UML-Based Integration Testing

Jean Hartmann
Siemens Corporate Research
755 College Road East
Princeton NJ 08540
++1 609 734 3361
jhartmann@scr.siemens.com

Claudio Imoberdorf
Siemens Corporate Research
755 College Road East
Princeton NJ 08540
++1 609 734 3688
claudio@scr.siemens.com

Michael Meisinger
Technical University, Munich
Arcisstraße 21
80333 München
++49 (89) 289-01
meisinger@gmx.de

Abstract

Increasing numbers of software developers are using the Unified Modeling Language (UML) and associated visual modeling tools as a basis for the design and implementation of their distributed, component-based applications. At the same time, it is necessary to test these components, especially during unit and integration testing.

At Siemens Corporate Research, we have addressed the issue of testing components by integrating test generation and test execution technology with commercial UML modeling tools such as Rational Rose; the goal being a *design-based* testing environment.

In order to generate test cases automatically, developers first define the dynamic behavior of their components via UML Statecharts, specify the interactions amongst them and finally annotate them with test requirements. Test cases are then derived from these annotated Statecharts using our test generation engine and executed with the help of our test execution tool. The latter tool was developed specifically for interfacing to components based on COM/DCOM and CORBA middleware.

In this paper, we present our approach to modeling components and their interactions, describe how test cases are derived from these component models and then executed to verify their conformant behavior. We outline the implementation strategy of our T*n*T environment and use it to evaluate our approach by means of a simple example.

Keywords

Distributed components, functional testing, test generation, test execution, UML statecharts, COM/DCOM, CORBA.

1. Introduction

While standardized testing strategies and tools have been available for IC (hardware) components for many years, the research and development of standardized testing techniques and tools for distributed *software* components has just begun [18]. Three key technological factors are not only allowing developers to design and implement components, but at the same time, they are contributing towards the development of such testing strategies.

These factors include:

- The definition of the Unified Modeling Language (UML) [14,15], a standardized way of modeling the static structure and dynamic behavior of components and their interfaces;
- The standardization of object-oriented middleware, for example, Microsoft's COM/DCOM and OMG's CORBA;
- The continued refinement of object-oriented programming languages, such as Java and C++, and integrated development environments that provide extensive support for creating distributed components.

As a result, developers are implementing large numbers of components ranging from relatively simple graphical user interface (GUI) components to sophisticated server-side application logic [4]. In this paper, we are focusing on the latter type of component development.

As developers are delivering these complex, server-side components, they must also ensure that each component is delivered with a concise and unambiguous definition of its *interfaces*, and the legal order in which operations may be invoked on them. Component interfaces and their protocol specifications are being described or modeled in a variety of ways. For example, in the case of the Enterprise Java Beans Specification [10], this is achieved through contracts and UML Sequence Diagrams (also known as Message Sequence Charts). While a Sequence Diagram is useful at describing a specific interaction scenario, it may require a large number of such diagrams to completely specify the interaction of a complex component with its client(s). A more concise and compact way, however, of representing these scenarios is to depict them using a *UML Statechart Diagram*. It is this dynamic, behavioral description that is used in this paper as a basis for modeling components and their interfaces.

Stimulating the interfaces to individual or groups of distributed components with inputs and monitoring as well as verifying the resulting responses is considered to be a functional or black-box testing strategy [2]. In our approach, we use Statecharts as a basis for generating such black-box tests, which developers can use for unit and integration testing. If the components have not been developed in-house or the source code is not available, as in the case of third-party components, then it may be possible to derive an abstracted Statechart for each or a subsystem of components.

In Section 2, we describe our approach for modeling components and their interactions by means of Statecharts. Section 3 outlines the way in which the individual Statecharts are combined into a global behavioral model. In Section 4, we present the way in

which test cases are generated and executed using the global model. For Section 5 and 6, we provide an overview of the T*n*T environment that realizes the approach and apply it to an example. Related work, conclusions and future work are presented in Section 7 and 8.

2. Modeling Components in UML

In this section, we describe the use of UML Statecharts in modeling the dynamic behavior of components as well as the communication between them[1]. To better convey the concepts, we have illustrated the paper with an example.

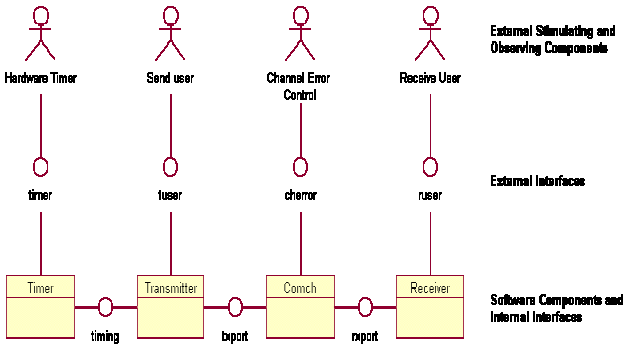


Figure 1: Alternating Bit Protocol Example

The example in Figure 1 represents an alternating bit communication protocol[2] in which there are four separate components *Timer*, *Transmitter*, *ComCh* (*Communication Channel*) and *Receiver* and several internal as well as external interfaces and stimuli.

