Delta-Oriented FSM-Based Testing

Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi*

Center for Research on Embedded Systems
Halmstad University, Sweden
{mahsa.varshosaz, harsh.beohar, m.r.mousavi}@hh.se

Abstract. We use the concept of delta-oriented programming to organize FSM-based test models in an incremental structure. We then exploit incremental FSM-based testing to make efficient use of this high-level structure in generating test cases. We show how our approach can lead to more efficient test-case generation, both by analyzing the complexity of the test-case generation algorithm and by applying the technique to a case study.

Keywords: Model-Based Testing, FSM-based Testing, HSI Method, Software Product Lines, Delta-Oriented Programming, DeltaJava

1 Introduction

Software product lines (SPLs) have become common practice thanks to their potential for mass production and customization of software. Testing software product lines, and in particular, their model-based testing are topics of increasing relevance in the research literature and also industrial practice [4, 10, 17]. In this paper, we propose the formal foundations of a delta-oriented framework for model-based testing. Delta-oriented programming (DOP) and in particular, DeltaJava [14], is a framework for SPLs, in which a product line is specified in terms of applications of a number of deltas (changes: additions, removals and modifications of member objects, methods, and classes) from a core product. The overall goal of the research commenced by this paper is to allow for efficient test-case generation and test-case execution for delta-oriented models and their corresponding programs. In this paper, we focus on test-case generation and show whether and how test-case generation for delta-oriented models can be made more efficient by benefiting from their incremental structure.

To this end, we use finite state machines (FSMs) as test models whose structure is based on DeltaJava: there is a test-model for the core product, which includes abstraction of state valuations as its states and the method calls, their return values and their effect on the abstract state as its transitions. Then, test models for different products are obtained by incrementally modifying the details of the core model (e.g., adding models for classes, member objects and

* The work of M.R. Mousavi has been partially supported by the Swedish Research Council award number: 621-2014-5057 and the Swedish Knowledge Foundation in the context of the AUTO-CAAS HöG project.

methods). In this paper, we focus on the incremental subset of DeltaJava, in which the core represents a minimal set of features and the deltas incrementally add to the core or the composition of core with other deltas (but do not remove anything from them). We also adopt the well-known Harmonized State Identification (HSI) method [13] and adapt it to the delta-oriented structure of our test models.

The remainder of this paper is organized as follows. In Section 2, we review several pieces of related work and identify their similarities and differences with the present paper. In Section 3, we recall some preliminary notions regarding FSM-based testing and the syntax of delta-oriented models. We specify the syntactic structure of our running example in Section 4, which we use throughout the rest of the paper to illustrate various formal definitions. Subsequently, we define the semantic domain of our test models in Section 5 and show how the test models of various products can be obtained from the semantics of the core model by applying a delta composition operator. In Section 6, we show how test cases can be generated from the test models of various products and analyze the complexity of test-case generation. In Section 7, we provide some empirical results obtained from comparing the effectiveness of the application of the delta-oriented testing method with the HSI-method for a case study. We conclude the paper and present the directions of our ongoing research in Section 8.

2 Related Work

Incremental FSM-Based Testing The closest line of research to that of the present paper is incremental FSM-based testing, which is extensively researched in the past few years [3, 6, 9, 11, 15]. This line of research aims at modularizing the test-case generation and/or test-case execution process with respect to changes such as adding, removing, or modifying transitions or states in test models. Such a modularization should eventually lead to saving time and effort in re-generating or re-executing tests by focusing on those parts that are influenced by the change. The approaches of [3, 6] differ from our approach in that they assume that the behavior of the core implementation is unchanged after each and every delta and focus on the effect of changes on the extended part of the implementation; we have no assumption about the behavior of the implementation due to the application of a delta. Our focus in this paper is on test-model semantics and test-case generation rather than test-case selection and execution. The approach of [15] is different from ours in that it aims at completing a given set of test cases, but does not per se address the changes in the test model. Our approach is mostly based on [9, 11] and applies it at a higher level of abstraction to delta-oriented models inspired by the DeltaJava framework of [14].

Model-Based Testing of SPLs In a recent survey, Oster et al. [10] observe that there is a considerable gap regarding testing in the current software engineering approaches to SPLs. Despite this gap, there is already some body of research on the theory and application of model-based testing for SPLs (see, e.g., [4, 10,

17] for recent surveys). Among these approaches, the closest to our approach are those developed by Malte Lochau, Ina Schaefer, et al. [8]. They propose a delta-oriented and state-machine-based testing methodology for SPLs and instantiate this methodology in a case study using IBM Rational Rhapsody and Automated Test-case Generator (ATG). Our approach follows the same structure and formalizes the part that has been implemented in IBM Rhapsody, by means of ideas from incremental FSM-based testing. This paves the way for further formal analyses of the technique proposed in [8], as well as further improvements by considering more relaxed fault models.

Object-Oriented Model-Based Testing There is a large body of literature regarding model-based testing of object-oriented programs by using sequence- or state-diagrams as test models (see, e.g., [1, 12, 18]). We follow object-oriented principles such as encapsulation and data-hiding in our modeling framework and organize our test models based on specification of class instantiations and dependencies. In this sense, our work builds upon earlier work in this direction such as [5, 18]; in particular, our test models are reminiscent of class state machines (CSMs) introduced in [5]. Our work differs from this line of work in two ways: firstly, our focus is on incremental changes in test models and not so much on testing object oriented programs. Secondly, in our approach the system under test need not be implemented as an object-oriented program; the abstract test-cases from our test-models can be used to test different types of implementation. This is achieved by means of adapters that turn the abstract test-cases into concrete test-cases for different programming languages and implementation platforms.

