using the edges from the *EdgeTable*, the fill operation interpolates a value for each of the 90,000 pixels. Note that the *Interpolator* uses *ScanLine* to fill the *FrameBuffer*. After all the polygons have been fully interpolated, the *CreateFile* operation

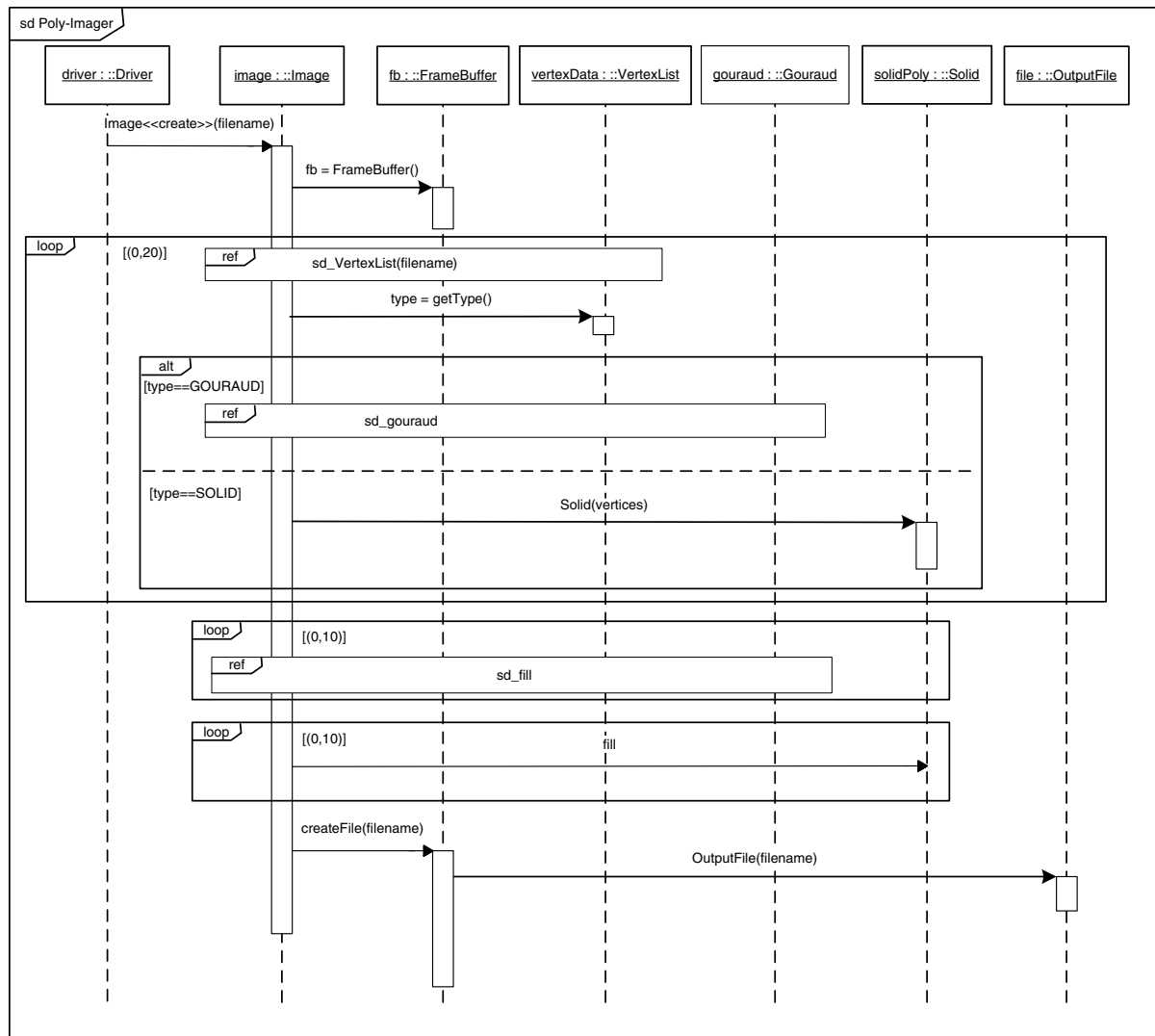


Fig. 9. Sequence Diagram. Overview of the Polygon Imager Software.

helps focus attention on the vertices that are involved in decisions or loops. The TAM shows multiple edges stemming from vertices where decisions take place. For example, vertex 28 and 50 are two alternatives that can be called from vertex 25 based upon different input values (GOURAUD or SOLID). A vertex with a prefixed with an “x” represents a sub-DG. The loop numbers in 13 show the minimum and maximum number of times a loop can be executed. For example, the loop described by sub-DG x11 must be executed a minimum of three times, but no more than six times. As an alternative the loop described by sub-DG x67 only has one number because its lower bound is zero.

