Verifying OCL Operational Contracts via SMT-based Synthesising

Hao Wu and Joseph Timoney
Computer Science Department,
Maynooth University
haowu@cs.nuim.ie and joseph.timoney@mu.ie

Keywords: OCL Synthesis, Call Sequence, SMT

Abstract: The set of operational contracts written in the Object Constraint Language can be used to describe the behaviour of a system. These contracts are specified as pre/post conditions to constrain inputs and outputs of operation calls defined in a UML class diagram. Hence, a sequence of operation calls conforming to pre/postconditions is crucial to analyse, verify and understand the behaviour of a system. In this paper, we present a new technique for synthesising property-based call sequences from a set of operational contracts. This technique works by reducing a synthesis problem to a satisfiability modulo theories (SMT) problem. We distinguish our technique from existing approaches by introducing a novel encoding that supports high levels of expressiveness, flexibility and performance. This encoding not only allows us to synthesise call sequences at a much larger scale but also maintains high performance. The evaluation results show that our technique is effective and scales reasonably well.

1 Introduction

UML models are central to Model Driven Engineering (MDE) based software development. They provide software engineers with different ways of visualising structural and behavioural aspects of a system. For example, UML class diagrams are used to depict entities, attributes and relationships in a system. On the other hand, Object Constraint Language (OCL) is designed to describe formal rules or queries that cannot be captured by UML models. For example, a pre/postcondition written in OCL can constrain the inputs and outputs of an operation call defined in a class diagram. The combination of UML and OCL is widely used for modelling a software system not only because of their expressiveness but also formality. Hence, the correctness of models annotated with OCL are crucial for MDE based software development. However, the tasks of verifying UML along with OCL constraints remain a challenge in the modelling community.

Though numerous approaches have been proposed to tackle this challenge (Berardi et al., 2005; Büttner et al., 2012; Cabot et al., 2014; Wille et al., 2012), many of them focus on verifying UML class diagrams annotated with a set of OCL class invariants (structural aspects of a system). In general, OCL operational contracts (behavioural aspects of a system) are specified as pre/post conditions of an operation call. These pre/postconditions capture constraints over a set of call sequences and system states. For example, finding a sequence with respect to pre/postconditions is the same as checking whether a valid system state can be derived. In this paper, we introduce a novel technique that allows us to verify OCL operational contracts via synthesising property-based call sequences. Our technique works by reducing it to a satisfiability modulo theories (SMT) problem. More specifically, our synthesis technique provides a high level of expressiveness and flexibility for not only data types but also property-based synthesis.

In general, our synthesis constraints are in first-order form and this gives us several advantages; (1) We can perform efficient satisfiability checks by taking full advantages of SMT first-order reasoning capabilities. (2) It allows us to specify queries and properties over unrolled states without regenerating the synthesis constraints.

Contributions. In particular, we first present our initial idea in (Wu, 2019) and we now show our full technical details including detailed SMT-encodings in this paper. Hence, our contributions of this paper are summarized as follows:

1. We design our synthesis constraints to be expressive enough so that users can use quantified invariants and queries over a collection of transitions (Section 4).

2. We then introduce a set of different properties based on an intermediate representation (Section 5). To optimise overall formula size and performance, we also introduce three simplification rules (Section 6).
3. We evaluate our technique on a benchmark but with much larger sequences, and show that our technique outperforms existing approaches in terms of expressiveness, flexibility and performance (Section 7).

2 A Running Example

In this section, we introduce a real world scenario that a software engineer uses a UML class diagram (as shown in Figure 1) to model an online shopping system. This example will be used throughout this paper to illustrate our approach. Except for the attributes and classes depicted in Figure 1. This engineer also defines 5 operation calls to further constrain the model. These operation calls allow users to change their shoppingcart, checkout, confirm and cancel[1] their orders. For each operation call, the corresponding pre or postcondition(s) are also defined. For example, the *checkout* operation requires a *shoppingcart* to be non-empty before a user can submit an order.

In general, it is very easy to introduce a mistake or omit a condition when writing a specification. Unfortunately, the class diagram presented in Figure 1 also contains a mistake. The operational contracts here allow a scenario that after an order is placed and confirmed, users can still modify their orders. In fact, this mistake is quite difficult to be identified. However, with the help of visualised call sequences this mistake becomes obvious. It can now be spotted in the synthesised call sequence shown in Figure 2. One way to fix this mistake is to add $self.cart.confirmed = false$ as precondition for both $addItem$ and $removeItem$.

3 Background

Before presenting our techniques, we first review the background here. Formally, synthesising a call sequence $G_{\langle k,P\rangle}$ from OCL operational contracts is defined in (Wu, 2019), and it is shown as follows:

$$G_{\langle k,P\rangle} \stackrel{\text{def}}{=} \Phi \wedge \Psi$$

where k is the length of the sequence, P describes a set of properties, Φ describes synthesis constraints and Ψ is a set of property constraints. Visually, a sequence $G_{\langle k,P\rangle}$ can be viewed as follows:

$$G_{\langle k,P\rangle} : S_0 \xrightarrow{\delta_1} S_1 \xrightarrow{\delta_2} S_2 \xrightarrow{\delta_3} S_3 \cdots \xrightarrow{\delta_k} S_k$$

where each δ_i in a sequence represents an operation call selected from the set of operation calls (denoted as Δ), and each S_j is a (system) state that is triggered by an δ_i in the previous state S_{j-1}. S_0 is the initial state and S_k is the final state that is derived by the kth operation call (δ_k) in a sequence [2]. In each state S_j, there is *exactly one* operation call δ_i invoked to make the transition to the next state S_{j+1}. Therefore, this transition captures all possible sequences that have a length of k.

For example, we can define a synthesised call sequence $G_{\langle 6,P\rangle}$ (with a length of 6) for our online shopping model in Figure 1 with the following properties:

- $addItem$ and $removeItem$ cannot be called after $confirmOrder$.
- each operation must be called at least once.
- $checkout$ and $confirmOrder$ must be part of the sequence.

