Automatic Generation of Just-in-time Online Assessments from Software Design Models

Imran A. Zualkernan[1], Salim Abou El-Naaj[2], Maria Papadopoulos[3], Budoor K. Al-Amoudi[4] and Charles E. Matthews[5]

[1]American University of Sharjah, Sharjah, U.A.E. ‘’ Tel: 971-50-6260243 // Fax: 971-6-5152979 // izualkernan@aus.edu

[2]EDS, Dubai, General Motors DWTC, P.O. Box 9233, Dubai, U.A.E. // Tel: 971-50-8115582 // Fax: +971-4-331 4102 // salim.abouelnaaj@gm.com

[3]International School of Choueifat, P.O. Box 15679, Dubai, U.A.E. // Tel: 971-4-2220560 // Fax: 971-4-2257955 // maria.mitchae84@gmail.com

[4]Thuraya Satellite Telecommunications Company, P.O. Box 29222, Sharjah, U.A.E. // Tel: 971-6-8080482 // Fax: 971-6-8828484 // b_alamoudi@thuraya.com

[5]Fifth Generation Systems, Ltd., Markham, Ontario, Canada // Tel: (905) 887-8522 // charles.matthews@acm.org

ABSTRACT

Computer software is pervasive in today’s society. The rate at which new versions of computer software products are released is phenomenal when compared to the release rate of new products in traditional industries such as aircraft building. This rapid rate of change can partially explain why most certifications in the software industry are generic as opposed to those in the aircraft-building industry where engineers and technicians are certified to work on a specific aircraft. For example, a software engineer may be certified on a database management system, but not on a specific implementation based on the database management system. Hence, software engineers are allowed to make critical changes to specific designs for the next release of a software product with little formal assessment of their understanding of the design. This paper presents a system that automatically generates just-in-time online assessments for judging a software engineer’s comprehension of artifacts representing software designs. The assessments thus generated are compliant with the IMS-QTI 2.1 standard. The system is based on the AXIS web-services architecture and provides a priori statistical estimates of effectiveness of each individually generated assessment.

Keywords

Design certification, Assessment generation, QTI, Web services

Introduction

New versions of software products are released every few months. Given the mission-critical nature of software products today, it is reasonable to expect that software engineers should be “certified” on a software design to ensure that they understand the design before being allowed to change it for the next release. However, this is often not the case. Most software design methodologies employ a “tell and pray” pedagogy in which the engineers are given the software product along with its design documents and are expected to pick up the design as they work; the engineers are almost never assessed to ensure that they understand the design.

In many situations, however, engineers are provided with visualization tools to help them understand software designs (Hundausen, Douglas & Stasko, 2002; Stasko, Dominique, Brown, & Price, 1988). These tools range from automatic help-file generators at the code level to browsers at the design and specification level (Confora, Cimitile, Carlini, & De Lucia, 1998; Hendrix, Cross, & Maghsoodloo, 2002; Lanza & Ducasse, 2003; Lehman, 1989; Tonella, 2003; Luqi, Berzins, & Qiao, 2004). While these tools help an engineer explore existing designs, they provide little guidance on how well an engineer understands a design.

Similarly, inspection and walkthrough methods (Anderson, Reps, & Teittelbaum, 1989; Brykczynski, 1999; Miller & Yin, 2004; Traore & Aredo, 2004) are often used as collaborative and reflective design exercises in which engineers are forced to articulate, reflect, and defend their software designs. Inspection and walkthrough methods are effective in ensuring quality of new software designs. Like visual tools, however, inspection and walkthrough methods also offer little in establishing how well an engineer understands a particular design in any meaningful way.

Using formal methods is another approach to understanding properties of software designs (Apvrille, Courtiat, Lohr, & Saqui-Sannes, 2004; Eshuis & Wieringa, 2004. However, like other approaches mentioned earlier, these methods also fail to provide an objective assessment of an engineer's level of understanding of a design.

Obviously, it is possible to manually construct exams to judge an engineer's level of understanding of a software design. However, every few months, typical computer software products release a newer version. Therefore, such an approach is not cost-effective: exams would need to be rewritten every time the design changes.

This paper presents an automated approach to judging a software engineer's level of comprehension of artifacts created during software design. This approach is cost-effective because online assessments to determine an engineer's level of understanding of a software design are automatically generated and graded; every time the design changes, assessments can be automatically re-generated.

The paper first presents a framework to formally define the problem and outline an approach. The framework is followed by a description of architecture and a prototype system based on this approach. Examples of automatically generated assessments are provided next. The paper ends with a discussion of limitations and conclusions.

Framework

The primary objective of this research is to automatically generate assessments for checking a software engineer's level of understanding of software design artifacts or models. For example, the Unified Modeling Language (UML) (OMG-UML, 2003) currently supports thirteen such models. Each design model is typically an approximation of some aspect of the software or the world. The task of checking a model against the world is typically called *validation*. Similarly, the task of checking a model for internal consistency and completeness is called *verification*. The goal of this research is neither validation nor verification. Rather, it is to determine how closely the "mental model" or *adaptation* (Simon, 1983) of a software engineer matches the actual design model. Hence, if M is a design model, the purpose of an assessment is to determine how closely M', the understanding of a software engineer, matches the actual model M. In other words, the goal of assessment is to approximate $(M – M')$ or the degree to which the understanding of a software engineer differs from the actual design model.

The meaning of a software engineer's understanding of a model, M, depends on an "authentic" context of its use (Brown, Collins, & Duguid, 1989; Lave & Chaiklin, 1993). This research assumes that this understanding is inherently related to the use of model M in the context of design. For example, if a model is used to emphasize logical gaps in reasoning, an understanding of this model consists of a software engineer's ability to perform the task of finding such logical gaps. More generally, an understanding of a particular model M is related to a set of related design tasks.

The framework has two components. The first component describes how to generate assessments from a design model, and the second describes how to determine the effectiveness of such automatically generated assessments.

Generation of assessments

On surface, the problem of automatic generation of assessments seems related to the problem of automatically generating test cases from design models (Andrews, France, Ghosh, & Craig, 2003; Bigot et al., 2004; Chow, 1978; Nebut, Fleurey, Le Traon, & Jézéquel, 2006). However, unlike test cases, which are used for validation and verification, assessments check the depth of an engineer's comprehension of a design model. Therefore, this framework follows Bloom's (1956) pedagogical categories as the foundation for generating assessments. Bloom provides generic categories of levels of learning for cognitive tasks, such as knowledge, comprehension, application, analysis, synthesis, and evaluation. Each successive level of learning requires a higher level of understanding. Roughly, each question in an assessment is generated using some of the following action verbs corresponding to each category:

- **Knowledge:** arrange, define, duplicate, label, list, memorize, name, order, recognize, relate, recall, repeat, reproduce, or state.

- **Comprehension:** classify, describe, discuss, explain, express, identify, indicate, locate, recognize report, restate, review, select, or translate.
- **Application:** apply, demonstrate, dramatize, employ, illustrate, interpret, operate, practice, schedule, sketch, solve, transcribe, use, or write.
- **Analysis:** analyze, calculate, categorize, compare, contrast, correlate, criticize, diagram, differentiate, discriminate, distinguish, examine, experiment, prove, question, or test.
- **Synthesis:** arrange, assemble, collect, compose, construct, create, design, develop, formulate, integrate, manage, organize, plan, prepare, propose, or set up.
- **Evaluation:** appraise, argue, assess, attach, choose, defend, estimate, evaluate, judge, predict, rate, select, support, or value.

Therefore, for a particular design model, an application of Bloom's categories leads to the generation of assessment questions at each of the six levels of understanding.

This research is currently focused on automatic generation of assessments for activity diagrams. Activity diagrams are a type of model commonly used in software design. An activity diagram for making a beverage is shown in Figure 1. As Figure 1 shows, an activity diagram typically captures control flow in a situation. For example, Figure 1 shows that a coffee machine has to be turned on *before* brewing coffee. An activity diagram also captures coordination or syncing points when activities are allowed to occur concurrently until some point in time. For example, in Figure 1, putting coffee in filter, adding water, and getting cups can happen concurrently. However, all three activities must finish before coffee can be poured. This condition is indicated by the horizontal bar in the figure. Finally, activity diagrams also include "guards" or conditions such as "coffee not found" that lead to different branching based on decision points indicated by the diamond symbol.

