1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

Usage Based Test Scenario Generation from Object-Z Formal Specification

Chun-Yu Chen, Richard Chapman, and Kai H. Chang
Department of Computer Science and Engineering
Auburn University, Auburn, AL 36849, USA
{*chunchen,chapman,kchang*}*@eng.auburn.edu*

Abstract

Usage-based testing is a technique for more effective generation of test cases based on the relative frequency of usage of different test scenarios. More frequently executed scenarios give rise to greater numbers of test cases than do rarely executed scenarios. Thus, the distribution of test cases among the scenarios mirrors the actual usage that the system is expected to receive. We use a formal specification as a basis for usage-based testing of programs that use Object-Oriented Technology (OOT). OOT is widely considered the best methodology for programs that are large. OOT has gained wide acceptance in the software engineering community, however better test methods are still needed. This paper introduces an approach to usage-based test scenario generation based on an Object-Z formal specification. The paper presents the methodology and applies it to an example.

1. Introduction

Testing is the most critical and time/cost consuming process in software development. An effective software testing methodology lets the user use the software without fear, and an efficient testing methodology saves a lot of development time. Although testing is not perfect, a sound testing methodology can greatly reduce the risk of postponing the delivery and reduce the cost for after-delivery maintenance. Nowadays, software development has gradually changed from traditional waterfall lifecycle model to incremental development model. Testing is no longer the last step in a software development life cycle. Instead, testing should begin as early as design starts.

Within the last ten years, because of its reusability, scalability, and the possibility of a direct mapping between the problem and solution domains, object-oriented languages, such as C++, Java and Ada95, have become main-stream computer languages, and object-oriented design and analysis have dominated software development. The object-oriented technology not only gives many advantages for developing new software systems but also brings some problems. Testing is one of the problems that the object-oriented technology brings to us due to the differences between it and the procedure-oriented technology. The traditional testing methods do not automatically adapt to use with object-oriented technology.

This paper is an extension of the work introduced in [8]. That work introduced a framework for object-oriented program testing by using a formal specification to conduct usage-based testing. This paper emphasizes usage-based test scenario generation for object-oriented programs from an Object-Z formal specification [2]. The rest of this paper is organized in the following manner: section two gives a brief discussion of the background information; section three discusses the detailed test scenario generation approach; section four gives an example of applying the approach discussed in section three; and section five concludes this paper.

2. Background information

Software testing can be classified into three categories. The first is code-based testing. Code-based testing tests against a piece of source code. The most commonly used code-based testing technique is probably McCabe's "*Basis-Path Testing*" [10]. The second category is specification-based testing. The idea of specification-based testing comes from the question "*Does the built software product behave exactly as it is expected?*" Formal specification languages can provide correct, consistent, and unambiguous specifications. The technique not only can be used for software product development, but it can also be used for guiding software testing. The third category is usage-based testing. Musa pointed out in the first sentence of [11]: "*A Software-based product's reliability depends on just how a customer will use it. Making a good reliability estimate depends on testing the product as if it were in the field.*"

The objective of our project is to build a framework for object-oriented program testing. It provides a common basis for both design and testing. In this

framework, test can be done to determine if the implementation conforms to the specification. Also, the design and testing teams can both start their work at the same time. A formal specification is included in the framework so that specification-based testing can be performed, while the usage profile is used to conduct the usage-based testing.

2.1. Object-oriented technology

Object-oriented technology also has an overall impact on how should a software product be developed. Not only should the software product be implemented by an object-oriented programming language, but the product should be developed with the help of object-oriented analysis and design also. Edward Berard noted in [1], "*The benefits of object-oriented technology are enhanced if it is addressed early-on and throughout the software engineering process,*" and, "*An overall object-oriented approach appears to yield better results than when object-oriented approaches are mixed with other approaches.*"

Until the beginning of 1990s, the majority of object-oriented research was still focused on the front-end of the software development life cycle. People started to realize in the beginning of 1990s that although object-oriented technology brought many new, powerful features (such as encapsulation, inheritance, polymorphism, and dynamic binding), it also introduced new problems in software testing and maintenance caused by these features [7, 14, 5].

