Automated Size Analysis for OCL*

Fang Yu Tevfik Bultan Erik Peterson
Computer Science Department
University of California
Santa Barbara, CA 93106, USA
{yuf,bultan,wombatty}@cs.ucsb.edu

ABSTRACT

An essential tool in object oriented modeling is the specification of cardinalities of associations between classes. In Object Constraint Language (OCL) such constraints are expressed as conditions on the sizes of the collections that correspond to associations. In this paper we present tools and techniques for automated verification of size properties of collection types in OCL. We automatically verify invariants related to the sizes of the collections of a class with respect to the pre and post-conditions of the methods of that class. Our approach is based on a *size abstraction* that abstracts away the contents of the collections, but preserves the constraints on their sizes. We implemented a tool which automates this abstraction by converting OCL expressions on collections to arithmetic expressions on their sizes. Following this translation, we employ an infinite state model checker, called Action Language Verifier (ALV), for *size analysis*. Size abstraction reduces the state space of the system and, hence, the cost of automated verification, and by focusing on size properties, enables us to use efficient, domain specific model checking techniques for automated verification. To demonstrate the effectiveness of our approach we conducted a case study on the OCL specification of the Java Card API [9]. The OCL specification of the Java Card API consists of 31 classes and 150 methods. Using our tool, we translated the OCL specification of each class to Action Language and verified the size properties using ALV. Verification with ALV took only a few seconds per class and we revealed errors in 26 out of the 150 method specifications.

Categories and Subject Descriptors: D.2.4 [Software/ Program Verification]: Class invariants, Model checking

General Terms: Verification

Keywords: size abstraction, size analysis, OCL

*This work is supported by NSF grants CCF-0341365 and CCF-0614002.

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.

1. INTRODUCTION

One of the most basic tools in object oriented modeling is the specification of cardinalities of associations. These specifications correspond to arithmetic constraints on the number of objects that can be associated with another object. We refer to these invariants as *size properties*, since they constrain the sizes of the associations. We believe that specification and analysis of size properties is important and promising for several reasons:

- Size properties are commonly used in object oriented models and do not require an extra specification effort from the software developers who use object oriented modeling languages.
- Violation of size properties is the cause of many types of security vulnerabilities such as buffer overflows.
- Effective automated verification of size properties can be achieved by first reducing the state space of a specification using abstractions that focus on size properties, and then by using domain specific and efficient automated verification techniques that target verification of systems characterized by arithmetic constraints.

We present tools and techniques for size analysis of Object Constraint Language (OCL) specifications. OCL [10, 13] is a specification language for describing constraints on object-oriented models. OCL is primarily used for specifying class invariants on fields and associations, and for specifying pre and post conditions of class methods.

Being one of the components of the Unified Modeling Language (UML) [11], OCL is a commonly used formal language for object oriented modeling, especially for expressing precise constraints that cannot be expressed using UML diagrams. Still, most software developers prefer to use UML as an informal specification language rather than adding more precision with the OCL constraints. We believe that effective automated verification and analysis of OCL/UML models can significantly improve the benefits of precise object oriented modeling, and, therefore, increase the use of precise object oriented models prior to implementation. Moreover, it is well known that identifying errors before the implementation stage is cost effective, hence an effective verification approach employed before the implementation stage is worthy of investigation.

Previous efforts in verification of OCL specifications have ranged from simulation of objection oriented models [12], to interactive verification via automated theorem prover support [1]. The approach taken in this paper is to use specialized techniques for automated verification of size properties rather than provide a general verification framework

for OCL which would require significant human interaction during verification. Our approach is more automated compared to approaches based on theorem proving [1], and provides stronger guarantees compared to approaches based on simulation [12] or bounded verification [7].

The OCL type system consists of basic types (i.e., booleans and integers), user-defined types (i.e., classes), and collection types. A *Collection* is an essential data type in object-oriented modeling, since an association between two classes corresponds to a relationship between one object and a collection of other objects. OCL supports three types of collections: *Set*, *Bag* and *Sequence*. In this paper, we present tools and techniques for verification of class invariants that express constraints on the sizes of these three collection types.

We present an automated abstraction technique for size analysis of OCL specifications. Size abstraction removes contents of collection types and replaces each collection in an OCL specification with an integer variable that represents the size of the collection. We define the size abstraction using an abstraction function that transforms OCL expressions by mapping expressions on collection types to expressions on integers. The abstract OCL specification conservatively approximates the behavior of the concrete specification which means that if the abstract system satisfies a size property it is guaranteed that the concrete system also satisfies the property.

After generating the abstract system, we use the Action Language Verifier (ALV) [14] to verify the size properties. ALV is an infinite state model checking tool that can verify or falsify (by generating counter-examples) a property using approximate fixpoint computations. Using ALV we are able to verify size properties of OCL specifications.

We applied our method to the Java Card API specification, which is the first open application programming interface for smart cards. Since smart cards are designed to be the next generation IDs, correctness of the Java Card API specification is extremely important. The OCL specification of the Java Card API [9] contains 31 classes with 150 methods. Using ALV we were able to verify or falsify all the classes in the Java Card API specification. For the falsified classes, we identified the errors in the corresponding method specifications by tracing the counter examples generated by ALV. This work consists of the following contributions:

- We propose a novel size abstraction and analysis for object-oriented systems. The experiments indicate our abstraction is precise enough to verify/falsify target systems, while coarse enough to perform complex model checking techniques efficiently.
- We implement an automatic translator from OCL to Action Language, in order to check the size properties of OCL specifications using ALV.
- We provide a detailed analysis of the Java Card API specification. We identify a set of falsified methods as well as the errors, and propose corrections for the pre-post conditions of these methods. To our best knowledge, this is the first attempt to successfully verify and falsify all classes and methods of Java Card API.

2. SIZE ABSTRACTION

Our goal is to automatically verify the size properties (i.e., properties about the sizes of the collection types) that are specified as class invariants with respect to pre and post-conditions of the methods in a class. We achieve this using a size abstraction that abstracts away the contents of collections but preserves the constraints about their sizes. In order to explain the size abstraction we first introduce a simple formal model for class specifications in OCL.

An OCL class specification $C = (P, A, M)$ consists of a set of properties P which represent the class invariants, a set of attributes (fields of the class) A, and a set of methods M, where each method is specified with one pre-condition expression and one post-condition expression. We formalize the semantics of an OCL class C as a transition system $\|C\| = (I, S, R)$ where S is the set of states, $R \subseteq S \times S$ is the transition relation and $I \subseteq S$ is the initial states, of the class C, respectively. We define the set of states of a class C as the Cartesian product of the sets of values that attributes of C can take

$$S = dom(type(a_1)) \times dom(type(a_2)) \times \ldots \times dom(type(a_{|A|}))$$

where $type(a_i) \in T$ denotes the type of the attribute $a_i \in A$, T denotes the set of OCL types, and $dom(type(a_i))$ denotes the set of values that the attribute a_i can take. I.e., each state $s \in S$ defines a valuation for all the attributes in A.