The protocol is a unidirectional, reliable communication protocol. A user invokes a *Transmitter* component to send data messages over a communication channel and to a *Receiver* component, which then passes it on to another user. The communication channel can lose data messages as well as acknowledgements. The reliable data connection is implemented by observing possible timeout conditions, repeatedly sending messages, if necessary, and ensuring the correct order of the messages.

2.1 UML Statecharts

The Unified Modeling Language (UML) is a general-purpose visual modeling language that is used to specify, visualize, construct and document the artifacts of a software system.

In this paper, we focus on the *dynamic* views of UML, in particular, Statechart Diagrams. A Statechart can be used to describe the dynamic behavior of a component or should we say *object* over time by modeling its lifecycle. The key elements described in a Statechart are *states*, *transitions*, *events*, and *actions*.

States and transitions define all possible states and changes of state an object can achieve during its lifetime. State changes occur as reactions to events received from the object's interfaces. Actions correspond to internal or external method calls.

[1] The nomenclature in this paper refers to UML, Revision 1.3.

[2] The name *Alternating Bit Protocol* stems from the message sequence numbering technique used to recognize missing or redundant messages and to keep up the correct order.

Figure 2 illustrates the Statechart for the *Transmitter* object shown in Figure 1. It comprises six states with a start and an end state. The transitions are labeled with call event descriptions corresponding to external stimuli being received from the *tuser* interface and internal stimuli being sent to the *Timer* component via the *timing* interface and received from the *ComCh* component via the *txport* interface. These internal/external interfaces and components are shown in Figure 1. Moreover, the nomenclature used for labeling the transitions is described in the next section and relates to the way in which component interactions are modeled.

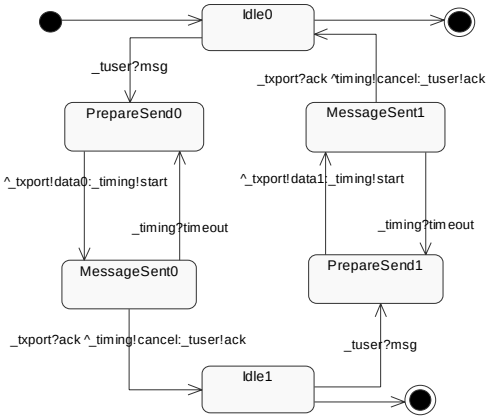


Figure 2: Statechart Diagram for the *Transmitter* Object

2.2 Communicating Statecharts

In the following section, we describe how a developer would need to model the communication between multiple Statecharts, when using a commercial UML-based modeling tool. At present, UML does not provide an adequate mechanism for describing the communication between two components, so we adopted concepts from CSP (Communicating Sequential Processes) [6] to enhance its existing notation.

2.2.1 Communication Semantics

In our approach, we wanted to select communication semantics that most closely relate to the way in which COM/DCOM and CORBA components interact in current systems. While such components allow both synchronous and asynchronous communications, we focus on a *synchronous* mechanism for the purposes of this paper.

In addition, there are two types of synchronous communication mechanisms. The first, the shared (or global) event model, may broadcast a single event to multiple components, all of which are waiting to receive and act upon it in unison. The second model, a point-to-point, blocking communication mechanism, can send a single event to just one other component and it is only these two components that are then synchronized. The originator of the event halts its execution (blocks) until the receiver obtains the event. It is *this* point-to-point model that we adopted, because it most closely resembles the communication semantics of COM/DCOM and CORBA.

2.2.2 Transition Labeling

In order to show explicit component connections and to associate operations on the interfaces with events within the respective Statecharts, we defined a transition labeling convention based on the *notation* used in CSP for communication operations[3]. A unique name must be assigned by the developer to the connection between two communicating Statecharts[4]. This name is used as a prefix for trigger (incoming) and send (outgoing) events. A transition label in a Statechart would be defined as follows:

```
_timing?timeout ^_txport!data0
```

This transition label can be interpreted as receiving a trigger event `timeout` from connection `timing` followed by a send event `data0` being sent to connection `txport`. Trigger (also known as receive) events are identified by a separating question mark, whereas send events are identified by a leading caret (an existing UML notation) and a separating exclamation mark. In Figure 3 below, two dark arrows indicate how the `timing` interface between the two components is used by the send and receive events. Connections are considered bi-directional, although it is possible to use different connection names for each direction, if the direction needs to be emphasized.

Transitions can contain *multiple* send and receive events. Multiple receive events within a transition label can be specified by separating them with a plus sign. Multiple send events with different event names can be specified by separating them by a colon.

2.2.3 Example

Figure 3 shows two communicating Statecharts for the *Transmitter* and *Timer* components. The labels on the transitions in each Statechart refer to events occurring via the internal *timing* interface, the interface *txport* with the *ComCh* component and two external interfaces, *timer* and *tuser*.