2.2. Operational profile

Operational (or usage) profiles have become an essential component of software reliability engineering. Used in software development, it can increase the productivity and reliability, and can also speed up development as well. Studies have shown the advantages and benefits of applying operational profile [9]. Musa [11] developed operational profiles by breaking system use down into up to five levels. He defined the term *profile*: "*A profile is simply a set of disjoint (only one can occur at a time) alternatives with the probability that each will occur.*" He used a *call tree* to represent the usage profile. A node in a *call tree* represents the current use of the system. A branch that diverges from a node shows the next possible use of the system. Each node is also given a key value which shows the probability the node will be visited from its parent node. A usage scenario is thus a traversal path from the root of the *call tree* to a leaf node. The probability of this scenario is the product of the key values of all nodes in the path. The probability of a usage scenario is then used to determine the number of test cases that should be generated to test this scenario.

2.3. Object-Z formal specification

The Z formal specification language [12] is based on first-order logic and set theory. Z was developed by Oxford University's Programming Research Group in the late seventies and early eighties, and it has become a well-accepted formal specification language, accepted as an ISO standard. Object-Z [3] is a specification language based on Z with extensions to support an object-oriented specification style. Object-Z is not the only approach providing Z with an object-oriented structuring mechanism. Other approaches such as Hall's style , ZERO, MooZ, OOZE, Schuman & Pitt, Z++, and ZEST can be found in [13]. Object-Z was chosen for this research because (1)Object-Z is fully object-oriented and (2)Object-Z appears to be the most mature object-oriented specification language [6]. A detailed description of Object-Z syntax and semantic can be found in [3].

3. Test Scenario Generation

A stepwise approach to test scenario generation from an Object-Z specification and a usage profile is explained in this section. A successful test oracle should contain the function invocation sequence and the input/output of each function for each test case. It is also important to have a certain degree of automation in conducting software testing for efficiency. Based on these two factors, our approach finds all possible function invocation sequences (test scenarios), with constraints, from an Object-Z specification as test scenarios. A high degree of automatic test scenario generation can be achieved by parsing the Object-Z specification. The detailed step by step procedure follows.

Step 1: Object-Z Specification Generation. Although it is the Object-Z specification that we use for test scenario generation, the generation of an Object-Z specification is not the focus of this paper. We assume the Object-Z specification is readily available for the process.

Step 2: Top-Level System Operation Diagram Derivation. A top-level system operation diagram describing the top-level behavior can be obtained from object-oriented analysis and the Object-Z specification. The necessity of this diagram is due to the fact that an Object-Z specification does not always capture the top-level behavior of the system.

In the Simple Bank Account System (*SBAS*) example of next section, although the specification shows that there are six visible operations in the system, it does not give any relationships between these six operations nor does it tell how will the system be started. This problem may be solved by changing the six operations from visible to invisible and adding the following visible operation

$BAS \mathrel{\widehat=} Deposit \ [] \ Withdrawal \ [] \ Inquiry$
$[] \ CalculateInterest \ [] \ AddInterest \ [] \ ResetWithdrawal.$

This specification shows that these six operations can be invoked independently. However, adding this operation does not show the fact that these six operands will be invoked continually until the system is shut down.

The top-level operation diagram should only show the behavior that is not described by the Object-Z specification. It should be kept as simple as possible. Except for the added pseudo-nodes like start and end nodes, each node of the top-level diagram should have a corresponding operation defined in the specification, i.e. each node is associated with an operation defined in the specification. A tree-structure-like call tree such as that mentioned in section 2.2 or a directed graph can be used as the top-level system operation diagram.

Step 3: Partial Invocation Sequence Generation. Object-Z provides two styles of class operation schemata: box and horizontal. The box style operation schema is used to define atomic class operations, which are can be used to examine and/or change the state of the object. The horizontal style operation schema is used to define new operations by combining more than one operation of the same class and/or by invoking operations from other objects. The horizontal style operation schema is used for hiding and renaming purposes also [4]. Because the box style operation schema is used for defining atomic class operations only, it is used as a server class operation. It can not request services from other operations of the same class nor from other objects. An invocation sequence is a list of operations to be invoked in an operation. An operation may have zero or more invocation sequences. The zero-length (denoted *nil*) invocation sequence means the operation does not request service from other operations.

A discussion of the kinds of operations that the horizontal style operation schema provides, as well as the invocation order of the combined operations, is needed before we discuss the generation of the partial invocation sequence for each operation schema. Object-Z provides four basic construction operators for defining the horizontal style operations. They are conjunction operator '$\wedge$', parallel operator '$\|$', sequential operator '$\fatsemi$', and choice operator ' [] '.

The conjunction operator '$\wedge$' conjoins two operands, and the invocation sequence of the conjoined operands is insignificant. For example, the operation

$Inc\_x\_y \mathrel{\widehat=} Inc\_x \wedge Inc\_y$

does not indicate whether Inc_x will be invoked first or not. Therefore, in the implementation, either Inc_x could be invoked first or Inc_y could be invoked first.