3 Preliminaries

3.1 FSM-Based Testing

In this section, we explain the basic concepts of FSM-based testing and delta-oriented modeling techniques used throughout the rest of the paper. We use the Harmonized State Identification (HSI) method [13] as the basis of our model-based testing technique. In the HSI method, test models are Finite State Machines (FSMs), specifying the desired behavior of systems. The formal definition of an FSM, borrowed from [2], is as follows.

Definition 1. *(Finite State Machine) A Finite State Machine (FSM) M is a 6-tuple $(S, s_0, I, O, \mu, \lambda)$, where S is a finite set of states, $s_0 \in S$ is the initial state, I and O are, respectively, finite nonempty sets of input and output symbols, $\mu : S \times I \to S$ is the transition function and $\lambda : S \times I \to O$ is the output function.*

Intuitively, whenever a machine receives input a at state s, it deterministically traverses to state $\mu(s, a)$ and generates output $\lambda(s, a)$. A transition from state s to state s' with input i and generated output o is represented by quadruple (s, i, o, s'), or alternatively by $s \xrightarrow{i/o} s'$. For a sequence $x \in I^*$, we define $\mu(s, x)$

and $\lambda(s, x)$ in the standard manner to denote, respectively, the final state that the machine ends in and the sequence of generated outputs, after receiving the input symbols in x one by one. Furthermore, we also informally recall that two states are X-equivalent ($X \subseteq I^*$) if and only if the two states produce the same output for every input sequence $\sigma \in X$ (see [2] for a formal definition). Lastly, two machines M, M' are X-equivalent, denoted by $M \equiv_X M'$, if and only if for every state of M there is an X-equivalent state of M' and vice versa. Machine M is said to conform to machine M' if and only if they are I^*-equivalent.

The main idea of the HSI method is to establish conformance between an FSM test model M and an unknown machine M', modeling the implementation, by generating a finite test case from M and applying it to M'. There are a set of assumptions that should hold for these machines, which are specified next.

Definition 2. *(HSI method assumptions) The HSI method can be applied on machines M and M', which satisfy the following assumptions:*

1. *Both M and M' are deterministic, i.e., for each state and each input i, there is at most one outgoing transition labeled with i.*
2. *Both M and M' are minimal, i.e., there are no distinct I^*-equivalent states in either of them. Note that if M is not minimal, an equivalent minimal machine can be generated using a minimization algorithm such as [7].*
3. *All states in M are reachable from its initial state s_0.*
4. *Both machines M and M' have reliable reset sequences, which take the respective machine from the current state to the initial state.*
5. *M' has at most as many states as M.*

The HSI method consists of two phases. The first phase comprises checking the existence of states in the implementation that are I^*-equivalent to the ones in the test model. In the second phase, the output and the target of the transitions for the corresponding states are tested for conformance. In order to reach all the states in the machine, the HSI method uses a set of input sequences, *state cover set*, denoted by Q, which is defined below.

Definition 3. *(State Cover Set) Consider an FSM $M = (S, s_0, I, O, \mu, \lambda)$; a state cover set of M, denoted by Q, is a set of sequences such that:*

$$\forall_{s \in S} \cdot \exists_{x \in Q} \cdot \mu(s_0, x) = s$$

A state cover set of an FSM can be obtained by building a spanning tree such that, the nodes are states of the FSM and the edges are chosen from the set of transitions in the FSM. The set of sequences obtained as the state cover set are then the paths from the initial state to the nodes in the spanning tree.

As another ingredient of the first phase, i.e., checking the existence of test-model states in the implementation, the HSI method uses a *separating family of sequences*, which is denoted by Z and comprises sets of separating sequences for all states. A set of separating sequences identifies and tests the target states after running each element of the state cover set. The separating set for a state is defined as follows.

Definition 4. *(Separating Sequences) Consider an FSM $M = (S, s_0, I, O, \mu, \lambda)$; the set of separating sequences for a state $s \in S$, is denoted by z_s and includes sequences that can distinguish s from all other states in S, that is:*

$$\forall_{s,s' \in S} \cdot s \neq s' \;\Rightarrow\; \exists_{x \in Pref(z_s) \cap Pref(z_{s'})} \cdot \; \lambda(s,x) \neq \lambda(s',x),$$

where $Pref(.)$ denotes the set of prefixes of a set of sequences.

A separating family of sequences for an FSM, is a set comprising the separating sequences of all states, that is $Z = \bigcup_{s \in S} \{z_s\}$.

Hence, the set of test cases executed in the first phase are generated as follows. For each state $s \in S$, let q_s and z_s denote, respectively, the sequence in the state cover set which leads to s and the set of separating sequences generated for s. Then, the test cases generated in the first phase is given by $\bigcup_{s \in S} r.q_s.z_s$, where r is the reset sequence of the FSM and for two sets A and B of sequences, $A.B$ denotes the concatenation of two sets and is defined as $\{\alpha\beta | \alpha \in A \wedge \beta \in B\}$. This way, in addition to checking the existence of the states, the output and target state of the transitions which are included in the spanning tree are checked for conformance to the specification.

In the second phase of the HSI method, the output and the target state of the remaining transitions, not visited while traversing the state cover set, are checked using the following set of test cases. For each of the remaining transitions such as $s \xrightarrow{i/o} s'$, the set of all $r.q_s.i.z_{s'}$ sequences is added to the set of test cases.