Several exceptions were raised while constructing the TAM in Fig. 13. One fault occurs at vertex 25. The Image class attempts to retrieve the polygon type from the *VertexList* by calling *getType()*. However, when the Class Diagram is consulted, the designer never specified this function. This is an easy exception to fix, just add the *get-*

Type() operation to the *VertexList* class. Another fault is detected on vertices 42 and 44. An instance of *Interpolator* calls *condense()* and *sort()* on an instance of *EdgeTable*. The problem is that both of these functions are private. It is not possible to determine the visibility of these operations using only the Sequence Diagram, but when the Sequence Diagram and the Class Diagram are combined into a TAM it can be easily seen that this is an error, and a fault is detected. To fix this fault, both of these function calls could be simply moved into the *EdgeTable* constructor. These faults can be detected in current UML design tools.

4.3. Determine input model and generate test cases

There are several different inputs to Poly-Imager. The main input source is a list of files from which the specifications for each polygon are read. There is a limit to the number of polygons that can be read by the program. Because

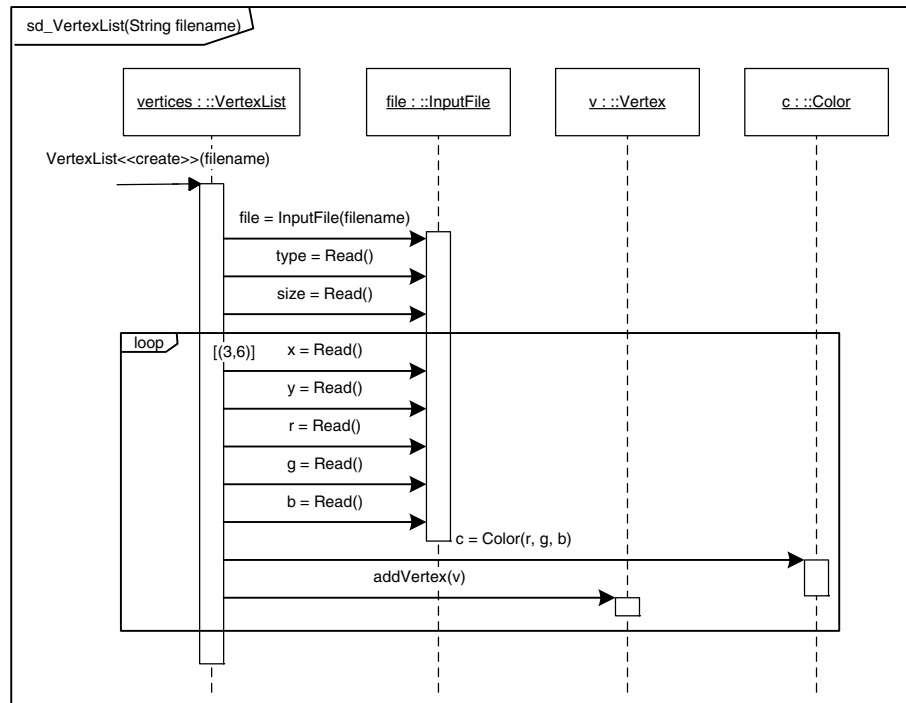


Fig. 10. Sequence Diagram. VertexList shows how a list of vertices is created from a file containing locations and colors.