The parallel operator '$\|$' conjoins two operands to achieve inter-object communication, provided that (1) these two operands have a local variable with the same basename, (2) one of them is an input (denoted by '?'), and (3) the other an output (denoted by '!'). The parallel operator supports communication in either direction (or both). In an invocation sequence the operand with the output variable must be invoked first so that the output value can be produced and passed to the operand with input variable. The operation

$Deposit \mathrel{\widehat=} SelectAccount \bullet (GetAmount \parallel account!.Deposit)$

defined in the *SBAS* in the next section will first invoke *SelectAccount*. Then, *GetAmount* is invoked to get *amt*! and pass it as an input variable to *account*!.*Deposit*. Therefore the invocation sequence would be *SelectAccount*, *GetAmount* then *account*!.*Deposit*. The '$\bullet$' in the above operation is the scope operator in Object-Z. The scope operation has the form $ScopeOp \mathrel{\widehat=} OpExp \bullet OpExp$. In the scope operation, the signature scope of the left operand is extended to the right operand and therefore the left operand will be invoked first.

The sequence operator '$\fatsemi$' provides another way for communication in Object-Z. It behaves like the parallel operator except that the communication is from the left to the right. An alternative definition for the above *Deposit* operation is:

$Deposit \mathrel{\widehat=} SelectAccount \bullet (GetAmount \fatsemi account!.Deposit).$

The invocation sequence for *Deposit* is obviously *SelectAccount*, *GetAmount* then *account*!.*Deposit*.

The last operator is the choice operator ' [] '. This operator indicates non-deterministic choice of one operand from the definition if more than one operand's preconditions are satisfied. The operation may fail if no precondition is satisfied. The operation

$PushOne \mathrel{\widehat=} Push_1 \ [] \ Push_2$

may have three different outcomes: $Push_1$, $Push_2$, or nothing.

Operators can be composed to define new operations. The precedence of these four operators from high to low is: ('$\wedge$', '$\|$'), '$\fatsemi$', ' [] '. The conjunction operator '$\wedge$', parallel operator '$\|$', and sequential operator '$\fatsemi$' are left associative. The association of the choice operator ' [] ' is not important in invocation sequence generation.

To generate partial invocation sequence from Object-Z automatically, some restrictions must be imposed.

- Each operation must have a one-to-one mapping between its specification and implementation.
- Bidirectional communication in the parallel operator is prohibited.

Based on the discussion and restrictions above, a partial invocation sequence for each operation is generated in the following manner: assume *Op* is the newly defined operation and Op_1, Op_2, and Op_3 are three operands used in defining the new operation *Op*. Op_1 has an output variable $var_1!$. Op_2 has an input variable $var_1?$ and an output variable $var_2!$. Op_3 has an input variable $var_2?$. In the following description, the newly

defined operation is shown on the left, and all its possible invocation sequences are given on the right. The invocation sequence is enclosed by < >.

1. conjunction Operator ($\wedge$).
   $Op \mathrel{\widehat{=}} Op_1 \wedge Op_2$ — $< Op_1, Op_2 >$, $< Op_2, Op_1 >$
   $Op \mathrel{\widehat{=}} Op_1 \wedge Op_2 \wedge Op_3$ — $< Op_1, Op_2, Op_3 >$, $< Op_2, Op_1, Op_3 >$, $< Op_3, Op_1, Op_2 >$, $< Op_3, Op_2, Op_1 >$
2. Parallel Operator ($\|$).
   $Op \mathrel{\widehat{=}} Op_1 \| Op_2$ — $< Op_1, Op_2 >$
   $Op \mathrel{\widehat{=}} Op_1 \| Op_2 \| Op_3$ — $< Op_1, Op_2, Op_3 >$
3. Sequential Operator ($\fatsemi$).
   $Op \mathrel{\widehat{=}} Op_1 \fatsemi Op_2$ — $< Op_1, Op_2 >$
   $Op \mathrel{\widehat{=}} Op_1 \fatsemi Op_2 \fatsemi Op_3$ — $< Op_1, Op_2, Op_3 >$
4. Choice Operator ([]).
   $Op \mathrel{\widehat{=}} Op_1 \,[]\, Op_2$ — $< nil >$, $< Op_1 >$, $< Op_2 >$
5. Composed Operators. Precedence and associative are used to determine the sequence.
   $Op \mathrel{\widehat{=}} Op_1 \,[]\, Op_2 \wedge Op_3$ — $< nil >$, $< Op_1 >$, $< Op_2, Op_3 >$, $< Op_3, Op_2 >$
6. Polymorphic Object Declaration. Assume A, B, and C are three defined classes. Class B is derived from class A, and class C is derived from class B. The polymorphic object $a! : \downarrow A$ is declared in the operation *SelectOne*. Op_a is an operation of class A, and is inherited by classes B and C.
   $Op \mathrel{\widehat{=}} SelectOne \bullet a!.Op_a$ — $< SelectOne, a!(A).Op_a >$, $< SelectOne, a!(B).Op_a >$, $< SelectOne, a!(C).Op_a >$