3.2 Delta-Oriented Syntactic Structure

Inspired by DeltaJava [14], our test-models for an SPL are structured into a *core model* and a set of *delta models*. The core model describes the correct behavior of a valid configuration in the SPL. The implementation of other products is obtained by applying delta models to the core model. The structure of our models is defined by the syntax of DeltaJava, which is described below.

A core model comprises a set of Java classes and a set of interfaces, that is:

$$\textbf{core}\ \langle \textit{Feature names} \rangle \{ \langle \textit{Java classes and interfaces} \rangle \},$$

where feature names specify the set of features which are included in the configuration corresponding to the core model.

Delta models describe sets of changes to the core model. The structure of a delta in the DeltaJava language is given in Fig. 1. In this syntax, a delta model may add/remove fields, methods, or interfaces from classes in the core model. Also, it can modify the existing ones. A class can also be added or removed from a core model by applying a delta model. The keyword *after* can be used in order to specify the order of the application of a set of delta model to the core model. The *when* keyword is used to specify that this delta can be applied when a set of features are being included in the configuration. In the remainder of this paper, we only consider incremental delta-oriented models, i.e., those models that only

```
delta ⟨ name ⟩ [after ⟨ delta names ⟩]
      when ⟨application condition⟩ {
removes ⟨class or interface name⟩
adds class ⟨ name ⟩ ⟨ standard Java class⟩
adds interface ⟨ name⟩ ⟨standard Java interface⟩
modifies interface ⟨ name ⟩
      { ⟨ (remove | add| rename) method header clauses⟩ }
modifies class ⟨name⟩
      { ⟨ (remove | add | rename) field clauses⟩ |
        ⟨ (remove | add | rename) method clauses }⟩ }
```

Fig. 1. DeltaJava syntax.

add model classes, methods or fields. In this paper, we focus on an incremental subset of the syntax, designated in blue, which assumes a minimal core and incremental additions by various deltas. Particularly, in Section 5, we provide a semantic domain in terms of FSMs for a subset of these syntactic structures, which covers adding classes, methods and fields to a core FSM model.

4 Running Example

In this section, we present the syntax of a DeltaJava example, which is used throughout the rest of this paper. The core model of this example consists of one class, named *Bridge*. This class has a field that represents the availability of the bridge and also a set of functions, which manipulate and report the value of this field. The syntax of the core model is given in Fig. 2.

```
Core Bridge{
      Class Bridge{
        private boolean Avl;
        public Bridge() {Avl=true;}
        public void SetAvl(){Avl=true;}
        public void ResetAvl(){Avl=false;}
        public boolean CheckAvl(){return(Avl);}
      }
}
delta DPedLight when pedestrian Light {
      modifies Class Bridge{
        adds boolean Psig
        adds boolean CheckPsig(){return(Psig);}
        adds void SetPsig(){Psig=true;}
        adds void ResetPsig(){Psig=false;}
      }
}
```

```
delta DController when controller {
   adds Class Controller{
     private boolean Lsig,Rsig;
     public bridge b;
     public controller(){
     Lsig=false; Rsig=false;}
     public int CheckLsig(){return(Lsig);}
     public int CheckRsig(){return(Rsig);}
     public void GetReq(int id){
       if(b.CheckAvl()==true){
         if(id==0){
         Lsig=true;Rsig=false;}
       else{
         Rsig=true;Lsig=false;
       }
       b. ResetAvl();}
     }
     public void SetPassed(){
       Lsig=false;Rsig=false;
       b.SetAvl();}
   }
}
```

Fig. 2. Core- and delta models of the running example.

We consider two different delta models to be added to the core model given in Fig. 2. The first delta model consists of the addition of a class. The class *controller* controls the status of the lights in both side of the bridge in order to guarantee a mutually exclusive access to the bridge. This delta is added when the feature controller is included in a product. The second delta model is added to the core model when the pedestrian light feature is included in the product. This delta model consists of adding a field to the bridge class, which represents the status of the pedestrian light, as well as two methods, which can set and reset the value of the pedestrian light.

5 Delta-Oriented FSM Modeling

In this section, we define a semantic domain based on FSMs for the syntactic structure of DeltaJava models. We assume that the transitions in our test models concern the call / return behavior of a set of modules. The states in a test model concern a symbolic aggregation of concrete states, where each concrete state corresponds to a valuation of variables. The granularity of this abstraction is modeler's choice, as long as it respects the HSI assumptions. Moreover, it is assumed that the set of fields used and manipulated by a method call, its possible return values and its effect on the value of these fields are known.

To start with, we define the following basic concepts for our semantic domain.

