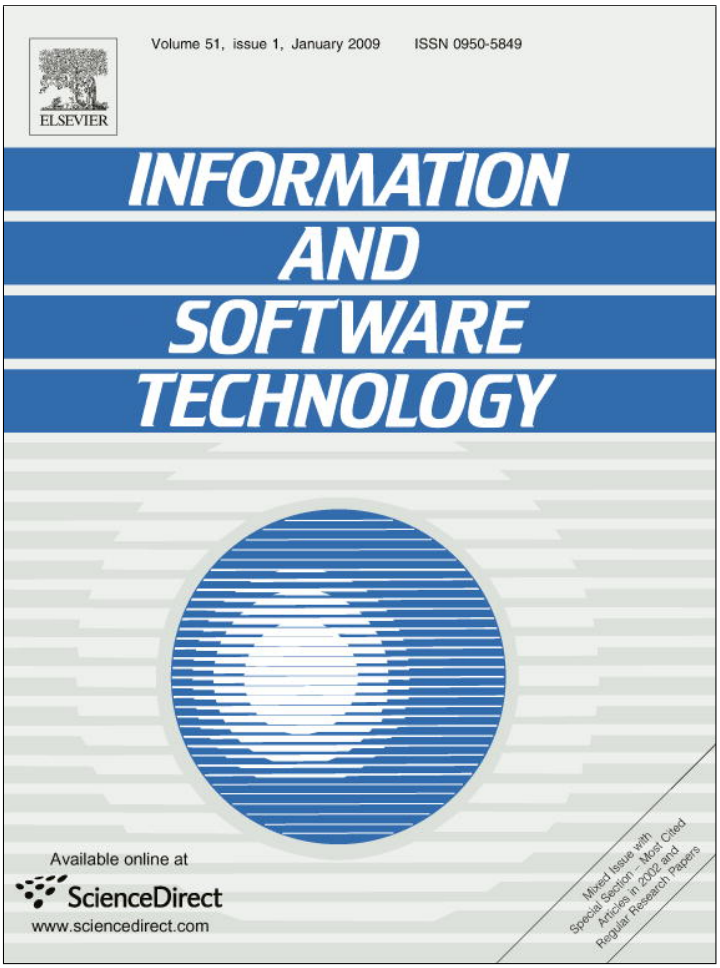

Information and Software Technology 51 (2009) 16–30

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Automating regression test selection based on UML designs

L.C. Briand [b,*], Y. Labiche [a], S. He [a]

[a] Carleton University, Software Quality Engineering Lab, 1125 Colonel By Drive, Ottawa, ON K1S 5B6, Canada
[b] Simula Research Laboratory and University of Oslo, P.O. Box 134, Lysaker, Norway

ARTICLE INFO

Keywords:
Regression testing
Test selection
Object-oriented software engineering
UML

ABSTRACT

This paper presents a methodology and tool to support test selection from regression test suites based on change analysis in object-oriented designs. We assume that designs are represented using the Unified Modeling Language (UML) 2.0 and we propose a formal mapping between design changes and a classification of regression test cases into three categories: Reusable, Retestable, and Obsolete. We provide evidence of the feasibility of the methodology and its usefulness by using our prototype tool on an industrial case study and two student projects.

1. Introduction

The purpose of regression testing is to test a new version of a system so as to verify that existing functionalities have not been affected by new system features [12,19]. Regression test selection is the activity that consists in choosing, from an existing test set, test cases that can and need to be rerun to ensure existing, unmodified functionalities are still working correctly. Reducing the number of regression test cases to execute is an obvious way of reducing the cost associated with regression testing, which is usually substantial [19].

The main objective of selecting test cases that need to be rerun is to identify regression test cases that exercise modified parts of the system. This is referred to as *safe regression testing* [27] as, in the ideal scenario, it identifies all test cases in the original test set that can reveal one or more faults in the modified program. In order to achieve such an objective, we need to classify test cases in an adequate manner. Adapting definitions in [18], we aim to automatically classify test cases as follows:

- *Obsolete*: A test case that cannot be executed on the new version of the system as it is 'invalid' in that context. Classifying a test case as obsolete may lead to either modifying the test case and corresponding test driver or removing the test case from the regression test suite altogether.
- *Retestable*: A test case is still valid but needs to be rerun for the regression testing to be safe.
- *Reusable*: A test case that is still valid but does not need to be rerun to ensure regression testing is safe.

Regression test selection can be based on source code control flow and data flow analysis. In this case, based on information about the code of the two versions of the program, one selects test cases that execute new or modified statements (in the new version of the program) to be rerun, or formerly executed statements that have been deleted from the original version of the program [28]. This selection is based on an analysis of the changes at the source code level to determine their impacts on test cases. A drawback is that it requires that the changes be already implemented but it can be very precise in terms of selecting a minimum regression test set as complete change information is available. (Precision varies among code-based regression test selection strategies [27].) An alternative, and complementary approach, is to use architectural/design information available in design models [31]. In this case, selected test cases execute new or modified model elements (e.g., class operations in the case of a UML model), or model elements formerly executed but deleted from the original version. The impact of possible changes is first assessed on the design of the last version of the system, by comparing what would be the new design with the existing design. The change impact magnitude is then assessed and a change management group decides whether to implement it in the next version of the source code. Assuming there is traceability between the design and regression test cases, we can, at the end of the design impact analysis, automatically determine what regression test cases will need to be rerun and what test cases should be removed from the regression test suite as they are no longer valid. Therefore, one main advantage of a design-based approach is the possibility of performing early regression test planning and effort estimation.

Another motivation for working at the architecture/design level is in part motivated by efficiency as discussed in [12,19]. Leung and White note that the cost of selecting regression test cases to rerun must be lower than the cost of running the remaining test cases for

* Corresponding author.
E-mail addresses: briand@simula.no (L.C. Briand), labiche@sce.carleton.ca (Y. Labiche), siyuan@sce.carleton.ca (S. He).

0950-5849/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2008.09.010

test selection to make sense. In [12], it is suggested that working closer to the architectural level may be more efficient than at the source code level. To summarize, the motivations for investigating test selection techniques at the architectural or design level are fourfold, the last two points being related to efficiency:

- We can estimate the extent of the effort required for regression testing earlier on, at the end of the design of the new system version. Estimating regression test effort is an important part of impact analysis and one of the decision criteria to include a change in an upcoming version (the modification-request problem [12]).
- Regression test tools can be largely programming language independent and they can be based on a standard, widely used design notation such as the UML.
- Traceability between code and test cases requires to store and update dependencies between test cases and code statements or other representations of the code, e.g., control flow graphs. Managing traceability at the design level may be more practical than doing it at the code level as it enables the specification of dependencies between test cases and the system at a higher level of abstraction.
- No complex static and dynamic code analysis is required (e.g., data flow, slices). The latter analysis being usually necessary for identifying possible dynamic bindings between methods at run-time [29]. Using UML designs enables the easy retrieval of relevant static and dynamic information (e.g., class interactions at run-time from sequence diagrams) since they provide information at a higher level of abstraction than the source code.

There are, of course, potential drawbacks too. For example, using designs for impact analysis and test selection requires the designs to be complete, internally consistent, and up-to-date. Though CASE tools are getting better at providing round-trip engineering capabilities, this is not always easy in practice. Another issue is that some (potentially faulty) changes to the source code may not be detectable from UML documents, e.g., a change in a method's body (a more efficient algorithm is implemented) may not be visible from class, sequence or statechart diagrams, suggesting that model-based and code-based approaches are complementary. These issues will be discussed in further details in the following sections.

In this paper, we focus on automating regression test selection based on architecture and design information represented with the Unified Modeling Language (UML) and traceability information linking the design to test cases. Our focus on the UML notation is a practical choice as it has become the industry de-facto standard. The original test set from which to select can contain both functional and non-functional system test cases. From a UML standpoint, functional system test cases test complete use case scenarios.

The rest of the paper is structured as follows. Since UML is only a notation, we first precisely describe the assumptions we make regarding the way it is used (Section 2). The following section describes the detected changes from UML class and use case/sequence diagrams as well as their impact on the classification of test cases (Section 3). To do so, we provide both intuitive definitions and a formal mapping using set theory. In Section 4, we analyze our model-based regression test selection strategy in the light of the framework proposed in [27], though this framework has been originally defined for white-box regression test selection strategies. Section 5 briefly introduces the functionality of the Regression Test Selection Tool (RTSTool) we built based on the principles introduced in Section 3. Sections 6 and 7 report the details of case studies and further discuss related works, respectively. Conclusions and future directions are then drawn in Section 8.

2. Assumptions on the use of the UML notation

This section focuses on the testability of UML diagrams, that is the extent to which they can be used to support test automation. As UML is only a notation, we need to make a number of assumptions about the way UML diagrams are used [6] to automate their analysis and facilitate traceability between test cases and the UML models. Though what we write in this section should not be surprising to the experienced UML practitioner, it needs to be clarified so as to automate our regression test selection methodology.

2.1. Consistency assumption and design by contract

First of all, we assume the different UML diagrams we rely on, i.e., use case, sequence and class diagrams, are consistent with each other. Otherwise, one cannot guarantee the validity of any UML-based analysis. For instance, if an operation has been deleted from a class in the class diagram, we assume the sequence diagrams in which the operation appears in the label of a message have been updated. Consistency checking can be easily implemented [4] and is a separate issue from the focus of the current paper. Note that modern modeling environments, such as Rational Software Architect [14], already support such consistency analysis.

Following the well-known Fusion method [9] and the Design By Contracts principles [20,21], we assume that class operations are described by providing their precondition and postcondition. In the context of UML, such contracts are typically described using the Object Constraint Language (OCL [32]). We also assume that class invariants are provided in OCL.

2.2. From use cases to sequence diagrams

Another issue is related to the combined use of use cases and sequence diagrams. We assume that with each use case we associate a unique sequence diagram specifying the possible object interactions that realize all possible use case scenarios. In practice, scenarios can be specified across several sequence diagrams to improve their readability but we assume there is only one, complete sequence diagram.[1] We also assume, following best practices, that sequence diagrams be named [16], and that they be named after the use cases they realize. As a notational convention, sequence diagram A refers to the sequence diagram for use case A.