Step 4: Usage Profile Generation. Usage profile generation consists of two parts: hierarchical invocation diagram (HID) generation and usage weight assignment to the HID. The HID is generated in the following 4 steps:

1. Use the result from **Step 2** above as the top-level HID. The top-level HID can be either a tree structure diagram or a directed graph. Except for the pseudo nodes, every other node corresponds to an operation defined in Object-Z specification and each operation has a list of invocation sequences obtained in **Step 3**.
2. Make each operation defined in the Object-Z specification an Operation Invocation Diagram (OID) according to its invocation sequences. Each operation in the invocation sequence becomes a node in the OID. The OID is enclosed by a pseudo start and a pseudo end nodes (denoted Pl_i). Each path from the pseudo start node to the pseudo end node in an OID represents an invocation sequence of the operation. For example, Op_1 has two possible invocation sequences, $< Op_a, Op_b >$ and $< Op_b, Op_a >$, generated in **Step 3** above and, therefore, the OID of Op_1 has two paths Pl_1-Op_a-Op_b-Pl_2 and Pl_1-Op_b-Op_a-Pl_2 as shown in Figure 1.
3. Expand every node in the HID with its OID, except pseudo nodes, and link the OID with its corresponding node by dash directed edges. In Figure 1, Op_1 is expanded with its corresponding OID and they are linked by a dash directed edge from Op_1 to Pl_1 and the other dash directed edge from Pl_2 to Op_1.
4. Repeat 3 for every newly added non-pseudo node until every newly added node has no corresponding OID. This will build an OID hierarchy. Figure 1 is an example HID.

The second part of the usage profile generation is the usage weight assignment to the resulting HID. This requires pre-collected usage information. This best usage information is from the actual usage of an early version of the system or from existing similar systems. Otherwise, simulated usage patterns according to system usage projections should be applied. Equal weight assignment is not recommended since it negates the advantage of the usage-based testing.

In assigning the usage weights we start from the top-level of the HID. When a tree structure is used, a usage weight is assigned to each branch. When a directed graph is used, the usage weight of each edge in the graph may vary if the diagram contains loops. A loop introduces the potential that system behavior may be affected by previous operations and that it may iterate forever. Two constraints can be set to handle this problem. First, a maximum loop count (n) can be set for each loop. Every outgoing edge in the loop should be given an n-tuple $(a_1, a_2, ..., a_n)$ usage weight for a_k represents the usage weight of the edge in the loop's kth iteration and all a_ks should sum up to 1. For example, node Op_1 in Figure 1 has two outgoing edges and each edge has a 3-tuple usage weight. The maximum loop count is set to 3 and the usage weight of the two outgoing edges of Op_1 is sum up to 1 in each iteration (0.7 + 0.3, 0.1 + 0.9, and 0.05 + 0.95). Second, a minimum usage weight for each possible path can be set such that any test scenario with usage weight below the minimum usage weight can be filtered out. These two constraints can be imposed at the same time.

After the usage weights in top-level HID are assigned, the rest in HID that need to have usage weight assigned are all OIDs. Since an OID contains no loop, the weight of each path in an OID can be easily assigned.

Step 5: Test Sequence Generation. After the completion of the usage profile, test scenarios can be generated. A test scenario includes two parts: a hierarchical invocation sequence and its usage weight. A hierarchical invocation sequence is any possible path of the HID and its usage weight is the multiplication of all usage weights of the edges along the path. Two possible test scenarios of Figure 1 would be

$(< Op_1 < Op_a, Op_b >, Op_2 < Op_e >, Op_3 < Op_h, Op_i < Op_j >>>, 0.07840)$ and

$(< Op_1 < Op_b, Op_a >, Op_2 < Op_f >, Op_1 < Op_a, Op_b >, Op_3 < Op_h, Op_i < Op_j >>>, 0.03175)$.