We define the set of initial states $I \subseteq S$ as all the states that satisfy all the class invariants[1], i.e., for all $p \in P$, $I \subseteq \|p\|$. We denote the set of OCL expressions as E, where $P \subseteq E$, and for any OCL expression $e \in E$ we use $\|e\|$ to denote the set of states for which e evaluates to true.

We assume that each method $m \in M$ contains a pre-condition expression $m.pre \in E$ and a post-condition expression $m.post \in E$ where $\|m.pre\| \subseteq S$ and $\|m.post\| \subseteq S \times S$. Note that, the post condition expressions are evaluated on pairs of states, one that corresponds to the state at the beginning of the method execution, and one that corresponds to the state at the end of the method execution. The transition relation of the class C is defined in terms of the transition relations of its methods as

$$R = \bigcup_{m \in M} R_m$$

and transition relation of each method m is defined in terms of its pre and post-conditions as

$$R_m = \{(s_1, s_2) \mid s_1 \in \|m.pre\| \ \wedge \ (s_1, s_2) \in \|m.post\|\}$$

Given a set of states $Q \subseteq S$, let $R(Q)$ denote the image of Q with respect to R, i.e.,

$$R(Q) = \{s_2 \mid \exists s_1 \in Q \ , \ (s_1, s_2) \in R\}$$

and let R^* denote the reflexive transitive closure of the transition relation R. We define the correctness of OCL class specifications as follows:

DEFINITION 1. *An OCL class specification $C = (P, A, M)$ with the corresponding transition system $\|C\| = (I, S, R)$ is* correct*, if and only if, all the reachable states of the class satisfy all the class invariants, i.e., for all $p \in P$, $R^*(I) \subseteq \|p\|$.*

In general, automatically checking correctness of OCL classes is a difficult problem due to unbounded types in OCL. Automated abstraction is one of the key techniques

[1]Alternatively, we can define the initial states based on the constructors of the class, however, this does not lead to any significant modification in our framework.

for achieving scalable verification. In order to achieve scalable verification of size properties we define a size abstraction which reduces the state space of an OCL specification by abstracting away the contents of collection types.

Given an OCL class specification $C = (P, A, M)$ with the corresponding transition system $\|C\| = (I, S, R)$, size abstraction generates an abstract class specification $\widehat{C} = (\widehat{P}, \widehat{A}, \widehat{M})$ and its corresponding transition system $\|\widehat{C}\| = (\widehat{I}, \widehat{S}, \widehat{R})$. We formalize the size abstraction using the abstraction function $\alpha_{\mathcal{C}}$ where given an OCL class specification C, $\alpha_{\mathcal{C}}(C) = \widehat{C}$ denotes its abstraction. The concrete and abstract class specifications satisfy the following properties:

$$\begin{array}{lll} \forall s \in S, & s \in I & \Rightarrow \ \widehat{s} \in \widehat{I} \\ \forall s_1, s_2 \in S, & (s_1, s_2) \in R & \Rightarrow \ (\widehat{s_1}, \widehat{s_2}) \in \widehat{R} \\ \forall s \in S, \forall p \in P, & \widehat{s} \in \|\widehat{p}\| & \Rightarrow \ s \in \|p\| \end{array}$$

Note that we use $\widehat{s}$ to denote the abstract states that corresponds to s and $\widehat{p}$ to denote the abstract invariant that corresponds to p. The mappings between the concrete and abstract elements are defined by the abstraction function $\alpha_{\mathcal{C}}$. Based on the above properties we have the following:

THEOREM 1. *Given a class specification C and its abstraction $\alpha_{\mathcal{C}}(C) = \widehat{C}$, if $\widehat{C}$ is correct, then C is correct.*

One can prove this property using an induction on the length of the sequences of states generated during the execution of a class and the properties of the abstraction function discussed above.

The formalization we gave above is a general formalization for abstraction based verification. The specifics of the size abstraction is in the implementation of the above abstraction function. We implement the abstraction function $\alpha_{\mathcal{C}}$ using two other abstraction functions which transform OCL types (T) and expressions (E):

- $\alpha_{\mathcal{T}} : T \rightarrow T$ where T is the set of all OCL types.
- $\alpha_{\mathcal{E}} : E \rightarrow E$ where E is the set of all OCL expressions.

The abstraction functions $\alpha_{\mathcal{T}}$ and $\alpha_{\mathcal{E}}$ eliminate the collection types and expressions on collection types from OCL types and expressions, respectively. We will discuss the abstraction functions $\alpha_{\mathcal{T}}$ and $\alpha_{\mathcal{E}}$ in the following two subsections.

2.1 Transforming OCL Types

The OCL type system consists of basic types, user-defined types, and collection types. There are three collections types in OCL: *set*, *bag* and *sequence*:

- A *set* is a collection that contains instances of a valid OCL type. A *Set* does not contain duplicate elements; any instance can be presented only once.
- A *bag* is like a *set*, but it can contain duplicate elements; that is, the same instance can occur in a bag more than once.
- A *sequence* is like a *bag* but the elements are ordered. Elements in a *bag* or *set* are not ordered.

Size abstraction abstracts collection types by converting them to integer type. After size abstraction, information about the contents of a collection variable is lost, and the collection variable is replaced with an integer variable. This integer variable represents the size of the collection it replaces. Hence, after the size abstraction, an OCL specification does not contain any collection types.

In addition to the collection types, there are four basic types in OCL: *boolean*, *integer*, *real*, and *string*. Size abstraction does not effect the basic types.

We define the set of OCL types as $T = \{boolean, integer, real, string, set, bag, sequence\}$. Then the abstraction function $\alpha_{\mathcal{T}}$ is defined as follows:

$$\begin{array}{lll} t \in \{boolean, integer, real, string\} & \Rightarrow & \alpha_{\mathcal{T}}(t) = t \\ t \in \{set, bag, sequence\} & \Rightarrow & \alpha_{\mathcal{T}}(t) = integer \end{array}$$

Given a class specification $C = (P, A, M)$ and the corresponding transition system $\|C\| = (I, S, R)$, abstraction function $\alpha_{\mathcal{C}}$ transforms attributes A to abstract attributes $\widehat{A}$ by changing the types of the attributes as follows: For all $a_i \in A$, $type(\widehat{a_i}) = \alpha_{\mathcal{T}}(type(a_i))$. The abstract state space $\widehat{S}$ corresponds to the Cartesian product of the domains of the abstract attributes in $\widehat{A}$.

Semantically, the abstraction mapping $\alpha_{\mathcal{C}}$ maps each attribute that is a collection, to an integer variable that denotes the size of that collection. Hence, the mapping between the concrete and abstract states is defined as follows: Given a state s and its abstraction $\widehat{s}$, and an attribute a that is a collection, the value of the abstracted attribute $\widehat{a}$ in $\widehat{s}$ is equal to the size of the collection a in s.