Use cases relate to each other in the use case diagram by means of include, extend and generalization relationships [2]. For instance, in an Automated Teller Machine (ATM) system, the `DoTransaction` use case includes use case `InsertCard`. In other words, common functionalities across use cases are factorized out to reduce complexity in the use case diagram and use case textual descriptions. This also results in simpler sequence diagrams that focus only on the event flows of the corresponding use cases rather than on the event flows of included or extension use cases. Definitions for include and extend use case relationships (with extension points [24]) allow a clear identification of when in the corresponding flow of events a use case invokes another use case.[2] Moreover, UML 2.0 provides a mechanism, namely Interaction Uses, to translate use case relationships into sequence diagrams [24]. This mechanism applies to both include and extends use case relationships and allows: (1) Complete scenarios possibly exercising several use cases

[1] There is no technical or theoretical difficulty in merging sequence diagrams modeling different scenarios of a same use case into one complete sequence diagram.

[2] An include relationship between use cases means that the base use case explicitly invokes another use case at a location specified in the base. An extend relationship between use cases means that the base use case implicitly invokes another use case at a location specified indirectly (i.e., conditions, extension points) by the extending use case [2].

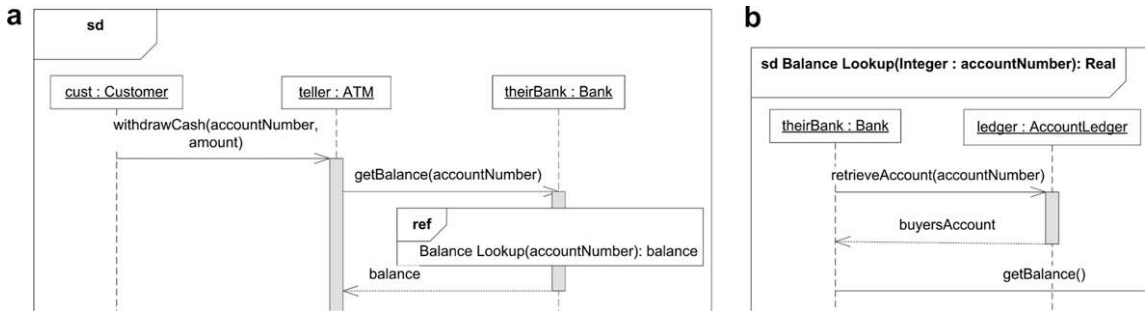

Fig. 1. Example of interaction use (excerpts from [1]).

to be derived when building test cases; (2) traceability between test cases and those scenarios when automatically classifying test cases.

The example in Fig. 1 shows a sequence diagram (a) that refers to another sequence diagram (b) using an interaction use (identified by the `ref` keyword). Note that the name of the referred sequence diagram and its formal parameters (Fig. 1b) match the name of the interaction use and its arguments in the referring sequence diagram (in the contents area of the `ref` box).

Generalization among use cases is similar to generalization among classes. The child use case inherits the behavior and meaning of the parent use case, the child may add to or override the behavior of its parent, and the child may be substituted any place the parent appears [2]. Again, we need to determine the impact of this relationship on the corresponding sequence diagrams. Due to the substitutability principle, both the parent and child use cases (i.e., their sequence diagrams) must have the same sequence of triggered `Fragments`, such that the target classes involved in the child's triggered `Fragments` are identical or child classes of the ones in the parent's triggered `Fragments`. Which one of the use cases is actually executed at run-time is usually determined by the inputs of an actor and the state of the system. A generalization between use cases then maps to a generalization in the class diagram.

As an example, consider an ATM system where a customer (actor) can perform transactions such as a withdrawal or deposit. The actor `Customer` is associated with use case `DoTransaction`, which includes use case `PerformTrans`, which is itself a parent use case for `PerformWithdrawal` and `PerformDeposit`. Fig. 2 provides partial sequence diagrams for use cases `DoTransaction` (a), `PerformWithdrawal` (b) and `PerformDeposit` (c). The sequence of triggered `Fragments` of `PerformTrans` (sequence diagram not shown) contains only message `doTrans()`. This operation is defined in abstract class `Transaction` (class diagram not shown) which is specialized by classes `Withdrawal` and `Deposit`. As shown in Fig. 2a, the only class that appears in the including use case is the parent class `Transaction` (the one that appears in included use case `PerformTrans`), and the actual target class of the message in the possibly included use cases (through generalization) is specific to the child use case, e.g., `Withdrawal`, `Deposit` in (b) and (c).

2.3. Applications

These above principles are applied in two different contexts to improve testability: deriving functional system test cases, and mapping test cases to complete scenarios (i.e., scenarios that may traverse several sequence diagrams).

The strategy we used for deriving functional system test cases in our case studies (see Section 6) has been described in [3]. This strategy is based on the derivation of use case dependency sequences that are transformed into test cases from the scenarios described in the corresponding sequence diagrams. Therefore, when extending a scenario in a base use case with a scenario from an included use case, it is then important to identify where in the former scenario (i.e., in the sequence of messages) the extension occurs.

When classifying test cases (Section 3.6), it is important to (i) first identify which sequence diagrams are triggered by a particular test case, and then (ii) verify that those test cases are valid: they execute feasible scenarios as described in the identified sequence diagrams.

(i) We assume classes are classified according to the types of functionality they provide. In [6], as in other OO development methods, one category is *Boundary*, also sometimes referred to as *Interface*. Those classes are managing the interaction with *actors* (e.g., GUI, device) and forward requests to other classes. Test drivers emulate the behavior of actors and execute system test cases by interacting solely with boundary classes: the only operations test drivers implementing test cases are directly invoking belong to boundary classes. A test case (using the messages to boundary classes it contains) can thus be uniquely mapped to a sequence diagram (using its messages to boundary classes).
(ii) To check whether a test case is valid, it is important to identify where in base sequence diagrams other sequence diagrams are invoked. This can be automated using the use case diagram and sequence diagrams, provided that the testability principles described in previous sections are followed. A tool can easily build complete sequence diagrams for the use cases directly triggered by actors, thus removing include and extend relationships. Note that in the case of an include or extend relationship of a parent use case, the tool has to record a possible set of invoked sequence diagrams (i.e., corresponding to the child use cases). Once we have complete sequence diagrams, we can then determine whether a test case is a valid sequence of actions for the sequence diagram it is mapped to in step (i).

3. Determining the impact of design changes

We present in this section the design changes that are being considered and detected to drive regression test selection (Sections 3.3 and 3.4). We then identify one critical issue involved in automating this process (Section 3.5) and precisely define the rules we follow to classify regression test cases (Section 3.6). The classification of changes and their use during test case selection are described informally as well as formally using set theory and first order logic. We limit the formalism employed (Section 3.2) to what we consider necessary to keep our definitions as simple and intuitive as possible while providing unambiguous definitions for the objective of this paper: regression test classification. To facilitate

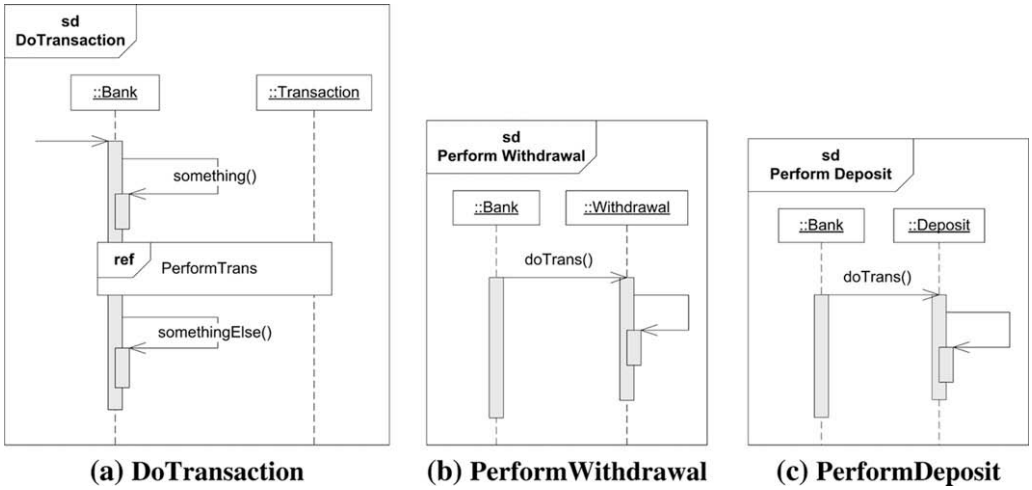

Fig. 2. Sequence diagrams for use cases with generalization relationship.

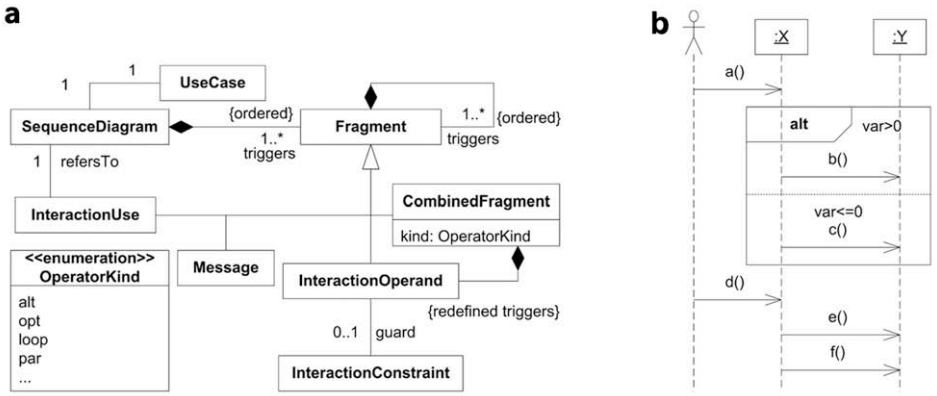

Fig. 3. Simplified UML 2.0 metamodel (a) and example sequence diagram (b).

this, we first present a simplified version of the UML 2.0 sequence diagram metamodel (Section 3.1).

As mentioned in Section 1, we consider changes on class diagrams, sequence diagrams and use case diagrams, as those are the most commonly used diagrams in practice. Future work will address other diagram types. In the text below, we assume the diagrams to be consistent.