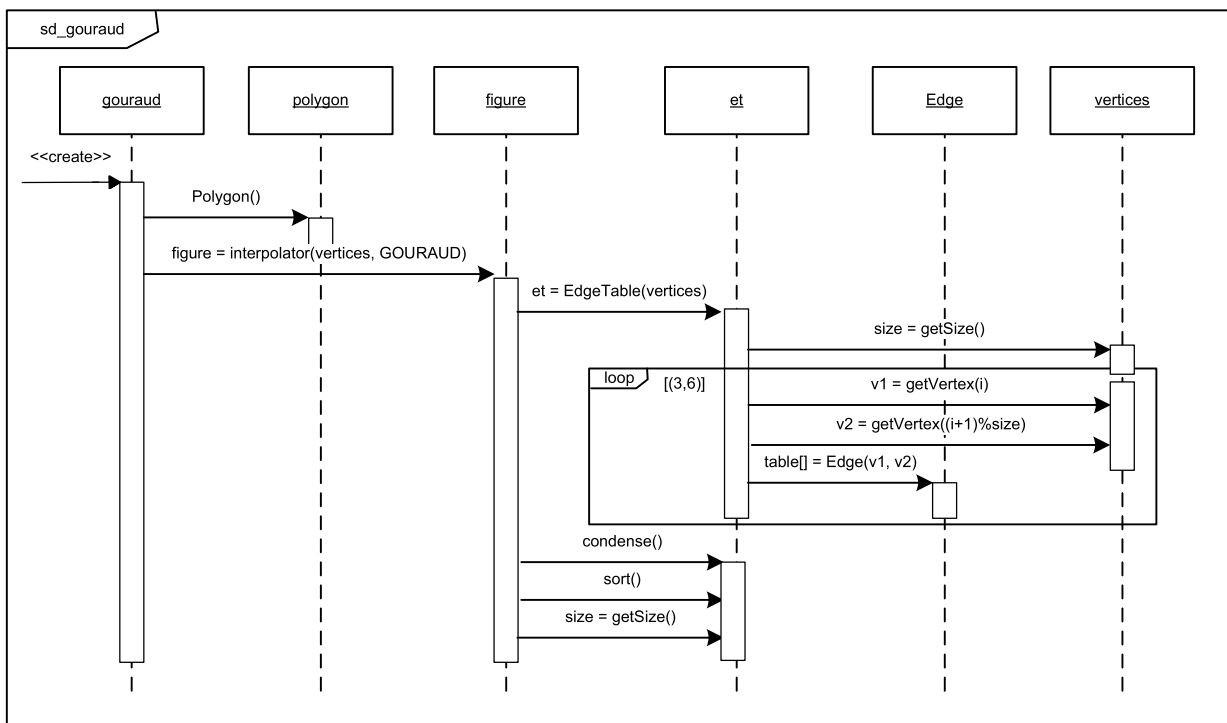


Fig. 11. Sequence Diagram, GouraudPoly, demonstrates how to create a Gouraud shaded polygon.

the mathematical operations performed to create a Polygon are quite intensive and the targeted system may have limited resources, the designers decided that there should only be 10 instantiated Polygons at any one time.

The first item in the input file specifies the type of polygon, which can be either Gouraud or Solid. The next item in each file is the size of the polygon. The lowest order polygon is a triangle. The highest order polygon has n sides.