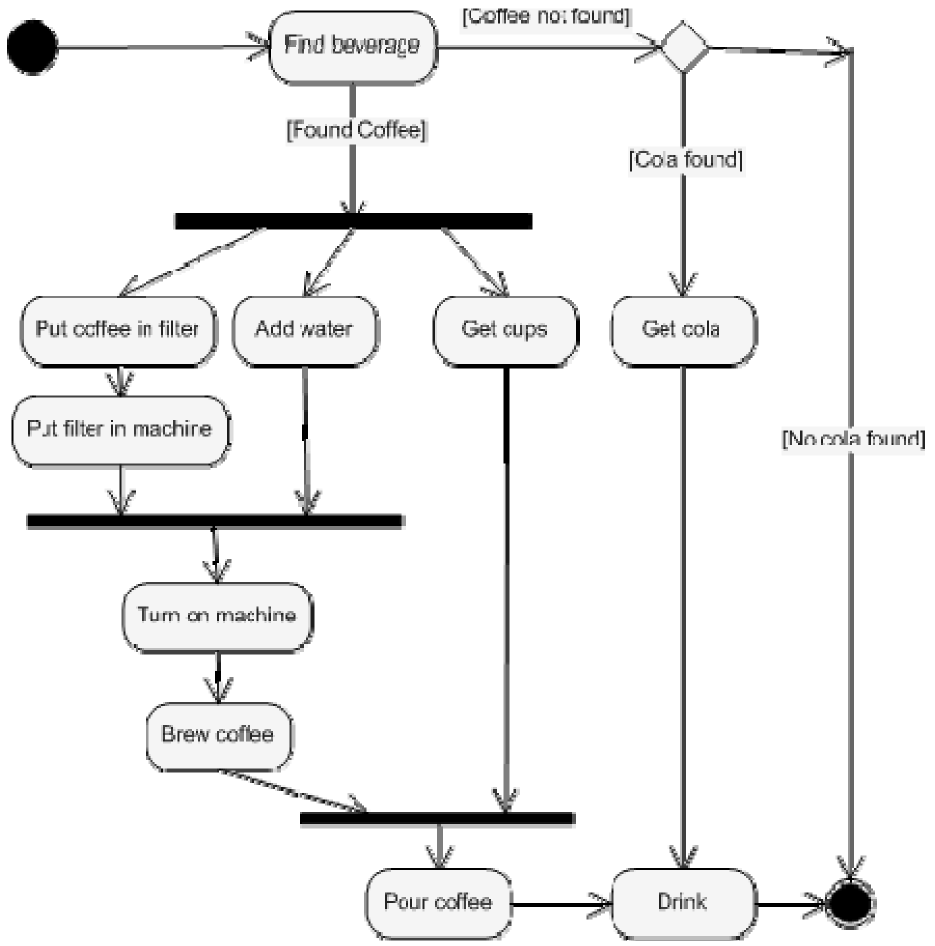


Figure 1. A sample activity diagram for drinking coffee (Adapted from OMG-UML, 2003)

The activity diagram shown in Figure 1 will now be used to illustrate how assessments at various levels of Bloom's hierarchy can be generated from an arbitrary activity diagram.

One type of knowledge-level assessment is a simple recall. For example, for the activity diagram shown in Figure 1, a recall-based assessment consists of asking an engineer to indicate if "pour coffee" was one of the activities in the diagram. Another variation may be to ask a software engineer to list the various activities in the diagram.

An assessment at the comprehension level is constructed by asking an engineer to describe and explain what is in the activity diagram. For example, a software engineer may be asked if coffee can be poured (i.e., "pour coffee") without both "brew coffee" and "get cups" finishing first. This information is embedded in the activity diagram via the horizontal bar (see Figure 1). As this example illustrates, to answer such questions correctly, an engineer needs to understand how to interpret the relationships among the various components of an activity diagram.

An assessment at the application level is constructed by asking a software engineer to apply what she knows from this activity diagram. For example, a software engineer may be asked to produce a sequence of activities (if any) that start from the "find beverage" activity and ultimately lead to the "pour coffee" activity, under the condition that no coffee was found. In order to perform correctly for this assessment, a software engineer needs to understand how to apply conditions to generate a specific path through the activity diagram.

An assessment at the analysis level consists of the ability to use facts and inferences to understand properties of the activity diagram. For example, a software engineer may be asked to describe conditions (if any) under which "get cola" and "get cups" happen concurrently. A successful performance of this assessment requires that the engineer be able to calculate the conditions under which specific paths can be taken concurrently in the activity diagram.

An assessment at the synthesis level requires building a structure or pattern from diverse elements to create new meaning in the activity diagram. For example, a software engineer may be asked to alter the activity diagram to enable the use of cups for drinking cola as well. A successful performance at this level requires the engineer to have the ability to successfully modify the existing activity diagram to satisfy additional functionality or constraints.

Finally, an assessment at the evaluation level requires judgment about the value of ideas or materials included in the activity diagram. For example, one may ask the software engineer if the activity diagram handles exceptional conditions such as the coffee machine not turning on appropriately. In general, various properties of the activity diagram such as usability, reliability, and security can be assessed at this level.

In summary, the general framework for generating assessments is based on Bloom's taxonomy. The description and guide words at each level are used to create specific heuristics that generate questions for an arbitrary activity diagram. A typical assessment consists of a number of questions from each level of understanding.

How many questions are sufficient to ensure that an engineer's understanding has been judged adequately? This issue is addressed by the creation of an assessment risk profile, described next.

Creation of assessment risk profile

The purpose of an assessment risk profile is to provide a reasonable measure of "goodness" of an automatically generated assessment for a particular activity diagram. More specifically, the purpose of an assessment risk profile is to provide an estimate of the relationship between the size of an assessment (i.e., the number of questions being asked) and the probability that an assessment of this size will catch or uncover particular types of misunderstandings.

Critical to the creation of a risk profile is the concept of a misunderstanding. Misunderstandings are commonly occurring differences between the actual model (e.g., the activity diagram) and how those differences are understood by a software engineer. For example, in the activity diagram shown in Figure 1, a software engineer may have the misunderstanding that brewing coffee comes before turning on the machine. Another misunderstanding may be that "pour coffee" does not exist at all. This research uses the Hazard Operators (HAZOP) Kim, Clark, & McDermid, 1999) scheme to generate classes of misunderstandings for activity diagrams, as shown in Table 1.

Each row in Table 1 represents a class of misunderstandings in an activity diagram. Each class of misunderstanding can have different manifestations. Table 2 shows different manifestations for each class of misunderstanding in a graphical fashion. The first column in Table 2 shows a manifestation of misunderstanding. The second column in Table 2 shows an example of a portion of an activity diagram. The third column shows how the portion of an activity diagram in the second column is misunderstood. For example, as Table 1 shows, the AS_WELL_AS class represents those misunderstandings that maintain the original intent of the activity diagram but adds additional spurious behaviors. Two manifestations of such misunderstandings (additional loop edge and additional reverse edge without removing the original edge) are shown graphically in the first and second rows of Table 2. The first row of Table 2 further shows an example of how a simple three-node activity diagram can be misunderstood by adding an additional loop edge (i.e., to node C).

Table 1. HAZOP classification for misunderstandings in an activity diagram

Class	Nature of misunderstanding	Manifestations of misunderstanding
AS_WELL_AS	The specific intent of the activity diagram is maintained but the misunderstanding yields additional results.	1. Additional loop edge 2. Additional reverse edge without removing the original edge
PART_OF	Only some of the intention in the activity diagram is achieved.	3. Missing an edge 4. Missing an activity (and reconnected edges)
REVERSE	Reverse flow — flow in the activity diagram is in the wrong direction.	5. Reversed edge 6. Two consecutive activities swapped
OTHER_THAN	A result other than initial intention is achieved while maintaining the structure of the activity diagram.	7. One activity substituted for another 8. An additional activity between two activities

Another general class of misunderstandings in Table 1 is PART_OF. Manifestations of this class of misunderstandings are missing edges or activities: the software engineer believes that the activity does not include a specific activity or an edge. A third class of misunderstandings in Table 1 is represented by the REVERSE class. In the case of edges, the REVERSE class translates into the belief by a software engineer that an edge is reversed. In the case of an activity, REVERSE misunderstanding may exist if the engineer believes that two contiguous activities are actually swapped. Finally, the OTHER_THAN misunderstandings are those cases in which the original structure of the activity diagram is believed to be about the same but additional content is substituted. For example, an engineer may have the right structure of the activity diagram, but may substitute or confuse one activity for another. A more extreme case may be the inclusion of an additional activity between two existing activities.

The "goodness" of an automatically generated assessment can be judged by the assessment's ability to catch a particular type of misunderstanding. This is done by using a variant of mutation testing (Boland, Singh, & Cukic, 2003; DeMillo, Lipton, & Sayward, 1978; Howden, 1982). Mutation testing has been used to evaluate test case suites (i.e., a set of test cases). A mutant (or variation) of a computer program is generated by deliberately applying a mutation operator to the program. Each mutation operator represents a fault (e.g., a misspelled variable). The mutated program is then tested using the test suite. If none of the test cases in the test suite fail, then the test suite has failed to kill the mutant. The failure to kill the mutant means that the specific test suite is not suitable for catching the fault that was introduced.

Since an assessment consists of a set of questions for a software engineer, the assessment acts much like a test suite. Similarly, the activity diagram is analogous to a computer program, and a misunderstanding is analogous to a fault. Therefore, a mechanism similar to mutation testing can be used to evaluate the effectiveness of the assessments thus generated.