3.1. Reminder on the UML 2.0 notation (Metamodel)

Fig. 3a is a (simplified) metamodel for the UML 2.0 sequence diagram notation [24]. As mentioned earlier, we limit the formalism employed to what we consider necessary for our notation and definitions (following subsections). We simplified the metamodel in a number of ways: (1) employing more straightforward names for concepts (e.g., `SequenceDiagram` and `Fragment` in Fig. 3a correspond to `Interaction` and `InteractionFragment` in UML 2.0); (2) simplifying associations (e.g., association between `Message` and `Fragment` does not exist in UML 2.0 but can be derived from other associations and classes), (3) limiting the information to classes and associations that are necessary for the rest of the paper. However, the information included in our (simplified) metamodel can always be retrieved from the UML 2.0 metamodel.

In a UML 2.0 sequence diagram, as specified by our metamodel, a `SequenceDiagram` is associated to a `UseCase` and is composed of a sequence of `Fragments` (ordered composition between `SequenceDiagram` and `Fragment`), which can be either `Messages`, `InteractionUses` (i.e., references to other sequence diagrams), or `CombinedFragments`.[3] For instance, in Fig. 3b, the sequence diagram triggers sequence `a(),d()` of `Messages`. A `Fragment` can trigger other `Fragments`[4] (self composition): in Fig. 3b, `a()` triggers a sequence composed of one alternative combined fragment. A `CombinedFragment` has a `kind` (e.g., `alt` stands for alternative) and is composed of a sequence of `InteractionOperands` (redefined association), which trigger `Fragments` (inherited association from `Fragment`) and can have a guard condition (association to `InteractionConstraint`).

3.2. Notation for the detection of changes

Let us first introduce some notation to formalize our definitions: U, S, and SB denote the sets of use cases, sequences of `Frag-`

[3] The generalization relations and self association on class `Fragment` are only used to simplify our definitions in Sections 3.3, 3.4 and 3.6. Note that an `InteractionOperand` can only be triggered by `CombinedFragments` and a `CombinedFragment` only triggers `InteractionOperands` (redefined composition).

[4] Self association `triggers` is derived in the context of `InteractionUse` from the association between `SequenceDiagram` and `Fragment`: the `Fragments` triggered by an `InteractionUse` (association `triggers` inherited by `InteractionUse`) are those triggered by the `SequenceDiagram` the `InteractionUse` refers to (association `refersTo`). (In the context of class `InteractionUse`, the following holds: `self.triggers =self.refersTo.triggers`.)

ment executions in all use cases (i.e., sequences in their corresponding sequence diagrams), and sequences of Fragments that include at least one message sent to boundary objects or classes. The sets of messages and messages sent to boundary objects are denoted as M and MB, respectively (MB ⊂ M). Any message in M can be represented by a 4-tuple: A message in M has a *source classifier* name (a classifier is usually an object, but can also be a class, or an actor when the message involves one), a *target classifier* name, a triggered *action* name, as well as a *triggering action* name corresponding to the action that triggers the message.[5] For instance, in Fig. 3b, message labeled b() has a source classifier name (i.e., instance of class X), a target classifier name (instance of class Y), a triggered action (b()), and a triggering action (a()). An action can be a *signal* (i.e., the class name of the signal object being asynchronously sent), a *call* to an operation, or special *create* and *destroy* actions [2]. (Note that we are not interested in reply messages.) Furthermore, a message can be specified by a return value and arguments.

Since we assume that UML diagrams are consistent, these 4-tuples uniquely identify messages in M. In other words, a 4-tuple that appears in different sequence diagrams (or in different sequences of the same sequence diagram) corresponds to only one message in M. A sequence diagram may be seen as a set of possible sequences of fragments. For a sequence $s \in S$ (or $s \in SB$) we denote the *i*th element of sequence s as s(i).

As specified in our metamodel (Fig. 3a), Fragments can be triggered in sequence ({ordered}) by another Fragment. We denote by T_{FF} the triggered relation on the set F of Fragments. $T_{FF}(f)$ therefore refers to the sequence of Fragments triggered by Fragment f and $T_{FF}(f)(i)$ refers to the *i*th element in sequence $T_{FF}(f)$.

In order to distinguish the above sets in the two versions of the system, we add the version number (i.e., 1 or 2) as a superscript[6]: set M^1 (resp. S^2) is the set of messages (resp. sequences) in the first/original (resp. second/modified) version of the system. A similar notation, adding character c as a superscript, is used to denote sets of changed elements: M^c denotes the set of changed messages from version 1 to version 2, i.e., elements that appear in both M^1 and M^2 but are changed (see Definition 1 below). Note that we do not need to refer to added or deleted messages since such modifications result in changed sequences (see Definition 5 below) as the result of our consistency assumption (Section 2.1). These sets have relations that are of interest to us, such as messages that appear in sequences. We thus define the *mathematical relations* R_{MS} and R_{MBS} to identify the messages and messages to boundary classes involved in sequences, respectively. Note that these relations' subscript indicates their domain, e.g., $R_{MS} \in M \times S$, thus making it easy to interpret a relation in a formula (this notation will be used throughout the article for other relations). This formalism is not only used to provide precise definitions but also to define constraint rules between the sets that are used as a basis for defining the underlying algorithms of our test selection tool (see Section 5).

Some notation is also required for comparison of class diagrams. A, O, C, and L denote the sets of attributes, operations (signatures), classes and relationships (between classes) in a class diagram, respectively. Note that attributes and operations are identified by their name and signature, respectively. Attributes (operations) with the same name (signature) but in different classes are distinguished using mathematical relations: R_{AC}, R_{OC}, and R_{CL} identify the attributes of classes, the operations of classes, and the relationships between classes, respectively.

Last, R_{MOC} identifies the operations (O) of classes (C) involved in messages (if the action is a call, create or destroy). Although only arguments (no type information) are shown in messages [30], a message (i.e., metaclass Message in the UML 2.0 metamodel) has a signature association to a NamedElement which "must either refer to an Operation [in which case the message is a call] or a Signal [in which case the message is a signal]" [24]. It is therefore possible to uniquely identify the operation that is used in a message, except when the operation is overwritten. In this case, for a given message m with target class c, there may exist more than one operation o in class c such that $(m, o, c) \in R_{MOC}$. In other words, action names in 4-tuples may correspond to lists of operations.

3.3. Changes between two versions of the same sequence diagram

Assuming that use case names are unique within one use case diagram, two versions of a given use case diagram are compared to detect the sets of added and deleted use cases. Using a similar notation as for changed elements, we denote these sets as U^a and U^d, respectively. Since each use case is described by one sequence diagram, two subsequent versions of sequence diagrams are compared to detect the set of changed messages and therefore the set of changed use cases.

3.3.1. Definition 1 – Changed message (M^c)

A changed message is a message that exists in both M^1 and M^2 (with identical 4-tuple) but with a different return value, arguments, or a changed action – i.e., either a changed operation (see Definitions 7–11) or a changed signal class (Definition 14) – as detected from class diagrams (Section 3.4).

Definition 1.

$$(\forall m_1 \in M^1)(\forall m_2 \in M^2)$$

m_1 and m_2 have the same 4-tuple but they have a different return value, arguments, or the message action (operation or signal) is changed $\Rightarrow m_1 \in M^c$

3.3.2. Definition 2 – Changed combined fragment (CF^c)

CF denotes the set of combined fragments. Combined fragments are uniquely identified by a 2-tuple: their triggering Fragment and their position in the sequence of Fragments triggered by their triggering Fragment. A changed combined fragment is a combined fragment that exists in both CF^1 and CF^2 (with identical 2-tuple) but with a different *kind* (e.g., alt, opt).

Definition 2.

$$(\forall cf_1 \in CF^1)(\forall cf_2 \in CF^2)$$

cf_1 and cf_2 have the same 2-tuple but a different kind $\Rightarrow cf_1 \in CF^c$

3.3.3. Definition 3 – Changed interaction operand (IOc)

IO denotes the set of interaction operands. Interaction operands are uniquely identified by a 2-tuple: their triggering combined fragment and their position in their triggering combined fragment's sequence of triggered interaction operands. A changed interaction operand is an interaction operand that exists in both IO^1 and IO^2 (with identical 2-tuple) but with a different *guard*. Modifications in OCL expressions (i.e., the guards) can easily be detected, as further discussed in Section 3.5, by parsing and comparing them.

[5] Messages in sequences triggered by sequence diagrams (association between SequenceDiagram and Fragment) do not have a triggering action. This also includes message sent by an actor. Similarly, messages sent to an actor do not have a triggering action. For instance, message labeled d() in Fig. 3 has a source classifier name (i.e., an actor), a target classifier name (i.e., class X), an action (i.e., d()), but no triggering action.

[6] This mechanism, adding the version as a superscript to identify a set, is also used for other sets or relations in the rest of the document. Note that the superscript is omitted when the version number is not relevant.

Definition 3.

$(\forall io_1 \in IO^1)(\forall io_2 \in IO^2)$

io_1 and io_2 have the same 2-tuple but a different guard $\Rightarrow io_1 \in IO^c$

3.3.4. Definition 4 – Changed interaction use (IU^c)

IU denotes the set of interaction uses. Interaction uses are uniquely identified by a 2-tuple: their triggering `Fragment` and their position in their triggering `Fragment`'s sequence of triggered `Fragments`. A changed interaction use is an interaction use that exists in both IU^1 and IU^2 (with identical 2-tuple) but with a different referred use case.

Definition 4.

$(\forall iu_1 \in IU^1)(\forall iu_2 \in IU^2)$

iu_1 and iu_2 have the same 2-tuple but a different referred use case $\Rightarrow iu_1 \in IU^c$

3.3.5. Definition 5 – Changed fragment (F^c, $F^c = M^c \cup CF^c \cup IO^c \cup IU^c$)

Recall that `Fragments` (i.e., `Message`, `CombinedFragment`, `InteractionOperand`, and `InteractionUse`) trigger sequences of `Fragments`. When any of the `Fragments` triggered by a `Fragment` is changed (Definitions 1–4) or the sequence triggered is changed (e.g., addition or deletion of triggered `Fragments`) we consider the triggering `Fragment` (either a `Message`, a `CombinedFragment`, an `InteractionOperand`, or an `InteractionUse`) to be changed. For example, in the case of a message with an operation as action, the implementation of that operation may change if the `Fragments` it triggers or their sequence change. When the action of the message is a signal, the signal object may not change as a result of changes in triggered fragments, but the processing of the signal in the target object will (e.g., the `run()` method in Java). In that case too, to be consistent, the message including the signal is considered changed. Note that such a change may not result in any change in the class diagram (e.g., the operations' contracts may not change).