To check whether there exists a counter-example, we negate Ψ and discover a sequence (shown in Figure 2) showing a scenario that a customer may add more items after confirming an order.

4 Basic Synthesis

Our synthesis constraints (Φ) consists of a series of formulas $\phi_{\langle i,j\rangle}$ and each encodes a δ_i (ith operation call) including pre/post conditions at state j.

Since a pre/post condition may refer to a set of specific features in a class diagram such as attributes and collections, our synthesis constraints also encode those features. In fact, the encoding of a feature is similar to the encoding described in (Wu, 2019). To be precise, we use a feature function to encode an attribute or a collection data type at a specific state. For example, $F_{id}(m,2) = 1001$ means that a feature function F_{id} sets the attribute id of an object m [3] to 1001 at state 2.

Quantified Invariants. To encode a class invariant, we use a quantified formula. This encoding allows us to reduce the number of individual unrolled class invariants in each state to a single quantified formula. For example, the class invariant in Figure 1 is represented using the following formula:

[1] Due to space restrictions, the operation cancel is not shown here.

[2] Note that δ_k is called in S_{k-1} and triggers S_k. Thus, a call sequence of length of k creates a total of $k+1$ states.

[3] Each object is encoded as a unique integer. Here, m represents an instance of $MembershipCard$.

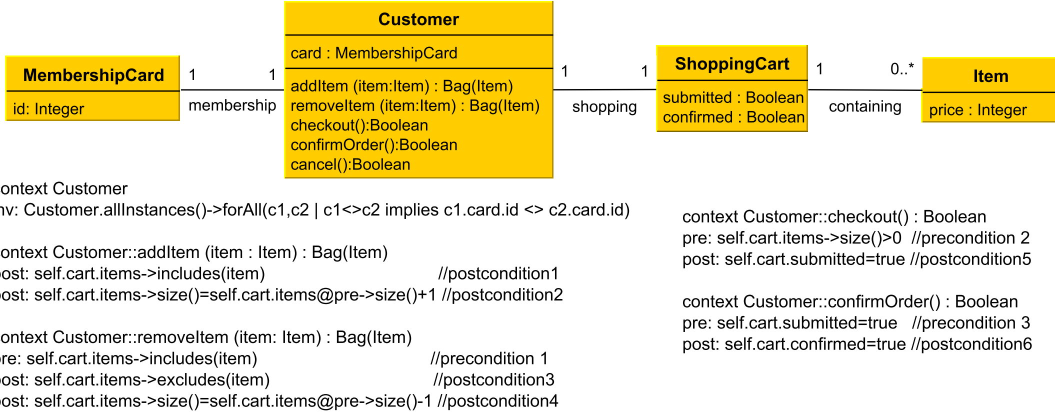

Figure 1: A UML class diagram annotated with a set of OCL operational contracts models an online shopping system. *inv* here specifies a class invariant. *pre* and *post* specify preconditions and postconditions of an operation respectively. @pre here denotes an object from previous state.

Figure 2: A call sequence shows a scenario that a customer may modify items after confirming an order.

$$\forall c1 : Customer, c2 : Customer, s : INT.\ \big(c1 \neq c2\big) \Rightarrow \big(F_{card}(c1,s) \neq F_{card}(c2,s)\big)$$

where s here is constrained as $0 \leq s \leq k$. This formula captures the semantics that no two customers ($c1$ and $c2$) can have the same card number in any state (denoted by s)[4]. This encoding gives us an advantage of producing compact formulas regardless of the increasing number of sequence length (k).

Queries. Similarly, users can also use quantifiers to specify queries over a set of states for specific purpose checking without affecting synthesis constraints. This significantly improves flexibility for users to debug and analyse a sequence without explicitly unrolling a query in each state. For example, a user can use the following query to check whether the number of items in a shoppingcart (in Figure 1) from a section of states is non-negative ($0 \leq i \leq j$).

$$\forall s : INT.\ \big(i \leq s \leq j\big) \Rightarrow \big(Size(items,\ s) \geq 0\big)$$

where $Size$ is defined as: $(Bag\ INT) \times INT \rightarrow INT$ and it returns the number of elements in a collection at a particular state s. $items$ encodes a collection of objects that are represented as a set of unique integers. The following diagram shows this query from state i to j. Note that the affected calls of this call sequence are $\delta_i, \delta_{i+1} \ldots \delta_j$.

$$S_0 \xrightarrow{\delta_1} \ldots \underbrace{S_i \xrightarrow{\delta_{i+1}\ \ldots\ \delta_j} S_j}_{query: |items| \geq 0} \xrightarrow{\delta_{j+1}\ \ldots\ \delta_k} S_k$$

Synthesis Constraints. To formally construct synthesis constraints, we simply conjoin each δ_i's pre/post conditions at state j with class invariants. In general, our synthesis constraint is in first-order form and it allows us to use unbounded collections and quantified invariants. For example, the two postconditions defined for the $1st$ operation $addItem()$ in Figure 1 at state j is encoded as follows:

$$\begin{aligned} \phi_{\langle 1,j\rangle} &\stackrel{\text{def}}{=} \Big(\exists x : INT.\ \big(Select(items,\ x,\ j)\big) = item0\Big) \\ \wedge &\quad \Big(Size(items,\ j) = Size(items,\ j-1)+1\Big) \end{aligned}$$

We use an existential quantifier to encode the *includes* operation, x denotes a specific position in a collection and j refers to current state and $j-1$ to previous state. Therefore, a transition can be established between two states via our synthesis constraints.

4.1 Transitions

Typically a transition from state S_{j-1} to S_j is triggered by a single operation call encoded by $\phi_{\langle i,j-1\rangle}$, assuming that no two calls can occur concurrently. To encode the transitions, we introduce a control variable $cv_{\langle i,j\rangle}$ for each $\phi_{\langle i,j\rangle}$. Each control variable controls the selection of its corresponding $\phi_{\langle i,j\rangle}$. More specifically, we constrain each control variable to be either 0 or 1. We say that $\phi_{\langle i,j\rangle}$ is enabled (δ_i at state j is called) only if $cv_{\langle i,j\rangle}$ is assigned to 1. In this way, we now can constrain the number of a selected operation calls at state j. Therefore, the following formula express the meaning of *selecting* exactly one operation

[4] Both $c1$ and $c2$ have their own unique object ids.