When applying the minimum usage weight constraint, the scenario

$(< Op_1 < Op_a, Op_b >, Op_2 < Op_g >, Op_1 < Op_a, Op_b >, Op_2 < Op_g >, Op_1 < Op_b, Op_a >, Op_3 < Op_h, Op_i < Op_j >>>, 0.00002)$

would be filtered out if the minimum usage weight is set to 0.00005.

One advantage of using a hierarchical representation of the test scenario is that some links can be turned off if the links are connected to a well-tested invocation sequence. This is particularly useful for integration testing, for it can greatly reduce the number of test scenarios.

4. Bank Account System Example

A simplified bank account system (*SBAS*) is used as an example to demonstrate the approach presented in the previous section. Three types of account are provided in *SBAS*: *CheckingAccount*, *SavingAccount*, and *SavingLimitedAccount*. *CheckingAccount* is the basic account and provides no interest, *SavingAccount* yields interest, and *SavingLimitedAccount* yields a better interest and is limited to three withdrawals in a week. Stepwise explanation is given below to show how the test scenarios can be generated according to the approach described in the previous section.

Step 1: *SBAS* Object-Z Specification. Figure 2 gives the Object-Z specification of the *SBAS*. *CheckingAccount* has a single attribute *balance* and provides three most basic account operations: *Deposit*, *Withdraw*, and *Inquiry*. *SavingAccount* is derived from *CheckingAccount*; it inherits the *Balance* attribute and all three operations from type *CheckingAccount*. *SavingAccount* introduces a new attribute *accumulatedInterest* to keep the accumulated interest of the account, and two new operations: *CalculateInterest* and *AddInterest*. *SavingLimitedAccount* is derived from *SavingAccount*. At most three withdrawals are allowed for a *SavingLimitedAccount* in a week. It introduces a new attribute *withdrawalInAWeek*, which is used to keep track how many withdrawals have been made so far in a week. *SavingLimitedAccount* defines a new operation *ResetWithdrawals* which sets the new defined attribute *withdrawalInAWeek* to 0. The whole *SBAS* is made up of some *CheckingAccounts*, some *SavingAccounts*, and some *SavingLimitedAccounts*. The variable *accounts* is made up of all existing accounts. Six visible operations serve as the interface of *SBAS* and the external world. Operations *SelectAccount* and *SelectSavingAccount* use a polymorphic type variable: *account!*. *SelectAccount* can select an account of any type. *SelectSavingAccount* can select either a *SavingAccount* or a *SavingLimitedAccount*.

Step 2: *SBAS* Top-Level System Diagram. Figure 4 shows the top- level system operation diagram of *SBAS*. The initial link denotes the invocation of the system, and node Pn_1 denotes that the system is waiting for an operation selection. The other six operations nodes are the visible operations listed in the Object-Z specification. From system analysis, it is found that these six operations can be repeated and the system may exit after the completion of any of the six operations. (For explanation purposes the analysis has been largely simplified.)

Step 3: *SBAS* Partial invocation sequence generation. All operations of *CheckingAccount*, *SavingAccount*, *SavingLimitedAccount* have the $< nil >$ invocation sequence. They all use the box style operation schema. No partial invocation sequence should be generated for *INIT* operation in any class for it is invoked only during object instantiation and can not be exercised by normal operation invocation. The partial invocation sequences for every operations in every class are listed in Figure 3.

Step 4: *SBAS* usage profile. Figure 4 shows a partial *SBAS* usage profile. The maximum loop count is set to 4.

Step 5: *SBAS* test scenarios. Two possible test scenarios are listed below. The minimum usage weight is 0.0001.

1.$(< Withdraw < SelectAccount, GetAmount, account!(CheckingAccount).Withdraw >>, 0.15)$

2.$(< Deposit < SelectAccount, GetAmount, account!(SavingAccount).Deposit >, Inquiry < SelectAccount, account!(SavingAccount).Inquiry >>, 0.0063)$

5. Conclusion and future work

This paper has presented an approach to test scenario generation using Object-Z formal specifications for object-oriented testing. The approach has five steps: Object-Z specification generation, top-level system operation diagram derivation, partial invocation sequence generation, usage profile generation, and test sequence generation. Invocation sequences of inherited operations are listed explicitly and the polymorphic types are handled by given each possible type its own invocation sequence. The approach has the following advantages:

- Automatic scenario generation: Although fully automated test scenario generation does not seem

feasible, a high degree automation can be achieved by parsing the Object-Z specification with the Object-Z grammar rules.