Definition 5. $(\forall f_1 \in F^1)$ $(\forall f_2 \in F^2)$, f_1 and f_2 have the same tuple (see Definitions 1–4)

$T_{FF}(f_1) \neq T_{FF}(f_2) \vee \exists k \in N, T_{FF}(f_1)(k) \in F^c \Rightarrow f_1 \in F^c$

3.3.6. Definition 6 – Deleted sequence of boundary messages (SB^d)

A deleted sequence of messages to boundary objects is a sequence that appears in the first version of the sequence diagrams, but not in the second version.

Definition 6.

$(\forall s \in SB) s \in SB^1 \wedge s \notin SB^2 \Rightarrow s \in SB^d$

3.4. Changes between two versions of the same class diagram

Two subsequent versions of a given class diagram, corresponding to different versions of an evolving system, are compared to detect the sets of changed attributes, operations, relationships, and classes. Note that, we do not identify additions and deletions of class diagram elements since, because of our consistency assumption, these modifications result in changes to operations (or changes to messages as seen in Section 3.3). For instance, additions/deletions of attributes, operations, or relationships require, in order to maintain consistency, that OCL expressions involving them in the first version of the system be changed, thus resulting in changed operations (see definitions below). Note that there are different ways a changed operation can be detected and this is why Definitions 7–11 all concern changed operations.

3.4.1. Definition 7 – Changed operation (R^c_{OC})

A changed operation has the same signature in the two class diagram versions (operation name, parameter names, types and direction), but with a different property or stereotype, or with different parameter default values.[7] We do not consider changes of visibility (public, private, protected) or scope (class, instance) because, since we assume diagrams are consistent, those changes would have either no impact on the classification of test cases or result in changes in sequence diagrams that are already handled as described in Section 3.1.

Definition 7. Some of the elements of R^c_{OC} are identified using the following rule:

$(\forall c \in C)(\forall o_1 \in R^1_{OC})(\forall o_2 \in R^2_{OC})$

o_1 and o_2 have the same signature but have a different property, stereotype or parameter default values[7] $\Rightarrow o_1 \in R^c_{OC}$

3.4.2. Definition 8 – Changed operation (R^c_{OC}) because of changed operation contracts

Assuming the design by contract approach is used to specify the operations preconditions and postconditions in the Object Constraint Language (OCL), changes to operation contracts (new precondition or postcondition) indicate a changed operation (the analysis of OCL expressions, required here, is further discussed in Section 3.5). For example, an operation may treat a wider set of situations (inputs), thus changing the precondition, though the postcondition may remain identical. We provide below a definition for postconditions though a very similar definition exists for preconditions.

Definition 8. Given P the set of postconditions and R_{POC} the relation that defines postconditions of operations in classes, some of the elements of R^c_{OC} are identified using the following rule:

$(\forall (o, c) \in R_{OC})(\exists p_1 \in P^1, p_2 \in P^2)$
$(p_1, o, c) \in R^1_{POC} \wedge (p_2, o, c) \in R^2_{POC} \wedge p_1 \neq p_2 \Rightarrow (o, c) \in R^c_{OC}$

Special case: If the operation is a constructor, we handle it in a different manner. Changed initialization of attributes, as shown in the constructor's postcondition, amounts to changed attributes (i.e., changed initial values).

3.4.3. Definition 9 – Changed operation (R^c_{OC}) because of changed class invariant

Assuming the design by contract approach is used to specify class invariants in the OCL, changes to the class invariant indicate changed operations. (This can be detected in a similar way to changes in preconditions and postconditions.) A new invariant may require that the operation be changed to preserve that new invariant after execution.[8]

Definition 9. Given I the set of class invariants and R_{IC} the relation that defines invariants in classes, some of the elements of R^c_{OC} are identified using the following rule:

$(\forall (o, c) \in R_{OC})(\exists i_1 \in I^1, i_2 \in I^2)(i_1, c) \in R^1_{IC} \wedge (i_2, c) \in R^2_{IC} \wedge i_1 \neq i_2$
$\Rightarrow (o, c) \in R^c_{OC}$

[7] Changes to other characteristics of an operation are also monitored: e.g., values to attributes `isQuery`, association to raised exceptions [24].

[8] This is a rough assumption since a changed invariant may not affect some operations. Future work will look into refining this definition and precisely identifying which operations are impacted or not by changes to invariants.

3.4.4. Definition 10 – Changed operation (R^c_{OC}) due to a changed attribute

If an operation accesses a changed attribute (Definition 12), as shown in the operation's contracts, the operation is considered changed. A similar definition exists if the operation's contract makes use of changed query[9] operations. Though no change may be visible in the operation's contracts or in the fragments the operation triggers (see the detection of changes in sequence diagrams in Section 3.3), its implementation is likely to require some changes. (A similar definition exists for changed attributes accessed in class invariants but is omitted here.)

Definition 10. Given R_{ACO} the relation that identifies attributes accessed by operations in classes (as shown in the operation's contracts), some elements of R^c_{OC} are identified using the following rule (note that the operation and the accessed attribute may not be in the same class):

$$(\forall(o,c_1)\in R_{OC})(\exists(a,c_2)\in R_{AC})(a,c_2)\in R^c_{AC}\wedge(a,c_1,o)\in R_{ACO} \Rightarrow (o,c_1)\in R^c_{OC}$$

3.4.5. Definition 11 – Changed operation (R^c_{OC}) due to a changed relationship

The operation contracts show which relationships are navigated in the operation (through the OCL navigation expressions [32]). If any of the navigated relationships in the operation's contract has changed (Definition 13(a)) in the new class diagram, then the operation is considered changed as its execution may be affected. (A similar definition exists for changed relationships traversed in class invariants but is omitted here.)

Definition 11. Let R_{OCL} capture relationships that are navigated in the contracts of operations in classes. Some elements of R^c_{OC} are identified as follows:

$$(\forall(o,c)\in R_{OC})(\exists l\in L)(o,c,l)\in R_{OCL}\wedge(c,l)\in R^c_{CL}\Rightarrow(o,c)\in R^c_{OC}$$

3.4.6. Definition 12 – Changed attribute (R^c_{AC})

A changed attribute (as identified by its name) exists in both versions of a given class but with a different scope (class or instance), type (i.e., datatype[10]), visibility (public, protected, private), multiplicity, initial value (declaration initialization or constructor changes), stereotype or property (e.g., property {frozen} indicates that the attribute's value may not change after an instance of the class holding it is created). Note that we only consider changes in datatypes since relationships between user-defined types (other than datatypes) should be modeled with associations rather than attributes (see the detection of changed associations below).

Definition 12. $R^c_{AC}=\{(a,c)\in R^1_{AC}\cap R^2_{AC}/$ a in class c has a different scope, type, visibility, multiplicity, initial value, stereotype or property in versions 1 and 2$\}$

3.4.7. Definition 13 – Changed relationship (L^c)

The process of identifying changed relationships depends on the kind of the relationships. A changed association is an association that exists in both class diagram versions but with a different name, kind (plain association, aggregation, composition), multiplicities, navigability, role visibility, or qualifier. Note that we do not consider changes in role names or stereotype for changed associations as associated classes along with their roles names and the association stereotype are used to uniquely identify associations. If we are dealing with an association carrying an association class, then any change in the association class results in a change in the association. Generalization and realization relationships can only be considered changed when they are described by a stereotype that changes. Given that dependencies are uniquely identified and fully specified by their source and target classes and their stereotype, it is impossible to observe changed dependencies since any change to a dependency results in a deletion and an addition.

Definition 13(a). Given La, Lg and Lr the sets of associations, generalizations and realizations, respectively ($La\cup Lg\cup Lr\subseteq L$), we have:($\forall l\in La$) (l has a different name, kind, multiplicities, navigability, role visibility, or qualifier in versions 1 and 2) $\Rightarrow l\in L^c$($\forall l\in Lg\cup Lr$) (l has a different stereotype in versions 1 and 2) $\Rightarrow l\in L^c$

Definition 13(b). Given C_{ac} the set of association classes ($C_{ac}\subseteq C$), C^c the set of changed classes (see Definition 14), and R_{CacL} the associations carrying association classes, we have:

$$(\forall l\in L)(\exists c\in C^c)(c,l)\in R_{CacL}\Rightarrow l\in L^c$$

3.4.8. Definition 14 – Changed class (C^c)

A changed class exists in both versions of the class diagram but with a changed attribute, operation, invariant (OCL), relationship, stereotype, property, multiplicity or template. This definition encompasses signal classes (used for asynchronous messages in sequence diagrams [11]).

Definition 14. Given the previous definitions on attributes, operations and relationships, the set of changed classes (C^c) is formally defined as follows:

($\forall c\in C$) (c has a different stereotype, property, multiplicity or template in versions 1 and 2) $\Rightarrow c\in C^c$

$$(\forall c\in C)(\exists a\in A,o\in O,l\in L,i\in I)(a,c)\in R^c_{AC}\vee(o,c)\in R^c_{OC}\vee((o,l)\in R_{CL}\wedge l\in L^c)\vee(i,c)\in R^c_{IC}\Rightarrow c\in C^c$$

3.5. Analysis of OCL expressions

OCL expressions have been used in the previous sections to detect changes in operations' contracts or in guards. In both cases, four kinds of changes are studied: the expression itself changes, a changed attribute or query operation is used in the expression, or a changed association is navigated in the expression; and we investigate here the challenges in detecting such changes.

When an attribute `y` of a class is used in one of class `X` operations' contracts, it appears under three possible forms: `y`, `self.y`, `expression.y` where `expression` is an OCL expression possibly involving navigations and operations on OCL collections. Note that, in the first two forms, `y` is an attribute of `X`, whereas it is not necessarily the case in the latter. Furthermore, the above discussion also applies to associations in which `X` is involved and that are used (i.e., navigated) in `X` operations' contracts. In this case, `y` denotes either the role name to the target class involved in the association or its name. Identifying the two first forms is straightforward, given a string representation of the OCL constraint, whereas identifying the third form is more complicated since it may require some type analysis (i.e., determining the type of `expression`) when several attributes bear the same name across different classes.