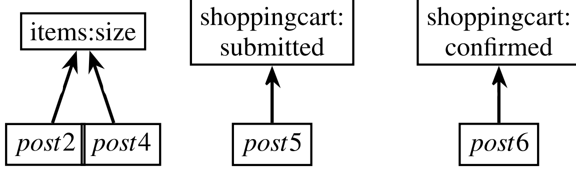

Figure 3: The dependency graph for postconditions defined in Figure 1.

call from Δ at each state.

$$\Big(\bigwedge_{i=1}^{|\Delta|}\bigwedge_{j=0}^{k}(cv_{\langle i,j\rangle}=1)\Rightarrow\phi_{\langle i,j\rangle}\Big)\wedge\Big(\bigwedge_{j=0}^{k}(\sum_{i=1}^{|\Delta|}cv_{\langle i,j\rangle})=1\Big) \quad (1)$$

where $cv_{\langle i,j\rangle}\in\{0,1\}$. On the other hand, if a control variable at state j is set to 0, then the corresponding operation call δ_i at that state is not selected. This implies that the attribute (affected by δ_i) of an object o (specified in a postcondition) has *not* been changed (since we have not selected δ_i). To compute the set of unchanged attributes, we work out the number of postconditions that affect an object's attributes via constructing a dependency graph as illustrated in Figure 3.

Let D be the set of nodes representing postconditions in a dependency graph. Each $d\in D$ is a specific postcondition of δ_i. We let T be the set of nodes that each d depends on. Now we can use the Formula 2 to express the set of attributes that have been changed [5].

$$\bigwedge_{j=1}^{k}\Big(\bigwedge_{d\in D}(cv_{\langle[d],j\rangle}=0)\Rightarrow\bigwedge_{t\in T}F_{[t]}(o,j)=F_{[t]}(o,j-1)\Big) \quad (2)$$

where F denotes a feature function and [ ] maps a node to the *ith* operation call or a specific feature. For example, we can construct a dependency graph (shown as Figure 3) for our online shopping model. The postcondition 2 and 4 (*post*2 and *post*4 in Figure 3) in Figure 1 both depend on the size of the object *items* [6] (since they change the number of items). We can say that when both postconditions 2 and 4 are *not* applied, then the number of *items* is *not* affected.

[5]Though the rule here only works on explicit changes, implicit ones can be further extracted using the technique presented in (Niemann et al., 2015).

[6]Note that the operations: *includes* and *excludes* only check the content of a collection rather than modifying its content.

5 Property-based Synthesis

In this section, we introduce a technique that allows us to synthesise a call sequence with respect to a set of properties. This technique uses a boolean matrix M as an intermediate representation for separating the synthesis constraints (Φ) from the property constraints (Ψ). Each property constraint (ψ_i) is then applied to constrain this boolean matrix. This provides a degree of flexibility that enables editing or expanding properties over a call sequence without affecting our synthesis constraints. This matrix M is shown in Figure 4. Essentially, this matrix captures all available operation calls from the initial state (S_0) to state S_{k-1}. The row of M represents all possible operation calls. The column of M indicates the transitional states. Each entry $cv_{\langle i,j\rangle}$ in the matrix is a control variable that denotes an operation call δ_i at state j. For example, the call sequence in Figure 2 is represented in the matrix in Figure 5.

5.1 Called-before

In many scenarios, a specific order of a series of calls is critical for verifying the behaviours of a system. For example, the *addItem* operation call in Figure 1 must be *called-before* the *confirmOrder*. Many potential problems in a design can be identified when an incorrect order is presented. In this subsection, we introduce a *called-before* property.

Given two operation calls δ_a and δ_b, the notation $\delta_a\rightarrow\delta_b$ defines a constraint that δ_a must be called-before δ_b. For example, $addItem\rightarrow confirmOrder$ defines that *addItem* must be invoked before $confirmOrder$. In order to encode this constraint, we look at it in an opposite way. A called-before relation $\delta_a\rightarrow\delta_b$ implies that an operation call δ_b *must not* be called before an operation call δ_a, that is $\delta_b\not\rightarrow\delta_a$. Thus, our encoding here works by blocking the possibilities of an operation call δ_b called before δ_a. More precisely, given a called-before sequence G_{cb}:

$$G_{cb}:\delta_1\rightarrow\delta_2\rightarrow\delta_3\rightarrow\ldots\delta_x\ldots\rightarrow\delta_n$$

Let x denote the xth δ in G_{cb} [7]. We now can start from the nth δ in G_{cb} and prevent each δ before nth position from being called until the $1st$ δ is reached. This constraint applies through from the initial state to the second last state since the number of states (where an operation is being called) is bounded from 0 to $k-$

[7]Note that δ_1 here denotes the $1st$ operation call in G_{cb}. That does *not* mean it also denotes the $1st$ operation call in Δ.

$$
\begin{array}{c} \\ \delta_1 \\ \delta_2 \\ \delta_3 \\ \vdots \\ \delta_n \end{array}
\begin{array}{c}
\begin{array}{cccc} S_0 & S_1 & \ldots & S_{k-1} \end{array} \\
\begin{bmatrix}
cv_{\langle 1,0\rangle} & cv_{\langle 1,1\rangle} & \cdots & cv_{\langle 1,k-1\rangle} \\
cv_{\langle 2,0\rangle} & cv_{\langle 2,1\rangle} & \cdots & cv_{\langle 2,k-1\rangle} \\
cv_{\langle 3,0\rangle} & cv_{\langle 3,1\rangle} & \cdots & cv_{\langle 3,k-1\rangle} \\
\vdots & \vdots & \ddots & \vdots \\
cv_{\langle n,0\rangle} & cv_{\langle n,1\rangle} & \cdots & cv_{\langle n,k-1\rangle}
\end{bmatrix}
\end{array}
$$

Figure 4: A boolean matrix M.