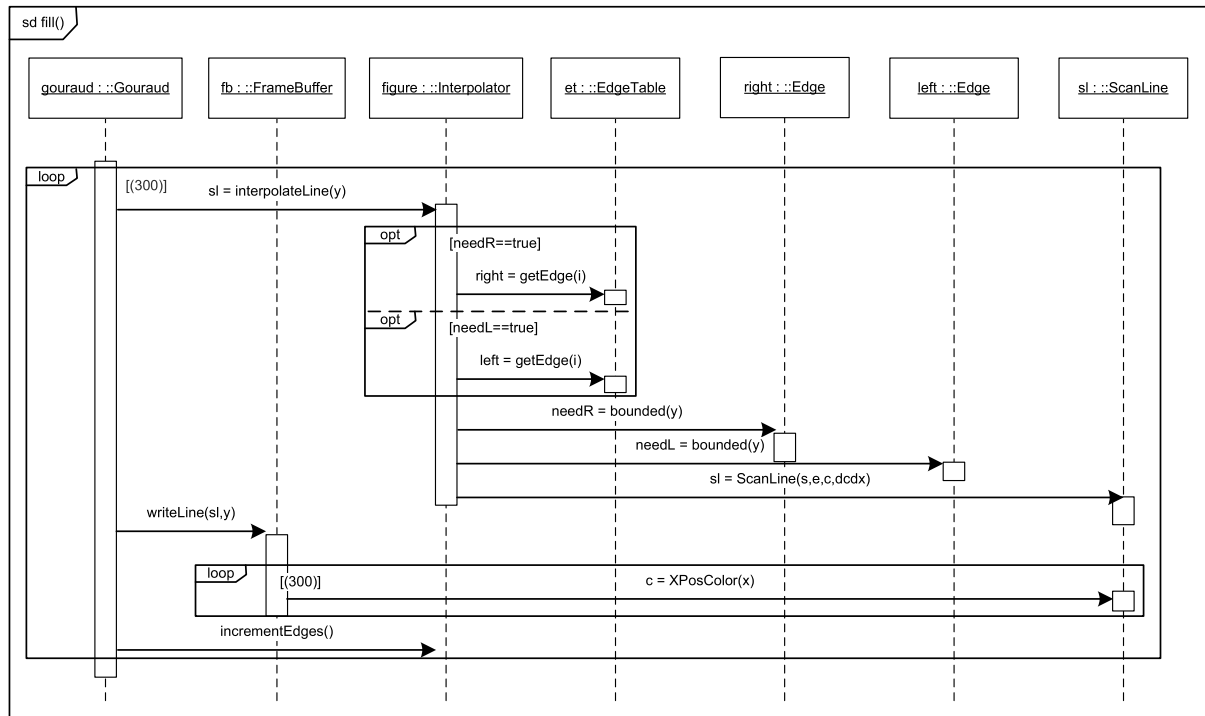


Fig. 12. Sequence Diagram – GouraudPoly.fill demonstrates how to fill a Gouraud polygon.

According to the UML 2.0 Specification [28], maximum and minimum values may be specified for loops.

The final set of inputs is a set of vertices: a Cartesian coordinate and an RGB color value. Each vertex must be within the 300×300 range, and all RGB values range from 0 to 255. Using these values, Table 7 shows the input domain for Poly-Imager.

The Input Model is determined from the conditional information in the TAM. To begin creating test cases, the set of variables that occur in the conditions in the TAM must be identified. The conditions from the input domain which affect the TAM, are the number of Polygons (also the number of each type of polygon), the type and size of each Polygon, and whether the individual color values are valid. The filenames are ignored for the purposes of these test cases, because the specific value does not affect any conditions in the TAM. There are some conditions in the TAM which will be left out of the domain analysis. Because the canvas size (300×300) does not change, the values cannot be altered to create a different path through the TAM. The *needL* and *needR* values in Fig. 12 are also ignored in this analysis, because they do not come directly from the input domain. Determining when an edge is done being processed is based upon a counter and the position values in the edge. This is determined through complex internal logic, which is not modeled in the Sequence Diagrams.

Next, the range and conditions of each variable need to be determined. These can be found by examining the conditions in the TAM. For instance, by following the size

input for each polygon, it can be seen that the size affects two loops (Figs. 10, 11). The conditions on the size both state that it has a minimum value of 3, and a maximum value of 6. Therefore, 3 and 6 are the ON points, and any value less than 3, or greater than 6 are the OFF points. Tables 8–10 show the domain analysis test cases based upon the number of each type of polygon, the position, and the color values.

4.4. Execute the tests

Some of the test cases in Tables 8–10, contain conditions which can lead to faulty states. These conditions show how a lack of integrated information may produce errors in design, and how this model can be used to find such errors.

Walking through Test Case T_{c2} (Table 8), uncovers an path fault contained in the design. The execution of T_{c2} begins with vertex 1 of the TAM. There are no special conditions or values in the first 3 vertices in the TAM. Because each vertex has only 1 child, selection of the next vertex is trivial. For each of the first 3 vertices, the Calling Operation, and the Object Name are recorded in the Trace Table. Because they do not access any values in the test cases, the attribute values column is empty. At vertex 1, an instance of *Image* is added to the Instance Table. Again, at vertex 2, an instance of *FrameBuffer* is added to the Instance table. At this point, the Trace Table is the first four lines of Table 11, and the Instance Table indicates 1 *Image* object and 1 *FrameBuffer* object have been instantiated.