A straightforward string comparison of the two versions of an OCL expression can be used to detect a change when the expression itself changes. However, this may produce false positives

[9] A query operation returns a value but does not change the state of any object [32].
[10] A datatype is a type whose values have no identity [2], and includes primitive built-in types (e.g., Integer) as well as classes with stereotypes ≪`enumeration`≫ or ≪`datatype`≫.

when the change is only a rewriting of the expression without any semantic change. Indeed it is often possible to express a constraint in different ways in OCL. False positives would result into detecting changes that should have no impact on the classification of test cases. A more complicated change detection technique, that would not produce such false positives, would require some semantic analysis to detect equivalent OCL expressions. An important question is then whether it would be worth it from a practical standpoint.

The current implementation of our tool (Section 5) does not perform type or semantic analysis of OCL expressions. However, none of our case studies required the above functionalities to be implemented and were therefore not affected by the current tool limitations.

3.6. Impact of changes on test cases

Using the traceability between test cases and sequence diagrams, the change analysis results obtained from comparing the two versions of the class and use case diagrams/sequence diagrams are used to automatically classify the regression test cases.

For a given set of test cases T, there exist relations between elements in T and elements in S (R_{TS}) and SB (R_{TSB}), as test cases execute sequences in sequence diagrams. The implementation and handling of these relations in our tool is presented in Section 5.2.3. In our classification of test cases, to simplify the notation, $R_{TSB}(t)$ represents the sequence of boundary messages triggered by test case *t*. A test case can be classified in one of three categories: Obsolete, Retestable, Reusable. We provide a mapping between these categories and the change impacts described in Sections 3.3 and 3.4.

3.6.1. Definition 15 – Obsolete test case (T_o)

A test case is obsolete if it consists of an invalid execution sequence of messages on boundary objects. This results from a change in the possible sequences according to which messages involved in the test case can be sent to boundary objects (including the addition or deletion of a message to a boundary object in the sequence diagram scenario that was mapped to the test case).

Definition 15. The set of obsolete test cases, T_o, can be defined as[11]

$$(\forall t \in T) R^1_{TSB}(t) \in SB^d \Rightarrow t \in T_o.$$

3.6.2. Definition 16 – Retestable test case (T_{rt})

A retestable test case is a test case which remains valid in the new version of the design in terms of sequence of messages to boundary objects. But one or more of these messages may have changed (e.g., operation postcondition), or the set of valid sequences of messages triggered by boundary class messages may have changed (this results in changed messages according to Definition 5).

Definition 16. The set of retestable test cases, T_{rt}, can be defined as

$$(\forall t \in T)(\exists (t,s) \in R_{TS})(\exists (m,s) \in R_{MS}) R^1_{TSB}(t) \notin SB^d \wedge m \in M^c \Rightarrow t \in T_{rt}$$

3.6.3. Definition 17 – Reusable test case (T_{re})

A reusable test case consists of a sequence of messages to boundary objects that has remained valid in the new version of the design. The sequences of messages triggered by the boundary messages have not changed either. In other words, none of the messages involved in the test case (sent to boundary or other classes) have been changed.

Definition 17. The set of reusable test cases, T_{re}, can be defined by

$$(\forall t \in T)(\forall (t,s) \in R_{TS})(\forall (m,s) \in R_{MS}) R^1_{TSB}(t) \notin SB^d \wedge m \notin M^c \Rightarrow t \in T_{re}$$

We can see from the above definitions that the three test case categories are mutually exclusive so that any given test case is either obsolete, retestable or reusable.

4. Analyzing our regression test selection strategy

Rothermel and Harrold [27] proposed an evaluation framework (referred to as the RH framework) for regression test selection techniques. The framework was originally designed for code-based techniques but most of the principles can be applied here. Four evaluation criteria are going to be discussed in this section: safety, precision, efficiency, generality. In particular, we address the impact of using UML designs instead of code on those criteria.

4.1. Safety

The RH framework shows that, when two assumptions hold, demonstrating that fault-revealing tests are selected is equivalent to demonstrating that all modification-revealing tests (i.e., that can cause the output of the two program versions to differ) are selected. The first assumption, that test cases halt and produce correct results on the original program version, was easy to verify in our case studies. The second assumption, that obsolete test cases are identified and removed, was also checked for all our case studies but was more complex. Though many of the obsolete test cases are automatically identified (Definition 15), the detection of some of them may require additional checking that is not yet implemented in our tool. For example, we would need to evaluate whether test case data (e.g., input values to operation calls) satisfy changed operations' preconditions.

Furthermore, under the controlled regression assumption (i.e., all the factors that influence the system outputs are held constant, except for the program modifications [27]), the non-obsolete modification-traversing tests are a superset of the modification-revealing tests, and therefore of the fault-revealing tests. In our context, modification-traversing tests take a different definition as we are dealing with UML designs, not code. A non-obsolete test case is modification-traversing for a given (original, modified) program pair if and only if it triggers a changed message, based on Definition 1 in Section 3.1.

In our context, showing that all modification-traversing test cases are selected entails that we show that (1) all design modifications to the UML diagrams are detected and properly classified and that (2) based on the detected modifications to the design, we correctly classify test cases as retestable, when the test case is modification-traversing. To address issue (1), we systematically look at all the diagram elements and consider how changing, deleting, and adding them would precisely translate into diagram changes (Section 3). For example, we carefully defined what a changed operation would be in a class diagram based on available information in the original and modified class diagrams. Those sets of changed elements were precisely and formally defined,[12] In terms of empirical evidence, we have not encountered in any of

[11] In this classification of test cases, $R_{TSB}(t)$ represents the sequence of boundary messages triggered by test case *t*.

[12] Recall that, putting aside the identification of deleted sequences of boundary messages to classify obsolete test cases, we do not identify additions and deletions of UML diagrams elements, as such changes results in changes we account for since we assume diagrams are consistent.

our case studies any changes to the diagrams (that were a priori known, as we defined and implemented system changes that translated into diagram changes) that were not detected and properly classified by the tool. With respect to issue (2), recall that a non-obsolete test case is classified as retestable if any of the actions in the test sequence, or any other element of its corresponding message in the sequence diagram (e.g., guard condition), are changed. How this is detected from the diagrams is described in Sections 3.3, 3.4 and 3.6 where we aim at being complete in accounting for all possible diagram changes and in determining which changes lead to changed messages and message sequences. Based on the traceability between sequences in sequence diagrams and test cases, we then determine if a test case is modification-traversing and therefore retestable.

There is one issue though, which was not relevant in the context of code-based techniques. Certain code modifications will not be visible in the UML design diagrams: A change to an operation implementation will not be visible if its contract (pre and postcondition) is not affected, its signature is not changed, or if no attribute (or query operation) used by the contract of the operation is changed. In that case, certain code modification-traversing (and therefore fault-revealing) tests may not be selected and, as a result, any technique based on UML designs cannot be safe in the sense of the HR framework. One simple solution would be to ask the people changing the UML diagrams to indicate if they expect such changes in operation implementations and make a note of it, in a predefined way (e.g., using UML note boxes or stereotypes on the class diagram). Though this issue is an inherent drawback of using design representations as a basis for impact analysis and regression testing, we have not encountered such cases in our case studies. It is clear, however, that such a situation is possible and should be dealt with in one of the ways suggested above and by extending our definitions to account for changes in operation implementations. Another alternative, that remains to be investigated in practice, is to consider the test selection based on design changes a first step and, once the changed code is available, to determine what additional test cases will need to be considered retestable based on code changes not visible in the design. If the number of such test cases happens to be too large, that would of course defeat the purpose of early test selection.

Our algorithm cannot be considered safe if there is any other mechanism, not accounted for in our definitions, that would lead to changed messages in sequence diagrams. Though we cannot prove (in a formal sense) that the selection is safe, none of the case studies have exhibited cases where a modification-traversing test case was not classified as retestable by our tool.

4.2. Precision

According to the RH framework, in order to show that our approach is precise, we have to show that we only select non-obsolete tests that can produce different program outputs in the modified system version. Program outputs in our context correspond to output data in messages sent by boundary classes to actors. As discussed in [27], there is no algorithm to determine, for arbitrary programs, changes, and test cases, the precision of a selection method. We therefore need to rely on empirical evidence. To analyze precision, for each test case classified as retestable in our case studies, we perform the following analysis:

- We determine whether the test case triggers a changed message.
- If this is the case, we determine whether any fault in the changed message (or any other message it triggers) could lead to a different program output. A constraint in finding such a fault is that the set of changes to the design should not be affected by the fault, i.e., we assume the code changes and design changes are consistent.
- If we can find at least one such fault, then we conclude that the test case was appropriately selected for safe regression testing (retestable).

If all the test cases selected in our case studies have been appropriately retained as retestable, then we can claim the method, based on the available empirical evidence, is precise.

Following the above procedure, in our case studies, we performed a manual analysis of all the test cases classified as retestable. We only found, across our three case studies, one occurrence of a test case that did not really need to be retested. The imprecision came from the fact that, for a given operation with alternative postconditions (disjuncts), we do not carefully analyze which postcondition applies in the context of a given test case. In that particular example, a query operation used in the postcondition of another operation was changed and the test case happened to execute that operation and was therefore considered retestable. However, in the scenario of the test case, that query operation was not actually executed. That case is described in detail in [5].

The results of this analysis suggest that cases of imprecision in classifying test cases as retestable are rare. More case studies are of course required to confirm our observation and a future research question is to determine whether a refinement in the analysis of postconditions would be worth the effort and additional time complexity.

4.3. Efficiency

According to the RH framework, several factors affect efficiency (i.e., space and time requirements) and should be investigated. The first one is the phase of the lifecycle where the regression test technique performs its activities. In our case, most of the analysis can be performed during the so-called preliminary phase, that is during the phase where corrections to the systems are being designed and implemented. Actually, we expect the analysis to be performed before any change code is even implemented. This should be, in practice, an important advantage over code-based techniques as the regression test effort can be planned for earlier on, when there is still flexibility concerning the release deadline and content.

The second factor is automatability. We show in Section 5 how the whole process can be automated. We developed a tool (RTSTool) that demonstrates the feasibility to automate in a comprehensive way all the principles introduced in the current paper. Though not industry strength, the tool has been applied to three case studies without any problem.

The third factor is the extent to which the technique must compute information on program modifications. We show, in Section 3, how the relevant sets involved in classifying test cases are defined. The time complexity of the corresponding algorithms is a function of the number of model elements in the original and modified UML models. In particular, a careful analysis shows it strongly depends on the number of operations and messages in the class and sequence diagrams, respectively.[13] Though these numbers may become large in real systems, they tend to be much smaller than the number of control flow paths or definition/use pairs in code control and data flow analysis. This is the main reason why a design-based impact analysis is likely to be more efficient than a code-based one [12].