Definition 5. *(Abstract Valuations) Assume a set V of variables and a set D of their possible values; for simplicity, we have left out typing information here and throughout the paper. Then $Val^V \subseteq 2^{V \to D}$, is an abstract valuation (i.e., a set of valuations) of V. The set of all such abstract valuations of V is denoted by VAL^V. We remove the superscript of an abstract valuation, if the set of variables is clear. For an abstract valuation $Val^V \subseteq 2^{V \to D}$ and for $V' \subseteq V$, we write $Val \downarrow V'$ to denote element-wise domain restriction of Val to V', that is leaving out the valuation of those variables not mentioned in V'.*

Definition 6. *(Object Structure) We formalize the structure of an object obj of class c, as a 3-tuple $(Id, Flds, Mtds)$, where Id is the object's unique identifier and Flds and Mtds, respectively, denote the set of fields and methods in the class c. (To avoid name clashes, we assume that all members of Flds and Mtds are prefixed with Id.) A method is represented by a 5-tuple $(Id, Inprms, Outprm, Clds, UsedVars)$, where Id, Inprms and Outprm, respectively, denote the name of the method and the list of the input parameters and the output returned by the method; Clds denotes the set of methods that are called in the body of this method and UsedVars is the set of variables read from or written to in the method. Note that UsedVars may comprise both members of Flds and* model variables. *The latter are variables that the test modeler has added to the model to capture unspecified details, e.g., associations and dependencies, without cluttering the model.*

In the rest of the paper, we recognize the components of the above-given tuples, by indexing the name of the intended component with the name of the

object or the method. For example, $Inprms_m$ denotes the input parameters of the method m. Next, we define the concept of post-condition for methods.

Definition 7. *(Effect and Return Value Functions) The effect of calling a method m is defined by a function $Effect_m : VAL^{Inprms_m \cup UsedVars_m} \rightarrow VAL^{UsedVars_m}$. Similarly, its set of admitted return values is defined by:*
$RetVal_m : VAL^{Inprms_m \cup UsedVars_m} \rightarrow 2^D$.

5.1 Core Model Semantics

In this section, we define the semantic domain for core models. The behavior of a core model results from execution of the methods called in the objects instantiated from the core model classes. (A conscious choice is to be made by the modeler as to which methods from which abstract states are included in the model.) Hence, the finite state machine describing the behavior of a set of objects is defined as follows.

Definition 8. *(Object FSM) An FSM $M(\mathcal{O}) = (S, s_0, I, O, \mu, \lambda)$ is a semantic model for a set $\mathcal{O}$ of objects from the set C of classes, if it satisfies the following conditions:*

- $S \subseteq VAL^V$ *where* $V \subseteq \bigcup_{o \in \mathcal{O}, m \in Mtds_o} UsedVars_m$ *is a subset of model variables and fields in $\mathcal{O}$; this means that each state in S is* an abstract valuation *of a subset of model variables and fields.*
- $I \subseteq \bigcup_{o \in \mathcal{O}, m \in Mtds_o} \{Id_m\} \times VAL^{Inprms_m}$*; this means that each input in the input symbols set comprises a method name and a set of passed arguments.*
- $O \subseteq D$ *is the set of possible return values of the method calls in I.*
- $\mu : S \times I \rightarrow S$*, is a transition function satisfying the following conditions:*

  (1) $\forall_{o \in \mathcal{O},\ m \in Mtds_o,\ val \in VAL^{Inprms_m},\ i \in I,\ s,s' \in S} \cdot \mu(s,i) = s' \wedge i = (Id_m, val)$
  $\Rightarrow Effect_m(s \downarrow UsedVars_m \times val) \subseteq s' \downarrow UsedVars_m$,
  (2) $\forall_{s \in S} \cdot \exists_{x \in I^*} \cdot\ (s_0, x) = s$
  (3) $\exists_{r \in I^*} \cdot \forall_{s \in S} \cdot\ \mu(s,r) = s_0$

- $\lambda : S \times I \rightarrow O$ *is an output function satisfying the following condition:*

$$\forall_{o \in \mathcal{O},\ m \in Mtds_o\ val \in VAL^{Inprms_m},\ i \in I,\ o \in O,\ s \in S} \cdot \lambda(s,i) = o \wedge$$
$$i = (Id_m, val)\ \Rightarrow RetVal_m(s \downarrow UsedVars_m \times val) = o.$$

Our notion of abstract states are reminiscent of similar notions (based on the category-partition method) in the literature [19]. Regarding the transition function, condition (1) specifies that there can be a transition from one state to another, labeled with a method call as input, only if this method call maps one of the concrete evaluations of the used variables in the source to another concrete valuation in the target state. Condition (2) requires that all states included in the set of states are reachable from the initial state. Condition (3) postulates that the given FSM has a reset sequence r. Regarding the output function, the

condition specifies that the output of the FSM for each given input is exactly the set of admitted outputs for the corresponding method.

A test model for core, defined below, is then an object FSM comprising a set of objects from the core model classes.

Definition 9. *(Test Model for Core) A test model for core is a minimal object FSM $M(\mathcal{O})$ such that each object in $\mathcal{O}$ is instantiated from a class in the core model.*

For example the FSM corresponding to the core model in the running example is demonstrated in Fig. 3 (a). This FSM is minimal and it satisfies the reachability condition. The reset sequence of this FSM is $SetAvl()$.