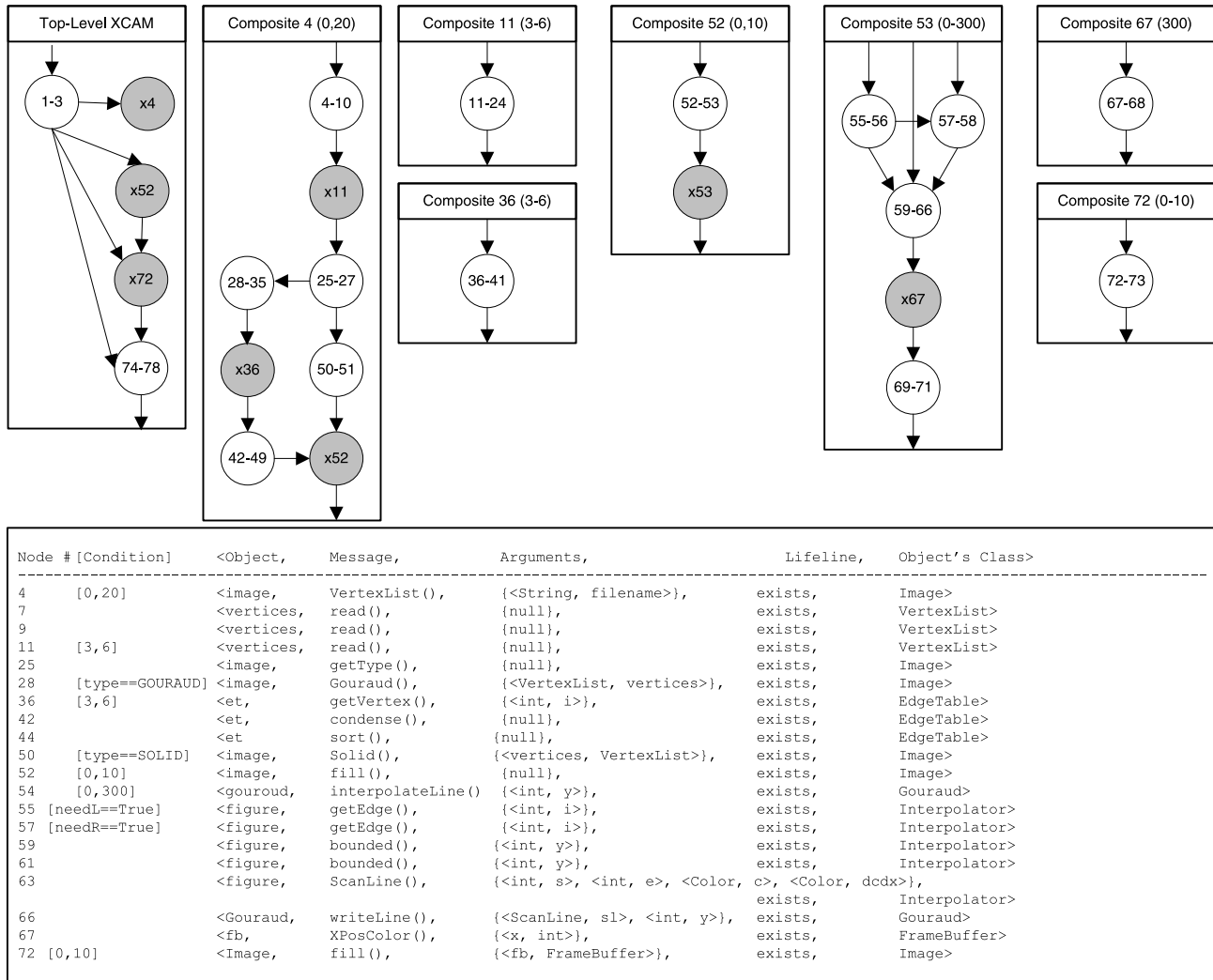


Fig. 13. Poly-Imager TAM.

After recording vertex 3, the first decision is reached, which is based upon the number of polygons. Because the Test Case T_{c2} indicates there is 1 polygon, execution passes to vertex 4, which satisfies this condition. Although vertex 4 is the beginning of a loop, it only has only one successor, since the value for the loop is one. After processing vertex 4, execution must pass to vertex 5, because it is the only child. Vertices 4–11 are added to the Trace Table just as before. Vertices 5 and 6 add an instance of *VertexList* and *InputFile* to the Instance Table. For vertices 7 and 9, T_{c2} is accessed to obtain values for the type and size. This is shown in the Attribute Values column of the Trace Table.

Vertex 11 signifies the beginning of another loop. The TAM illustrates execution must pass to vertex 12, because it is the only child of vertex 11. (In the Sequence Diagram it states this loop must be traversed at least three times). Vertices 11–20 are added to the Trace Table and the Test Case provides values for the position and color values.

After vertex 20 is processed, execution passes to vertex 21, which requires the r value, which is -1, to be between 0 and 255. While processing vertex 21, a path fault is found because c has not been assigned a value in the Trace Table, and cannot be used as a parameter in vertex 21. Table 11 shows the Trace Table at this moment. The Instance Table would contain one instance of the *Image*, *FrameBuffer*, *VertexList*, and the *InputFile* classes.

The reason for this fault lies in the error checking done around the color values, which prevent the *Color* object from being instantiated when invalid values are read from the file. This causes the color to be undefined when the *Vertex* is created, signifying a path fault in the TAM. We will reference this fault as *undefined r value*. This fault is caught through the Trace Table, and can be handled during the design phase. This fault would not be automatically detected by current UML tools.