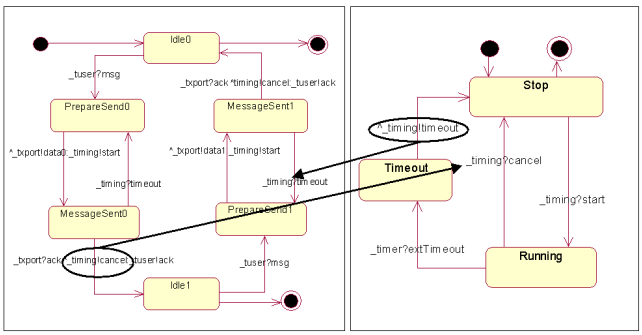


Figure 3: Communicating *Transmitter* and *Timer* Components

The *Transmitter* component starts execution in state `Idle0` and waits for user input. If a message arrives from connection `tuser`, the state changes to `PrepareSend0`. Now, the message is sent to the communication channel. At the same time, the `Timer` component receives a `start` event. The component is now in the state `MessageSent0` and waits until either the `Timer` component sends a `timeout` event or the `ComCh` component sends a message acknowledgement `ack`. In case of a timeout, the message is sent again and the timer is also started again. If an `ack` is received, an event is sent to the `Timer` component to `cancel` the timer and the user gets an acknowledgement for successful delivery of the message. Now, the same steps may be repeated, but with a different message sequence number, which is expressed by the event `data1` instead of `data0`.

In addition to modeling the respective Statecharts and defining the interactions between them, developers can specify test requirements, that is, directives for test generation, which influence the size and complexity of the resulting test suite. However, this aspect is not shown in this example.

3. Establishing a Global Behavioral Model

In the following section, we describe the steps taken in constructing a *global* behavioral model, which is internal to our tool, from multiple Statecharts that have been defined by a developer using a commercial UML-based modeling tool. In this global behavioral model, the significant properties, that is, behavior, of the individual state machines are preserved.

3.1 Definition of Subsystems

A prime concern with respect to the construction of such a global model is scalability. Apart from utilizing efficient algorithms to compute such a global model, we defined a mechanism whereby developers can group components into subsystems and thus help to reduce the size of a given model. The benefit of such a subsystem definition is that it also reflects a commonly used integration testing strategy described in Section 4.

Our approach allows developers to specify a subsystem of components to be tested and the interfaces to be tested. If no subsystem definition has been specified by a developer, then all modeled components and interfaces are considered as part of the global model.

3.2 Composing Statecharts

3.2.1 Finite State Machines

We consider Statecharts as Mealy finite state machines; they react upon input in form of receive events and produce output in form of send events. Such state machines define a directed graph with nodes (representing the states) and edges (representing the transitions). They have one initial state and possibly several final states. The state transitions are described by a function:

Definition: A *communicating* finite state machine used for component specification is defined as $A = (S, M, T, \delta, s_0, F)$, where

S is a set of states, unique to the state machine

$M \subset S$ are states marked as intermediate states

T is an alphabet of valid transition annotations, consisting of transition type, connection name and event name. Transition type $\in$ {INT, COMM, SEND, RECEIVE}

$\delta: S \times T \rightarrow S$ is a function describing the transitions between states

$s_0 \in S$ is the initial state

$F \subset S$ is a set of final states

[3] In CSP, operations are written as *channel1!event1* which means that *event1* is sent via *channel1*. A machine input operation is written as *channel2?event1* where *channel2* receives an *event1*.

[4] This is currently a limitation of our tool implementation.

Initial and final states are regular states. The initial state gives a starting point for a behavior description. Final states express possible end points for the execution of a component.

The transition annotations T contain a transition type as well as a connection name and an event name. Transition types can be INTernal, SEND, RECEIVE and COMMunication. Transitions of type SEND and RECEIVE are external events sent to or received from an external interface to the component's state machine. SEND and RECEIVE transitions define the external behavior of a component and are relevant for the external behavior that can be observed. An INTernal transition is equivalent to a ε-transition (empty transition) of a finite state machine [7]. It is not triggered by any external event and has no observable behavior. It represents arbitrary internal action. COMMunication transitions are special types of internal transitions representing interaction between two state machines. Such behavior is not externally observable. When composing state machines, matching pairs of SEND and RECEIVE transitions with equal connection and event names are merged to form COMMunication transitions. For example, the transitions highlighted by dark arrows in Figure 3 would be such candidates.

The definition of a state machine allows transitions that contain single actions. Every action expressed by a transition annotation is interpreted as an atomic action. Component interaction can occur after each action. If several actions are grouped together without the possibility of interruption, the states between the transitions can be marked as intermediate states. Intermediate states ($M \in S$) are introduced to logically group substructures of states and transitions. The semantics of intermediate states provide a behavioral description mechanism similar to *microsteps*. Atomic actions are separated into multiple consecutive steps, the microsteps, which are always executed in one run. These microsteps are the outgoing transitions of intermediate states. This technique is used in our approach as part of the process of converting the UML Statecharts into an internal representation. The result is a set of *normalized* state machines.

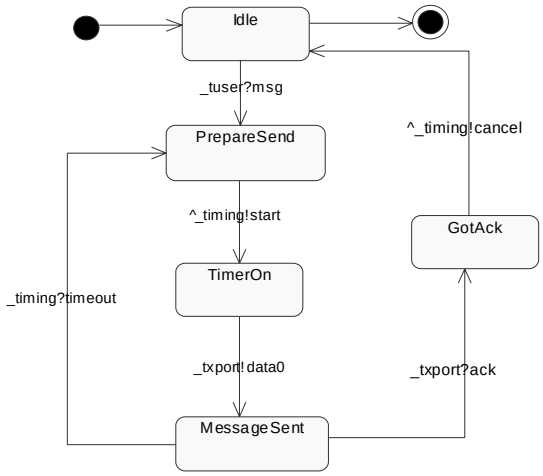


Figure 4: *Normalized* Transmitter Component

Figure 4 shows such a state machine for a simplified version of the *Transmitter* object. Two additional, intermediate states `TimerOn` and `GotAck` have been inserted to separate the multiple events `_txport!data0:^_timing_start` and `txport?ack^timing!cancel` between the *PrepareSend*, *MessageSent* and *Idle* states shown in Figure 2.