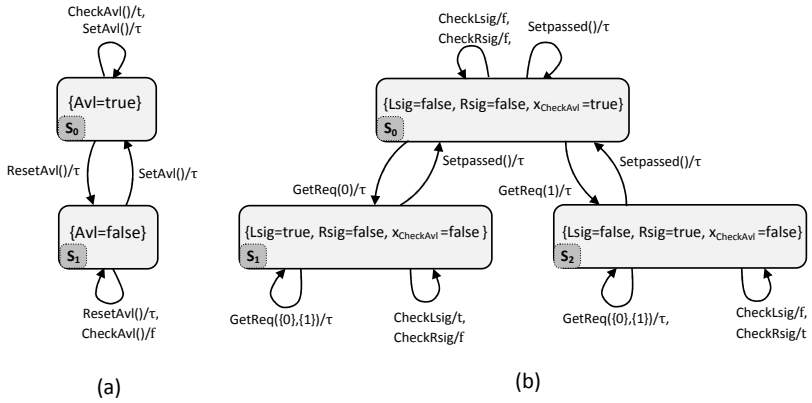

Fig. 3. (a) FSM modeling the bridge class, (b) FSM modeling the controller class

5.2 Delta Application

In this section, we define the semantic domain for delta models and the application of a delta to a core model. As mentioned in Sect. 3.2, a delta comprises a set of operations applying changes to the core model. In order to give a practical definition to a delta model and the type of changes that it can make to the core model, we focus on adding a class, on one hand and adding a set of fields and methods, on the other hand. The reason we combine adding fields and methods in one step is that often adding new methods requires some additional fields. Moreover, in several cases the new abstract valuations (additional state-partitions) due to the additional fields can only preserve minimality, if new methods are also added to tell them apart. We leave the deltas concerning removals and modifications of methods and removal of fields for future work. Hence, for now we

are assuming that the core model comprises the least mandatory set of features and the model regarding each product is generated incrementally from the core model.

We proceed by defining the effect of applying a delta containing each of the above-mentioned changes on the core model's FSM.

Adding a Class The test model for the added class has the structure and abides by the constraints of object FSMs given in Definition 8. Hence, we assume that the test model for the added class c is given as a minimal object FSM $M_d(\mathcal{O}_d)$ where $\mathcal{O}_d$ only contains objects of class c with a fresh identifier (not mentioned among the identifier of core objects and other deltas).

For example, the FSM describing the behavior of an object of the controller class is depicted in Fig. 3 (b). In this figure, $x_{CheckAvl}$ is an extra model variable included in the state, representing the returned value of *CheckAvl*() and cutting the dependency with the core model. The result of adding a class to the core model is defined as follows.

Assume that the test models for the core and the delta models are object FSMs $M(\mathcal{O}) = (S, s_0, I, O, \mu, \lambda)$ and $M_d(\mathcal{O}_d) = (S_d, s_0^d, I_d, O_d, \mu_d, \lambda_d)$, respectively. In order to define the composition of the core and the delta, we first specify the possible connections between the model variables of delta and core. Assuming that V and V_d, respectively, denote the variables in the domain of the states in S and S_d, then, the (partial) composition function $\gamma : V_d \to V$ specifies which (model) variables in V_d should match which variables in V. Moreover, the methods of the delta class can initiate method calls to instances of the core class included in the delta class (if any). Here, for the sake of simplicity, we consider that each delta method can contain at most one method call to the core, but the generalization to a sequence of core method calls is straightforward. We assume that the set of methods in the core model and the set of methods in the delta model are denoted, respectively, by *MTD* and *Mtds*.

Definition 10. *The result of composing the above-given models M and M_d with regards to γ is an FSM $M'(\mathcal{O}') = (S', s_0', I', O', \mu', \lambda')$, where:*

- $S' = \{val \in VAL^{V \cup V_d} \mid val \downarrow V \in S \wedge val \downarrow V_d \in S' \wedge \ \forall_{v_d \in V_d,\ v \in V} \cdot\ \gamma(v_d) = v \Rightarrow val \downarrow \{v_d\} = val \downarrow \{v\}\}$*; for the composition to be well-defined, we assume V and V_d to be disjoint,*
- s_0' *is the initial state such that* $s_0' \downarrow V = s_0$ *and* $s_0' \downarrow V_d = s_0^d$,
- $I' = I \cup I_d$
- $O' = O \cup O_d$
- $\mu' : S' \times I' \to S'$*, is the transition function. For each $i \in I'$, we distinguish the following three cases:*
  - $i \in I$ *concerns a method call from the core; then, the following condition should be satisfied*

$$\begin{array}{l}\forall_{m \in MTD,\ s_1', s_2' \in S'} \cdot i = (Id_m, val) \Rightarrow \big(\ \mu'(s_1', i) = s_2' \Leftrightarrow \\ \exists_{s_1, s_2 \in S} \cdot s_1 \downarrow UsedVars_m = s_1' \downarrow UsedVars_m \wedge \\ s_2 \downarrow UsedVars_m = s_2' \downarrow UsedVars_m \ \wedge\ \mu(s_1, i) = s_2\ \big)\end{array}$$

- *$i \in I_d$ concerns a method call from delta that does not have any nested call to the core; then, the following condition should be satisfied*

$$\forall_{m \in Mtds,\ s'_1, s'_2 \in S'} \cdot i = (Id_m, val) \Rightarrow \big(\mu'(s'_1, i) = s'_2 \Leftrightarrow \exists_{s^d_1, s^d_2 \in S_d} \cdot s^d_1 \downarrow UsedVars_m = s'_1 \downarrow UsedVars_m \wedge s^d_2 \downarrow UsedVars_m = s'_2 \downarrow UsedVars_m \ \wedge \ \mu_d(s^d_1, i) = s^d_2 \big)$$