2.2 Transforming OCL Expressions

The key step in size abstraction is the transformation of OCL expressions. As with the type transformations, size abstraction does not modify the expressions on basic types. Size abstraction converts the expressions on collection types to boolean and integer expressions.

An OCL collection expression consists of constants, variables (i.e., class attributes) and OCL collection operations. Each OCL collection expression evaluates to one of the following types: *boolean*, *integer*, *set*, *bag*, *sequence*. Given an OCL collection expression $o \in E$ we use $type(o)$ to denote the type of the result of the expression (i.e., $type(o)$ denotes the type of the value that results from evaluating o).

During size abstraction of an expression, for each subexpression $o \in E$, we generate an *auxiliary variable* $o.v$ and a *size constraint* $o.c$ where $type(o.c) = boolean$. The type of the auxiliary variable $o.v$ is defined as $type(o.v) = \alpha_{\mathcal{T}}(type(o))$, i.e., expressions that evaluate to collection types are assigned auxiliary variables of type *integer*, whereas the expressions that evaluate to boolean or integer values are assigned auxiliary variables of the same type.

For expressions that evaluate to boolean or integer values (i.e., $type(o) \in \{boolean, integer\}$), auxiliary variable $o.v$ represents the result of evaluating the expression o and the constraint $o.c$ represents the constraints on the auxiliary variable $o.v$. I.e., if the constraint $o.c$ holds, then the value of $o.v$ corresponds to the result of evaluating the expression o.

For expressions that evaluate to collections (i.e., $type(o) \in \{set, bag, sequence\}$) $o.v$ represents the size of the resulting collection, i.e., if the constraint $o.c$ holds, then the value of $o.v$ corresponds to the size of the collection that is the result of evaluating the expression o. Note that, since we abstract away the contents of the collections, we cannot always pre-

cisely figure out the sizes of the resulting collections. For example, given an expression o in the form o_1->union(o_2), if $type(o_1) = type(o_2) \in \{bag, sequence\}$, then $o.c$ is

$$o.v = o_1.v + o_2.v \wedge o_1.c \wedge o_2.c$$

and $type(o.v) = integer$. However, for the same expression, if the arguments are sets (i.e., $type(o_1) = type(o_2) = set$), then the best we can do for $o.c$ is:

$$max(o_1.v, o_2.v) \leq o.v \leq o_1.v + o_2.v \wedge o_1.c \wedge o_2.c$$

which states that the size of the resulting collection (i.e., $o.v$) can take any value between the size of its largest argument and the addition of the sizes of both of its arguments.

Tables 1, 2, and 3, define how the constraint $o.c$ is computed for each OCL expression $o \in E$. Table 1 describes the case where $type(o) \in \{set, bag, sequence\}$. Table 2 describes the case where $type(o) = integer$, and Table 3 describes the case where $type(o) = boolean$.

We use the following shorthand notation for types in Tables 1, 2, and 3: i denotes type *integer*, b denotes type *boolean*, s denotes type *set*, and m denotes type *bag* or *sequence*. We use m to represent both *bag* and *sequence* types since the treatment of *bag* and *sequence* types are identical in our size abstraction. Note that, since size abstraction abstracts the contents of each collection, the ordering information among the elements in a collection are also lost, and, hence, the behavior of the *bag* and *sequence* types become equivalent after the size abstraction.

Table 1 shows the construction of constraints for expressions with $type(o) \in \{set, bag, sequence\}$. In the first column, we show the topmost OCL operator used in the expression. The second column shows the type of the result of the expression and its arguments, respectively. Finally, the third columns shows $o.c$, the constraint generated by the size abstraction. Note that for this type of expressions $o.v$ represents the size of the resulting collection and, hence, $type(o.v) = integer$.

Table 2 shows the construction of constraints for expressions with $type(o) = integer$ and $type(o.v) = integer$. Finally, Table 3 describes the construction of constraints for expressions with $type(o) = boolean$ and $type(o.v) = boolean$.

We define the abstraction function $\alpha_{\mathcal{E}}$ based on the size constraints defined in Tables 1, 2, and 3. Given a class specification C, we use the abstraction function $\alpha_{\mathcal{E}}$ to generate the abstract pre and post-conditions for the methods of the abstraction class specification $\alpha_{\mathcal{C}}$. Note that, pre and post-condition expressions are always boolean expressions, i.e., for all m, $type(m.pre) = type(m.post) = boolean$. Let $o \in E$ be an OCL expression where $type(o) = boolean$. In order to compute the abstraction of o we first compute the size constraint $o.c$ based on the rules given in Tables 1, 2, and 3. Let V denote the set of all the auxiliary variables that are introduced during the computation of $o.c$, and let $o.v$ be the auxiliary variable for the whole expression o, then we define $\alpha_{\mathcal{E}}(o)$ as follows:

$$\alpha_{\mathcal{E}}(o) = \exists V, o.v \wedge o.c$$

The abstraction function $\alpha_{\mathcal{E}}$ satisfies the following property:

$$\forall s \in S, s \in \|e\| \Rightarrow \widehat{s} \in \|\alpha_{\mathcal{E}}(e)\|$$

I.e., for any concrete state for which e evaluates to true, there exists a valuation of the auxiliary variables where, for the corresponding abstract state, $\alpha_{\mathcal{E}}(e)$ evaluates to true. This property holds since the existential quantification of the auxiliary variables allows the evaluation of the expression $o.v \wedge o.c$ in the most conservative manner (i.e., allowing all the nondeterminism to be resolved in a way to make expression $o.v \wedge o.c$ to evaluate to true when possible).

Given a class specification $C = (P, A, M)$, we compute the pre and post-condition of the methods of its abstraction $\alpha_{\mathcal{C}}(C) = (\widehat{P}, \widehat{A}, \widehat{M})$ using the expression abstraction function $\alpha_{\mathcal{E}}$. Let $m \in M$ be a concrete method and let $\widehat{m} \in \widehat{M}$ be the abstraction of method m, the pre and post-condition expressions for the abstract method $\widehat{m}$ are defined as follows:

$$\forall m \in M, \widehat{m}.pre = \alpha_{\mathcal{E}}(m.pre)$$
$$\forall m \in M, \widehat{m}.post = \alpha_{\mathcal{E}}(m.post)$$

Based on this definition and the above discussions we establish the following property mentioned earlier:

$$\forall s_1, s_2 \in S, (s_1, s_2) \in R \Rightarrow (\widehat{s_1}, \widehat{s_2}) \in \widehat{R}$$

i.e., the abstract transition relation is a conservative abstraction of the concrete transition relation.

So far, we described the abstraction of attributes and methods. The only remaining piece in a class specification is the set of class invariants. First, we define the initial states of the abstract transition system as all the abstract states that satisfy the following property: $\bigwedge_{p \in P} \alpha_{\mathcal{E}}(p)$. This definition establishes another abstraction property mentioned earlier:

$$\forall s \in S, s \in I \Rightarrow \widehat{s} \in \widehat{I}$$

i.e., the set of initial states in the abstract transition system is a conservative abstraction of the initial states in the concrete transition system.