	S_0	S_1	S_2	S_3	S_4	S_5
addItem	1	1	0	0	0	0
removeItem	0	0	0	0	1	0
checkout	0	0	1	0	0	0
confirmOrder	0	0	0	1	0	0
cancel	0	0	0	0	0	1

Figure 5: A matrix representing the call sequence in Figure 2.

1 (excluding the last state) for a sequence that has a length of k.

$$\bigwedge_{x=n}^{2}\bigwedge_{i=0}^{k-2}\Big(\big(cv_{\langle [\![x]\!],i\rangle}=1\big)\Rightarrow\big(\bigwedge_{y=x-1}^{1}\bigwedge_{j=i+1}^{k-1} cv_{\langle [\![y]\!],j\rangle}=0\big)\Big) \tag{3}$$

Formula 3 captures this constraint and $[\![]\!]$ here maps the ith δ in G_{cb} to the jth δ in Δ. For example, Figure 6 shows an example of $confirmOrder \not\rightarrow (addItem \wedge removeItem)$. That is, if $confirmOrder$ (blue shaded area) is called at state S_2, then $addItem$ and $removeItem$ (red shaded area) cannot be called from state S_3 to S_5.

	S_0	S_1	S_2	S_3	S_4	S_5
addItem	x	x	x	x	x	x
removeItem	x	x	x	x	x	x
checkout	x	x	x	x	x	x
confirmOrder	x	x	x	x	x	x
cancel	x	x	x	x	x	x

Figure 6: An example matrix shows that $confirmOrder$ is called at state 2. This implies no $addItem$ or $removeItem$ should be called in the subsequent states. Each x in the matrix represents a control variable. The control variables in the red shaded area are switched off while the control variable in the blue shaded area is switched on.

5.2 Called Exactly n Times

In general, we use Formula 4 to capture that an operation call occurs exactly n times within a sequence. Let $cv_{\langle c,j\rangle}$ denote a control variable that represents the cth operation call at state j. We can count the number of control variables $cv_{\langle c,j\rangle}$ from the initial state to state $k-1$, and constrain this number to be n. Formula 4 provides a general form for other numeric constraints such as ensuring that an operation call must occur in a sequence *at least* n or *no more* than n times. This can be achieved by setting $=$ in the Formula 4 to $\geq$, $\leq$, $>$ and $<$.

$$\sum_{j=0}^{k-1} cv_{\langle c,j\rangle} = n, \text{where } c \text{ is a constant and } n \leq k-1. \tag{4}$$

5.3 Full Reachability

We can easily extend the called exactly n times property to check whether every single operation call δ_i defined in Δ can be reached by some operations within a sequence at least once. We say such a sequence has *full reachability*. This is achieved by using Formula 5. This property is quite useful because it allows users to synthesise a call sequence that covers each operation call at least once. Therefore, it also requires a sequence call to be a minimum length of $|\Delta|$.

$$\bigwedge_{i=1}^{|\Delta|}\bigvee_{j=0}^{k-1} cv_{\langle i,j\rangle} = 1, \text{ where } k \geq |\Delta|. \tag{5}$$

For Formula 5, we can also fix j to specifically check whether an operation call can be reached at the specific position in a sequence. For example, $removeItem$ cannot be placed at the beginning of a sequence, if a shoppingcart in the initial state is empty.

5.4 Partial Sequence

In many situations, users may like to have a property that they can perform analysis on a certain fixed portion of a call sequence. We say such sequence is partially known or a partial sequence. A partial sequence is a *subsequence* $L = \langle \delta_{1+i}, \ldots, \delta_{m+i}\rangle$ of a sequence $S = \langle \delta_1, \delta_2, \ldots, \delta_k\rangle$ such that $i \geq 0$ and $m+i \leq k$. This property requires that every synthesised call sequence must contain that partial sequence. Thus, this allows users to analyse the behaviours of a system under a particular portion of a call sequence. For example, Figure 2 shows that this synthesised call sequence must contain a partial sequence *checkout* and $confirmOrder$.

Formula 6 expresses our partial sequence property. Let L denote a partial sequence and $|L| \geq 1$ that consists of a series of operation calls $\langle \delta_1, \delta_2, \ldots, \delta_k\rangle$.

For each $\delta_a \in L$ at state i encoded by a control variable $cv_{\langle a,i \rangle}$, we ask the SMT solver to select this control variable starting from the initial state ($i = 0$) until the length of a partial sequence can no longer fit in the remaining states. This encodes all possible ways that a partial sequence L could happen within a sequence. We then make sure at least one of them is chosen by the solver.

$$\bigvee_{i=0}^{m} \bigwedge_{\delta_a \in L}^{|L|} cv_{\langle a,i \rangle} = 1, \text{where } m = k - |L| + 1. \quad (6)$$

For example, we let $L = \langle checkout, confirmOrder \rangle$ be a partial sequence and generate a call sequence that must contain L. Each blue shaded area in Figure 7 represents a way of selecting this partial sequence by enabling the corresponding control variables. In addition, Figure 7 shows that there is a total of 5 ways of synthesising this partial sequence L. Selecting any one of the five ways satisfies the partial sequence property. Therefore, the disjunction in Formula 6 makes sure that at least one of the possible subsequences is selected.

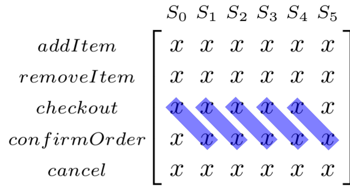

Figure 7: There are 5 possible ways of synthesising a call sequence (a length of 6) that contains a partial sequence consisting of *checkout* and *confirmOrder* for the running example in Figure 1. Here, each x represents a control variable.

6 Formula Simplifications

Removing Invariants. We notice that in many cases a class invariant may *not* affect the features used by the postconditions of an operation. In other words, a class invariant may not depend on the features that a postcondition depends on. In this case, we can *safely* remove this class invariant from the model. The general rule here is to construct two dependency graphs for postconditions (G_1) and invariants (G_2). If a feature f that appears in G_2 does not appear in G_1, then we can remove all the class invariants that depend on this f [8]. Otherwise, we reduce each class invariant to a single quantified formula.