- *$i \in I_d$ concerns a method call from delta that has a nested method call n_i to the core; then the following condition should hold:*

$$\forall_{m \in Mtds, n \in MTD, s'_1, s'_2 \in S'} \cdot i = (Id_m, val) \wedge n_i = (Id_n, val_n) \ \Rightarrow \mu'(s'_1, i) = s'_2 \Leftrightarrow \exists_{s^d_1, s^d_2 \in S^d} \cdot s^d_1 \downarrow UsedVars_m = s'_1 \downarrow UsedVars_m \wedge s^d_2 \downarrow UsedVars_m = s'_2 \downarrow UsedVars_m \wedge \mu_d(s^d_1, i) = s^2_d \wedge \exists_{s_1, s_2 \in S} \cdot s_1 \downarrow UsedVars_n = s'_1 \downarrow UsedVars_n \wedge s_2 \downarrow UsedVars_n = s'_2 \downarrow UsedVars_n \wedge \mu(s_1, n_i) = s'_2$$

– *$\lambda' : S' \times I' \to O'$ is the output function; for each $i \in I'$, we distinguish the following two cases:*
  - *either $i \in I$, then the following condition should hold:*

$$\forall_{m \in MTD,\ o \in O', s' \in S'} \cdot i = (Id_m, val) \ \Rightarrow \ \lambda'(s', i) = o \Leftrightarrow \exists_{s \in S} \cdot s \downarrow UsedVars_m = s' \downarrow UsedVars_m \wedge \ \lambda(s, i) = o$$

  - *or $i \in I_d$, then the following condition should hold:*

$$\forall_{m \in Mtds,\ o \in O', s' \in S'} \cdot i = (Id_m, val) \Rightarrow \lambda'(s', i) = o \Leftrightarrow \exists_{s_d \in S_d} \cdot s_d \downarrow UsedVars_m = s' \downarrow UsedVars_m \wedge \ \lambda_d(s_d, i) = o$$

In the definition of transition function, a case distinction is made based on whether the method calls (in the delta model) have a nested method call or not. In the former case the valuations of the variables belonging to both core and delta models can change in the target state while in the latter case only the valuation of the variables belonging to the delta model can change. In the definition of output function these two cases are defined as one since the effect of the output of the inner method calls, if any, of a method call in the delta model is captured by the corresponding model variables which are included in the states of the delta model.

Fig. 4. (a) demonstrates the FSM resulting from the addition of the controller class to the bridge class. Note that the γ function is defined to match the valuation of the model variable $x_{CheckAvl}$ in the delta with the variable Avl in the core.

Theorem 1. *Based on the assumptions made about the core model and the delta model, the resulting FSM of Definition 10 satisfies the assumptions (1)-(4) of Definition 2.*

Note that the last constraint of Definition 10 is implementation-dependent and hence, it can only proven without sufficient assumptions on the implementation. This is out of the scope of the present paper.

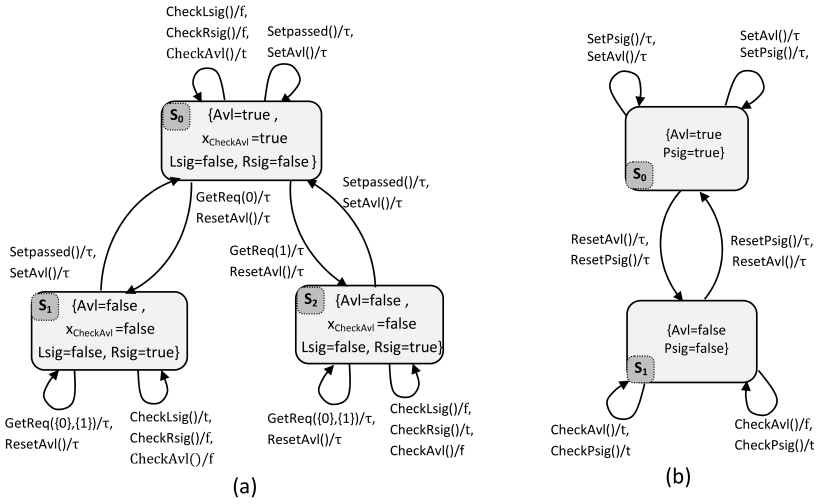

Fig. 4. (a) FSM resulted from adding the delta model *DController*, to the core model,(b) FSM resulting from adding delta *DPedLight* to the core model

Adding Fields and Methods In this section, we discuss the effect of adding a set of fields and methods to the core module.

Let X and E, respectively, denote the set of fields and methods added by a delta. Also, assume that V denotes the variables in the domain of the states in the core model, and that a method can comprise method calls. The addition of X and E to the core FSM results in another FSM in which the abstract states and transitions accommodate X and E. The formal definition of the application function has a similar structure to the case of adding a class.

Theorem 2. *Assumptions (1)-(4) of Definition 2 are preserved under the addition a set of fields and methods to a core FSM model.*

As an example, Fig.4. (b) demonstrates the FSM resulting from the addition of the delta *DPedLight*, to the core model. This delta adds a new field, namely, *Psig*, and two methods, namely, *SetPsig* and *ResetPsig*, to the class *Bridge*.

6 Delta-Oriented Testing

In this section, we explain the incremental test-case generation method. In the remainder of this section, we assume that the core model is an object FSM such as $M(\mathcal{O}) = (S, s_0, I, O, \mu, \lambda)$ and the set of all methods of the classes in this core model are denoted by MTD. The state cover set and the separating family of sequences computed for M are, respectively, denoted by Q and Z. We assume that $q_s \in Q$ denotes a sequence in the state cover set that ends in state s and z_s denotes the set of sequences which separate s from other states. For example,

the state cover set and the separating family of sequences for the core model represented in Fig. 3 are, respectively, $Q = \{\varepsilon, \mathit{ResetAvl}()\}$ and $Z = \{z_{s_0}, z_{s_1}\} = \{\{\mathit{CheckAvl}()\}, \{\mathit{CheckAvl}()\}\}$.