The Trace Table is designed to locate faults dealing with values assigned to attributes and arguments. Another fault

Table 7
Input Domain

Input	Valid range
# Polygons	$1 \leq \#Gouraud + \#Solid \leq 10$
# Gouraud	$1 \leq \#Gouraud \leq 10$
# Solid	$1 \leq \#Solid \leq 10$
Input variables for each polygon	
Filename	Any valid filename
Size	$3 \leq Size \leq 6$
Type	Gouraud, Solid
Input variables per vertex per polygon	
Position (X, Y)	$0 \leq X \leq 300$
	$0 \leq Y \leq 300$
	$0 \leq R \leq 255$
Color (R, G, B)	$0 \leq G \leq 255$
	$0 \leq B \leq 255$

that could be caught using the Trace Table concerns the constraints on the Position and Vertex class. When the Test Case T_{p2} from Table 9 is executed, it will begin producing a Trace Table and an Instance Table. Like the Test Case T_{c2} , it will halt after processing vertex 21, although for com-

pletely different reasons. Table 12 is a snap shot of the trace table created by Test Case T_{p2} when the OCL design error is detected. The designer added error checking on the input from the file, forcing the value of the (x,y) position variables to be between 0 and 300. The problem is, according to the OCL, the values need to be between 1 and 300. This *off by one* error is easily detected by the Trace Table, and can be addressed in the design phase. In addition, we find that the OCL constraint in the *Image* class has been violated. The image OCL constraint states that for a file to exist, a vertex must be in existence as well. However, we can see from the trace table that this is not the case. The designer needs to reconcile this inconsistency. We will reference this fault as *the off by one fault*. This fault would not be automatically detected by current UML tools.

While the Trace Table can catch OCL errors, other errors can only be caught by examining the Instance Table. In the design, the Polygon class has an OCL constraint stating it can only be instantiated 10 times. Both of the subclasses inherit this constraint. There are also OCL constraints stating that only 10 Gouraud Polygons can be instantiated. Likewise only 10 Solid Polygons can be

Table 8
Sample test cases – variant = Color.r

Variable	Condition	Bound	T _{c1}	T _{c2}	T _{c3}	T _{c4}	T _{c5}
# Polygons	Typical	IN	1	1	1	1	1
Size	Typical	IN	3	3	3	3	3
Type	Typical	IN	G	G	G	G	G
Color _{[1],r}	≥ 0	ON	0				
		OFF		–1			
	≤ 255	ON			255		
		OFF				256	
Color _{[1],g}	Typical	IN					128
	Typical	IN	128	128	128	128	128
Color _{[1],b}	Typical	IN	128	128	128	128	128
Color _[1–6]	Typical	IN	(r,g,b)	(r,g,b)	(r,g,b)	(r,g,b)	(r,g,b)
Position _[1–6]	Typical	IN	(x,y)	(x,y)	(x,y)	(x,y)	(x,y)
Expected results			PP	SP	PP	SP	PP

Table 9
Sample test cases – variant = Position.x

Variable	Condition	Bound	T _{p1}	T _{p2}	T _{p3}	T _{p4}	T _{p5}
# Polygons	Typical	IN	1	1	1	1	1
Size	Typical	IN	3	3	3	3	3
Type	Typical	IN	G	G	G	G	G
Color _[1–6]	Typical	IN	(r,g,b)	(r,g,b)	(r,g,b)	(r,g,b)	(r,g,b)
Position _{[1],x}	≥ 1	ON	1				
		OFF		0			
	≤ 300	ON			300		
		OFF				301	
Position _{[1],y}	Typical	IN					150
	Typical	IN	150	150	150	150	150
Position _[2–6]	Typical	IN	(x,y)	(x,y)	(x,y)	(x,y)	(x,y)
Expected results			PP	SP	PP	SP	PP

Table 10

Sample test cases – variant = Number of each polygon type

Variable	Condition	Bound	T _k 1	T _k 2	T _k 3	T _k 4	T _k 5	T _k 6	T _k 7	T _k 8
# Gouraud	≥0	ON	0							
		OFF		−1						
	≤10	ON			10					
		OFF				11				
	Typical	IN					4	4	4	4
# Solid	>0	ON					0			
		OFF						−1		
	<10	ON							10	
		OFF								11
	Typical	IN	4	4	4	4				
Size _{all}	Typical	IN	3	3	3	3	3	3	3	3
Expected results			PP	–	SP	SP	PP	–	SP	SP