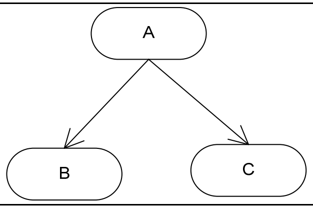
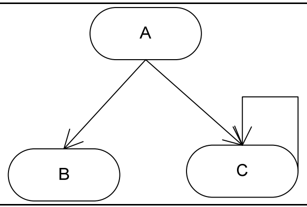
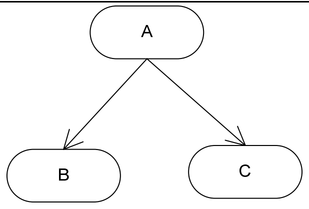
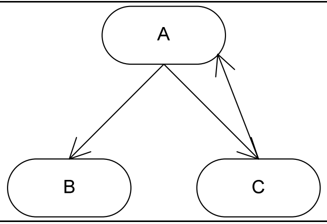
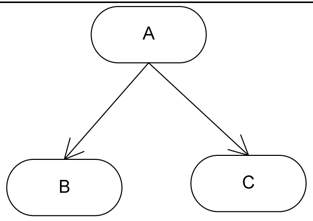
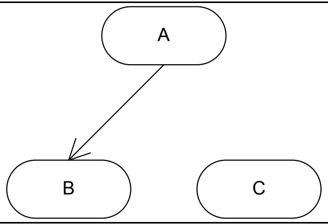
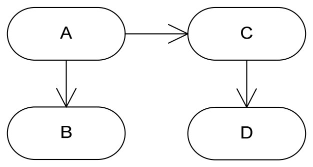
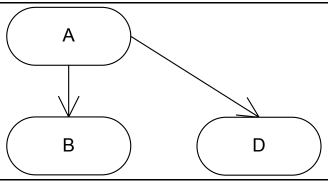
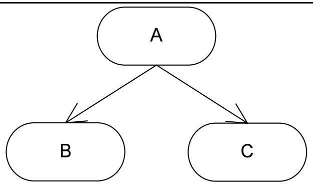
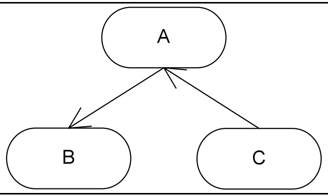
Let $a \in A$ represent an arbitrary activity diagram. A mutation operator $\mu \in M$ is applied to the activity diagram a to generates a set of mutated activity diagrams A' where $A' = \{a' \mid a' \leftarrow \mu(a)\}$. The list of mutation operators (i.e.,

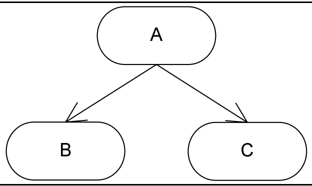
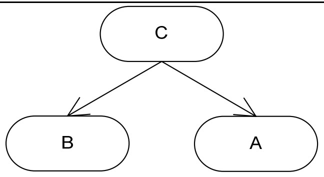
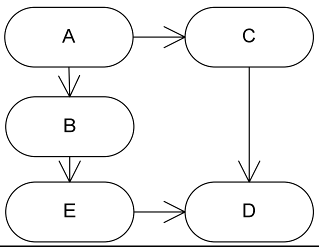
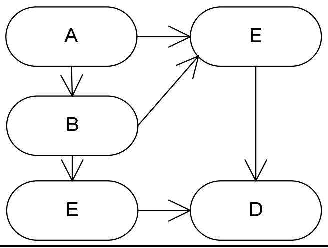
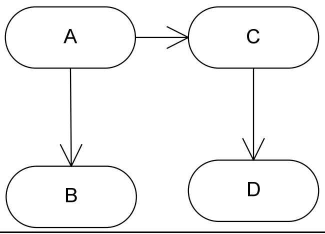
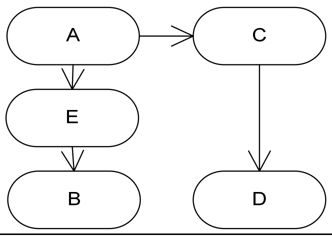
M) is based on the misunderstandings shown in Table 2. For example, one mutation operator adds an additional loop edge to an activity diagram, another mutation operator reverses an edge, and so on.

Let $Q = \{q \mid q \leftarrow BAG(a)\}$ represent an assessment or a set of automatically generated questions using the approach described in the previous section. (The approach is called Bloom's assessment generator [BAG].) Each $q \varepsilon Q$ represents one question for the software engineer based on Bloom's taxonomy.

Finally, for a particular $q \varepsilon Q$, $a \in A$ and $a' \in A'$, let $ORACLE(q, a, a')$ represent an oracle that returns *true* if the question $q \in Q$ produces the same answer for both a and a' , and *false* if it does not. For example, suppose the question "Is 'add water' an activity in the activity diagram?" was generated from the activity diagram shown in Figure 1. The correct answer for this question, based on the original activity diagram, is obviously "yes." However, suppose a misunderstanding were introduced in the original activity diagram by removing the activity "add water." This is done by applying the mutation operator (4) from Table 2. The resulting mutated activity diagram would be identical to the original diagram except that the "add water" activity and associated edges would be missing. If one were to answer the original question using the mutated diagram, the answer would be "no." This means that the oracle answers *false* in this case because the answer to the question in the original and the mutated diagrams are different. A *false* answer from the oracle means that a misunderstanding was caught by the question.

Table 2. Example manifestations of misunderstandings in an activity diagram

Manifestation	**Actual**	**Misunderstood as**
1. Additional loop edge		
2. Additional reverse edge without removing the original edge		
3. Missing an edge		
4. Missing an activity and reconnected edges		
5. Reversed edge		

6. Two consecutive activities swapped		
7. One activity substituted for another		
8. Additional activity between two activities		

An algorithm to generate a risk profile for a specific activity diagram $a \in A$ takes a set P = {*a, M, k, n*} as input, in which k represents the number of questions in the assessment being generated, n represents the number of replication representing statistical accuracy, and M is a set of mutation operators. The algorithm carries out binomial experiments that produce an estimated probability that an assessment with k questions generated from activity diagram a will catch a particular type of misunderstanding. An outline of the algorithm for the system is shown below.

```
Given P = {a, M, k, n}
For each $\mu \in M$ do // for all mutation operators
$A' = \{a' \mid a' \leftarrow \mu(a)\}$ // generate the set of mutated activity diagrams for $a$
For each $a' \in A'$ do // for all mutations of $a$
$success(a') \leftarrow 1$
For $i = 1, n$ do // repeat the experiment $n$ times
// randomly generate an assessment with $k$ questions
$Q = \{q \mid q \leftarrow BAG(a)\}$ where $|Q| = k$
For each $q \varepsilon Q$ do
If $(ORACLE(a, a', q) \equiv true)$ // the mutated and original gave
// the same answer
then $success(a') \leftarrow 0$ // one misunderstanding was not caught
$p(\mu, a, k) \leftarrow \left( \sum_{n \times |A'|} success(i) \right) / \left( n \times |A'| \right)$
```

The algorithm returns the probability $p(\mu, a, k)$, that for a specific activity diagram a, an assessment with k questions will catch a particular misunderstanding injected by a mutation operator μ. Since each application of the oracle represents a binomial experiment, the statistical bounds on the probability p are given by

$$p \pm z_{\alpha/2}\sqrt{p(1-p)/(n \times |A'|)}$$

where z represents the normalized score while α represents the Type I error rate.

In summary, the risk profile for each activity diagram is generated by introducing known misunderstandings repeatedly in the same activity diagram, randomly generating assessments of a particular size and determining how many of these assessments are able to catch the introduced misunderstandings. This procedure leads to a statistical profile of how well the technique works for a specific activity diagram in terms of unearthing various types of misunderstandings.

System architecture

A prototype system based on the framework presented earlier was constructed to automatically generate online assessments from arbitrary activity diagrams. As shown in Figure 2, the system architecture is based on the Apache Axis framework for building web services (Axis, 2008). The system relies on Apache Tomcat (Tomcat, 2008) and uses MySQL (MySQL, 2008) as the back-end database. The system has one primary server that provides assessment generation services by optionally using distributed clients to calculate the risk profile for an arbitrary activity diagram. The assessment generation is not computationally expensive because it relies on known graph algorithms and pattern matching. The calculation of risk profile, however, is computationally expensive because for each profile all possible mutations of a particular type (say, reverse edge) are applied to an activity diagram, and this process is replicated depending on the statistical accuracy desired. Distributed clients essentially help tackle this portion of the computational complexity. The server supports two types of clients. Submit clients (SC) are used to submit activity diagrams, and processing clients (PC) are used to help generate the risk profile for a particular activity diagram. Each of these is described below.

Server

The server supports two SOAP (Soap, 2008) interfaces. The first interface is a submit activity diagram interface (S). This interface allows any SOAP client to submit an activity diagram to generate an assessment. In addition to submitting an activity diagram, the client also specifies the number of questions in the assessment (k), the mutation operators to be used, and a sample size (n) for the risk profile. The server, in turn, generates and returns an assessment and a risk profile to the client. The second interface is a processing client interface (P) that distributes and runs mutations for the risk profile and returns the results to the server.