6.1 Test-Case Generation for Class Addition

Let $M_d(\mathcal{O}_d) = (S_d, s_0^d, I_d, O_d, \mu_d, \lambda_d)$, be the FSM that is composed with core model, with regards to the composition function γ, as a result of adding the new class to the core module. We assume that the state cover set and the family of separating sequences for this FSM are, respectively, denoted by Q_d and Z_d. The resulting object FSM is $M'(\mathcal{O}') = (S', s'_0, I', O', \mu', \lambda')$, as defined in Sect. 5.2, and the set of test cases for this FSM are computed as follows.

In order to compute the new state cover set, denoted by Q', we need to build the spanning tree of M'. Assuming that $P_d(S_d, E_d)$ is the spanning tree built for M_d, where S_d denotes the set of vertices and $E_d \subseteq S_d \times I_d \times S_d$, denotes the set of edges in this tree, and $P(S, E)$ is the spanning tree built for M, where S and $E \subseteq S \times I \times S$, are, respectively, the set of vertices and edges in this tree. Moreover, we assume that V and V_d, respectively, denote the set of variables included in S and S_d. The spanning tree for M', denoted by $P'(S', E')$, where $E' \subseteq S' \times I' \times S'$, is built using P and P_d as follows. Note that each state $s' \in S'$ can be represented by (s, s_d), where $s \in S$ and $s_d \in S_d$, that is $s' \downarrow V = s \downarrow V$ and $s' \downarrow V_d = s_d \downarrow V_d$.

Starting from the root of the tree, that is (s_0, s_0^d), for each state such as (s, s_d), we add the following child nodes:

1. (s', s_d), where for some $i \in I$, we have $(s, i, s') \in E$
2. (s, s'_d), where for some $i \in I'$ which is corresponding to a method call that does not contain any nested method calls, we have $(s_d, i, s'_d) \in E_d$
3. (s', s'_d), where for some $i \in I'$ that contains a method call denoted by $j \in I$, we have $(s_d, i, s'_d) \in E$ and $\mu(s, j) = s'$.

Assuming that $|S| = n$, $|S_d| = m$ and $|S'| = n'$, then the worst-case complexity of computing the spanning tree is $O(n'(m + n))$. The state cover set is computed by traversing the resulting spanning tree.

The family of separating sequences Z' is defined as $\bigcup_{s' \in S}\{z'_{s'}\}$, where for each state $s' = (s, s_d) \in S'$, we have that $z'_{s'} = z_s \cup z_{s_d}$.

For example, the state cover set and the family of separating sequences for the FSM corresponding to the *controller* class in Fig.4. (b) are as follows: $Q_d = \{\varepsilon, \mathit{GetReq}(0), \mathit{GetReq}(1)\}$, $Z_d = \bigcup_{i=0}^{2} \big(z_{s_i} = \{\mathit{CheckLsig}(), \mathit{CheckRsig}()\}\big)$.

Hence, the state cover set and the family of separating sequences for the FSM resulted adding the class are: $Q' = Q = \{\varepsilon, \mathit{ResetAvl}()\}$, $Z' = \bigcup_{i=0}^{2} \big(z_{s_i} = \{\mathit{CheckAvl}(), \mathit{CheckLsig}(), \mathit{CheckRsig}()\}\big)$, respectively.

A special case of adding a class is when there are no nested method calls. In such a case the state cover set is equal to the state cover set of the core model that is $Q = Q'$. The computation of separating sequences remains intact with respect to the general case.

Complexity Analysis The difference of complexity of the delta-oriented testing approach compared to the HSI method, in this case, is in the computation of the family of separating sequences. As explained above, in this case the delta-oriented approach obtains the family of the separating sequences for the new FSM, just using Z_d and Z. Hence, defining $m = |S_d|$, and $q = |I_d|$, the complexity of computing Z', using the delta-oriented approach, is $O(qm^2) + f_u$, where f_u is the the complexity of computing the union of two sets. Assuming that the delta has n' states where $n' \leq m \cdot n$, and $p = |I'|$, the complexity of computing the family of separating sequences, using the HSI method, for this FSM is $O(pn'^2)$. It should be noticed that this computation is done for each product in a product line separately, where the number of the products can increase exponentially in terms of the features. Practically, in a product line we have $m \ll n$, hence $O(qm^2) + f_u \ll O(pn'^2)$. In other word, there can be a substantial gain in calculating the separating sequences using the delta-oriented approach.

7 Empirical Results

In order to check the efficiency of the proposed algorithm, we applied our method to a software system from the health-care domain. In order not to reveal the structure of the commercial system, we dispense with the details that are not necessary for understanding the experimental results. The core logic of this system includes six classes and its main functionality is to detect devices in the surroundings and control users' access to them. Each user can create and complete a set of tasks after accessing a device. We considered the proportion of time required to generate test cases for 4 different models in two cases: using the delta-oriented approach, and using the plain monolithic HSI method. (In this work, we only consider the reduction in the test-case-generation time; we leave the study of the test-case-execution time as future work).

In order to compute the test-case generation time, we performed the algorithms in both methods in a step-by-step manner and manually, while counting the basic computation steps in these algorithms. Because these basic steps are common to both methods and consume a constant amount of time, we could hence come up with a precise comparison of the time required for test-case generation.