[8]Removed invariants can be reasoned by using a separate procedure.

For example, the class invariant in Figure 1 depends on the attribute *id* in the *membership* class. However, *id* does not appear in the dependency graph in Figure 3. Therefore, we can remove this class invariant from the synthesis constraints.

Implicit Invariants. The set of pre/postconditions may impose constraints on a set of common features in a model. Unrolling each of them produces similar-structured formulas and therefore increases the overall formula size. For example, it is implicit that each operation call in Figure 1 uses one common constraint: the number of items contained in a *shopping cart must not* be negative. In fact, this constraint is an implicit class invariant. When users discover this constraint via the analysis of synthesised sequence, users may elevate this constraint to a class invariant and introduce a $\forall$ quantifier. Hence, we introduce the following simplification rule:

$$\bigwedge_{j=0}^{k} F_i(o,\ j) \equiv \forall o : T, s : INT.\ F_i(o,\ s) \qquad (7)$$

Formula 7 implies that if there exists a constraint (F_i) on an object (o) over bounded states, then this constraint can be reduced to a single quantified one.

Symmetric Relations. Relations between two classes are modelled as associations in a UML class diagram. However, many relations are symmetric and naively unrolling formulas over these relations doubles the overall formula size. For example, a marriage relationship between two people is symmetric since $marriage(person0,\ person1)$ is the same as $marriage(person1,\ person0)$. Thus, we can halve the formula size by unrolling only one of them. We now provide a simplification rule (Formula 8) for expressing such symmetric relations over unbounded states.

$$\left(\bigwedge_{j=0}^{k} Rel(o_1,\ o_2,\ j)\right) \wedge \left(\bigwedge_{j=0}^{k} Rel(o_2,\ o_1,\ j)\right) \equiv \left(\forall o_1 : T_1, o_2 : T_2, s : INT. Rel(o_1,\ o_2,\ s) = Rel(o_2,\ o_1,\ s)\right) \qquad (8)$$

Formula 8 uses $\forall$ to quantify a (binary) symmetric relation Rel[9]. With this rule, we now can unroll one of the $Rel(o1,\ o2, j)$ and therefore reduce the overall formula size.

[9]For an *n-ary* relation, it can be decomposed into multiple binary relations.

7 Implementation and Evaluation

We have implemented a prototype that generates synthesis and properties constraints for a given input model. Currently, this prototype is semi-automatic. We first generate a smaller template for a given model. The template includes information such as the number of the objects and the length of a sequence to be synthesised. Then, our tool uses a formula reasoning engine from MaxUSE for instantiating those templates to produce the concrete synthesis and property constraints (Wu, 2017a; Wu, 2017b). By default, our tool generates SMT2 standard formulas and uses the Z3 SMT solver for constraint solving (De Moura and Bjørner, 2008).

In order to evaluate the scalability of our technique, we collect 4 models from recent literature and one from the example shown in Figure 1 to form a benchmark. Since our prototype is semi-automatic, we first generate a smaller template for each model in the benchmark and manually verify the correctness of these formulas. We then scale them into much longer sequences.

7.1 Evaluation

We evaluate our technique on an Intel(R) Xeon(R) machine with eight 3.2GHz cores and 16 GB memory. However, our evaluation only uses one single core. The evaluation is divided into two phases. First, we evaluate the synthesis constraints with property constraints with respect to pre/post conditions. Second, we evaluate the effectiveness of our simplification rules by applying appropriate rules on each model. We now here to discuss our evaluation results. All of our generated results can be found at:

`http://www.cs.nuim.ie/~haowu/synseq.html`

Results. The rows in non-gray colour in Table 1 shows the results of our evaluation on synthesis and property constraints. The 'Time' and 'Size' columns show the time spent (the unit here is second) on synthesising sequences and number of formulas generated, respectively. In general, our technique is able to synthesise large sequences quite efficiently. For each model, our technique can handle a number of objects and pre/post conditions. The most challenging model in this benchmark is the *Bank* model. This is because this model contains quite a lot of numeric constraints and this imposes a great challenge to the SMT solver. To determine how effective our simplification rules are, we apply suitable simplification rules on each model. For example, we use full reachability (FR) property constraint and removing invariants (RI) simplification rule to discover a counter-example (for the $TrafficLights$ model) that a pedestrian light and a car light are both in a dead state (cannot progress with further operation when $k \geq 4$). The rows in gray colour in Table 1 shows the difference after applying suitable simplification rules. The results here show that by using simplification rules in Section 6 can effectively reduce overall formula size and increases performance in most of the cases. Interestingly, though the symmetric relation rule can halve the formula size, it does not always increase performance. This phenomenon is observed when we set $k \geq 250$ for $Marriage$ model. We surmise that when use symmetric relation (SR) rule it introduces additional quantifiers and this may cause the solver to spend quite amount of time on instantiating those quantifiers.

Comparison. Comparing to existing approaches (Soeken et al., 2011b; Przigoda et al., 2015), our technique can synthesise much larger sequences. Here, we compare our technique against bit-vector based approaches (Soeken et al., 2011a; Soeken et al., 2011b; Przigoda et al., 2015). In general, it is difficult to compare performance. This is because: (1) The performance typically depends on specific models, properties (deadlock, reachability), and the individual SMT solver used. (2) It is *extremely difficult* to reimplement other's formulas due to the lack of *publicly accessible data* (Soeken et al., 2011a; Soeken et al., 2011b; Przigoda et al., 2015). In order to perform a fair performance comparison, we compare our approach against bit-vector based approaches on the benchmark but select the bank model as a representative model here [10]. In general, our comparison results reveal that the bit-vector based approaches have good performance only when the size of bit-vectors are relatively small.