Table 11

Trace Table T_c2

Calling object	Operation call	Attribute values
Driver	Image()	
Image	FrameBuffer()	
fb	return	
Image	VertexList(—)	
vertices	InputFile(—)	
file	return	
vertices	read()	
file	return	vertices.type = GOURAUD
vertices	read()	
file	return	vertices.size = 3
vertices	read()	
file	return	x = 1
vertices	read()	
file	return	y = 1
vertices	read()	
file	return	r = −1
vertices	read()	
file	return	g = 128
vertices	read()	
file	return	b = 128
vertices	Vertex(0,150,c)	Path Fault (c not defined)

Table 12

Trace Table T_p2

Calling object	Operation call	Attribute values
Driver	Image()	
Image	FrameBuffer()	
fb	return	
Image	VertexList(—)	
vertices	InputFile(—)	
file	return	
vertices	read()	
file	return	vertices.type = GOURAUD
vertices	read()	
file	return	vertices.size = 3
vertices	read()	
file	return	x = 0
vertices	read()	
file	return	y = 150
vertices	read()	
file	return	r = r
vertices	read()	
file	return	g = g
vertices	read()	
file	return	b = b
vertices	color(r,g,b)	
color	return	color. <r, g, b>
vertices	Vertex(0,150,c)	OCL Fault (x = 0)

instantiated. From Table 10, the test cases for T_k2 and T_k6 cannot happen, because it is impossible to have a negative number of polygon files. The test cases for T_k4 and T_k8 lead to faulty states, which will be caught by examining the constraints on Solid and Gouraud, because we are not allowed to create more than 10 of each type of Polygon. This error is prevented in the Sequence Diagram by the maximum value of 10 for the loops surrounding the fill() calls in Fig. 9. However, test cases T_k3 and T_k7 lead to invalid states, allowed by the Sequence Diagram, which are more tricky to detect.

The number of instantiated Gouraud Polygons, and the number of instantiated Solid Polygons both pass the OCL constraints on their class. However, it calls for 15 total polygons to be created, which violates a constraint of the parent class. This fault can be found by creating an Instance Table. Table 13 shows the Instance Table for

T_k3 when the fault is first logged. At this point an exception is raised stating a design error allows the Polygon constraint to be broken. We will reference this design fault as the *polygon constraint error*. To fix this error, one must change the max value in the first loop in Fig. 9 from 20 to 10. This fault would not be automatically detected by current UML tools.

While these errors may be caught and fixed through inspection, we can see that some of the errors that are automatically detected by our approach are very subtle. As a design's complexity and size increases inspection techniques may become less effective [5]. Thus, as designs increase in size and complexity, our systematic approach becomes much more appealing.

Table 13
Instance Table – based upon T_{k3}

Class	Number
Color	33
Position	33
File	1
Vertex	33
VertexList	1
Edge	30
EdgeTable	10
Interpolator	10
FrameBuffer	1
<i>Polygon</i>	<i>11</i>
Polygon. Gouraud	10
Polygon. Solid	1
Image	1

5. Conclusion

This paper described an approach to integrate multiple UML views, to build an integrated model, to generate test cases for this model, and to execute the model on these test cases. During model construction and validation, a variety of faults can be found. The test cases meet several of the test criteria defined in [6].

In doing this work, we realized that the integrated model has many more potential uses than just for testing. We were able to use it for Component definition and component qualification in Component-based Software Development [29]. With the instance table and the trace information, we were able to collect and compute a multitude of dynamic design metrics [30]. If we can find estimates for storage needs of class instantiations and time estimates for operation invocations, the existing instance and trace information can be used to estimate performance related to space and time. We have found the model quite flexible for a variety of evaluations beyond testing.

One of the limitations of our approach is that the OCL constraints need to be present in the design. In the future, we plan on using test oracles defined at a different level of abstraction, such as Use-Cases. In addition, we found in our case study that the lack of completeness in a design cannot be assessed using our approach. The design in the example did not contain detailed information concerning edge removal. When the design was implemented, this is where most of the faults occurred.

Obviously, we can only test information that exists in the diagrams we integrated. Other types of UML views provide further information and testing needs. Thus our future work will include evaluating which types of diagrams would be useful in evaluating a multi-view UML design, how to integrate those other types of diagrams in the current model, and how to generate and execute tests.