3.2.2 Composed State Machines

A composed state machine can be considered as the *product* of multiple state machines. It is itself a state machine with the dynamic behavior of its constituents. As such, it would react and generate output as a result of being stimulated by events specified for the respective state machines. Based on the above definition of a finite state machine, the structure of a composed state machine can be defined as follows:

Definition: Let $A = (S_1, M_1, T_1, \delta_1, s_{01}, s_{f1})$ and $B = (S_2, M_2, T_2, \delta_2, s_{02}, s_{f2})$ be two state machines and $S_1 \cap S_2 = \emptyset$. The composed state machine $C = A\#B$ has the following formal definition:

$A\#B = (S', M', T', \delta', s_0', F')$

$S' = S_1 \times S_2$

$M' \subset (M_1 \times S_2) \cup (S_1 \times M_2)$

$T' \subset T_{12} \cup T_{COMM}$

$T_{12} = (T_1 \cup T_2) \setminus$ {SEND, RECEIVE with connections between A and B}

T_{COMM} = {COMM for matching events from T_1 and T_2}

δ': $S' \times T' \rightarrow S'$

δ' is generated from δ_1 and δ_2 with the state machine composition schema

$s_0' = (s_{01}, s_{02}) \in S'$.

$F' = \{(s_1, s_2) \in S' \mid s_1 \in F_1 \wedge s_2 \in F_2\}$

For example, a global state for A#B is defined as a two-tuple (s_1, s_2), where s_1 is a state of A and s_2 is a state of B. These two states are referred to as *part states*. Initial state and final states of A#B are element and subset of this product. The possible transition annotations are composed from the union of T_1 and T_2 and new COMMunication transitions that result from the matching transitions. Excluded are the transitions that describe possible matches. Either COMMunication transitions are created from them or they are omitted, because no communication is possible.

3.2.3 Composition Method

A basic approach for composing two state machines is to generate the product state machine by applying generative multiplication rules for states and transitions. This leads to a large overhead, because many unreachable states are produced that have to be removed in later steps. The resulting product uses more resources than necessary as well as more computation time for generation and minimization.

Instead, our approach incorporates an incremental composition and reduction algorithm that uses reachability computations. A global behavioral model is created stepwise. Beginning with the global initial state, all reachable states and all transitions in between are computed. Every state of the composed state machine is evaluated only once. Due to the reachability algorithm, the intermediate data structures are at no time larger than the result of one composition step. States and transitions within the composed state machines that are redundant in terms of external observation are removed. By applying the reduction algorithm using heuristic

rules, it is possible to detect redundancies and to reduce the size of a composed state machine before the next composition step.

Defined subsystems are processed independently sequentially. For each subsystem, the composition algorithm is applied. The inputs for the composition algorithm are data structures representing the normalized communicating state machines of the specified components within the current subsystem. The connection structure between these components is part of these data structures. The order of the composition steps determines the size and complexity of the result for the next step and therefore the effectiveness of the whole algorithm. The worst case for intermediate composition products is a composition of two components with no interaction. The maximum possible number of states and transitions created in this case resembles the product of two state machines.

It is therefore important to select the most suitable component for the next composition step. The minimal requirement for the selected component is to have a common interface with the other component. This means that at least one connection exists to the existing previously calculated composed state machine.

A better strategy with respect to minimizing the size of intermediate results is to select the state machine with the highest relative number of communication relationships or interaction points. A suitable selection norm is the ratio of possible communication transitions to all transitions in a state machine. The component with the highest ratio exposes the most extensive interface to the existing state machine and should be selected.

1	2	3	4	5	6	7	8	9
GROUP	# BWO	Transition s1 → t1		Transition s2 → t2			Resulting transitions and states	
		Connection is to FSM B	Transition type 1	Connection is to FSM A	Transition type 2	Equal event	New transition descriptor	Success or state
1	1	YES	SEND	YES	RECV	YES	COMM, connection, event	(t1,t2)
	1	YES	RECV	YES	SEND	YES	COMM, connection, event	(t1,t2)
2	1	YES	SEND	YES	SEND	YES	-	-
	1	YES	RECV	YES	RECV	YES	-	-
	4	YES	SEND/RECV	YES	SEND/RECV	NO	-	-
3	8	NO	SEND/RECV	NO	SEND/RECV	*	A: SEND/RECV, connA, eventA B: SEND/RECV, connB, eventB	(t1,s2) (s1,t2)
4	8	NO	SEND/RECV	YES	SEND/RECV	*	A: SEND/RECV, connA, eventA	(t1,s2)
	8	YES	SEND/RECV	NO	SEND/RECV	*	B: SEND/RECV, connB, eventB	(s1,t2)
5	16	*	INT/COMM	YES	SEND/RECV	*	A: INT/COMM	(t1,s2)
	16	YES	SEND/RECV	*	INT/COMM	*	B: INT/COMM	(s1,t2)
6	16	*	INT/COMM	NO	SEND/RECV	*	A: INT/COMM B: SEND/RECV, connB, eventB	(t1,s2) (s1,t2)
	16	NO	SEND/RECV	*	INT/COMM	*	A: SEND/RECV, connA, eventA B: INT/COMM	(t1,s2) (s1,t2)
7	32	*	INT/COMM	*	INT/COMM	*	A: INT/COMM B: INT/COMM	(t1,s2) (s1,t2)

Table 1 : Decision Table for Computing Successor States and Transitions

This incremental composition and reduction method also specifies a composition schema. For every combination of outgoing transitions of the part states, a decision table (shown in Table 1) is used to compute the new transitions for the composed state machine.