To conduct a meaningful experiment, we *reimplement* the bit-vector based approaches presented in (Soeken et al., 2011b; Przigoda et al., 2015) for the bank model (Przigoda et al., 2015) and set up three groups [11]. Each group uses different sized bit-vectors: 8,16 and 32. Hence, each account in three groups has an upper bound of 255,65535 and 4294967295, respectively. For each group, we then instantiate 3 bank accounts, constrain the initial and final state within a sequence. We ask SMT solvers to synthesise a sequence with a length (k) from 10 up to 50 (with an interval of 10). We use Boolector for the bit-vector based approaches. Boolector is a specially crafted

[10] Due to the page limit, we use bank model as our representative model to explain the comparison results.

[11] Though the work in (Przigoda et al., 2015) checks the concurrent behaviours of a system, the fundamental encoding is the same as the work in (Soeken et al., 2011b).

Length	Company		TrafficLights		Bank		Marriage		Onlineshop	
	FR, RI		FR, RI		CEnT, RI		CEnT, SR		PS, CB, RI, II	
	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size
$k = 100$	0.47	2302	0.91	912	10.83	2499	4.48	7549	0.09	1314
	0.24	2201	0.42	810	8.40	1599	2.29	4305	0.09	1013
$k = 150$	3.56	3450	4.63	1362	18.12	3748	10.30	11575	1.24	1963
	3.50	3302	3.46	1210	14.72	2398	7.51	5994	1.12	1509
$k = 200$	5.50	4603	14.72	1810	80.56	5203	18.39	15288	4.56	2611
	4.48	4402	7.22	1597	45.72	3401	8.51	8598	3.22	2247
$k = 250$	6.57	5752	21.87	2262	154.58	6428	19.53	21840	5.45	3011
	5.86	5500	15.61	2010	151.61	3998	20.21	9994	5.70	2509
$k = 300$	7.71	6901	32.06	2704	167.93	7498	25.83	21849	6.30	3613
	7.38	6704	23.70	2408	109.74	4798	26.39	11994	6.21	3008

Table 1: The evaluation results of synthesis and property constraints with simplification rules for models collected in the literature: Company(Gogolla et al., 2007), TrafficLights(Soeken et al., 2011b), Bank(Przigoda et al., 2015) and Marriage (Gogolla et al., 2017). The time unit here is seconds. The row in gray colour shows the improvements in both performance and formula size after applying simplification rules. CB, FR, CEnT and PS denotes called-before, full reachability, called exactly n times and partial sequence properties respectively. RI, II and SR denotes removing invariants, implicit invariants and symmetric relations simplification rules respectively.

SMT solver for bit-level reasoning (Niemetz et al., 2015). We then use Z3 SMT solver to solve the synthesis constraints generated using our technique.

Table 2 shows the performance comparison results on the bank model. It can be seen that bit-vector based encoding works quite well with the width of 8. However, when the size of each bit-vector increases the performance significantly decreases. This is because the solver needs to create a boolean variable for each bit in a bit-vector (bit-blasting), before it performs a series of arithmetic operations for each bit. This is fine for a bit-vector that has a smaller fixed width. In Table 2, the bit-vector based encoding (with 16 and 32 bit width) can only synthesise a call sequence of length up to 20. On the other hand, our technique allows much better scaling. This is much more realistic as in the real world many data structures work with 32 bit long integers. For example, a bank can set up an account using a 32 bit integer (or even larger) representing the amount of money that an account can hold. s

7.2 Discussion

Compared to existing approaches, our evaluation results show that our synthesis and property constraints can be solved efficiently at a much larger scale (Soeken et al., 2011b; Przigoda et al., 2016; Przigoda et al., 2015). Our simplification rules also effectively reduce the overall formula size. In comparison to bit-vector based approaches, the trade off here is among three aspects: expressiveness, flexibility and performance. Therefore, we compare our technique to bit-vector based approaches in these three aspects. Table 3 shows detailed comparisons in different criterion. In general, encoding a model along with OCL operational contracts into quantifier-free bit-vectors should possess high performance during the SMT solving. However, this is not always the case. In particular, when the model is required to be encoded into larger size bit-vectors or involves complicated arithmetic computations, the performance decreases significantly due to bit-blasting. Our technique provides a better solution to balance the three aspects by using the first-order encoding. This allows quantifiers to be used for class invariants, queries and triggered system states. This helps to reduce the number of formulas unrolled in each state and provides much more expressiveness, flexibility and in the meanwhile maintains high performance.

Automation. Though our technique is semi-automatic, we provide a set of python scripts that is able to process a formula template generated by MaxUSE and instantiate it with concrete synthesis and property constraints. The amount of user intervention here is quite little. However, adding extra constraints for a particular state may require users to manually insert the formula into a template. We are now building a tool that is able to automatically insert the formula at appropriate place in a template.

OCL. Our technique currently supports a range of OCL language constructs. These include: constraints on attributes, different operations over collection data types and quantified class invariants. However, the models collected in the benchmark (Table 1) do not cover full OCL features. Hence, there is a gap between covering full OCL language features and our technique. Further, OCL is a 4-valued logic language

Approach	$k = 10$	$k = 20$	$k = 30$	$k = 40$	$k = 50$
BV(8)	0.41	1.22	3.25	4.67	6.32
BV(16)	0.54	10.7	TO	TO	TO
BV(32)	0.93	37.6	TO	TO	TO
Our Technique	0.13	0.54	0.62	3.58	5.67

Table 2: The performance comparison between bit-vector based approaches and our technique. The time unit here is seconds and TO indicates timed out. BV(8), BV(16) and BV(32) indicates each bit vector has a width of 8, 16 and 32 bits respectively. The timed out setting here is 180 seconds for both Boolector and Z3 SMT solvers.