- Reduced number of test scenarios during integration testing: By searching for class names, object names, and operation names contained in the newly integrated functions, scenarios affected by the newly integrated functions can be identified. Only these affected scenarios should be tested in each increment during integration testing. A number of test scenarios can thus be eliminated, which means that the number of test cases can be reduced.
- Conforms to modern software engineering concepts:
  1. Testing is independent from development, but both are based on the same foundation: the Object-Z specification.
  2. Testing preparation need not wait for implementation. Test design can start as soon as the Object-Z specification is ready.

Formal specification can help through out most steps in the framework for object oriented testing. Test scenario generation is the first step. A number of test scenarios will be generated during test scenario generation. Test cases are then generated for each scenario. For test case generation, one must know the input data and the ordering. The testing oracle expects the outcome of the test cases including the output and the state change. The Object-Z specification provides input/output data and the pre/post-conditions of each operation defined. This information is just what we need in test case generation and test oracle derivation. Our future work is to incorporate test scenarios with information extracted from the Object-Z specification for test case generation and test oracle derivation.

References

[1] E. Berard. *Essays on Object-Oriented Software Engineering.* Prentice Hall, New Jersey, 1993.

[2] D. Carrington, R. Duke, P. King, G. Rose, and G. Smith. "Object-Z: an object-oriented extension to Z". In *Proc. IFIP TC/WG 6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols*, pages 281–296, Vancouver, Canada, December 1990.

[3] R. Duke, P. King, G. Rose, and G. Smith. "The Object-Z specification language, Version 1". Technical Report 91-1, Software Verification Research Centre, Department of Computer Science, University of Queensland, May 1991.

[4] Wendy Johnston. "A Type Checker for Object-Z". Technical Report 96-24, Software Verification Research Centre, Department of Computer Science, University of Queensland, September 1996.

[5] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. "Change impact identification in object oriented software maintenance". In *Proc. International Conference on Software Maintenance*, pages 202–211, Victoria, British Columbia, Canada, September 1994.

[6] K. Lano and H. Haughton. *Object-Oriented specification case studies.* Prentice Hall, 1994.

[7] M. Lejter, S. Meyers, and S.P. Reiss. "Support for maintaining object-oriented programs". *IEEE Trans. on Soft. Eng.* , 18(12):1045–1052, December 1992.

[8] S. Liao, K. H. Chang, and C. Chen. "An Integrated Testing Framework for Object-Oriented Programs". *the Informatica Journal* , 21:135–145, 1997.

[9] M. Lyu. *Handbook of Software Reliability Engineering.* IEEE Computer Society Press, 1995.

[10] T. J. McCabe. "A software complexity measure". *IEEE Transactions on Software Engineering* , 2(6):308–320, December 1976.

[11] J. D. Musa. "Operational profiles in software reliability engineering". *IEEE Software*, pages 14–32, March 1993.

[12] J. M. Spivey. *Understanding Z* . Cambridge University Publishing, 1988.

[13] S. Stepney, R. Barden, and D. Cooper (Eds.). *Workshops in computing series: object-orientation in Z.* Springer-Verlag, 1992.

[14] N. Wilde and R. Huitt. "Maintenance support for object-oriented programs". *IEEE Trans. on Soft. Eng.* , 18(12):1038–1044, December 1992.