References

- [1] R. Binder, Testing Object-Oriented Systems Models, Patterns, and Tools, Addison Wesley, Reading, MA, 1999.
- [2] M. Fagan, Design and code inspections to reduce errors in program development, IBM Systems Journal (1987) 259–265.
- [3] T. Gilb, D. Graham, Software Inspection, Addison-Wesley, Reading, MA, 1993.
- [4] I. Traore, D. Aredo, Enhancing structured review with model-based verification, IEEE Transactions on Software Engineering (2004) 736–753.
- [5] P. Runeson, A. Andrews, Detection or isolation of defects? An experimental comparison of unit testing and code inspections, in: Proceedings of the IEEE Symposium on Software Reliability Engineering, 2003, pp. 3–14.
- [6] A. Andrews, R. France, S. Ghosh, G. Craig, Test adequacy criteria for uml design models, in: Journal of Software Testing, Verification, and Reliability, 2003, pp. 95–127.
- [7] Apache xml project, batik svg toolkit (2004). URL: <http://xml.apache.org/batik/>
- [8] E. Arisholm, L. Briand, A. Foyen, Dynamic coupling measurements for object-oriented software, IEEE Transactions on Software Engineering (2004) 491–506.
- [9] J. Offutt, A. Abdurazik, Generating test from uml specifications, in: Proceedings of the 2nd International Conference on the UML, 1999, pp. 416–429.
- [10] A. Abdurazik, J. Offutt, Using uml collaboration diagrams for static checking and test generation, in: Proceedings of the 3rd International Conference on the UML, 2000, pp. 383–395.
- [11] L. Briand, Y. Labiche, A uml-based approach to system testing, Software and Systems Modeling (2002) 10–42.
- [12] L. Briand, J. Cui, Y. Labiche, Towards automated for deriving test data from uml statecharts, in: Proceedings from the 6th International Conference on the UML, 2003, pp. 249–264.
- [13] M. Scheetz, A. von Mayrhauser, R. France, E. Dahlman, A.E. Howe, Generating test cases from an OO model with an ai planning system, in: Proceedings of the IEEE Symposium on Software Reliability Engineering, 1999, pp. 250–259.
- [14] M. Gogolla, J. Bohling, M. Richters, Validation of uml and ocl models by automatic snapshot generation, in: Proceedings from the 6th International Conference on the UML, 2003, pp. 265–279.
- [15] D. Riehle, S. Fraleigh, D. Bucka-Lassen, N. Omorogbe, The architecture of a UML virtual machine, in: Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01), ACM Press, 2001, pp. 327–341.
- [16] S. Mellor, M. Balcer, Executable UML: A Foundation for Model Driven Architecture, Addison Wesley Professional, 2002.
- [17] D. Harel, E. Gery, Executable object modeling with statecharts, Computer 30 (7) (1997) 31–42.
- [18] G. Engels, R. Huckling, S. Sauer, A. Wagner, UML Collaboration Diagrams and Their Transformations to Java, in: Proceedings of the 2nd International Conference on the UML, Fort Collins, CO, 1999, pp. 473–488.
- [19] Borland Software Corporation, Together 6.0. URL: <http://borland.com/together/>
- [20] U.A. Nickel, J. Niere, R.P. Wadsack, A. Zundorf, Roundtrip Engineering with FUJABA, in: Proceedings of the 2nd Workshop on Software-Engineering, Bad Honnef, Germany, 2000.
- [21] J. Offutt, A. Xiong, Y. Liu, Criteria for generating specification-based tests, in: Proceedings of Fifth IEEE International Conference on Engineering of Complex Computer Systems, 1999, pp. 119–129.
- [22] J. Chilenski, S. Miller, Applicability of modified condition decision coverage to software testing, Software Engineering Journal (1994) 193–200.
- [23] Use, a uml-based specification environment (2005). URL: <http://www.uni-bremen.de>.
- [24] K. Krause, R. Smith, M. Goodwin, Optimal software test planning through automated network analysis, in: Proceedings of the IEEE Symposium on Computer Software Reliability, 1973, pp. 18–22.
- [25] T. Ostrand, M. Balcer, The category-partition method for specifying and generating functional tests, Communications of the ACM (1988) 676–686.

- [26] E. Weyuker, T. Ostrand, Theories of program testing and the application of revealing subdomains, *IEEE Transactions on Software Engineering* (1980) 236–246.
- [27] M. Grindal, J. Offutt, S. Andler, Combination testing strategies: A survey, *Software Testing, Verification and Reliability* (2005) 167–199.
- [28] O.M. Group, Uml 2.0 specification (2006). URL <http://www.uml.org>.
- [29] A. O’Fallon, O. Pilskalns, A. Knight, A. Andrews, Defining and qualifying components in the design phase, in: *Proceedings of the Software Engineering and Knowledge Engineering Conference*, 2004, pp. 192–134.
- [30] O. Pilskalns, D. Williams, A. Andrews, Defining maintainable components in the design phase, in: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2005, pp. 49–58.