If a new transition leads to a global state that is not part of the existing structure of the composed state machine, it is added to an *unmarked* list. The transition is added to the global model. Exceptions exist, when part states are marked as intermediate. Every reachable global state is processed and every possible new global transition is inserted into the composed state machine. The algorithm terminates when no *unmarked* states remain. This means that every reachable global state was inserted into the model and later processed. The schema we used was based on a composition schema developed by Sabnani *et al.* [16]. We enhanced it to include extensions for connections, communication transitions, and intermediate states.

We are assuming throughout this composition process that the individual as well as composed state machines have deterministic behavior. We also ensure that the execution order of all component actions is sequential. This is important as we then wish to use the global model to create test cases that are dependent on a certain flow of events and actions; we want to generate linear and sequential test cases for a given subsystem.

3.2.4 Complexity Analysis

As we are composing the product of two state machines, the worst case complexity would be $O(n^2)$ assuming *n* is the number of states in a state machine. However, our approach often does much better than this due to the application of the heuristic reduction rules that can help to minimize the overall size of the global model during composition and maintain its observational equivalence [11].

Typically, the reduction algorithm being used has linear complexity with respect to the number of states [16]. For example, it was reported that the algorithm was applied to a complex communication protocol (ISDN Q.931), where it was shown that instead of generating over 60,000 intermediate states during composition, the reduction algorithm kept the size of the model to approximately 1,000 intermediate states. Similar results were reported during use of the algorithm with other systems. The algorithm typically resulted in a reduction in the number of intermediate states by one to two orders of magnitude.

3.2.5 Example

Taking the normalized state machine of the Transmitter component in Figure 4 and the Timer component in Figure 3, the composition algorithm needs to perform only one iteration to generate the global behavioral model in Figure 5.

A global initial state `Idle_Stopped` is created using the initial states of the two state machines. This state is added to the list of unmarked states. The composition schema is now applied for every state within this list to generate new global states and transitions until the list is empty. The reachability algorithm creates a global state machine comprising six states and seven transitions. Three COMMunication transitions are generated, which are identified by a hash mark in the transition label showing the communication connection and event.

The example shows the application of the decision table. In the first global state, `Idle_Stopped`, part state `Idle` has an outgoing receive transition to `PrepareSend` using an external connection. Part state `Stopped` has also an outgoing receive transition to `Running` with a connection to the other component. According to the Decision Rule #4 of the table, the transition with the external connection is inserted into the composed state machine and the other transition is ignored. The new global receive transition leads to the global state `PrepareSend_Stop`.

For the next step, both part states include transitions, which use internal connections. They communicate via the same connection `timing` and the same event - these are matching transitions. According to Decision Rule #1 of the table, a communication

transition is included in the composed state machine that leads to the global state `TimerOn_Running`. These rules are applied repeatedly until all global states are covered.

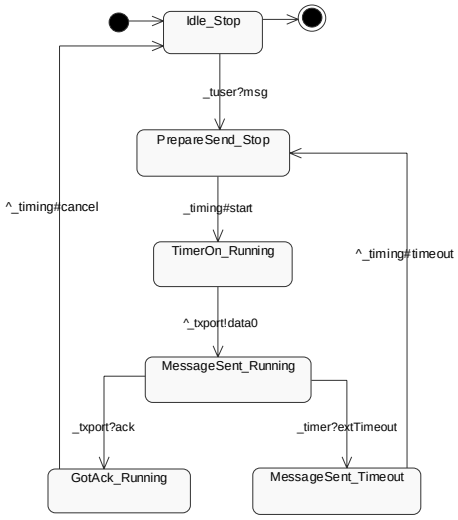


Figure 5: Global Behavioral Model for the *TransmitterTimer* Subsystem

4. Test Generation and Execution

In the preceding sections, we discussed our approach to modeling individual or collections of components using UML Statecharts, and establishing a global behavioral model of the composed Statecharts. In this section, we show how this model can be used as the basis for automatic test generation and execution during unit and integration testing.

4.1 Unit and Integration Testing

After designing and coding each software component, developers perform unit testing to ensure that each component correctly implements its design and is ready to be integrated into a system of components. This type of testing is performed in isolation from other components and relies heavily on the design and implementation of *test drivers* and *test stubs*. New test drivers and stubs have to be developed to validate each of the components in the system.

After unit testing is concluded, the individual components are collated, integrated into the system, and validated again using yet another set of test drivers. At each level of testing, a new set of custom test drivers is required to stimulate the components. While each component may have behaved correctly during unit testing, it may not do so when interacting with other components. Therefore, the objective of integration testing is to ensure that all components interact and interface correctly with each other, that is, have no interface mismatches. This is commonly referred to as *bottom-up* integration testing.