The last abstraction property we need to establish is:

$$\forall s \in S, \forall p \in P, \widehat{s} \in \|\widehat{p}\| \Rightarrow s \in \|p\|$$

Note that we cannot establish this property by simply replacing a class invariant $p \in P$ with $\alpha_{\mathcal{E}}(p)$ in the abstract class specification. Since $\alpha_{\mathcal{E}}(p)$ is a conservative abstraction of p, this would violate the above property. We resolve this problem by computing a conservative abstraction of the negations of the invariant properties and then looking for property violations. Note that, since $\alpha_{\mathcal{E}}(\neg p)$ is a conservative abstraction of $\neg p$, we have the following property:

$$\forall s \in S, \forall p \in P, \widehat{s} \notin \|\alpha_{\mathcal{E}}(\neg p)\| \Rightarrow s \in \|p\|$$

which means that if an abstract state does not violate the abstraction of an invariant then we can conclude that the corresponding concrete state satisfies the invariant. In general, if we cannot find a violation of an invariant in the abstract transition system, then we can conclude that the concrete transition system does not violate the property either.

3. SIZE ANALYSIS

In order to perform size analysis, we translate the abstract OCL specifications to Action Language and then use Action Language Verifier (ALV) to check the class invariants. ALV consists of 1) a compiler that converts Action Language specifications into symbolic representations, and 2) an infinite-state symbolic model checker which verifies or falsifies (by generating counterexamples) CTL properties of Action Language specifications [14].

OCL Expression o	Type $type(o) : type(o_1)[type(o_2)][type(o_3)]$	Size Constraint $o.c$
o_1->including(e)	$s : s$ $m : m$	$o_1.v \leq o.v \leq o_1.v + 1 \wedge (o_1.v = 0 \Rightarrow o.v = 1) \wedge o_1.c$ $o.v = o_1.v + 1 \wedge o_1.c$
o_1->append(e)	$m : m$	$o.v = o_1.v + 1 \wedge o_1.c$
o_1->prepend(e)	$m : m$	$o.v = o_1.v + 1 \wedge o_1.c$
o_1->insertAt(e)	$m : m$	$o.v = o_1.v + 1 \wedge o_1.c$
o_1->excluding(e)	$s : s$ $m : m$	$max(0, o_1.v - 1) \leq o.v \leq o_1.v \wedge o_1.c$ $max(0, o_1.v - 1) \leq o.v \leq o_1.v \wedge o_1.c$
o_1->union(o_2)	$s : s, s$ $m : \{s, m\}, \{s, m\}$	$max(o_1.v, o_2.v) \leq o.v \leq o_1.v + o_2.v \wedge o_1.c \wedge o_2.c$ $o.v = o_1.v + o_2.v \wedge o_1.c \wedge o_2.c$
o_1->intersection(o_2)	$\{s, m\} : \{s, m\}, \{s, m\}$	$0 \leq o.v \leq min(o_1.v, o_2.v) \wedge o_1.c \wedge o_2.c$
o_1-o_2	$s : s, s$	$max(0.o_1.v - o_2.v) \leq o.v \leq o_1.v \wedge o_1.c \wedge o_2.c$
o_1->symmetricDifference(o_2)	$s : s, s$	$0 \leq o.v \leq o_1.v + o_2.v \wedge o_1.c \wedge o_2.c$
o_1->select($expr$)	$s : s$ $m : m$	$0 \leq o.v \leq o_1.v \wedge o_1.c$ $0 \leq o.v \leq o_1.v \wedge o_1.c$
o_1->reject($expr$)	$s : s$ $m : m$	$0 \leq o.v \leq o_1.v \wedge o_1.c$ $0 \leq o.v \leq o_1.v \wedge o_1.c$
o_1->collect($expr$)	$s : s$ $m : m$	$0 \leq o.v \leq o_1.v \wedge o_1.c$ $0 \leq o.v \leq o_1.v \wedge o_1.c$
o_1->subSequence(o_2, o_3)	$m : m, i, i$	$((o1.v \geq o_3.v \geq o_2.v \wedge o.v = o_3.v - o_2.v + 1) \vee$ $(\neg(o1.v \geq o_3.v \geq o_2.v) \wedge o.v = o_1.v)) \wedge o_1.c \wedge o_2.c \wedge o_3.c$
o_1->at(o_2)	$m : m, i$	$((o_1.v \geq o_2.v \geq 0 \wedge o.v = 1) \vee$ $(\neg(o_1.v \geq o_2.v \geq 0) \wedge o.v = o_1.v)) \wedge o_1.c \wedge o_2.c$
o_1->first	$s : m$	$o.v = 1 \wedge o_1.c$
o_1->last	$s : m$	$o.v = 1 \wedge o_1.c$
o_1->asSet	$s : s$ $s : m$	$o.v = o_1.v \wedge o_1.c$ $((o_1.v > 0 \wedge 1 \leq o.v \leq o_1.v) \vee (o_1.v = o.v = 0)) \wedge o_1.c$
o_1->asBag	$m : \{s, m\}$	$o.v = o_1.v \wedge o_1.c$
o_1->asSequence	$m : \{s, m\}$	$o.v = o_1.v \wedge o_1.c$

Table 1: Interpretation of OCL expressions that return collections.

OCL Expression o	Type $type(o) : type(o_1)$	Size Constraint $o.c$
o_1->size	$i : \{s, m\}$	$o.v = o_1.v \wedge o_1.c$
o_1->count(e)	$i : s$ $i : m$	$0 \leq o.v \leq 1 \wedge o_1.c$ $0 \leq o.v \leq o_1.v \wedge o_1.c$

Table 2: Interpretation of OCL expressions that return an integer value.

OCL Expression o	Type $type(o) : type(o_1)[type(o_2)]$	Size Constraint o.c
$o_1 = o_2$	$b : \{s, m\}, \{s, m\}$	$(o.v = false \vee o.v = (o_1.v = o_2.v)) \wedge o_1.c \wedge o_2.c$
o_1->includes(e)	$b : \{s, m\}$	$(o.v = false \vee o.v = (o_1.v \geq 1)) \wedge o_1.c$
o_1->exists($expr$)	$b : \{s, m\}$	$(o.v = false \vee o.v = (o_1.v \geq 1)) \wedge o_1.c$
o_1->forAll($expr$)	$b : \{s, m\}$	$(o.v = false \vee o.v = (o_1.v \geq 1)) \wedge o_1.c$
o_1->includesAll(o_2)	$b : \{s, m\}, \{s, m\}$	$(o.v = false \vee o.v = (o_1.v \geq o_2.v)) \wedge o_1.c \wedge o_2.c$
o_1->isEmpty	$b : \{s, m\}$	$(o.v = (o_1.v = 0)) \wedge o_1.c$
o_1->notEmpty	$b : \{s, m\}, \{s, m\}$	$(o.v = (o_1.v > 0)) \wedge o_1.c$
not(o_1)	$b : b$	$(o.v = \neg(o_1.v)) \wedge o_1.c$
o_1 and o_2	$b : b, b$	$(o.v = (o_1.v \wedge o_2.v)) \wedge o_1.c \wedge o_2.c$
o_1 or o_2	$b : b, b$	$(o.v = (o_1.v \vee o_2.v)) \wedge o_1.c \wedge o_2.c$

Table 3: Interpretation of OCL expressions that returns a boolean value.