Criterion	Bit-vector	First-order
Data Types (**E**)	Bounded	Unbounded
Class Invariants (**E**)	Unrolled in each state	Quantified
Query (**F**)	No direct support	Quantified states.
Property (**F**)	Unrolled with synthesis constraints.	Separated from synthesis constraints
Formula Size (**P**)	Compact	Reduced
Solving Time (**P**)	Fast on small size bit-vectors	Fast first-order reasoning
Underlying Solver (**P**)	BV solver	SMT Solver

Table 3: The criterion for comparing our technique (first-order encoding) against bit-vector based approaches. E, F and P here denote three aspects: expressiveness, flexibility and performance.

that includes undefined and invalid values. Currently, we do not support encodings for undefined and invalid values. However, we are investigating a new encoding that allows us to support multi-valued logic.

Our Findings. We have two main findings: (1) Our synthesis and property constraints possess high expressiveness and flexibility. This significantly improves possibilities of debugging or analysing different sequences by using quantified first-order forms without regenerating synthesis constraints for each state. (2) Our simplification rules are effective for reducing and boosting performance. This also shows users a general template for further expanding or creating more customised rules.

Limitations. We identify two limitations. (1) Our property constraints currently works on bounded states. Hence, when the length of a sequence increases the regeneration of the property constraints is inevitable. Though it is possible to introduce quantified forms for the property constraints, rewriting the property constraints unrolled in each state to first-order form may require a more sophisticated encoding for the synthesis constraints. (2) Our simplification rules in general reduce overall formula size. However, the introduced quantifiers may lead to quantifier alternations (If the formula is already quantified). This typically places a great challenge on current SMT solvers. One of the ways to tackle this is to use a specialised decision procedure designed for synthesis problems only (Reynolds et al., 2017).

Threats to Validity. There are two threats to validity in our evaluation. (1) Our benchmark is formed from existing approaches covering a great deal of OCL constructs including: navigation, nested quantified invariants and queries or operations over collection data types. However, they do not cover all aspects of OCL constructs such as closure operator. (2) The comparison results may not be precise enough. This is because we reimplement bit-vector based approaches based on interpretations of published articles rather than the actual concrete formulas due to the lack of publicly accessible data.

8 Related Work

SMT solving techniques (Przigoda et al., 2015; Soeken et al., 2011b) and filmstripping (Gogolla et al., 2014; Hilken and Gogolla, 2016; Gogolla et al., 2017) are two major approaches for generating (synthesising) call sequences from OCL operational contracts. In general, the two approaches complement each other and a detailed comparison has been conducted in (Hilken et al., 2014). The filmstripping approach translates the source model into so-called filmstrip models. These models essentially are UML class diagrams annotated with OCL constraints. These constraints are frame conditions that specify the changes to the model. One of the major advantages using filmstrip is that the low level (SAT solver) is not explicitly exposed to users. Thus, it does not require users to have the knowledge at the solver level. However, the cost here is the substantial manual interaction at

model level. Therefore, the filmstripping approach is suitable for very dedicated tasks. In comparison, our technique presented here is more general and it focuses on increasing expressiveness, flexibility and maintaining high performance through a novel first-order encoding.

Constraint programming (CP) is another popular approach to verifying dynamic aspects of UML models (Cabot et al., 2009). Typically, CP provides a high-level programming language so that a particular problem can be programmed into a constraint satisfaction problem (CSP) (Cabot et al., 2009; González Pérez et al., 2012; Cabot et al., 2014). In (Cabot et al., 2009), they program the verification tasks into a CSP that is solved later using constraint solvers. In their work, they propose a range of different properties to be checked such as weak and strong satisfaction. Their work mainly focus on generating proofs rather than synthesising call sequences. Our work distinguishes from theirs by proposing a first-order reasoning technique so that we can synthesise call sequences at large scale via SMT solving.

Alloy as a model finding tool (Jackson, 2002; Torlak and Jackson, 2007), is popularly used in many domains including verifying UML models. However, the majority of the work uses Alloy as its basis focuses on verifying/solving structural constraints of a system (Anastasakis et al., 2007; Shah et al., 2009; Garis et al., 2011; Kuhlmann and Gogolla, 2012). For example, Kyriakos et al. maps a range of OCL constructs to Alloy's specification (Anastasakis et al., 2007), and Kuhlmann et, al. integrates kodkod (Alloy's solving engine) into the USE modelling tool and this enables them to be able to verify and analyse UML models annotated with different types of OCL constraints.

Other approaches have also sought to translate UML and OCL into different types of formalisms. These include interactive theorem provers such as Isabelle and KeY (Ahrendt et al., 2007; Brucker and Wolff, 2009; Balaban and Maraee, 2013; Dania and Clavel, 2013; Dania and Clavel, 2016). For example, Brucker et, al. translate OCL into high-order logic and prove them using Isabelle (Brucker and Wolff, 2008). Others formalises UML models into PVS (Kyas et al., 2005). Compare to SAT/SMT solving (Wu et al., 2013; Wu, 2016), interactive based approaches allow users to input/define their own theorems or axioms to guide the solver. This could be particularly helpful if the solvers are stuck and cannot progress during the prove.

9 Conclusion

In this paper, we present a new SMT-based technique that allows us to synthesise call sequences at a much larger scale than previously possible. This technique uses first-order form based synthesis constraints. To enable property-based synthesis with reduced formula size, we have designed a set of property constraints and simplification rules. Our evaluation results show that in comparison to existing approaches, our technique provides a much more general solution for synthesising call sequences in terms of scalability, expressiveness, flexibility and performance. Currently, we are investigating new algorithms and techniques so that both alternative quantified synthesis and property constraints can be efficiently reasoned.

REFERENCES

Ahrendt, W., Beckert, B., Hähnle, R., and Schmitt, P. H. (2007). Key: A formal method for object-oriented systems. In *Formal Methods for Open Object-Based Distributed Systems*, pages 32–43. Springer Berlin Heidelberg.

Anastasakis, K., Bordbar, B., Georg, G., and Ray, I. (2007). UML2Alloy: A challenging model transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, pages 436–450, Nashville, TN. Springer.

Balaban, M. and Maraee, A. (2013). Finite Satisfiability of UML Class Diagrams with Constrained Class Hierarchy. *ACM Transcation on Software Engineering and Methodology*, 22(3):24:1–24:42.

Berardi, D., Calvanese, D., and De Giacomo, G. (2005). Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1-2):70–118.