Our approach aims at minimizing the testing costs, time and effort associated with *initially* developing customized test drivers, test stubs, and test cases as well as *repeatedly* adapting and rerunning them for regression testing purposes at each level of integration.

4.2 Test Generation

Before proceeding with a description of the test generation and execution steps, we would like to emphasize the following:

- Our approach generates a set of conformance tests. These test cases ensure the compliance of the design specification with the resulting implementation.
- It is assumed that the implementation behaves in a deterministic and externally controllable way. Otherwise, the generated test cases may produce incorrect results.

4.2.1 Category-Partition Method

For test generation, we use the Test Development Environment (TDE), a product developed at Siemens Corporate Research [1]. TDE processes a *test design* written in the Test Specification Language (TSL). This language is based on the category-partition method, which identifies behavioral equivalence classes within the structure of a system under test.

A category or partition is defined by specifying all possible data choices that it can represent. Such choices can be either data values or references to other categories or partitions, or a combination of both. The data values may be string literals representing fragments of test scripts, code, or case definitions, which later can form the contents of a test case.

A TSL test design is now created from the global behavioral model by mapping its states and transitions to TSL categories or partitions, and choices. States are the equivalence classes and are therefore represented by partitions. Each transition from the state is represented as a choice of the category/partition. Only partitions are used for equivalence class definitions, because paths through the state machine are not limited to certain outgoing transitions for a state; this would be the case when using a category. Each transition defines a choice for the current state, combining a test data string (the send and receive event annotations) and a reference to the next state. A final state defines a choice with an empty test data string.

4.2.2 Generation Procedure

A recursive, directed graph is built by TDE that has a root category/partition and contains all the different paths of choices to plain data choices. This graph may contain cycles depending on the choice definitions and is equivalent to the graph of the global state machine. A test frame, that is, test case is one instance of the initial data category or partition, that is, one possible path from the root to a leaf of the (potentially infinite) reachability tree for the graph.

An instantiation of a category or partition is a random selection of a choice from the possible set of choices defined for that category/partition. In the case of a category, the same choice is selected for every instantiation of a test frame. This restricts the branching possibilities of the graph. With a partition, however, a new choice is selected at random with every new instantiation. This allows full branching within the graph and significantly influences test data generation. The contents of a test case consist of all data values associated with the edges along a path in the graph.

4.2.3 Coverage Requirements

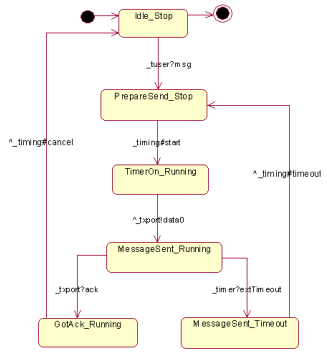
The TSL language provides two types of coverage requirements:

- *Generative requirements* control which test cases are instantiated. If no generative test requirements are defined, no test frames are created. For example, coverage statements can be defined for categories, partitions and choices.
- *Constraining requirements* cause TDE to omit certain generated test cases. For example, there are maximum coverage definitions, rule-based constraints for category/partition instantiation combinations, instantiation preconditions and instantiation depth limitations. Such test requirements can be defined globally within a TSL test design or attached to individual categories, partitions or choices.

TDE creates test cases in order to satisfy all specified coverage requirements. Input sequences for the subsystem are equivalent to paths within the global behavioral model that represents the subsystem, starting with the initial states. Receive transitions with events from external connections stimulate the subsystem. Send transitions with events to external connections define the resulting output that can be observed by the test execution tool. All communication is performed through events. For unit test purposes, the default coverage criterion is that all transitions within a Statechart must be traversed at least once. For integration testing, only those transitions that involve component interactions are exercised. If a subsystem of components is defined as part of the modeling process, coverage requirements are formulated to ensure that those interfaces, that is, transitions are tested.

4.2.4 Example

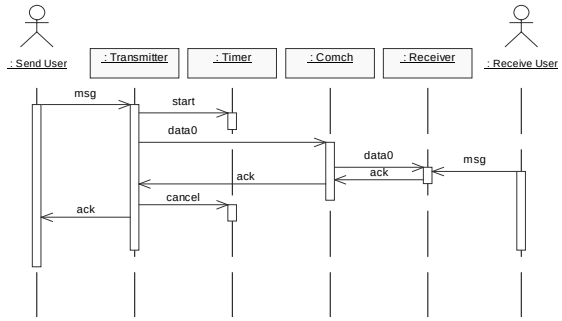
Figure 6 presents the test case that is derived from the global behavioral model shown in Figure 5. This one test case is sufficient to exercise the interfaces, `txport`, `tuser` and `timer` defined for the components. Each line of this generic test case format represents either an input event or an expected output event. We chose a test case format where the stimulating events and expected responses use the strings `SEND` and `RECEIVE` respectively, followed by the connection and event names. Currently, the events have no parameters, but that will be remedied in future work.