The last factor is the ability of the technique to handle cases where the new version results from multiple modifications, without any redundant computations when working separately on

[13] The specific results of the worst-case complexity analysis are not reported here as they strongly vary, depending on detailed algorithmic and implementation choices. Suffice to say that the relationship between computational time and, say, number of operations or messages, is not exponential and can therefore scale up to large models.

individual modifications (as it is the case for some white-box techniques [27]). We have seen in Section 3 how modifications are being detected. In our method and tool, there is no difference between cases where one or multiple modifications are performed to create the new system version. All the information used for the test classifications is computed once by comparing the original and new design versions.

4.4. Generality

This criterion relates to the ability of the selection technique to apply to a wide and practical range of situations. Our technique obviously applies to software that is designed using the UML. This is a practical restriction but as the use of UML spreads [25], that should become less and less constraining.

Our selection technique handles all possible modifications to the UML diagrams we consider. As mentioned above, we consider the diagrams that are the most likely to be used in practice: use case diagram, class diagram, and interaction (sequence) diagram. Assuming state machine diagrams would be used, any change to a statechart should result in a change in the sequence diagrams and/or the OCL contracts. That would therefore not increase our capability to detect faults and would impose additional constraints on the use of UML.

The only assumption we make regarding the testing or maintenance environment is that a UML case tool be used to ensure diagrams are consistent and that it has the capability of exporting XMI files that can be imported into RTSTool.

5. Regression Test Selection Tool (RTSTool)

The first subsection describes the functionality of the Regression Test Selection Tool (RTSTool) we built based on the principles described in the previous sections, and highlights some of the most interesting technical details, whereas the second subsection focuses on how traceability between design and test cases is implemented. More technical details on the RTSTool architecture can be found in [5].

5.1. Functionality

The RTSTool main functionality is to classify regression test cases as obsolete, retestable, and reusable, based on the design information of the old and new system versions and traceability information between the UML design and test cases. Its inputs are the UML diagrams of two system versions (XMI files produced by UML case tools) along with the original regression test suite. It then compares the two versions of each diagram type (class, use case, and sequence) and classifies test cases. Future functionalities that can be easily added to the current architecture include the generation of new regression test cases based on the new versions of UML diagrams (e.g., [3]). Furthermore, the results of the impact analysis (i.e., added, deleted, and changed model elements) can easily be used for other purposes than regression test selection, e.g., to assess the effort of producing the new version or to make a decision on whether to include a change in the next version [4].

RTSTool is independent of any specific UML case tool since it uses XMI as a data interchange format. It can be extended to account for refinements to the test case classification strategy (e.g., considering additional UML diagrams), changes in UML or XMI standards, or changes to the test case representation, since those aspects are encapsulated in their respective packages and unknown to other packages. Additionally, RTSTool uses an object-oriented database management system to store the different versions of UML models and test cases, thus allowing the reuse of previously loaded diagrams or test cases. The interested reader is referred to [5] for more details on RTSTool.

The RTSTool tool is implemented with Java (Java 2 Platform, Standard Edition version 1.4), and consists of 131 classes and 11704 lines of code (without comments). Our implementation of the UML metamodel counts for 97 classes and 5705 lines of code.

5.2. Test cases and traceability

We describe here how the traceability between test cases and sequence diagrams is represented and implemented.

5.2.1. Representation of test cases

Any test case is associated with a sequence of triplets (action name, source classifier name, target classifier name). It specifies the sequence of actions resulting from a test case. In the test driver, a functional test case will consist of operation invocations, signals being sent (e.g., placed in a queue), and object creations as well as destructions, when the language permits. All the messages to boundary classes [6] will directly or indirectly trigger subsequent actions so as to complete a use case scenario. We associate the complete action sequences to test cases (as opposed to just action sequences on boundary objects) as determining changes in non-boundary actions will be necessary to classify test cases as reusable or retestable (Section 3.6).

5.2.2. Representation of sequence diagrams

In the same way as the test sequences, messages in sequence diagrams are triplets (message label, source classifier name, target classifier name). However, the information about messages is more complete as, in addition to action names, we have possible arguments in message labels. Furthermore, in order to represent every possible message sequence in sequence diagrams, each sequence diagram is represented using a regular expression whose alphabet is composed of the above triplets [3]. This facilitates automation in our algorithms since we can then easily check whether a test case is a legal sequence of a regular expression (i.e., a sequence diagram), and therefore whether a test case can be executed given the design described by a sequence diagram.

5.2.3. Traceability

To automate test selection, we need to have traceability between the UML design and regression test cases (i.e., implement R_{TS} and R_{TSB} defined in Section 3.6), so that we can determine the effect of design changes on those test cases. Traceability is simply an association between test cases and sequence diagrams, each test case testing a use case scenario. We therefore implement traceability as a mapping between sequence diagram scenarios (i.e., a complete message sequence through the sequence diagrams) and test cases. A test case exercises, for each use case it executes, only one scenario but one scenario can be exercised by several test cases.

6. Case studies

In this section we apply our methodology, using the RTSTool, on three different case studies. The first one is a real system developed by a Telecom company, which underwent a major design change. The second and third studies are systems developed by teams of students that were subsequently modified. The advantage of using student systems was that we could define a variety of changes so as to make the studies more diverse and interesting. In a complementary fashion, the industrial case study provided a more realistic context in which to apply our tool. In all three cases, we first describe the system and then discuss the changes that were per-

formed and present the results of the regression test selection using RTSTool.

Though of modest sizes, more specifically of the size of a typical subsystem, our three case studies provide insight into how our model-based approach performs. They are also, to the best of our knowledge (Section 7), larger in size than all the case studies that have been reported (with sufficient details) in the literature on model-based regression testing. For instance, the work in [31] reports a case study with 19 classes and 31 class relationships, three versions with 32 and 10 model changes from version 1–2 and version 2–3, respectively. In [7] the authors used three components of IBM Websphere that required 306 test cases (no technical detail is reported). An extended finite state machine with five states and eleven transitions is used in [15]. In other related work such as [10,17,33], no evaluation is reported. Only [22] in a recent study used a larger model: a labeled transition system with 80 states and 244 transitions. More experiments are definitely required to evaluate these different alternative model-based strategies.

6.1. An IP Router

This system, developed by a Telecom company, is part of an IP router. It provides a generic mechanism for software agents to discover and subsequently communicate with one another in the network. Its main function is to register new agents as they join the network and add them to the discovered set of agents. It also removes the terminated agents. The system is involved in constructing and distributing a global table during the startup process of the router. The table maps each process ID in the network to its socket address (IP address, UDP port). The system maintains and distributes the table to other agents in the network: each agent receives a copy of the global table during the agent initialization process. Agents in the network use this table to send messages to other agents.

The first version of the system is specified by 11 use cases, corresponding to 11 sequence diagrams, and contains nine classes defined by a total amount of 18 attributes and 70 operations. The original test suite contains 596 test cases, that is 546 functional test cases built according to the strategy defined in [3], and 50 non-functional test cases mostly looking at performance and scalability issues.

The second version of the class diagram contains the same nine classes as the first version, but 16 attributes and 75 operations. Detailed statistics on the changes between versions are shown in Table 1. Many of the changes in the class diagram consisted in moving responsibilities (i.e., operations) from one class to another. Some operation parameters were also merged to reduce the size of the data exchanged over the network (e.g., the table), thus reducing the messaging overhead associated with the downloading of this data. Two sequence diagrams show a deleted boundary operation. In the nine remaining sequence diagrams, the boundary class operations and their invocation sequences remain unchanged. However, we find they all contain added or deleted non-boundary class operations by analyzing the class diagrams. From these change descriptions, we can see that all the 11 use cases were changed in one way or another.

Table 1
Impact analysis and regression test selection results (IP Router).

	Total (V.1)	Added	Changed	Deleted	Total (V.2)
Detected changes					
Attributes	18	1	3	3	16
Operations	70	21	0	16	75
Classes	9	0	6	0	9
Use cases	11	0	11	0	11
Regression test selection					
Test cases					
	Total	Obsolete	Retestable	Reusable	
Functional	546	8	538	0	
Non-functional	50	0	18	32	
Total	596	8	556	32	

Eight of the 546 functional test cases execute scenarios that map to the two sequence diagrams that show a deleted boundary class operation. Their execution is no longer possible and they are classified as obsolete. Since all the other sequences are modified, all the remaining 538 functional test cases are classified as retestable (Table 1) and no functional test case is reusable. None of the non-functional test cases implies a change in the sequence of boundary operations. Only 18 contain added or deleted operations in their sequence. Then 32 non-functional test cases are reusable and 18 are retestable.

From the above results we can see that no significant selection gain could be obtained for functional test cases. The changes performed between the two versions affected all test cases and required them to be tested to achieve safe regression testing. Though their number is much smaller, the reduction in non-functional test cases was, however, substantial as 32/50 test cases were reusable. Those results do not allow us to draw any formal conclusion regarding the precision of test selection but they were reviewed by developers and were considered to make sense considering the extent of changes between the two versions. The fact that such a test selection could be automated based on UML design information was also considered a significant advantage.

6.2. An automated teller machine system

The second case study is an Automated Teller Machine (ATM) system, developed by a team of students. The ATM design model contains 20 classes, 74 operations, 31 attributes and 15 use cases. The ATM's main function is to perform transactions based on the user's inputs. Four types of transactions can be carried out: Deposit, Withdraw, Transfer, and Inquiry. What is specific to the use case diagram for the ATM is that all of the use cases, except for two of them, depend on one main use case, `doTransaction`, that describes the details of how the system performs transactions (Fig. 4). All the other use cases are either inclusions or extensions of `doTransaction`. The two other use cases describe the start up and shut down procedures of the ATM. The test set for the system contains 30 functional test cases, developed using the methodology described in [3]. Most test cases test a different transaction, combination of transactions or error conditions which may arise when performing a transaction. However almost all of these test cases execute that main high-level use case, `doTransaction`, therefore we can already foresee that if there is a change to `doTransaction` all these test cases will be classified as retestable. Two test cases exercise the start up and shut down procedures of the ATM.

Four different logical changes were performed from this original design, and we present them, as well as the result in terms of regression test selection, in the following subsections.