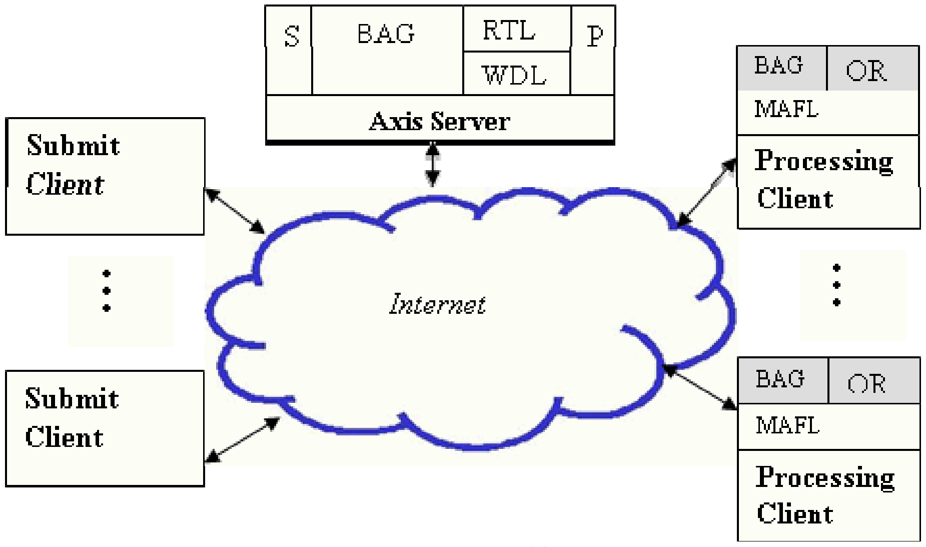


Figure 2. System architecture

The server has three primary components: the Bloom assessment generator (BAG), the work distribution layer (WDL), and the results tabulation layer (RTL). A SOAP client connects to the submit interface (S) of the server to provide a job to execute. A typical job consists of an activity diagram in the form of an XMI file (OMG-XMI, 2006), the mutation operators to apply, the number of questions in each assessment, and the sample size. The BAG takes the XMI file representing an activity diagram as input and generates an assessment for the activity diagram as an XML file based on the IMS QTI format (IMS-QTI, 2006).

```
Generate_order_know_13(activity_diagram a)
{
    $a_x$ = randomly select an activity from
         activity diagram a
    $a_y$ = randomly select another activity from
         activity diagram a

  {$a_1$, … $a_m$} = Generate m activities that
               lie on path between $a_x$ and $a_y$

   Generate the QTI 2.1 code for the
   following question template:

     "Arrange $a_1$, $a_m$, ..$a_i$ performed
      between activities  $a_x$ and $a_y$ in the
      correct sequential order."

   with the correct answer being the
   order 1,..m.
}
```

Figure 3. A knowledge-level heuristic generating a multiple-choice question

```
Generate_t/f_comp_3(activity_diagram a)
{
 Randomly jump to first: or second:

first:
{$a_i$, $a_j$} = Find a random pair of activities
$a_i$ and $a_j$ that share a join in a
correct_answer = true;
goto generate:

second:
{$a_i$, $a_j$} = Find two activities $a_i$ and $a_j$
that are strictly sequential (ai follows aj)
correct_answer = false;

generate:

Generate the QTI 2.1 code for the following
question template:

"Are the activities $a_i$ and $a_j$ potentially
concurrent?"

with correct answer being correct_answser;

}
```

Figure 4. A comprehension-level heuristic for generating a true/false question

BAG is implemented using Java, SWI-Prolog (Clocksin & Mellish 1994; SWI-Prolog, 2008), and JPL Java APIs (http://sourceforge.net/projects/jpl). BAG consists of a collection of heuristics that are organized according to Bloom's levels. Each level incorporates a number of pedagogical heuristics. From a learning perspective, a software engineer needs to learn the sequencing and conditions under which particular activities can occur in an activity diagram. This is reflected in most heuristics. These heuristics are encoded as rules in the Prolog programming language. Each heuristic is parameterized with respect to the various components of an activity diagram. Figure 3 shows the pseudo-code for one such heuristic at the knowledge level. This heuristic is able to generate questions that ask a software engineer to order a set of activities between two arbitrarily selected activities (a_x and a_y) for a particular activity diagram.

Figure 4 shows an example of a heuristic that generates a true/false question asking the user to select activities that can be potentially concurrent. As the figure shows, the heuristic randomly generates both positive (true) and negative (false) questions.

Finally, Figure 5 shows an example of a heuristic at the synthesis level. This heuristic taps into a software engineer's ability to effect changes to the existing activity diagram and hence is at the synthesis level. In order to be able to answer questions generated from this heuristic, the software engineer needs to work through the consequences of changes in the activity diagram.

```
Generate_t/f_syn_12(activity a)
{
  a_x = randomly select an activity from
        activity diagram a.
  a_y = randomly select an activity that
        follows activity a_x.
  a_z = randomly select another activity
        that follows activity a_x.

Generate all paths between a_x and a_z after
deleting an edge between a_x and a_y. If there
is at least one path then correct_answer =
true. Otherwise, the correct_answer = false.

Generate the QTI 2.1 code for the following
question template:

"If an edge between activity a_x and activity
a_y is deleted, will the user still be able to
reach activity a_z?"

   with the correct answer
   being correct_answer.

}
```

Figure 5. A synthesis-level heuristic for generating a true/false question

The WDL takes the activity diagram, the generated assessment, and the sample size representing statistical accuracy and splits and assigns various mutation tasks for generating the risk profile to different processing clients. Finally, the results tabulation layer (RTL) consolidates the results received from various processing clients.

Once an assessment is generated, the WDL divides and assigns the job to various processing clients to generate the risk profile for the job. For example, the sample of jobs submitted to the server shown in Table 3 for two activity diagrams will be distributed as tasks to various processing clients as shown in Table 4.

Table 3. Sample jobs submitted to the server

Job	M	n	k
1	{1, 2}	100	20
2	{2, 3}	50	10

M is a set of mutation operators
n is the number of replication representing statistical accuracy
k is the number of questions in the assessment being generated

Table 4. Mapping of jobs to tasks

Task	Job	μ	n	k
1	1	1	50	20
2	1	1	50	20
3	1	2	50	20
4	1	2	50	20
5	2	2	25	10
6	2	2	25	10
7	2	3	25	10
8	2	3	25	10

μ is a mutation operator
n is the number of replications representing statistical accuracy
k is the number of questions in the assessment being generated

The server waits for processing clients to attempt connection and assigns a unique identifier to each client. Once the client is connected, the server sends the parameters for the task, along with the required files, to the processing client. When the client sends back the results for the task it received, the server records these results in its database and sets the task to be completed.

Submitting clients

Since the server supports the SOAP protocol, clients can be written in any programming language that supports the SOAP protocol. These clients connect to the server through the *S* interface as shown in Figure 2. Figure 6 shows one such SOAP client (implemented in Java) that allows the user to select the mutation operators and submit an activity diagram. The submitting client is able to watch the progress on its job via a status bar.

Processing clients

Client nodes that implement the algorithm described earlier have three layers as shown in Figure 2. These clients connect to the server through the processing client (P) interface. The mutation analysis framework layer (MAFL) provides services related to creating mutations and applying the ORACLE (OR) to determine the results. In addition, each client includes BAG to randomly create an assessment of a particular size (number of questions) while the ORACLE determines whether a question yields the same results for the normal and the mutated activity diagrams.

Each processing client initiates communications with the server. When the client has resources, it continuously attempts to connect the server in order to receive a unique identifier. Once the connection is established, the client requests the task from the server and, upon receiving the job, the client starts mutation testing based on the parameters it has received. After completion, the client calls the return result method to give back the results to the server. After dispatching the results, the processing client probes the server for any additional tasks to process. The MAFL is organized according to the HAZOP classification described earlier. Currently, the MAFL includes all the misunderstandings in Table 2 implemented as mutation operators.