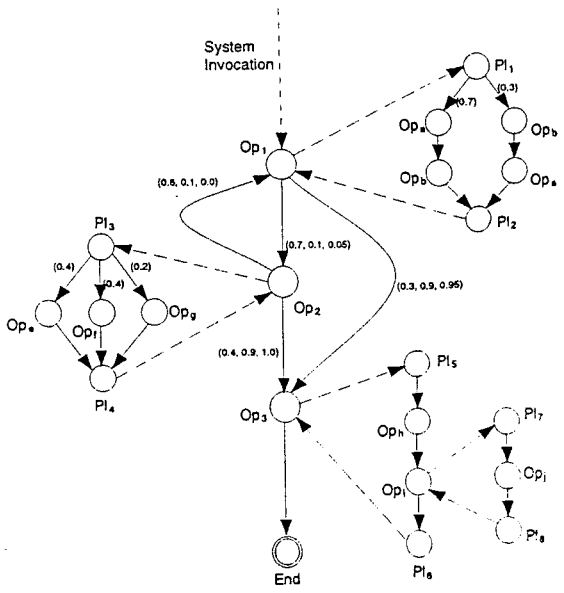

Figure 1. Hierarchical invocation Diagram

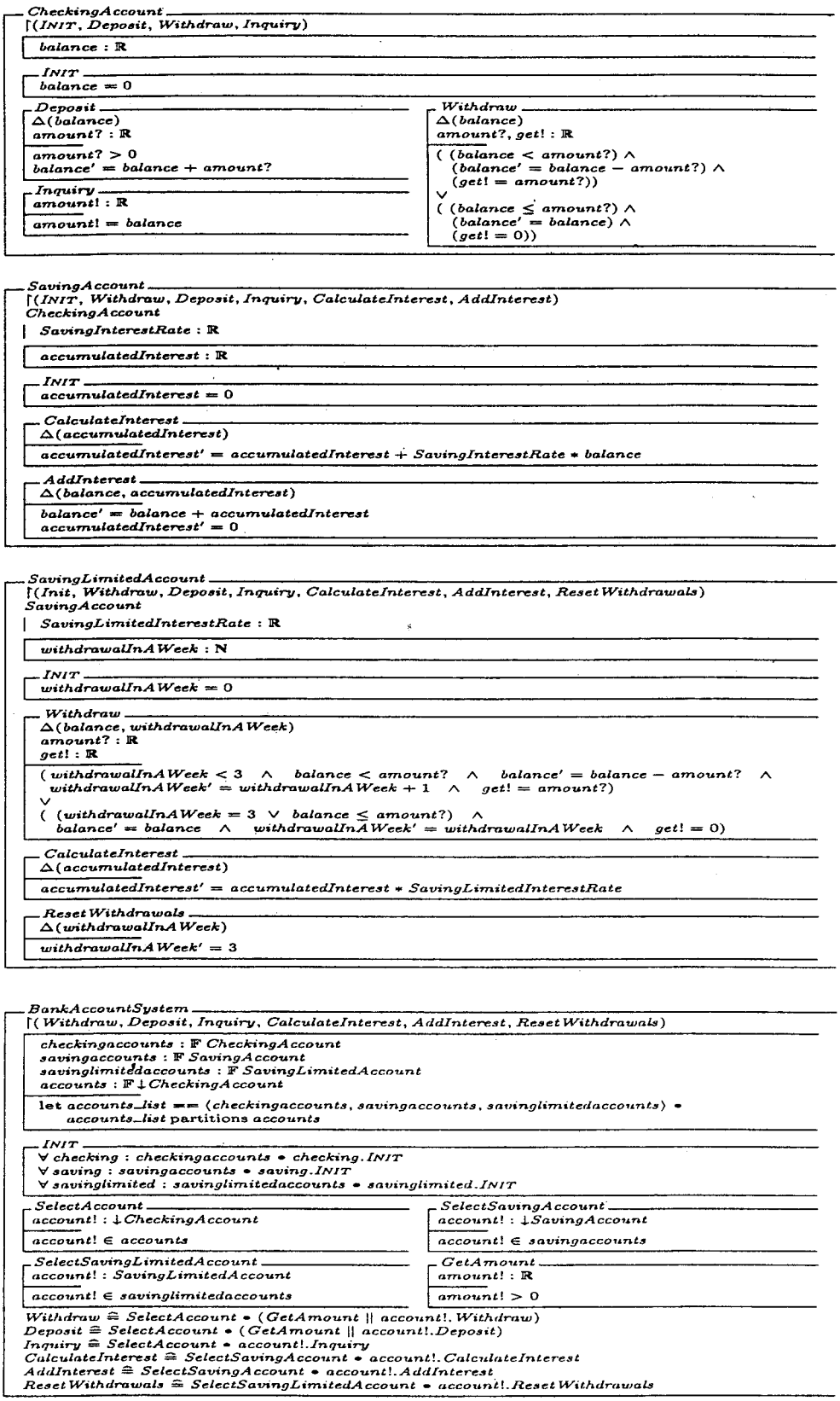

Figure 2. *SBAS* Object-Z Specification

CheckingAccount
  Deposit : < *nil* >
  Withdraw : < *nil* >
  Inquiry : < *nil* >
SavingAccount
  Deposit : < *nil* >
  Withdraw : < *nil* >
  Inquiry : < *nil* >
  CalculateInterest : < *nil* >
  AddInterest : < *nil* >
SavingLimitedAccount
  Deposit : < *nil* >
  Withdraw : < *nil* >
  Inquiry : < *nil* >
  CalculateInterest : < *nil* >
  AddInterest : < *nil* >
  ReserWithdrawals : < *nil* >
BankAccountSystem
  SelectAccount : < *nil* >
  SelectSavingAccount : < *nil* >
  SelectSavingLimitedAccount : < *nil* >
  GetAmount : < *nil* >
  Withdraw :
    < *SelectAccount*, *GetAmount*,
      account!(*CheckingAccount*).*Withdraw* >
    < *SelectAccount*, *GetAmount*,
      account!(*SavingAccount*).*Withdraw* >
    < *SelectAccount*, *GetAmount*,
      account!(*SavingLimitedAccount*).*Withdraw* >
  Deposit :
    < *SelectAccount*, *GetAmount*,