First, we considered a core FSM with 11 abstract states and 74 transitions. This core model included a set of objects, which model a group of users, devices and tasks created by users. Then, we applied a delta which comprised the addition of a method to a class in order to enable modification of a field in the core model. The result of applying this delta is another FSM with the same number of states and 85 transitions. Using the delta-oriented approach for generating test cases resulted in a 50-percent reduction in test-case generation time. This difference is due to that the spanning tree and the family of separating sequences are computed anew in the HSI method, while the delta oriented approach reuses the sequences computed for the core model.

We also applied a delta concerning the addition of an object of a task to the core model which resulted in another FSM with 16 states and 89 transitions. In this case, applying the delta-oriented approach resulted in a 40-percent reduction in test-case generation time. (For more detailed data, we refer to Fig. 5.)

Subsequently, we considered another core model including 21 abstract states and 167 transitions. We applied a delta comprising the addition of the same method as above to the core model, which resulted in the same number of states and 188 transitions. Applying the delta-oriented approach results in a 50-percent reduction in the test-case generation time.

The last delta in this software product line comprised the addition of an object of a device to the last core model, with 37 states and 215 transitions. The reduction in the test-case generation time in this latter case is 30 percent.

The results show that in cases that we can reuse the separating sequences and the state cover set of the core model, such as the addition of a set of methods that do not change the number of states, the delta-oriented approach can be very efficient. The above-mentioned results are summarized in Fig. 5.

Core Model		After Applying Delta		HSI Test Case	Delta-Oriented Test	Reduction
Num. States	Num. Trans.	Num. States	Num. Trans	Generation Steps	Case Generation Steps	
11	74	11	85	194	388	50
11	74	16	89	253	421	40
21	167	21	188	419	824	50
21	167	37	215	731	1043	30

Fig. 5. Results obtained from test-case generation for the case study

8 Conclusions and Future Work

In this paper, we introduced test models and test-case generation methods for delta-oriented FSM-based testing, based on the DeltaJava syntax. Our test-case generation method is a lifting of the incremental test-case generation for the HSI method, using a higher level of abstraction suitable for our DeltaJava-based models. We showed, both using complexity analysis and by application to a case study, that the delta-oriented approach can increase the efficiency of test-case generation.

We are studying realistic, yet more relaxed fault models (than those underlying the HSI method). Such a fault model can capture the possible mutual effects of different behavior in deltas and core. Then, we will identify parts of test cases that need not be re-executed and also independent pieces of behavior that can be reduced, e.g., using partial-order reduction [16]. Moreover, we intend to extend our approach to the full syntax of DeltaJava and in particular, consider modifying and removing methods, building upon the results of [9, 11]. Finally, we plan to implement our approach in a programming environment and organize more extensive experiments with our industrial partner.

References

1. R. Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley, 2000.
2. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures*, vol. 3472 of LNCS. Springer, 2005.
3. K. El-Fakih, N. Yevtushenko, and G. von Bochmann. FSM-based incremental conformance testing methods. *IEEE TSE*, 30(7):425–436, 2004.
4. E. Engström and P. Runeson. Software product line testing - a systematic mapping study. *Inf. Softw. Technol.*, 53(1):2–13, 2011.
5. H. S. Hong, Y. R. Kwon, and S. D. Cha. Testing of object-oriented programs based on finite state machines. In *Proc. of APSEC 1995*, pp. 234–241. IEEE CS, 1995.
6. A. Jääskeläinen. Filtering test models to support incremental testing. In *Proc. of TAIC PART 2010*, vol. 6303 of *LNCS*, pp. 72–87. Springer, 2010.
7. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proc. of the IEEE*, 84(8):1090–1123, 1996.
8. M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. Incremental model-based testing of delta-oriented software product lines. In *Tests and Proofs*, vol. 7305 of *LNCS*, pp. 67–82. Springer, 2012.
9. G. A. Németh and Z. Pap. The incremental maintenance of transition tour. *Fund. Inf.*, 129:279–300, 2014.
10. S. Oster, A. Wübbeke, G. Engels, and A. Schürr. Model-based software product lines testing survey. In *Model-based Testing for Embedded Systems*, pp. 339–381. CRC Press, 2011.
11. Z. Pap, M. Subramaniam, G. Kovács, and G. A. Németh. A bounded incremental test generation algorithm for finite state machines. In *Proc. of TestCom/FATES 2007*, pp. 244–259, 2007.
12. B. Rumpe. Model-based testing of object-oriented systems. In *Proc. of FMCO 2002*, vol. 2852 of *LNCS*, pp. 380–402. Springer, 2003.
13. K. Sabnani and A. Dahbura. A protocol test generation procedure. *Comput. Netw. ISDN Syst.*, 15(4):285–297, 1988.
14. I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. of SPLS 2010*, pp. 77–91, Springer, 2010.
15. A. Simão and A. Petrenko. Fault coverage-driven incremental test generation. *The Computer Journal*, 53:1508–1522, 2010.
16. S. Tasharofi, R. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *Formal Techniques for Distributed Systems*, vol. 7273 of *LNCS*, pp. 219–234. Springer, 2012.
17. T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis strategies for software product lines. Technical report FIN-004-2012, School of Computer Science, University of Magdeburg, 2012.
18. M. Utting and B. Legeard. *Practical model-based testing: a tools approach.* Morgan Kaufmann, 2010.
19. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. In *Proc. of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '02)*, pp. 112–122. ACM, 2002.