An Action Language specification contains integer, Boolean and enumerated variables, parameterized integer constants, and a set of modules and actions which are composed using synchronous and asynchronous composition operators. Semantically, each Action Language module corresponds to a transition system. The variable declarations of a module define the module's state set. A module's *initial expression* defines the module's set of initial states. Each *action expression* corresponds to a single execution step and a *module expression* defines the transition relation of the module in terms of its actions and submodules using composition operators.

We translate abstract OCL specifications to Action Language by mapping method specifications in OCL to module specifications in Action Language. We implemented an automatic translator which parses the OCL class specification (in USE format [12]), performs the size abstraction and automatically emits an Action Language module which corresponds to the abstract OCL specification. We use ALV to check the correctness of the automatically generated Action Language specification.

ALV is a symbolic CTL model checker and uses the least and greatest fixpoint characterizations of CTL operators to compute the truth set of a given CTL property. It iteratively computes the fixpoints starting from the innermost temporal operator in the input CTL formula. Since Action Language allows specifications with unbounded integer variables, fixpoint computations are not guaranteed to converge. To achieve convergence, ALV uses conservative approximation techniques such as widening and bounded fixpoint computations [14].

Since ALV can only handle boolean, enumerated, and integer variables, in the current version of our size analysis tool we interpret only the boolean and integer basic types from OCL. However, with an appropriate back-end verification tool (that can support other basic types such as reals and strings) size analysis can easily be extended to other types.

However, boolean and integer types are the most important in terms of size analysis. It is important to support the boolean type since any bounded type (such as enumerated types or bounded strings) can be automatically mapped to boolean variables without increasing the state space of the specification. For verification of size properties it is crucial to handle integer variables during analysis since most size properties express a relationship between the size of a collection and the value of an integer variable (i.e., an integer attribute of a class). On the other hand, real variables are unlikely to be involved in size properties since the size of a collection is always a discrete value. As we mentioned above bounded string types can be handled using boolean variables. Moreover, size properties on strings can be verified by interpreting strings as a sequence of characters.

4. CASE STUDY: JAVA CARD API

In this section, we discuss the application of size analysis to verification of the OCL specification of the Java Card API. Java Card is a platform for developing applications that run on smart cards. Java Card API is a library that handles smart card features such as data units, identifiers, PIN codes, etc. In [9], Larsson and Mostowski give OCL specification of all the classes in the Java Card API based on the specification provided by Sun. The Java Card API specification contains 31 classes and 150 methods. We applied the size analysis techniques proposed in this paper to verification of the Java Card API specification.

The front end of our size analysis toolset uses the OCL parser from the USE tool [12]. Hence, in order to analyze the Java Card API specification, we first had to convert it to a form compatible with USE format. This required two types of modifications to the original Java Card API specification. First one involves using default OCL types and operations instead of Java types and utilities, and the second involves the specification of exceptions.

Assume that we want to compare a segment of two integer arrays `pin1` and `pin2`. In Java Card API, utility functions are defined to support element comparison. Hence, we can specify a method for array comparison with the following post-condition:

```
Util.arrayCompare(pin1,0,pin2,offset,length)==0.
```

Alternatively, instead of using Java types and utilities (which may or may not have OCL specifications of their own), we can use matching OCL types and operations. Since our size analysis tool knows how to interpret default OCL types and operations, using them instead of Java types and utilities can add precision to the size analysis (especially for the cases where OCL specifications of Java utilities are not available). For example, instead of invoking utility functions in the above post-condition, we can treat arrays as sequences and replace the above post-condition with the following one:

```
pin1->subSequence(0,length)=
pin2->subSequence(offset,offset+length)
```

Our size analysis tool would automatically interpret this post-condition based on OCL semantics without needing any other specification. Whenever we found cases like this we modified the Java Card API specification by replacing the Java types and utilities with the equivalent OCL types and operations in order to improve the precision of our analysis.

The second modification we made to the Java Card API specification is about the exception handling. Exception handling is an essential part of Java programs. The typical behavior of a Java method can be specified using a *precondition*, a *postcondition*, and a set of condition, exception pairs: (*condition_1*, *exception_1*), (*condition_2*, *exception_2*) . . . , (*condition_n*, *exception_n*). The behavior of the method is then defined as follows: If the *precondition* holds at the beginning of the method invocation, then the method either terminates normally and the *postcondition* holds, or it terminates abruptly by throwing some listed exception and the corresponding exception condition holds.

In order to represent this behavior in OCL, for each class which may throw exceptions, we induce an auxiliary attribute, named *thrownExceptions*, which is a bag of type exception. Then the exception handling semantics can be captured by the following OCL specification:

```
context SomeClass::someMethod()
pre: <precondition>
post: (thrownExceptions = thrownExceptions@pre
         and <postcondition>)
      or (thrownExceptions = thrownExceptions@pre
         ->including(e: exception |
         e.oclType = Exception_1)
         and <condition_1>)
      ...
      or (thrownExceptions = thrownExceptions@pre
         ->including (e: exception|e.oclType = Exception_n)
         and <condition_n>)
```

Considering that the goal of the size analysis is to verify size properties, we can further simplify the post condition as follows

```
post: (thrownExceptions=thrownExceptions@pre
        and <postcondition>)
      or((thrownExceptions->size()=
        thrownExceptions@pre->size()+1)
        and(<condition_1> or...or <condition_n>))
```

This specification keeps track of the number of exceptions thrown. One can specify that a class invariant holds when no exception is thrown as:
`thrownExceptions->isEmpty() implies`"`invariant`".

Correct interpretation of the exception handling semantics is important for verification of such properties.