      account!(*CheckingAccount*).*Deposit* >
    < *SelectAccount*, *GetAmount*,
      account!(*SavingAccount*).*Deposit* >
    < *SelectAccount*, *GetAmount*,
      account!(*SavingLimitedAccount*).*Deposit* >
  Inquiry :
    < *SelectAccount*, *account*!(*CheckingAccount*).*Inquiry* >
    < *SelectAccount*, *account*!(*SavingAccount*).*Inquiry* >
    < *SelectAccount*, *account*!(*SavingLimitedAccount*).*Inquiry* >
  CalculateInterest :
    < *SelectSavingAccount*,
      account!(*SavingAccount*).*CalculateInterest* >
    < *SelectSavingAccount*,
      account!(*SavingLimitedAccount*).*CalculateInterest* >
  AddInterest :
    < *SelectSavingAccount*,
      account!(*SavingAccount*).*AddInterest* >
    < *SelectSavingAccount*,
      account!(*SavingLimitedAccount*).*AddInterest* >
  ResetWithdrawals :
    < *SelectSavingLimitedAccount*,
      account!(*SavingLimitedAccount*).*ResetWithdrawals* >

Figure 3. *SBAS* **partial invocation sequences.**

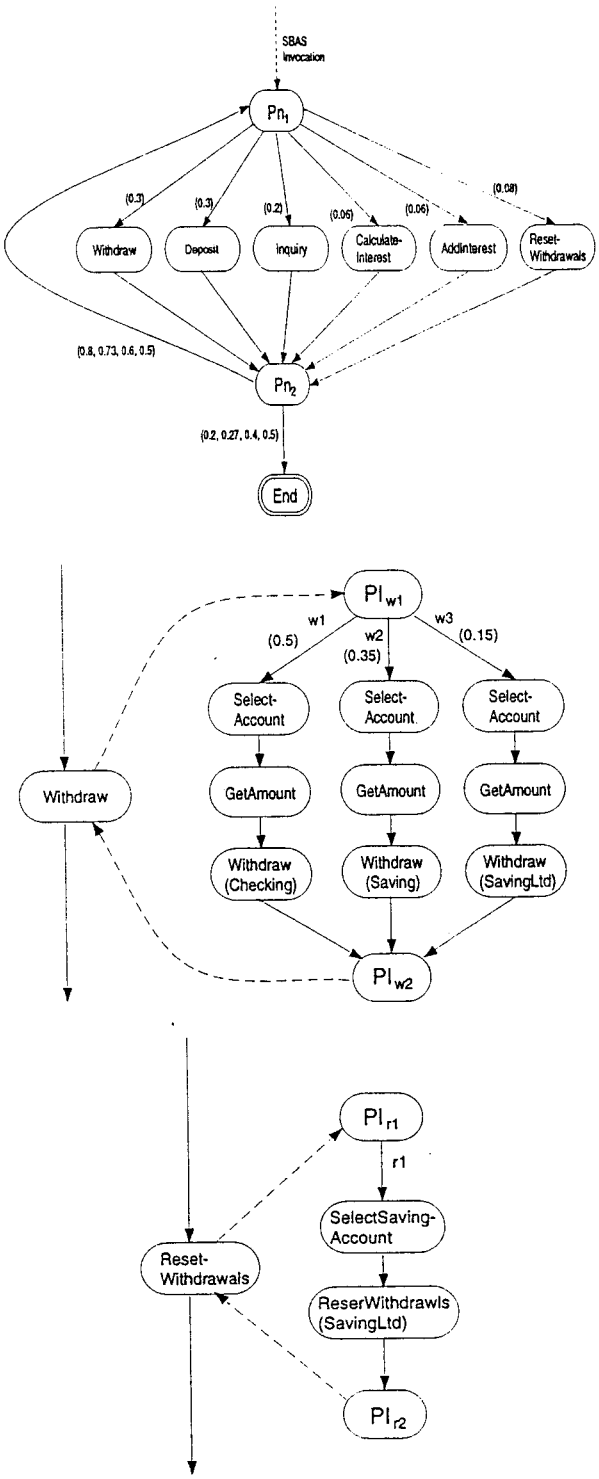

Figure 4. *SBAS* **hierarchical invocation diagram**