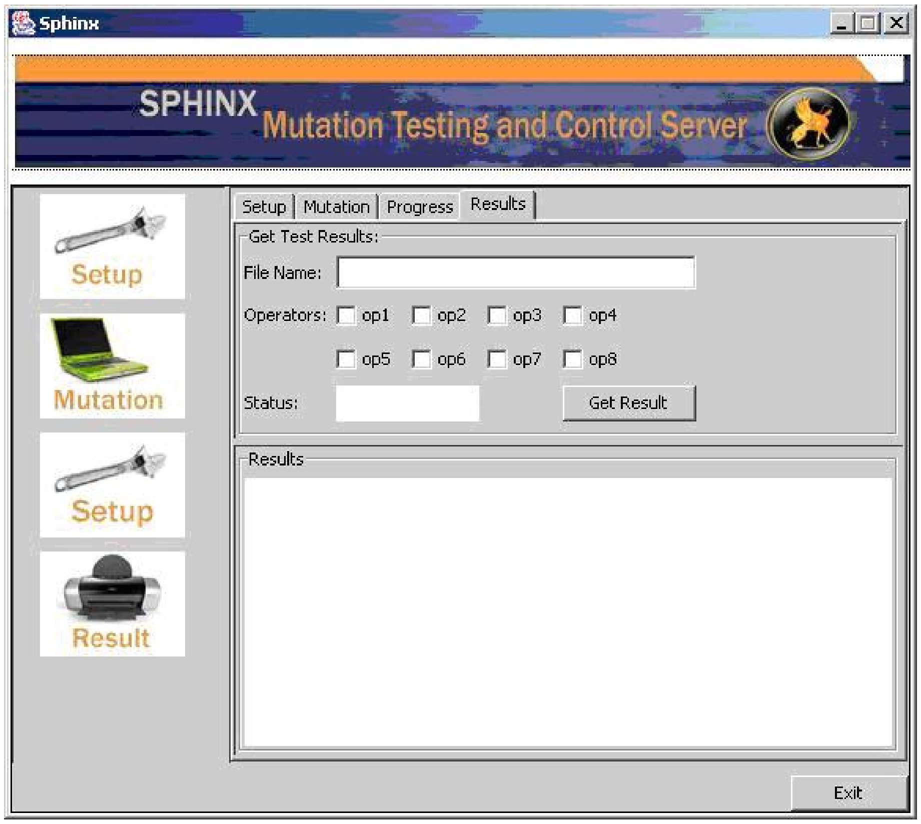
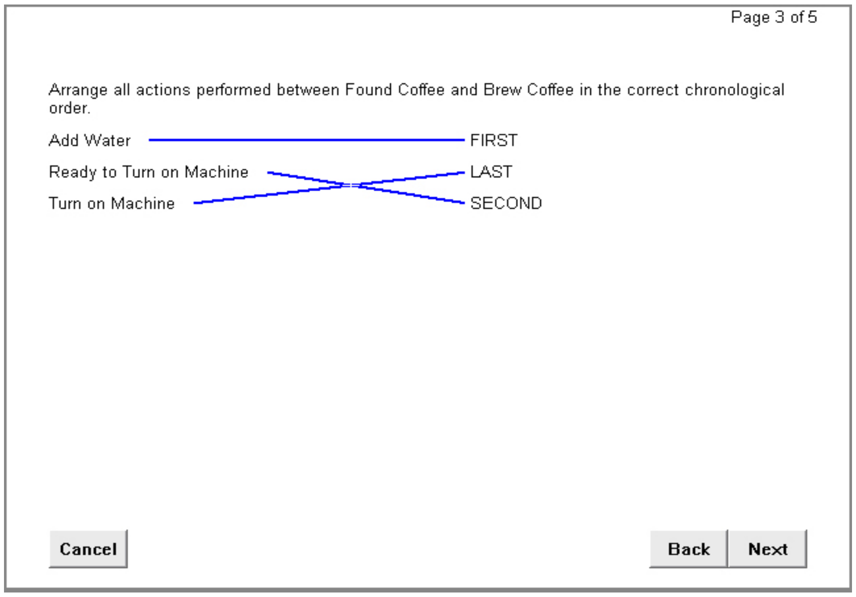


Figure 6. A sample submit client

Evaluation

The system currently contains 115 rules for generating questions at various levels of the Bloom's taxonomy. Figures 7 and 8 show sample questions from an assessment that was automatically generated from the activity diagram shown in Figure 1.



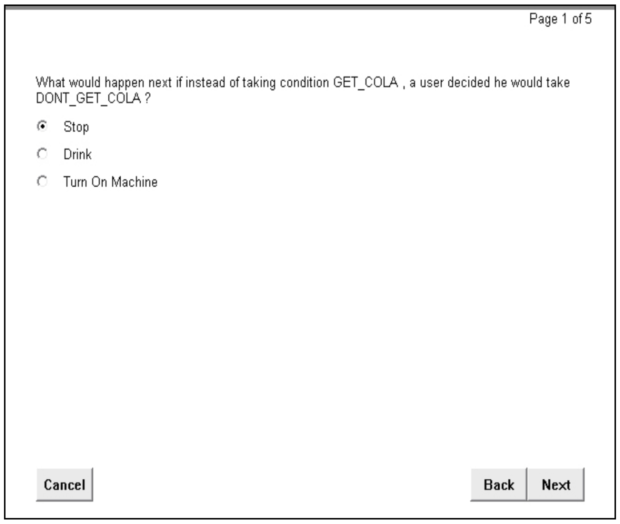
Page 3 of 5

Arrange all actions performed between Found Coffee and Brew Coffee in the correct chronological order.

Figure 7. A knowledge-level sample question for the drink-coffee activity diagram

Figure 9 shows how the results of the assessment are automatically scored and presented to a software engineer through a commercial learning management system supporting the IMS QTI assessment format.

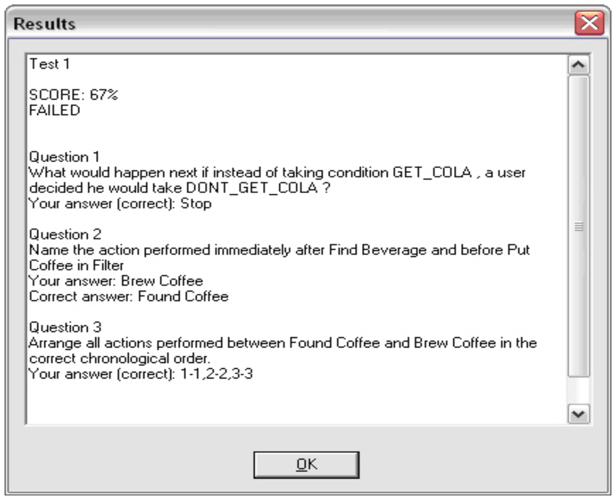
The risk profile for the drink-coffee activity diagram shown in Figure 1 (labeled Activity B) is shown in Figure 10. Figure 10 shows that as the number of questions in an assessment increases, so does the probability of catching each type of misunderstanding. However, based on the current set of rules, some misunderstandings are easier to catch than others. For example, as Figure 10 shows, the reversed-edge misunderstanding, where an engineer believes that an edge is pointing in the wrong direction, is fairly easy to catch because the number of questions in an assessment is increased beyond five. On the other hand, misunderstandings such as an additional reversed edge, where an engineer believes that an additional reversed edge exists between two activities, are difficult to catch; the probability of catching such misunderstandings for this specific activity diagram is less than 30 percent even with an assessment containing twenty questions. In addition, as the number of questions is doubled from ten to twenty, the probability of catching this misunderstanding only goes up by about 10 percent.


Page 1 of 5

What would happen next if instead of taking condition GET_COLA , a user decided he would take DONT_GET_COLA ?

[✓ Stop]
Drink
Turn On Machine

Figure 8. A comprehension-level sample question for the drink-coffee activity diagram



```
Test 1

SCORE: 67%
FAILED


Question 1
What would happen next if instead of taking condition GET_COLA , a user
decided he would take DONT_GET_COLA ?
Your answer (correct): Stop

Question 2
Name the action performed immediately after Find Beverage and before Put
Coffee in Filter
Your answer: Brew Coffee
Correct answer: Found Coffee

Question 3
Arrange all actions performed between Found Coffee and Brew Coffee in the
correct chronological order.
Your answer (correct): 1-1,2-2,3-3
```

Figure 9. Results of an assessment from the drink-coffee activity diagram

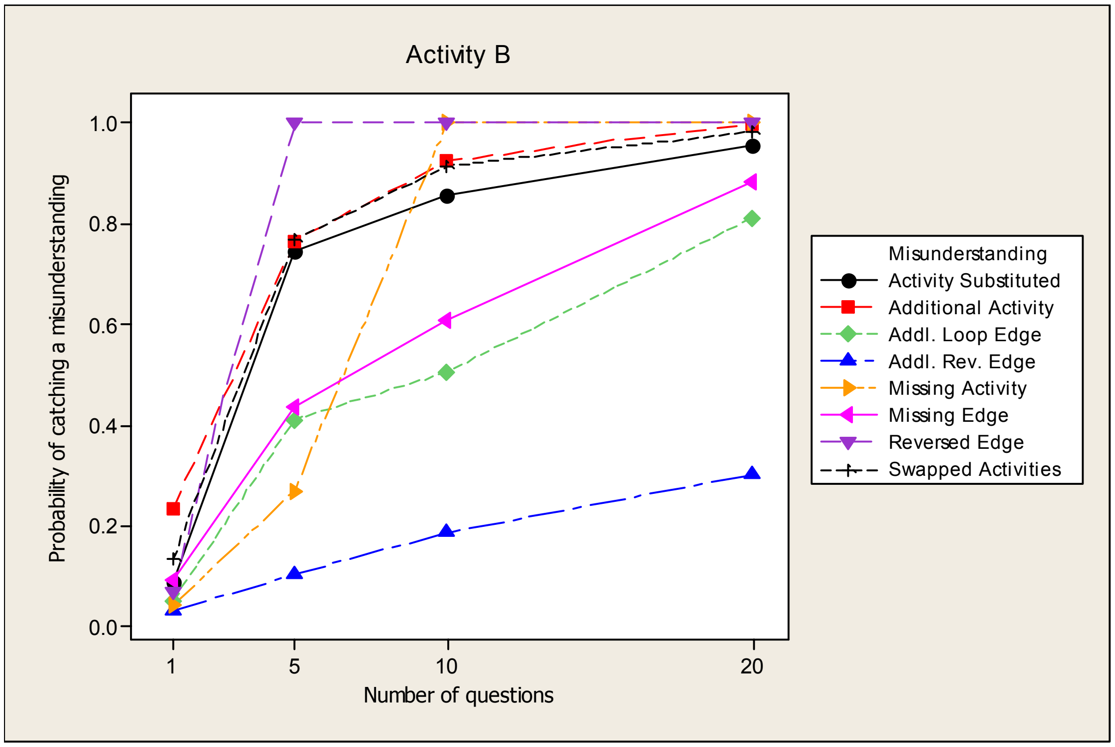


Figure 10. Risk profile for the drink-coffee activity diagram

The risk profile for the activity diagram shown in Figure 10 also provides additional useful information. For example, if an engineer generates an assessment with ten questions, the risk profile shows that there is about a 50 percent probability ($p = 0.5044$ with a 95 percent confidence interval of +/−0.0224) that a misunderstanding like the additional loop edge will be missed by the assessment.