Below, we give the OCL specification of the update method of the PIN class in the Java Card API 2.1.1. The OCL specification is based on the specification in [9], updated with the modifications discussed above.

```
context OwnerPIN::update(newpin: Sequence(Integer),
offset:Integer, length:Integer, e:Integer)

pre: newpin->notEmpty()
     and offset >= 0
     and offset+length <= newpin->size()
     and length >= 0
post:(
1:     thrownExceptions=thrownExceptions@pre
2:     and self.pin->subSequence(0,length)
          =newpin->subSequence(offset, offset+length)
      )or(
3:     thrownExceptions=thrownExceptions@pre->including(e)
4:   and length > self.maxPINSize
      )or(
5:     thrownExceptions=thrownExceptions@pre->including(e)
6:     and systemInstance->notEmpty()
      )
```

The corresponding automatically generated Action Language specification is as follows:

```
module updateMod()
  updateMod:
pre: newpin > 0 and offset >= 0
        and length + offset <= newpin
        and length >= 0 and
post:(
1:   (thrownExceptions' = thrownExceptions
2:   and tmp8 = tmp9
     and ((tmp8 = length - 0 + 1 and pin' >= length)
         or (tmp8 = pin' and pin' < length))
     and ((tmp9 = length + offset - offset + 1
           and newpin' >= length + offset)
         or (tmp9 = newpin'
           and newpin'< length + offset))
     ) or (
3:   thrownExceptions' = tmp10
     and tmp10 = thrownExceptions + 1
4:   and length > maxPINSize'
     ) or (
5:   thrownExceptions' = tmp11
     and tmp11 = thrownExceptions + 1
6:   and systemInstance' > 0) );
endmodule
```

For each method, we generate a corresponding Action Language module based on the pre and post condition of the method. In the example above we labeled the Action Language and OCL specifications to indicate the parts that correspond to each other. Note that the Action Language module above corresponds the abstraction of the OCL specification based on the size abstraction we defined earlier.

5. EXPERIMENTS

Using our size analysis toolset, we checked all classes with non-trivial OCL specifications in the Java Card API specification [9]. These include 31 classes and 150 methods contained in `javacard.framework`, `javacard.security`, `javacard.framework.services`, and `javacardx.crypto` packages.

In the first phase of the verification we try to verify that a class is correct. If the verification phase fails then we move to the falsification phase and look for counter-examples. In the following, we discuss the verification phase.

We take the conjunction of all the class invariants and generate a single invariant property that implies all the class invariants. Let us call this invariant property p. We check that the property "p holds in all reachable states of the class". For the classes that throw exceptions, we check the property that "p holds in all reachable states where no exception has been thrown", i.e., we check that "`thrownExceptions->isEmpty()` $\Rightarrow p$" is an invariant.

We generate an Action Language module for each method in a class and then compose these modules asynchronously to obtain the class specification. This means that when we check a class using ALV, we are checking all states that can be reached by all possible interleavings of all the class methods.

The verification results are shown in Table 5. ALV verified 26 out of the 31 classes and falsified the other 5 classes. For those classes that are verified, the class invariant holds with respect to pre and post-conditions of the methods. I.e., if one implements these methods according to pre and post-conditions in the OCL specifications, the class invariants are guaranteed to hold. For the five classes that were not verified, we conducted the second phase of our analysis and looked for counter-example behaviors.

For these classes that were falsified we were able to generate a counter-example behavior. Recall that since we over approximate the behavior of a class, the falsified classes may be correct in the concrete system, i.e., the generated counter example may be spurious. After tracing the counter example manually, we determined that none of the reported counter-examples were spurious. This demonstrates that the size abstraction is relatively precise and does not introduce too many spurious behaviors. We will discuss the falsification phase and discuss the errors we found in the Java Card API specification in the next section.

ALV performs very well with its default parameters while verifying most classes. This indicates that the size abstraction is effective in reducing the size of the state space and generates compact modules that can be analyzed efficiently. However, recall that infinite state model checking is undecidable in general and ALV's fixpoint computations are not guaranteed to converge. In fact we observed this while checking `DSAKey`. ALV did not terminate with its default parameters. ALV can conservatively approximate a fixpoint using the widening technique. When we used the widening technique provided by the ALV, the fixpoint converged and ALV was able to verify the correctness of the `DSAKey` class.

We mentioned above that we identified five classes of Java Card API that might contain errors. In order to identify the

Class	M	R	tran+ver	Mem
AID	7	F	0.06s+0.03s	2273k
		Y	0.06s+0.06s	2322k
APDU	14	V	0.38s+0.12s	18248k
Applet	7	V	0.06s+0.01s	1532k
CardException	4	V	0s+0s	406k
CardRuntimeException	4	V	0s+0s	323k
Cipher	6	V	0.02 s+2.05s	2998k
CryptoException	2	V	0s+0s	135k
DESKey	2	V	0.01s+0.01s	422k
Dispatcher	5	V	0.01s+0.01s	635k
DSAKey	6	V	0.06s+6.2s	7840k
DSAPrivateKey	8	V	0.11s+2.61s	4170k
DSAPublicKey	8	V	0.11s+2.62s	4170k
CardRemoteObject	2	V	0s+0s	135k
JCSystem	11	F	1.08s+0.15s	18571k
		Y	1.09s+0.19s	18571k
KeyBuilder	1	V	0.01s+0s	135k
KeyEncryption	2	F	0.01s+0s	118k
		Y	0s+0s	131k
KeyPair	5	V	0s+0s	1044k
MessageDigest	3	V	0.01s+0s	397k
OwnerPIN	9	F	0.08s+0.52s	7725k
		Y	0.1s+0.4s	5091k
PIN	4	F	0.03s+0.33s	5693k
		Y	0.03s+0.23s	3670k
PINException	2	V	0.01s+0s	135k
RandomData	3	V	0s+0s	401k
RMIService	2	V	0s+0s	414k
RandomData	3	V	0s+0s	401k
RSAPrivateCrtKey	10	V	0.2s+7.31s	6087k
RSAPrivateKey	4	V	0.03s+0.05s	1008k
RSAPublicKey	4	V	0.03s+0.05s	1008k
SecurityService	3	V	0.01s+0s	520k
Service	3	V	0.01s+0s	270k
TransactionException	3	V	0s+0s	135k
UserException	3	V	0s+0s	270k

Table 4: Verification of the Java Card API OCL specification. M: No. of methods, R: Result (F:Falsify/V:Verify), tran: Translation time, ver: Verification Time. Y: Found a counter example

errors in these classes we analyzed their methods separately. For each method, we generated an Action Language module that can only execute that method, and checked if the class invariants can be violated by that method. For the methods we were able to falsify, we traced the generated counter example, identified the possible bugs, and fixed the pre and post condition of the method whenever we can. In the following subsections, we first summarize the errors we identified in these methods, and then detail our falsification analysis for each buggy method.

Identified errors.

The errors we found in the Java Card API specification fall into three categories: 1) Frame Error (FE), 2) Unsound Implication (UI), and 3) Design Error (DE).

Frame Error (FE) If a post-condition does not specify the next value of a variable, this unconstrained variable can take any value in the next step. If some variable that appears in an invariant is an unconstrained variable, it can take a value that violates the invariant. Frame errors can be fixed by appending $\bigwedge_i v_i = v_i@pre$ to the post-condition, where each v_i is an unconstrained variable. This error was identified mostly in our falsified classes.

Unsound Implication (UI) One common used structure in post conditions is implication conjunction, i.e., $\bigwedge_i (B_i$ `implies` $B_i')$. Unsound implication happens if $\bigvee_i B_i \neq true$. In this case, the state satisfying $\bigwedge_i \neg B_i$ is allowed to make unexpected transitions that can violate the invariant. Unsound implication can be fixed by adding an extra implication to specify the behavior for the states which satisfy $\bigwedge_i \neg B_i$.