Brucker, A. D. and Wolff, B. (2008). HOL-OCL: A formal proof environment for UML/OCL. In *11th International Conference on Fundamental Approaches to Software Engineering*, pages 97–100. Springer.

Brucker, A. D. and Wolff, B. (2009). Semantics, calculi, and analysis for object-oriented specifications. *Acta Informatica*, 46(4):255–284.

Büttner, F., Egea, M., and Cabot, J. (2012). On verifying ATL transformations using 'off-the-shelf' SMT solvers. In *15th International Conference on Model Driven Engineering Languages and Systems*, pages 432–448.

Cabot, J., Clarisó, R., and Riera, D. (2009). Verifying UML/OCL operation contracts. In *7th International Conference on Integrated Formal Methods*, pages 40–55, Düsseldorf, Germany. Springer.

Cabot, J., Clarisó, R., and Riera, D. (2014). On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software*, 93:1–23.

Dania, C. and Clavel, M. (2013). Ocl2fol+: Coping with undefinedness. In *OCL@MoDELS*.

Dania, C. and Clavel, M. (2016). Ocl2msfol: A mapping to many-sorted first-order logic for efficiently checking the satisfiability of ocl constraints. In *19th International Conference on Model Driven Engineering Languages and Systems*, pages 65–75. ACM.

De Moura, L. and Bjørner, N. (2008). Z3: an efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Budapest, Hungary. Springer.

Garis, A., Cunha, A., and Riesco, D. (2011). Translating Alloy Specifications to UML Class Diagrams Annotated with OCL. In *9th International Conference on Software Engineering and Formal Methods*, pages 221–236, Montevideo, Uruguay. Springer.

Gogolla, M., Büttner, F., and Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1-3):27–34.

Gogolla, M., Hamann, L., Hilken, F., Kuhlmann, M., and France, R. B. (2014). From application models to filmstrip models: An approach to automatic validation of model dynamics. In *Modellierung*.

Gogolla, M., Hilken, F., Doan, K., and Desai, N. (2017). Checking UML and OCL model behavior with filmstripping and classifying terms. In *11th International Conference on Tests & Proofs*, pages 119–128.

González Pérez, C. A., Buettner, F., Clarisó, R., and Cabot, J. (2012). EMFtoCSP: A tool for the lightweight verification of EMF models. In *Formal Methods in Software Engineering: Rigorous and Agile Approaches*, Zurich, Suisse.

Hilken, F. and Gogolla, M. (2016). Verifying linear temporal logic properties in UML/OCL class diagrams using filmstripping. In *2016 Euromicro Conference on Digital System Design*, pages 708–713.

Hilken, F., Niemann, P., Gogolla, M., and Wille, R. (2014). Filmstripping and unrolling: A comparison of verification approaches for uml and ocl behavioral models. In *Tests and Proofs*, pages 99–116. Springer International Publishing.

Jackson, D. (2002). Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodologies*, 11(2):256–290.

Kuhlmann, M. and Gogolla, M. (2012). From uml and ocl to relational logic and back. In *15th International Conference on Model Driven Engineering Languages and Systems*, pages 415–431. Springer.

Kyas, M., Fecher, H., de Boer, F. S., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., and Kugler, H. (2005). Formalizing UML models and OCL constraints in PVS. *Electronic Notes in Theoretical Computer Science*, 115:39–47.

Niemann, P., Hilken, F., Gogolla, M., and Wille, R. (2015). Extracting frame conditions from operation contracts. In *18th International Conference on Model Driven Engineering Languages and Systems*, pages 266–275.

Niemetz, A., Preiner, M., and Biere, A. (2015). Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58.

Przigoda, N., Hilken, C., Wille, R., Peleska, J., and Drechsler, R. (2015). Checking concurrent behavior in uml/ocl models. In *18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 176–185.

Przigoda, N., Soeken, M., Wille, R., and Drechsler, R. (2016). Verifying the structure and behavior in uml/ocl models using satisfiability solvers. *IET Cyber-Physical Systems: Theory Applications*, 1(1):49–59.

Reynolds, A., Kuncak, V., Tinelli, C., Barrett, C., and Deters, M. (2017). Refutation-based synthesis in smt. *Formal Methods in System Design*.

Shah, S. M. A., Anastasakis, K., and Bordbar, B. (2009). From UML to alloy and back again. In *6th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 4:1–4:10. ACM.

Soeken, M., Wille, R., and Drechsler, R. (2011a). Encoding OCL data types for SAT-based verification of UML/OCL models. In *5th International Conference on Tests and Proofs*, pages 152–170, Zurich, Switzerland. Springer.

Soeken, M., Wille, R., and Drechsler, R. (2011b). Verifying dynamic aspects of uml models. In *Design, Automation Test in Europe*, pages 1–6.

Torlak, E. and Jackson, D. (2007). Kodkod: a relational model finder. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, Braga, Portugal. Springer.

Wille, R., Soeken, M., and Drechsler, R. (2012). Debugging of inconsistent UML/OCL models. In *2012 Design, Automation Test in Europe Conference Exhibition*, pages 1078–1083.

Wu, H. (2016). Generating metamodel instances satisfying coverage criteria via SMT solving. In *The 4th International Conference on Model-Driven Engineering and Software Development*, pages 40–51.

Wu, H. (2017a). Finding achievable features and constraint conflicts for inconsistent metamodels. In *13th European Conference on Modelling Foundations and Applications*, pages 179–196. Springer.

Wu, H. (2017b). Maxuse: A tool for finding achievable constraints and conflicts for inconsistent UML class diagrams. In *Integrated Formal Methods*, pages 348–356. Springer.

Wu, H. (2019). Synthesising call sequences from OCL operational contracts. In *The 34th ACM/SIGAPP Symposium on Applied Computing*.

Wu, H., Monahan, R., and Power, J. F. (2013). Exploiting attributed type graphs to generate metamodel instances using an SMT solver. In *7th International Symposium on Theoretical Aspects of Software Engineering*, Birmingham, UK.