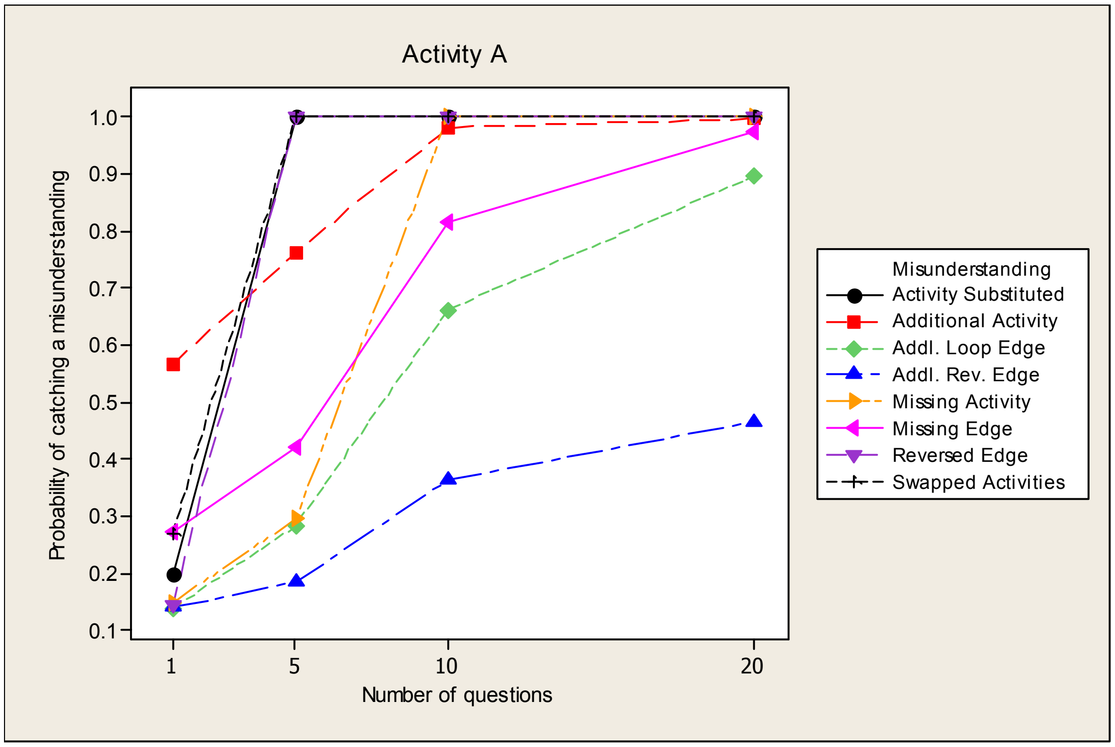


Figure 11. Risk profile for an activity diagram with eight activities

Figure 11 shows the risk profile for another activity diagram (Activity Diagram A) that is much smaller (only eight activities) and simpler than the Activity Diagram B. Figure 12 shows that for Activity Diagram A, the additional reversed-edge misunderstanding is also the most difficult (lowest probability) to catch. However, the probability of catching this misunderstanding with a ten-question assessment is much higher (p = 0.36 with a 95 percent confidence interval of +/−0.033) than the probability for Activity Diagram B (p = 0.18 with a 95 percent confidence interval of +/−0.017).

Each activity diagram has a unique risk profile that is not necessarily tied to its size. For example, Figure 12 shows the risk profile for an activity diagram (Activity Diagram C) with 24 activities. The probability of catching a reversed-edge misunderstanding for this activity diagram is actually higher (p = 0.28 with a 95 percent confidence interval of +/−0.18) than a smaller activity diagram (Activity Diagram B, for example).

In addition to the results shown above, the system presented in this paper has been successfully tested on one hundred arbitrary activity diagrams collected from published sources (mean number of activities/activity diagram = 15.56, SD = 8.22; mean number of edges/activity diagram = 13.65, SD = 8.26). Risk profiles were also generated for these diagrams by varying the size of the assessments from 1 to 20 questions in increments of five questions. Thirty Pentium IV machines were used as processing clients. As expected, each activity diagram resulted in a unique risk profile.

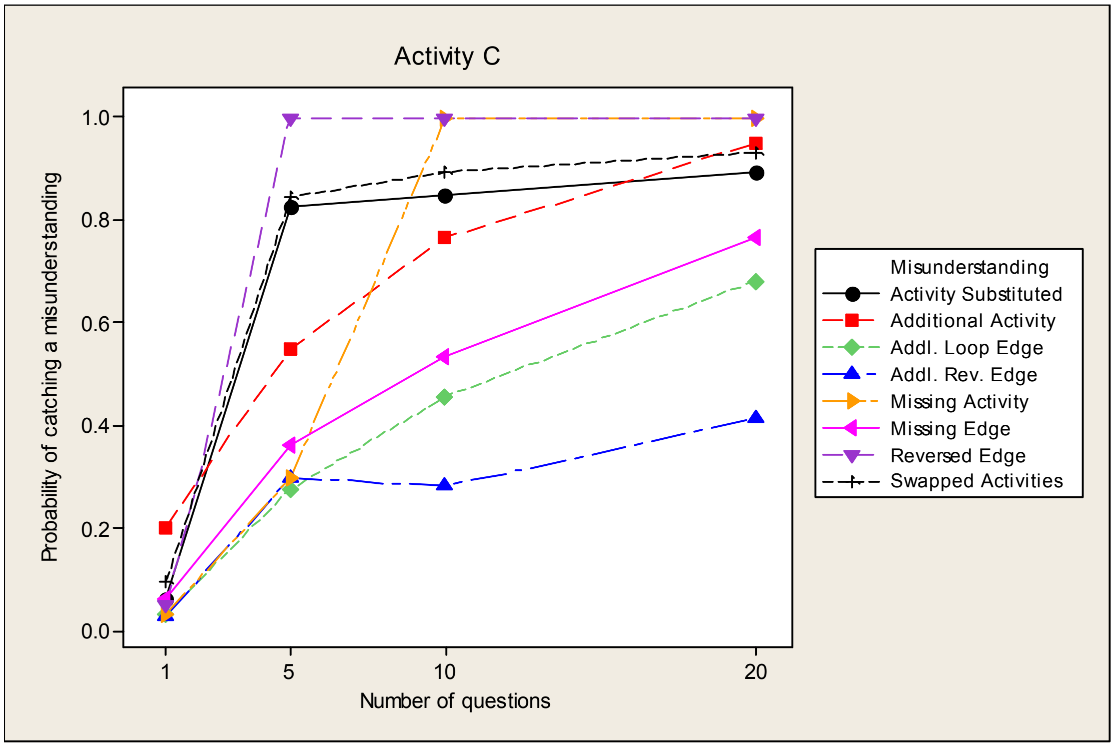


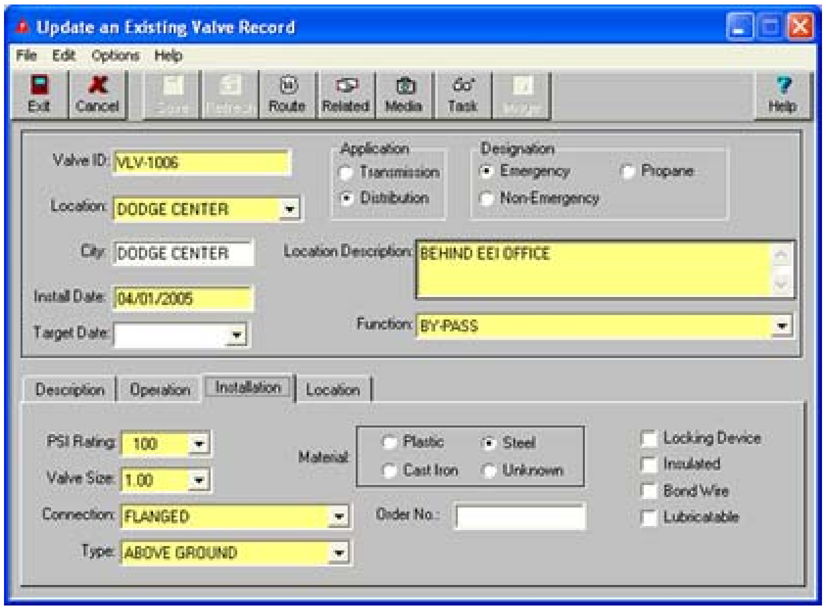
Figure 12. Risk profile for an activity diagram with 24 activities

The methodology and the tools presented in this paper were also applied to a portion of a mission-critical software system called the Gas Compliance System (or GCS Energy) (GCS, 2008). GCS Energy is an integrated suite of software modules designed to assist a natural gas utility with compliance tracking and auditing. In the United States, the Department of Transportation's Office of Pipeline Safety (OPS) mandates periodic inspection, testing, and maintenance tasks for gas pipeline systems through *DOT Part 192 — Transportation of Natural or Other Gas by Pipeline: Minimum Federal Safety Standards*. This federal standard requires that pipeline operators perform compliance tasks at thousands of sites, covering hundreds of miles. Upon completion, these tasks must be recorded, stored, and made available for review in the event of an OPS audit. Failing an audit can result in the leverage of large fines against a natural gas utility.