Design Error (DE) We put all the other errors in this category as design errors. One common design error is the use of an unchecked method parameter to define the next value of a variable that appears in an invariant. This may make the updated variable take unexpected values that can violate the invariant. This error may be fixed by restricting the values of the used parameter. We manually fixed errors in this category for each case.

In the following sections, we discuss our falsification analysis for each falsified class in Java Card API specifications respectively.

AID.

This class encapsulates the Application Identifier (AID) associated with an applet. An AID is defined in ISO 7816-5 to be a sequence of bytes between 5 and 16 bytes in length. In the OCL specification, an AID is treated as a sequence object, and the class invariant is

```
self.thrownExceptions->isEmpty() implies
(self.theAID->size() >= 5 and self.theAID->size() <= 16).
```

We separately checked each method in AID without any modification. The method `AID` was verified while the other six methods were falsified. We then inserted frame constraints for undefined variables, and checked each method again. The result is shown in table 5. Three methods: `equal`, `getPartialBytes`, and `RIDEquals` were verified after fixing frame errors, while the other three methods were still falsified. We attempted to identify errors of these three methods by tracing the generated counter examples manually. For the method `equals`, its post condition is an implication structure, B `implies` B', where B is as follows:

```
bArray->isEmpty()
or(offset>= 0 and
  length>= 0 and
  offset+length <= bArray->size() and
  offset+length >= 1)
```

Since $B \neq true$, this raises an unsound implication error. The counter example indicates that for some state that satisfies $\neg B$, `theAID` may have an arbitrary size that violates the invariant. A similar error happens in the method `partialEquals`.

For the method `getBytes`, a nontrivial design error was found. In one part of its post condition, `theAID` is set equal to a subsequence of `dest` with the size of `theAID`, which is specified as follows:

```
self.theAID = dest->
subSequence(offset, offset+self.theAID->size())
```

Note that this specification defines `theAID` with its own size, and results in `theAID` having an arbitrary size. The correct post condition can be specified as follows:

```
self.theAID = dest->
subSequence(offset, offset+self.theAID@pre->size())
```

Method	Err.	R	trans+ver.	Mem
AID	None	V	0.02+0.09s	2523k
equal	(FE)	V	0s+0s	299k
equals	UI	F	0.02s+0.02	610k
getBytes	DE	F	0.02s+0.02s	676k
getPartialBytes	(FE)	V	0.01s+0.02s	418k
partialEquals	UI	F	0.02s+0.01s	545k
RIDEquals	(FE)	V	0s+0s	324k

Table 5: Checking Methods in AID Class with Frame Constraints Inserted. None: the method is verified without any modification. (FE): Frame error is fixed in this case.

This postcondition enforces `theAID` and `dest` having the same size as `theAID` in the previous state.

ALV performed quite well while checking all methods in this class. This indicates that our size analysis generates compact modules that can be analyzed efficiently. On the other hand, the fact that we were able to discover nontrivial errors using ALV shows that our size analysis is precise enough to check considerable part of system correctness.

JCSystem.

The `JCSystem` class is the only system class in Java Card API. It includes methods to control applet execution, resource management, atomic transaction management, etc. in the Java Card environment. The class invariant is

```
self.thrownExceptions ->isEmpty() implies
(self.transactionDepth = 0 or self.transactionDepth = 1).
```

We checked each method without any modification. `abortTransaction` was verified while the other ten methods were falsified in our first attempt. We then inserted frame constraints for the ten falsified methods. The falsification results are shown in Table 6. Nine methods were verified after inserting frame constraints, while `getAppletSharableObjectInterface` remained falsified. We identified an unsound implication error by tracing the generated counter example.

ALV encounters some problems while checking `MakeTransientBooleanArray`, `MakeTransientByteArray`, `MakeTransientObjectArray`, and `MakeTransientShortArray` after inserting frame constraints. The fixpoint computation did not converge with the default option or widening approximation. ALV allows users to change widening seeds such that the widening operator is used only after a certain number of steps. For our cases, the fixpoint computation converged once we started widening after the third iteration (with the option -A -W 3), and all these four methods were successfully verified.

KeyEncryption.

`KeyEncryption` is an interface in `javacardx.crypto` package, which defines the methods used to enable encrypted key data access to a key implementation. In this simple interface, there is no exception thrown and the class invariant is `self.cipher` ≥ 0. Both methods were falsified, and `getKeyCipherMod` was verified after inserting frame constraints. The result is shown in table 7. For the method `setKeyCipherMod`, we identified a design error. In its postcondition, a parameter `KeyCipher` is used to define `cipheri`. This results that `cipher` may have arbitrary values and violates the invariant. A fixed post condition can be

Method	Err.	R	trans+ver.	Mem
abortTransaction	None	V	0s+0.01s	266k
beginTransaction	(FE)	V	0s+0.06s	266k
commitTransaction	(FE)	V	0s+0.01s	266k
getAppletSharable-ObjectInterface	UI	F	0.06s+0.03s	815k
getTransactionDepth	(FE)	V	0s+0s	270k
isTransient	(FE)	V	0s+0.01s	270k
lookupAID	(FE)	V	0.03s+0.07s	1028k
MakeTransientBooleanArray	(FE)	V	0.09s+1.61s	1147k
MakeTransientByteArray	(FE)	V	0.06s+1.73s	1487k
MakeTransientObjectArray	(FE)	V	0.06s+1.72s	1495k
MakeTransientShortArray	(FE)	V	0.07s+1.72s	950k

Table 6: Checking Methods in JCSystem Class with Frame Constraints Inserted

Method	Err.	R	trans+ver.	Mem
getKeyCipherMod	(FE)	V	0s+0s	115k
setKeyCipherMod	DE	F	0s+0s	123k

Table 7: Checking Methods in KeyEncryption Class with Frame Constraints Inserted.

```
keyCipher>=0 and self.cipher = keyCipher.
```

OwnerPIN and PIN.

`OwnerPIN` is a class defined in the `javacard.framework` package, which implements the PIN interface. The class provides methods for performing PIN operations, such as updating or verifying a PIN. In our OCL specification, the class invariant is specified as follows:

```
 this.thrownExceptions->isEmpty() implies
(this.maxPINSize > 0 and this.maxTries > 0
 and this.triesRemaining>= 0
 and this.triesRemaining<=this.maxTries
 and this.pin->notEmpty()}
 and this.pin->size()<=this.maxPINSize)
```

In our first attempt, `reset` was verified while the other eight methods were falsified. We then inserted frame constraints for these falsified methods, and checked each method again. The result is shown in table 8. Seven methods: `getValidatedFlag`, `setValidatedFlagare`, `OwnerPIN`, `update`, `resetAndUnblock`, `getTriesRemaining`, `isValidated` were verified after inserting frame constraints, while `check` remained falsified. Its post condition is an implication structure where B is `self.triesRemaining` ≥ 0 and an unsound implication error was identified.