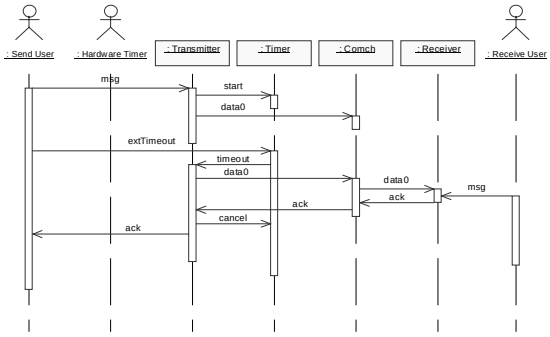

```
/* covers both successful and timed out
transmissions */

TEST CASE #1
 * SEND _tuser.msg();
 * RECEIVE _txport.data0();
 * SEND _txport.ack();
 * RECEIVE _tuser.ack();
 * SEND _tuser.msg();
 * RECEIVE _txport.data0();
 * SEND _timer.extTimeout();
 * RECEIVE _txport.data0();
 * SEND _txport.ack();
 * RECEIVE _tuser.ack();
```

Figure 6: Test Case for *TransmitterTimer* Subsystem

The Sequence Diagrams for the execution of this test case are shown in Figure 7. Note that the external connection `timer` has a possible event `extTimeout`. This event allows a timeout to be triggered without having a real hardware timer available.



(a) Successful Transmission



(b) Timed Out Transmission

Figure 7: Sequence Diagrams for the Example

4.3 Test Execution

In this section, we show how the generated test cases can be mapped to the COM/CORBA programming model. We describe how an executable test driver (including stubs) is generated out of such test cases.

As seen earlier, a test case consists of a sequence of SEND and RECEIVE events such as the following:

```
*SEND _tuser.msg();
*RECEIVE _txport.data0();
```

The intent of the SEND event is to stimulate the object under test. To do so, the connection `_tuser` is mapped to an object reference, which is stored in a variable `_tuser` defined in the test case[5]. The event `msg` is mapped to a method call on the object referenced by `_tuser`.

The RECEIVE event represents a response from the object under test, which is received by an appropriate sink object. To do so, the connection `_txport` is mapped to an object reference that is stored in a variable `_txport`. The event `data0` is mapped to a callback, such that the object under test fires the event by *calling back* to a sink object identified by the variable `_txport`. The sink object thus acts as a stub for an object that would implement the `txport` interface on the next higher layer of software.

[5] In the current implementation of TnT, the initialization code that instantiates the *Transmitter* object and stores the object reference in the variable `_tuser` has to be written manually.

Typically, reactive software components expose an interface that allows interested parties to subscribe for event notification[6].

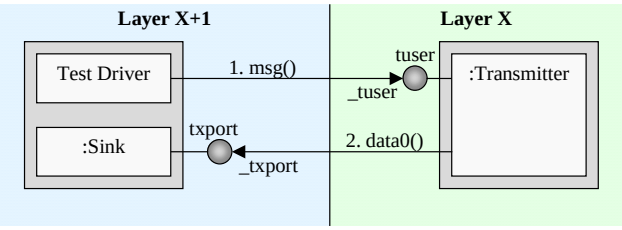


Figure 8: Interaction with the Object under Test

The interactions between the test execution environment and the *Transmitter* object are shown in Figure 8. The *TestDriver* calls the method *msg()* on the *Transmitter* object referenced through the variable `_tuser`. The *Transmitter* object notifies the sink object via its outgoing `_txport` interface.

Test case execution involving RECEIVE events not only requires a comparison of the out-parameters and return values with expected values, but also the evaluation of event patterns. These event patterns specify which events are expected in response to particular stimuli, and when they are expected to respond by. To accomplish this, the sink objects associated with the test cases need to be monitored to see if the required sink methods are invoked.

5. Implementation of T*n*T

The T*n*T environment was developed at Siemens Corporate Research in order to realize the work described above. This design-based testing environment consists of two tools, our existing test generation tool, TDE with extensions for UML (TDE/UML) and TECS, the test execution tool. Thus, the name - T*n*T. Our new environment interfaces directly to the UML modeling tools, Rose2000 and Rose Real-Time 6.0, by Rational Software. Figure 9 shows how test case generation can be initiated from within Rational Rose.

In this section, we briefly describe our implementation strategy.

5.1 TDE/UML

Figure 10 depicts the class diagram for TDE/UML. TDE/UML accesses both Rose applications through Microsoft COM interfaces. In fact, our application implements a COM server, that is, a COM component waiting for events. We implemented TDE/UML in Java using Microsoft's Visual J++ as it can generate Java classes for a given COM interface. Each class and interface of the Rose object model can thus be represented as a Java class; data types are converted and are consistent. The Rose applications export administrative objects as well as model objects, which represent the underlying Rose repository.

Rose also provides an extensibility interface (REI) to integrate external tools known as *Add-Ins*. A new tool, such as TDE/UML can be installed within the Rose application as an Add-In and invoked via the Rose *Tool* menu. Upon invocation, the current Rose object model is imported including the necessary Statecharts, processed using the techniques described in previous sections, and the files needed for test generation and test execution generated.