Based on principles found in the American Gas Association's GPTC Guide, GCS Energy's system manages compliance tracking for various facilities within a natural gas utility company. The various subsystem modules that track compliance activities include atmospheric corrosion, corrosion tracking, leak survey, leak tracking, pressure control stations, exposed pipe exam, pipeline patrol, and valve tracking.

One portion of the valve tracking system manages inspections. Pipeline operators must inspect every installed valve within a periodic time frame as determined by the *DOT Part 192 r*egulations. Various valve parameters determine not only the inspection frequency but also the inspection questions and whether the questions are mandatory or optional. Figure 13 displays some of these parameters in the valve detail window.

The corresponding activity diagram that models the valve inspection functionality for GCS Energy is shown in Figure 14.



Valve ID: VLV-1006

Location: DODGE CENTER

City: DODGE CENTER

Install Date: 04/01/2005

Target Date:

Application: Transmission [✓ Distribution]

Designation: [✓ Emergency] Non-Emergency Propane

Location Description: BEHIND EEI OFFICE

Function: BY-PASS

PSI Rating: 100

Valve Size: 1.00

Connection: FLANGED

Type: ABOVE GROUND

Material: Plastic [✓ Steel] Cast Iron Unknown

Order No.:

Locking Device

Insulated

Bond Wire

Lubricatable

Figure 13. Sample screen for entering valve parameters in GCS

Some of the questions automatically generated by the system described here for the activity diagram shown in Figure 15 are discussed below.

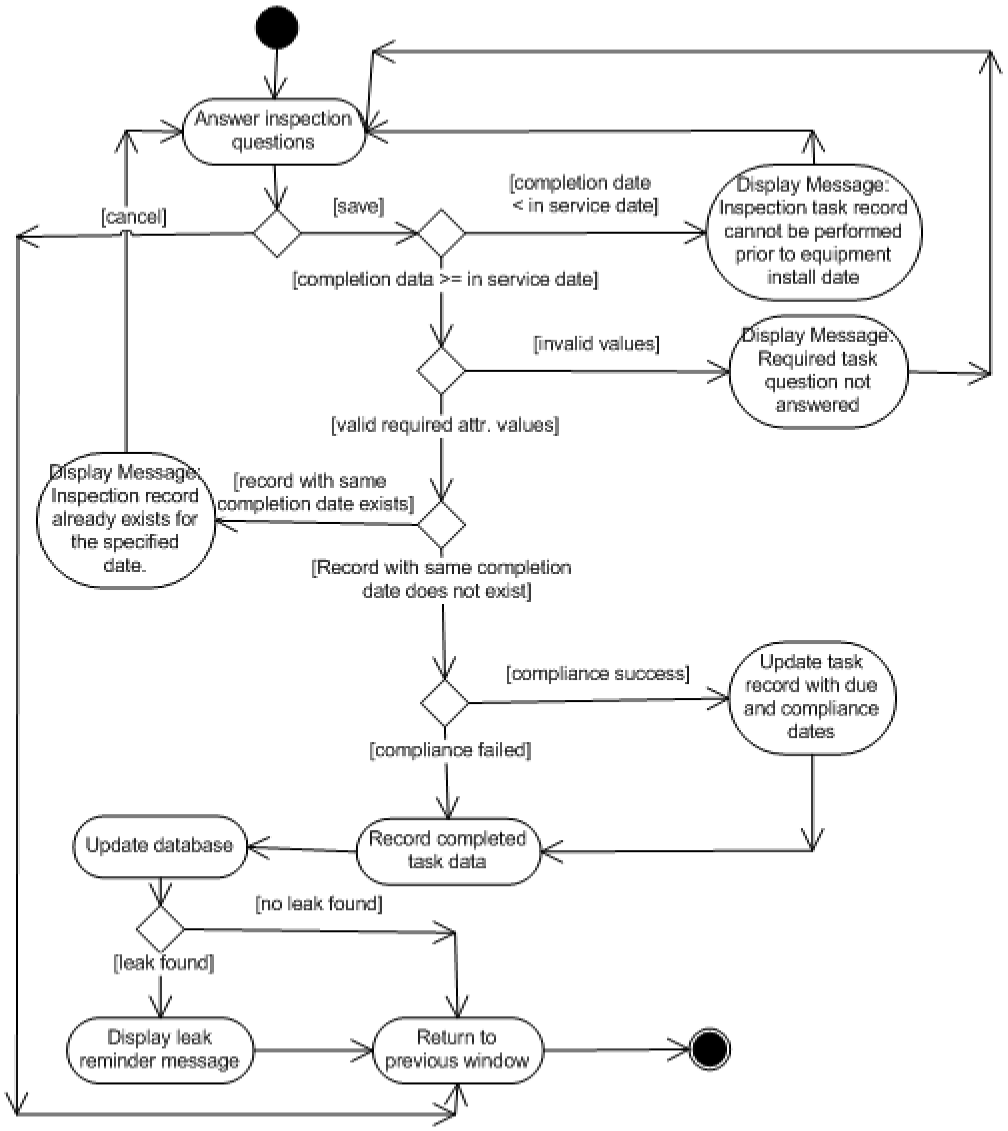
```
Q1: Starting from the initial state, can an operator do "Return to
previous window" without taking the decision 'valid required attr.
values'?
```

The correct answer to this question is "yes." The software engineer needs to know that an operator can choose to cancel the inspection task and return to the previous window. This operation skips any validation checks because the user is discarding the inspection task. An incorrect answer to this question exposes a serious lack of understanding on the part of the software engineer who does not understand that a user is able to cancel the inspection task and go back to the previous window before the required attribute values are validated.

```
Q2: What necessary action must be performed in order to do "Update
database"?

A) Record completed task data
B) Display Message: Required task question not answered
C) Return to previous window
```

The correct answer to this question is A. Again, the software engineer needs to know that C cannot occur before A and that receiving the error message in B means that the database cannot be updated. If a software engineer fails to answer this question, he does not understand that all inspection questions are answered before the database system is updated. In this context, a software engineer who selects C as the possible answer shows a complete lack of understanding of the system because returning to the previous window can only be done either if the inspection is cancelled or if all the requisite inspection tasks have been completed.



Answer inspection questions
[cancel]
[save]
[completion date < in service date]
Display Message: Inspection task record cannot be performed prior to equipment install date
[completion data >= in service date]
[invalid values]
Display Message: Required task question not answered
[valid required attr. values]
Display Message: Inspection record already exists for the specified date.
[record with same completion date exists]
[Record with same completion date does not exist]
[compliance success]
Update task record with due and compliance dates
[compliance failed]
Update database
Record completed task data
[no leak found]
[leak found]
Display leak reminder message
Return to previous window

Figure 14. An activity diagram for the valve inspection functionality in GCS

```
Q3: After the condition "completion date < in_service date", which of
the following are possible immediate successors?

A) Display Message: Inspection task record cannot be performed prior to
equipment install date
B) Update task record with new due and compliance dates
C) Update database
```

The correct answer to this question is A; the software engineer needs to know that A is the result if the condition test fails. In addition, the software engineer also needs to know that additional questions must be answered before the new due and compliance dates can be calculated or the database updated. A software engineer with the requisite knowledge of the design also knows that "completion date < in_service date" occurs before these other decision nodes. Therefore, A is the only correct answer.

The three questions discussed above show that questions generated by the assessment system probe for a deep understanding of a software engineer's knowledge of the software product design, and a failure of such understanding can lead to potentially catastrophic consequences.

Discussion and limitations

The prototype system constructed shows that it is possible to generate useful assessments automatically from activity diagrams. In addition, a unique risk profile is provided for each activity diagram.

As is evident, the assessments at the lower levels of Bloom's taxonomy are fairly context free and can be derived easily from the syntactic structure of activity diagrams. However, since activity diagrams are semi-formal in nature, a formal model should lead to a more robust set of questions. As one progresses to higher levels of Bloom's taxonomy, the task dimension also becomes more important in one's ability to generate the right rules. A model of the task can help in the generation of such questions.