Method	Err.	R	trans+ver.	Mem
getValidatedFlag	(FE)	V	0s+0.01s	385k
setValidatedFlag	(FE)	V	0.01s+0s	381k
OwnerPIN	(FE)	V	0.01s+0.05s	590k
update	(FE)	V	0.02s+0.7s	782k
resetAndUnblock	(FE)	V	0s+0.01s	381k
getTriesRemaining	(FE)	V	0.01s+0s	385k
isValidated	(FE)	V	0.01s+0s	381k
reset	None	V	0.01s+0s	381k
check	UI	F	0.03s+0.06s	877k

Table 8: Checking Methods in OwnerPIN Class with Frame Constraints Inserted

6. RELATED WORK

One attempt to check the correctness of UML/OCL models is UML-based Specification Environment (USE) [12, 5]. USE provides an environment where users can simulate the behavior of UML models and check OCL invariants and pre and post conditions during the simulation. One disadvantage of USE is lack of support for automatically guided simulation, and, hence, one can only cover a small portion of system behaviors with USE. In contrast to this type of simulation-based validation, we apply automated verification techniques to guarantee the correctness of UML/OCL models.

Another related work is the "Key" system [1, 9], which adopts interactive theorem proving to verify object-oriented software models. The OCL specification of the Java Card API was developed as a part of this project [9]. One important difference between our approach and the techniques based on interactive theorem proving is the level of automation. As our experiments demonstrate verification with ALV is very efficient and the only user interaction involves choosing some heuristics or parameters for verification. Another advantage of verification techniques based on model checking is the ability to generate counter-example behaviors. Our experiments demonstrate that this feature is essential for identifying errors in a specification.

Alloy [7] is another design language used to specify object oriented systems. It has formal syntax and semantics, and an automatic analyzer developed to facilitate formal verification of Alloy modules. Alloy analyzer analyzes concrete object-oriented systems and reduces the complexity of verification by bounding the number of instances of each object. Alloy analyzer reduces the first order logic reasoning over bounded domains to the boolean satisfiability problem and incorporates the modern SAT solver. Such an approach is not likely to be effective for verification of size properties for a couple of reasons: 1) errors in size properties may not be identified within a small bound (for example violations such as buffer overflow), 2) translation of numeric relationships among size variables to a boolean encoding may not be efficient. Note that, compared to the bounded verification approach used in Alloy which cannot guarantee correctness, our size analysis approach is able to guarantee correctness for size properties.

Various methods to verify size properties of systems had been proposed. However, most work in this area is in the area of programming languages. Hughes et al. [6] develop a sound semantic model of size types to verify the properties of reactive systems. They showed that various essential program properties, such as function productivity, memory leaks, array bounds and the termination of some restricted functions, could be reduced to type checking problems. The advantages of type analysis include a) the soundness proof and b) the efficient type checking algorithm. Hughes' work is the first paper using size types to analyze programs. Chin et al. [2] expand size types to verify object-oriented languages by annotations. They annotate an abstract data type for each object with size invariants, which can then be used to infer size properties among objects. An intermediate language, called OIMP, is proposed to capture the size information of real programs, such as C++/Java, via an annotated type system.

These earlier efforts in size analysis focus on programming languages and use type checking to establish the correctness of size properties. We, on the other hand, adopt automated abstraction and model checking techniques to verify size properties in UML/OCL models. We employ ALV [14] as the back-end model checker, which incorporates BDD and polyhedra/automata representations to verify systems with integer variables. Although ALV may not be able to give a conclusive answer when fixpoints fail to converge and the approximations are not precise enough, this did not happen in our case study and all specifications were successfully verified or falsified using ALV.

7. CONCLUSIONS

We presented tools and techniques for size analysis of OCL specifications. In order to achieve efficient size analysis we proposed a size abstraction. Size abstraction removes the collection types from the OCL specification and replaces them with integer variables that represent the sizes of the collections. To demonstrate the effectiveness of our approach, we conducted a case study on the OCL specification of the Java Card API [9]. All 31 class specifications were either verified or falsified, and various errors were identified in 26 out of the 150 method specifications.

8. REFERENCES

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hahnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. "The KeY Tool." *Software and Systems Modeling*, vol. 4, no. 1, pp. 32–54, 2005.

[2] Wei-Ngan Chin, Siau-Cheng Khoo, Shengchao Qin, Corneliu Popeea, Huu Hai Nguyen. "Verifying Safety Policies with Size Properties and Alias Controls." In *Proc. ICSE '05*, St. Louis, MO, USA, pp. 186–195, 2005.

[3] Andy Evans, Robert B. France, Ana M. D. Moreira, Bernhard Rumpe. "Using Alloy and UML/OCL to Specify Run-Time Configuration Management: A Case Study." In *Practical UML-Based Rigorous Development Methods UML01*, Oct 2001, Toronto, Canada.

[4] Jonathan Edwards, Daniel Jackson, Technology Emina Torlak. "A Type System for Object Models." In *Proc. of FSE '04*, Newport Beach, CA, pp. 189-199, 2004.

[5] Martin Gogolla, Jorn Bohling, and Mark Richters. "Validation of UML and OCL Models by Automatic Snapshot Generation." In *Proc. UML 2003*. Springer, Berlin, LNCS 2863, 2003.

[6] John Hughes, Lars Pareto, Amr Sabry. "Proving the Correctness of Reactive Systems Using Sized Types." In *Proc. POPL '96*, pp. 410-423, 1996.

[7] Daniel Jackson. "Alloy: A Lightweight Object Modelling Notation." *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 2, pp. 256-290, 2002.

[8] Viktor Kuncak and Daniel Jackson. "Relational Analysis of Algebraic Datatypes." In *Proc. ESEC/FSE 2005*, Lisbon, Portugal, September 5-9, 2005.

[9] Daniel Larsson and Wojciech Mostowski. "Specifying Java Card API in OCL." OCL 2.0 Workshop at UML 2003, San Francisco, *Electronic Notes in Theoretical Computer Science*, vol. 102 pp. 3-19, 2004,

[10] OMG. "Object Constraint Language Specification." In OMG Unified Modeling Language Specification, Version 1.3, June 1999.

[11] OMG. "OMG Unified Modeling Language Specification, Version 1.3." Object Management Group, Inc., Framingham, Mass., Internet:http://www.omg.org, 1999.

[12] Mark Richters and Martin Gogolla. "Validating UML models and OCL constraints." In *Proc. UML 2000*. Springer, York, UK, LNCS 1939, 2000.

[13] Jos Warmer and Anneke Kleppe. "The Object Constraint Language: Precise Modeling with UML." Addison-Wesley, 1998.

[14] Tuba Yavuz-Kahveci, Constantinos Bartzis, and Tevfik Bultan. "Action Language Verifier, Extended." In *Proc. CAV '05*, LNCS 3576, pp. 413-427, 2005.