6.2.1. First logical change (version 2 of the ATM)

The first logical change has to do with how many times a user could enter an incorrect PIN number. In the original system there was no limit to that number. In the new version a user has only three attempts to enter a valid PIN before their card is retained by the system. This logical change translates into one new attribute (to record the number of attempts to enter a PIN), four added operations (e.g., to manipulate that attribute), and three changed operations, resulting in two changed classes. Those new operations are used in three sequence diagrams (Table 2).

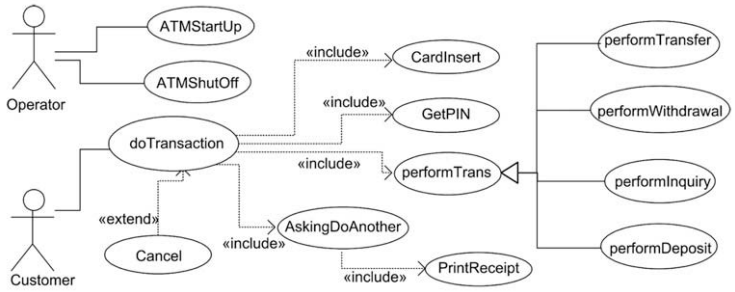

Fig. 4. Use case diagram for the ATM case study.

Table 2
Impact analysis and regression test selection results (ATM, V.2).

	Total (V.1)	Added	Changed	Deleted	Total (V.2)
Detected changes					
Attributes	31	1	0	0	32
Operations	74	4	3	0	78
Classes	20	0	2	0	20
Use cases	15	0	3	0	15
Regression test selection					
Test cases					
Total	Obsolete		Retestable		Reusable
30	0		28		2

As shown in Table 2, of the 30 test cases in this study, 28 were classified as retestable and only 2 were reusable. The reason for this lies in the type of change that was made. The 28 test cases that are retestable all explore situations where the user wants to perform a transaction, and to do so the user must first enter a card into the machine. Although the change description has to do with how many times the PIN is entered the `GetPIN` use case is not the only one affected: `CardInsert` and `doTransaction` are also affected by these changes. The two test cases that are reusable are the ones that test the start up and shut down procedures and do not involve the user putting a card in the machine and entering a PIN number.

6.2.2. Second logical change (version 3 of the ATM)

The second logical change imposes some extra restrictions on the savings type of account. In the original system savings and chequing accounts were identical in terms of which transactions could be performed on them. In the new version a user cannot withdraw money directly from an account of type savings. Therefore in order to remove their money from a savings account they must first transfer the money to a chequing account. This translates into the changes reported in Table 3.

Table 3
Impact analysis and regression test selection results (ATM, V.3).

	Total (V.1)	Added	Changed	Deleted	Total (V.3)
Detected changes					
Attributes	31	1	0	0	32
Operations	74	0	2	0	74
Classes	20	0	3	0	20
Use cases	15	0	1	0	15
Regression test selection					
Test cases					
Total	Obsolete		Retestable		Reusable
30	0		25		5

When first reading the change description one would come to the conclusion that only test cases involving a withdrawal transaction would need to be retestable. However since this change resulted in a new error condition which is checked after each transaction, all 25 test cases which perform a transaction are considered retestable. The five test cases which are reusable explore the following situations: startup, shutdown, card not readable, the user presses the cancel button when the PIN is requested and the user presses the cancel button when the transaction type is requested. In these five test cases no transaction is performed which is why they are classified as reusable.

6.2.3. Third logical change (version 4 of the ATM)

The third logical change has to do with the cash dispenser. The current dispenser only holds twenty dollar bills. The new dispenser will be able to handle twenty and five dollar bills. This results in the changes reported in Table 4.

There are only nine test cases which perform a successful withdrawal: they are classified as retestable. The other 19 test cases either perform startup, shutdown, a transaction other than withdrawal or a withdrawal in which an error occurs before the cash is dispensed. In contrast with the second change this version of the system behaves as one would intuitively think: A change has been made to the way a withdrawal transaction is executed and only the test cases which exercise the corresponding behavior need to be retested.

6.2.4. Fourth logical change (version 5 of the ATM)

In this version of the system, it was decided to merge two boundary classes for use cases `ATMStartUp` and `ATMShutOff`, and that the operator would be required to enter a password in order to complete the start up and shut down of the ATM. The RTSTool reports the changes of Table 5.

Since a boundary class involved only in use cases `ATMStartUp` and `ATMShutDown` has been removed, the `ATMStartUp` and the `ATMShutDown` test cases are obsolete (two test cases), and the other test cases are reusable (Table 5).

Table 4
Impact analysis and regression test selection results (ATM, V.4).

	Total (V.1)	Added	Changed	Deleted	Total (V.4)
Detected changes					
Attributes	31	0	0	0	31
Operations	74	4	1	0	78
Classes	20	0	1	0	20
Use cases	15	0	1	0	15
Regression test selection					
Test cases					
Total	Obsolete		Retestable		Reusable
30	0		9		21

Table 5
Impact analysis and regression test selection results (ATM, V.5).

	Total (V.1)	Added	Changed	Deleted	Total (V.5)
Detected changes					
Attributes	31	2	0	0	33
Operations	74	6	1	4	76
Classes	20	0	4	1	19
Use cases	15	0	2	0	15
Regression test selection					
Test cases					
Total	Obsolete		Retestable		Reusable
30	2		0		28

Table 6
Impact analysis and regression test selection results (CCS, V.2).

	Total (V.1)	Added	Changed	Deleted	Total (V.2)
Detected changes					
Attributes	62	0	0	0	62
Operations	64	1	2	0	65
Classes	37	1	1	0	38
Use cases	8	0	0	0	8
Regression test selection					
Test cases					
Total	Obsolete		Retestable		Reusable
323, 614	0		318, 508		5106

6.3. A cruise control and monitoring system

The cruise control system is made up of 37 classes, 62 attributes, 64 operations and 11 use cases. The main functionality of this system is to emulate the cruise control feature of an automobile. The user is able to turn on and off the car, press the gas and break pedals, set a desired cruising speed, accelerate the car to a new speed, turn off cruise control, resume to the desired speed and view the current speed, average trip speed and average trip fuel consumption. Three of the 11 use cases describe a behavior that is constantly being executed in the background (i.e., Update Shaft Rotation, Determine Distance and Speed and Calculate Trip Speed and Fuel Consumption) and are thus not considered in the experiment. The remaining eight use cases are `Turn On Engine`, `Turn Off Engine`, `Cruise`, `Resume`, `Accelerate`, `Cruise Off`, `Throttle Pressed` and `Brake Pressed`. The system testing strategy we used produced 323, 614 test cases [3,5]. This seemingly large number stems from the nature of the system and the method used to derive system test cases [3] and is not an issue in our context as we are looking at the *proportions* of obsolete, retestable, and reusable test cases. The testing method consists in identifying possible execution sequences of use cases from use case sequential dependencies when such constraints exist or from the interleaving of un-constrained use cases. Other than turning the cruise control on first and off to end, there are very few sequential constraints in the cruise control system, thus resulting in an extraordinary large number of possible use case sequences (due to interleavings). Then, when developing test cases for the cruise control we, additionally, arbitrary selected only sequences that exercise at most twice the use cases involved in sequences, in an attempt to control the number of test cases generated. (Note that the combinatorial explosion of use case sequences due to interleavings in an open issue in [3].)

6.3.1. First logical change (version 2 of the CCS)

In the original version of the system when the driver selects the 'accelerate' position of the cruise control lever, the car accelerates at a rate of 10 mph per second. The change that is introduced in version 2 allows the driver to set the rate of acceleration from 1 mph per second to 40 mph per second. A slider is added to the user interface and the position of the slider tells the system at which rate to accelerate. The results of this logical change are reported in Table 6.

These changes only affect the Accelerate use case. Therefore all test cases which exercise the accelerate use case are considered retestable and all the rest of the test cases are reusable.

6.3.2. Second logical change (version 3 of the CCS)

In the second change to this system the structure of the system is examined to determine the concurrent tasks. When examining the cruise control subsystem it is noted that the engine and brake are passive input devices that need to be polled every 100 milliseconds. The polling of these two devices is combined into one task called `AutoSensors` which will be triggered by a timer every 100 milliseconds. As a result the RTSTool reports the changes in Table 7: `The BrakePressed`, `TurnOnEngine` and `TurnOffEngine` use cases are deleted and replaced by a use case named `Brake` and `Engine Signals`.

Due to the deletion and addition of use cases all test cases that invoke a scenario of the use cases that were deleted are classified as obsolete. Since every test case contains the `TurnOnEngine` and `TurnOffEngine` use case, all of the test cases in the test suite are classified as obsolete.

6.3.3. Third logical change (version 4 of the CCS)

In the third change a button is added to enable the driver to cancel the desired speed that is saved in the system. This can be used when the driver wants to set a new desired speed. The changes are shown in Table 8.

These changes did not affect any existing functionality (i.e., messages sent while executing existing use cases) of the system therefore all of the test cases can be considered reusable.

Table 7
Impact analysis and regression test selection results (CCS, V.3).

	Total (V.1)	Added	Changed	Deleted	Total (V.3)
Detected changes					
Attributes	62	3	0	0	65
Operations	64	0	0	5	59
Classes	37	1	2	2	36
Use cases	8	1	0	3	6
Regression test selection					
Test cases					
Total	Obsolete		Retestable		Reusable
323, 614	323, 614		0		0

Table 8
Impact analysis and regression test selection results (CCS, V.4).

	Total (V.1)	Added	Changed	Deleted	Total (V.4)
Detected changes					
Attributes	62	0	0	0	62
Operations	64	3	0	0	67
Classes	37	0	2	0	37
Use cases	8	1	0	0	9
Regression test selection					
Test cases					
Total	Obsolete		Retestable		Reusable
323, 614	0		0		323, 614

Table 9
Impact analysis and regression test selection results (CCS, V.5).

	Total (V.1)	Added	Changed	Deleted	Total (V.5)
Detected changes					
Attributes	62	0	1	0	62
Operations	64	0	1	0	64
Classes	37	0	1	0	37
Use cases	8	0	2	0	8
Regression test selection					
Test cases					
Total	Obsolete		Retestable		Reusable
323, 614	0		318, 502		5106

6.3.4. Fourth logical change (version 5 of the CCS)

In the fourth change a lower limit was added to desired cruising speed: if the car is traveling at a speed less than 20 mph the driver will not be able to set the desired cruising speed. The changes are reported in Table 9.