The mutation operators used in this research are based on the HAZOP model. Additional operators can be derived from bug checklists (Thelin, Runeson, & Wohlin, 2003). Currently, the risk profile assumes a uniform distribution while generating the assessment questions. However, once a correspondence between the types of misunderstandings and their frequency in a software engineering environment is established, the assessment generation process can incorporate this information by increasing the proportion of questions that target frequently occurring misunderstandings in a particular environment.

Finally, the results presented in this paper are being extended to include the other twelve models of UML.

Conclusion

As designed artifacts like computer software become more complex and life cycles become shorter, the assessment techniques of the type presented here will become a necessary part of any design cycle. This paper describes a distributed system that automatically generates standards-based, tool-independent, and just-in-time online assessments from arbitrary activity diagrams. A key feature of this system is that, in addition to automatically generating an online assessment, the system provides a unique a priori risk profile for each activity diagram.

The results presented in this paper are limited to one specific model (i.e., the activity diagram) within one methodology (i.e., UML) in the context of one engineering discipline (i.e., software engineering). In addition, most assessment heuristics are at lower levels of Bloom's hierarchy. However, the results presented here have a potential application in any branch of engineering that constructs formal or symbolic by-products or artifacts as a natural design activity. For example, CAD/CAM systems for mechanical and civil engineering design are obvious candidates. Another significant area for application of this approach is process design. Increasingly, businesses' processes are not only being formalized but are in continuous flux as businesses continuously respond to changes in their environment. This flux creates a similar problem where the changes in a business's process are to be conveyed to all individuals that play a role or come in contact with the process. Again, rather than showing an individual a "picture" of a changed business process, an approach similar to the one presented in this paper can actually generate assessments to ensure that process changes have actually been understood.

Using mutation-analysis to generate an a priori estimate of risks associated with a particular assessment is also a promising side benefit of this approach. However, the effectiveness of this method relies on the availability of information on the types of misunderstandings that typically occur in specific contexts. The approach has the potential to be generalized to other disciplines as well. For example, one can imagine a common set of misunderstandings of a construction blueprint in the context of civil engineering: a missing door, the wrong type of HVAC pipe, etcetera. Once a taxonomy of such misunderstandings is established in a field, it can be used to create mutations of an existing design artifact (such as a blueprint) to judge the effectiveness of automatically generated assessments.

In hindsight, the difficulty of constructing heuristics to generate Bloom's higher level assessments is not surprising; it is difficult enough to construct such assessments by hand. However, in this research it was possible to arrive at some rudimentary heuristics at Bloom's higher levels, such as synthesis. Constructing high-level heuristics presents an interesting challenge for next phase of this research. Current attempts at constructing such higher-level heuristics suggest that ultimately domain-specific task models may be required to construct heuristics at higher level of Bloom's taxonomy. For example, a sophisticated heuristic at the synthesis level will have to generate synthesis tasks for a software engineer and automatically mark the engineer's performance on such tasks. The tasks need to go beyond the current synthesis heuristics that require an engineer to work through the consequences of various types of changes to a design artifact. Asking a software engineer to propose a novel solution in the context of an existing design artifact and automatically generating assessments to judge the feasibility of such solutions are non-trivial tasks that will likely require a sophisticated task and domain model.

In summary, this paper represents a first set of experiments in automatically generating just-in-time assessments in the limited domain of software engineering. This approach can be applied to a large number of engineering and business contexts. The ultimate success of such an approach, however, will depend on one's ability to generate specific heuristics for each domain.

Acknowledgements

This research was supported in part by a faculty research grant from the IBM Corporation.

References

Anderson, P., Reps, T., & Teittelbaum, T. (1989). Design and implementation of a fine-grained software inspection tool. *IEEE Transactions on Software Engineering, 29*(8), 721–733.

Andrews, A., France, R., Ghosh, S., & Craig, G. (2003). Test adequacy criteria for UML design models. *Software Testing, Verification and Reliability, 13*(2), 95–127.

Apvrille, L., Courtiat, J., Lohr, C., & Saqui-Sannes, P. (2004). TURTLE: A real-time UML profile supported by a formal validation toolkit. *IEEE Transactions on Software Engineering, 30*(7), 473–487.

Axis (2008). *The Apache Axis Project*. Retrieved November 12, 2008, from http://ws.apache.org/axis/

Bigot, C., Valot, Y., Gallois, J., Gérard, S., Terrier, F., & Lugato, D. (2004). Validation and automatic test generation on UML models: the AGATHA approach. *International Journal on Software Tools for Technology Transfer, 5*(2), 124–139.

Bloom, B. S. (Ed.). (1956). *Taxonomy of educational objectives: The classification of educational goals: Handbook I, cognitive domain*. New York: Longmans, Green.

Boland, P., H. Singh, H, & Cukic, B. (2003). Comparing partition and random testing via majorization and Schur functions. *IEEE Transactions on Software Engineering, 29*(1), 88–94.

Brown, J. S., Collins, A., & Duguid, B. (1989). Situated learning and the culture of learning. *Education Researcher, 18*(1), 32–42.

Brykczynski, B. (1999). A survey of software inspection checklist. *ACM Software Engineering Notes, 24*(1). 82–89.

Chow, T. C. (1978). Testing design modeled by finite-state machines. *IEEE Transactions on Software Engineering, 4*(3), 178–186.

Clocksin, W. F., & Mellish, C. S. (1994). *Programming in Prolog* (4th ed). Springer-Verlag.

Confora, G., Cimitile, A., Carlini, U., & De Lucia, A. (1998). An extensible system for source code analysis. *IEEE Transactions on Software Engineering, 24*(9), 721–740.

DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer, *IEEE Computer, 11*(4), 34–41.

Eshuis, R., & Wieringa, R. (2004). Tool support for verifying UML activity diagrams. *IEEE Transactions on Software Engineering, 30*(7), 437–447.

GCS. (2008). Gas Compliance System. Retrieved November 12, 2008, from http://www.eei.com

Hendrix, D., Cross, J. H., & Maghsoodloo, S. (2002). The effectiveness of control structure diagrams in source code comprehension activities. *IEEE Transactions on Software Engineering, 28*(5), 463–477.

Howden, W. E. (1982). Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering, 8*(4), 371–379.

Hundausen, C., Douglas S., & Stasko, J. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing, 13*(3), 259–290.

IMS-QTI (2006). *IMS Question & Test Interoperability Specification*. Retrieved May 22, 2006, from http://www.imsglobal.org/question/index.html.

Kim, S., J., Clark, A., & McDermid, J. A. (1999). The rigorous generation of Java mutation using HAZOP. *In Proceedings of the 12th International Conference on Software and Systems Engineering and Their Applications (ICSSEA '99)*, 9–10.

Lanza, M., & Ducasse, S. (2003). Polymetric views: A lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering, 29* (9), 782–795.

Lave, J., & Chaiklin, S. (Eds.). (1993). *Understanding practice: Perspectives on activity and context*, Cambridge: University of Cambridge Press.

Lehman, J. A. (1989). An empirical comparison of textual and graphical data structure documentation for COBOL programs. *IEEE Transactions on Software Engineering, 14*(9), 1131–1135.

Luqi, L., Berzins, V., & Qiao, Y. (2004). Documentation driven development for complex real-time systems. *IEEE Transactions on Software Engineering, 30* (12), 936–952.

Miller, J., & Yin, Z. (2004). A cognitive-based mechanism for constructing software inspection teams. *IEEE Transactions on Software Engineering, 30*(11), 811–825.

MySQL (2008). *MySQL*, Retrieved November 12, 2008, from http://www.mysql.com/

Nebut, C., Fleurey, F., Le Traon, Y., & Jézéquel, J. (2006). Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering, 32*(3), 140–155.

OMG-UML (2003). *Unified modeling language specification*. (version 1.5). Retrieved May 21, 2006, from http://www.omg.org/technology/documents–/formal/uml.htm.

OMG-XMI (2006). *XML metadata interchange (XMI) specification, version 1.1* Retrieved May 21, 2006, from http://www.omg.org/cgi-bin/doc?formal/2000-11-02.

Simon, H. (1983). *Sciences of the artificial*. Cambridge, MA: MIT Press.

Soap (2008). *The SOAP Interface*/ Retrieved November 12, 2008, from http://www.w3.org/TR/soap/

Stasko, J, Dominique, J. B., Brown, M. H., & Price, B. A. (1988). *Software visualization*. Cambridge, MA: MIT Press.

SWI-Prolog (2008). *SWI Prolog*. Retrieved November 12, 2008, from http://www.swi-prolog.org

Thelin, T., Runeson, P., & Wohlin, C. (2003). An experimental comparison of usage-based and checklist-based reading. *IEEE Transactions on Software Engineering, 29*(8), 687–704.

Tomcat. (2008). The Apache Tomcat Project. Retrieved May 21, 2008, from http://jakarta.apache.org/tomcat

Tonella, P. (2003). Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering, 29*(6), 495–509.

Traore, I., & Aredo, D. B. (2004) Enhancing structured review with model-based verification, *IEEE Transactions on Software Engineering, 30*(11), 736–753.