All test cases which exercised one of the changed use cases, either `Cruise` or `Resume` are considered retestable. The rest of the test cases are reusable.

6.4. Conclusions from the case studies

In some cases, the number of reusable test cases represented a large proportion (up to 100%): It seems to indicate that substantial savings can be obtained, especially since the whole process can be automated.

However, the case studies have shown that changes can have a widely variable impact on the resulting system. Large numbers of test cases may be obsolete, retestable, or reusable. In some cases the results are intuitive, in others the RTSTool was useful to uncover unexpected retestable test cases. But in general, we expect such a technology to be even more useful for large systems, involving many designers in diagram changes, when no one has a comprehensive understanding of all the use cases and their design. In such a system, a manual impact analysis would likely lead to errors, especially in a context with typical project pressures.

7. Further related work

As noted in [31], a large portion of the regression testing strategies are white-box strategies [13,28,32] and require code analysis. As in [31], our work focuses on model-based testing where changes related to functionality of a system are regression tested. The approach described in [31] uses a functional model (referred to as the "domain model") of the system under test to generate test cases and builds a mapping between changes to the domain model and the impact it has on test cases, so as to classify them. However, their working context is very different from ours as it is based on a non-object-oriented, black-box testing strategy. More recent model-based regression test selection techniques are more comparable to our work. In [7] regression tests are selected from UML activity diagrams representing software behavior. Since an activity diagram is similar to a control flow graph, the authors adapt a regression test selection algorithm originally defined for C++ source code regression test selection [29]. They also discuss a risk-based regression test selection technique, whereby additional tests are selected according to both their cost and the risk involved in not selecting them in terms of failure consequences. As suggested by the authors, this may be a way of coping for code changes that do not translate into design changes. In [33], a regression test selection strategy for systems designed with the UML (class, collaboration and state diagrams) is presented. It addresses a different objective from ours since the authors focus on component-based systems. Furthermore, the description of change identification and the description of how this information is actually used for test selection are rudimentary and does not account for as much of the UML notation as our solution. In [17] the authors also present a model-based test regression strategy for UML. They only consider the structural aspects of the UML design though (class diagram) and the regression strategy is only succinctly described. In [10] a model (UML-like) based regression test selection strategy is also succinctly described and there are not enough details available for us to really compare it with our approach. In [22,23] the architecture of a software system, comprising structural (topological) and behavioral (labeled transition system) models (these are not UML models though), is used to perform regression test selection. The labeled transition systems are combined with structural information to build graphs (unfortunately, not much details are provided on that aspect) and existing code-based regression test selection algorithms are used. More recently, control and data dependencies between states and transitions in an extended finite state machine have been used to identify the impact of model modifications and perform regression test selection ([15] extended in [8]). We consider this work as complementary to ours since it focuses on another type of model (state machine), though it is once again not directly applicable in the context of UML-based development.

One last work related to the use of UML has been presented in [26]. The objective is very different from all the previously discussed strategies, including ours, since the authors present a technique to regression test UML models rather than select regression tests from UML models: model (regression) testing vs. model-based (regression) testing of the executable system.

There exist a number of papers proposing object-oriented, code-based test selection techniques [13,28]. Rothermel et al. propose a solution for C++ based on interprocedural control flow graphs augmented with a careful analysis of non-executable statements, which can also affect test case executions. An algorithm is provided to select test cases for application programs using changed classes and their derived classes. Harrold et al. focus on Java and the problem of handling incomplete programs, due to the fact that third-party software or external libraries are commonly used. Xie et al. use program spectra, i.e., distributions of path executions (i.e., execution profiles), to perform regression test selection.

8. Conclusion

We propose here a methodology supported by a prototype tool to tackle the regression test selection problem at the architecture/design level in the context of UML-based development. Our main motivation is to enable, in the context of UML-based development, regression test selection based on design change information, early in the change process. We also present three case studies that were used as an initial feasibility and benefit assessment. These case studies are varied in the sense that they cover very different systems and changes, in both industrial and academic settings. Results show that design changes can have a complex impact on regression test selection and that, in many cases, automation is likely to help avoid human errors. Our objective has been to ensure that regression testing was safe while minimizing regression testing effort. But we have shown that certain changes may not be visible in the design and may require additional attention during coding or a special way to document them during design. Another limitation is that, based on UML design information, test selection may not be as precise as if it was based on detailed code analysis (see examples in [13,29]). Improving precision by analyzing in more detail guard conditions and OCL contracts is the focus of our current re-

search. However, our case studies have only shown one case of imprecision in classifying a test case as retestable.

Despite the above limitations, by providing a methodology and tool to perform impact analysis and regression test selection based on UML designs, we hope to achieve:

- Higher efficiency in test selection based on the automation of design change analysis and traceability between UML designs and regression test cases.
- Better support for assessing and planning regression test effort earlier in the change process, that is once design changes have been determined.

As future work, it is important to run additional case studies to assess the drawbacks and advantages of working with UML designs instead of code to support regression test selection. A detailed cost-benefit analysis is required. A comparison of UML-based and code-based techniques should also be investigated within realistic settings in order to realistically assess the loss caused by working at a design level.

Acknowledgement

This project was supported in part by a Canada Research Chair. We would like to thank Gregg Rothermel for helpful discussions on the evaluation framework.

References

[1] D. Bell, UML's Sequence Diagram, http://www.ibm.com/developerworks/rational/library/3101.html, 2004 (Accessed March 2008).
[2] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, second ed., Addison-Wesley, 2005.
[3] L.C. Briand, Y. Labiche, A UML-based approach to system testing, Software and Systems Modeling 1 (1) (2002) 10–42.
[4] L.C. Briand, Y. Labiche, L. O'Sullivan, Impact analysis and change management of UML models, in: Proceedings of IEEE International Conference on Software Maintenance, 2003, pp. 256–265.
[5] L.C. Briand, Y. Labiche, G. Soccar, Automating impact analysis and regression test selection based on UML designs, Carleton University, Technical Report SCE-02-04, http://www.sce.carleton.ca/Squall, March, 2002, a short version appeared in Proceedings of IEEE ICSM 02.
[6] B. Bruegge, A.H. Dutoit, Object-Oriented Software Engineering Using UML, Patterns, and Java, second ed., Prentice-Hall, 2004.
[7] Y. Chen, R.L. Probert, D.P. Sims, Specification based Regression test selection with risk analysis, in: Proceedings of Conference of the Center for Advance Studies on Collaborative Research, 2002.
[8] Y. Chen, R.L. Probert, H. Ural, Model-based regression test suite generation using dependence analysis, in: Proceedings of ACM International Workshop on Advances in Model-Based Testing, 2007, pp. 54–62.
[9] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. GilChrist, F. Hayes, P. Jeremaes, Object-Oriented Development – The Fusion Method Object-Oriented Series, Prentice-Hall Edition, 1994.
[10] D. Deng, P.C.-Y. Sheu, Model-based testing and maintenance, in: Proceedings of International Symposium on Multimedia Software Engineering, 2004.
[11] B.P. Douglass, Real Time UML, third ed., Addison-Wesley, 2004.
[12] M.J. Harrold, Testing evolving software, Journal of Systems and Software 47 (2–3) (1999) 173–181.
[13] M.J. Harrold, J.A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S.A. Spoon, Regression test selection for java software, in: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01), 2001.
[14] IBM-Rational: Rational Software Architect, 2005. www-306-ibm.com/software/awdtools/architect/swarchitect/.
[15] B. Korel, L.H. Tahat, B. Vaysburg, Model-based regression test reduction using dependence analysis, in: Proceedings of IEEE International Conference on Software Maintenance, 2002, pp. 214–223.
[16] C. Larman, Applying UML and Patterns, third ed., Prentice-Hall, 2004.
[17] Y. Le Traon, T. Jéron, J.-M. Jézéquel, P. Morel, Efficient object-oriented integration and regression testing, IEEE Transactions on Reliability 49(1) (2000) 12–25.
[18] H.K.N. Leung, L. White, Insights into regression testing, in: Proceedings of IEEE International Conference on Software Maintenance (ICSM), 1989, pp. 60–69.
[19] H.K.N. Leung, L.J. White, A cost model to compare regression test strategies, in: Proceedings of Conference on Software Maintenance, 1991, pp. 201–208.
[20] B. Meyer, Design by contracts, IEEE Computer 25 (10) (1992) 40–52.
[21] R. Mitchell, J. McKim, Design by Contract, by Example, Addison-Wesley, 2001.
[22] H. Muccini, M.S. Dias, D.J. Richardson, Reasoning about software architecture-based regression testing through a case study, in: Proceedings of Annual International Computer Software and Applications Conference, 2005, pp. 189–195.
[23] H. Muccini, M.S. Dias, D.J. Richardson, Towards software architecture-based regression testing, in: Proceedings of ICSE Workshop on Architecting Dependable Systems, 2005.
[24] OMG, UML 2.0 Superstructure Specification, Object Management Group, Final Adopted Specification ptc/03-08-02, 2003.
[25] T. Pender, UML Bible, Wiley, 2003.
[26] O. Pilskalns, G. Uyan, A. Andrews, Regression testing UML designs, in: Proceedings of IEEE International Conference on Software Maintenance, 2006.
[27] G. Rothermel, M.J. Harrold, Analysing regression test selection techniques, IEEE Transactions on Software Engineering 22 (8) (1996) 529–551.
[28] G. Rothermel, M.J. Harrold, A. Safe, Efficient regression test selection technique, ACM Transactions on Software Engineering and Methodology 6 (2) (1997) 173–210.
[29] G. Rothermel, M.J. Harrold, J. Debhia, Regression test selection for C++ software, Journal of Software Testing, Verification, and Reliability 10 (2) (2000) 77–109.
[30] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, second ed., Addison-Wesley, 2004.
[31] A. Von Mayrhauser, N. Zhang, Automated regression testing using DBT and Sleuth, Journal of Software Maintenance 11 (2) (1999) 93–116.
[32] J. Warmer, A. Kleppe, The Object Constraint Language, second ed., Addison-Wesley, 2003.
[33] Y. Wu, A.J. Offutt, Maintaining Evolving Component-based Software with UML, in: Proceedings of European Conference on Software Maintenance And Reengineering, 2003, pp. 133–142.