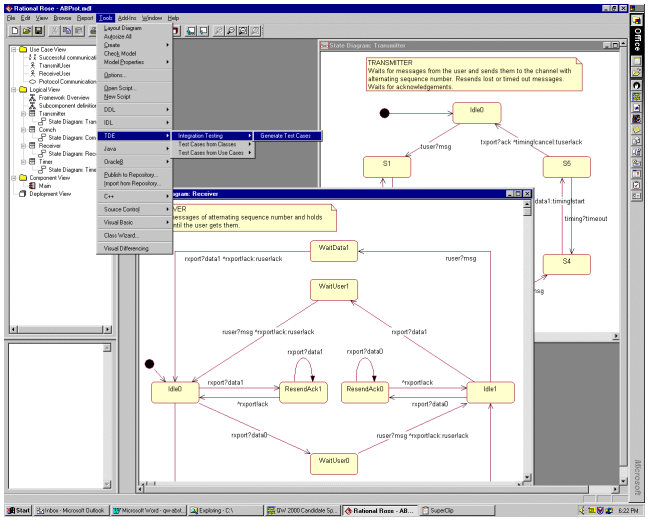


Figure 9: Generating Tests from within Rational Rose

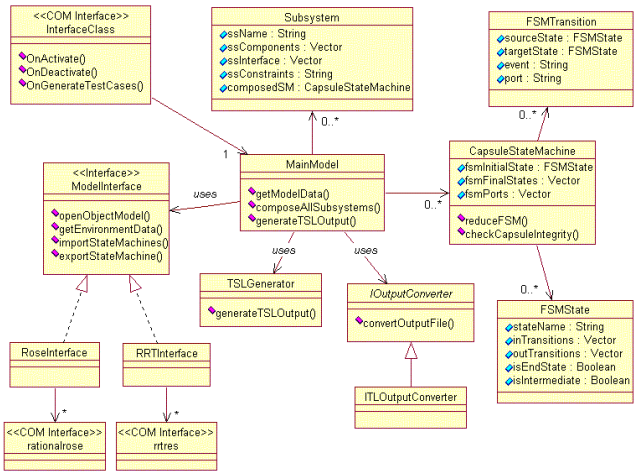


Figure 10: Class Diagram for TDE/UML

5.2 TECS

The **T**est **E**nvironment for Distributed **C**omponent-Based **S**oftware (TECS) specifically addresses test execution. While the test generation method described in Section 4.2 can only support components communicating synchronously, TECS already supports both synchronous *and* asynchronous communication[7].

The test environment is specifically designed for testing COM or CORBA components during unit and integration testing. The current version of TECS supports the testing of COM components. It can be used as part of the T*n*T environment or as a standalone tool, and includes the following features:

[6] In the current implementation of T*n*T, the initialization code for instantiating a sink object and registering it with the *Transmitter* component has to be written manually.

[7] With asynchronous communication, a component under test can send response events to a sink object at any time and from any thread.

- **Test Harness Library** – this is a C++ framework that provides the basic infrastructure for creating the executable test drivers.
- **Test Case Compiler** – it is used to generate test cases in C++ from a test case definition such as the one illustrated in Figure 6. The generated test cases closely co-operate with the Test Harness Library. A regular C++ compiler is then used to create an executable test driver out of the generated code and the Test Harness Library. The generated test drivers are COM components themselves, exposing the interfaces defined through the TECS environment.
- **Sink Generator** – it is used to generate C++ sink classes out of an IDL interface definition file. The generated sink classes also closely co-operate with the Test Harness Library.
- **Test Control Center** – it provides the user a means of running test cases interactively through a graphical user interface or in batch mode. The information generated during test execution is written into an XML-based tracefile. The Test Control Center provides different views of this data such as a trace view, an error list, and an execution summary. Further views can easily be defined by writing additional XSL style sheets.

6. Evaluating the Example

In this section, we describe an evaluation of our approach using the alternating bit protocol example. As discussed in Section 2, the example comprises of four components, each with its own Statechart and connected using the interfaces depicted in Figure 1.

We are currently applying this approach to a set of products within different Siemens business units, but results from our experimentation are not yet available. We are aiming to examine issues such as the fault detection capabilities of our approach.

6.1.1 Component Statistics

Table 2 shows the number of states and transitions for the four Statecharts before and after they were imported into TDE/UML and converted into a normalized global model by the composition steps described in Section 3.2. We realize that the size of these components is moderate, but we use them to highlight a number of issues. For the example, the normalized state machine for each component is never more than twice the size of its associated UML Statechart.

	State Diagram		Normalized FSM	
Component	**States**	**Transitions**	**States**	**Transitions**
Transmitter	6	8	12	14
Timer	3	4	4	5
Comch	5	8	5	10
Receiver	8	14	16	22

Table 2 : Component Statistics

6.1.2 Defining an Integration Test Strategy

An important decision for the developer is the choice of an appropriate integration test strategy. Assuming that a bottom-up integration test strategy is to be used, a developer may wish to integrate the `Transmitter` and `Timer` components first followed by the `Receiver` and `Comch` components. Afterwards, the two subsystems would be grouped together to form the complete system. In this case, only the interface between the two subsystems, `txport`, would need to be tested. Below, we show the subsystem definitions for the chosen integration test strategy.

```
subsystem TransmitterTimer {
        components: Transmitter, Timer; }
subsystem ComchReceiver {
        components: Comch, Receiver; }
subsystem ABProtocol {
        components: Transmitter, Timer, Comch,
        Receiver;
        interface: txport; }
```

6.1.3 Applying the Composition and Reduction Step

The time taken for the import of these four Statecharts as well as the execution time for the composition algorithm was negligible. Table 3 shows the number of states/transitions created during the composition step as well as the values for when the reduction step is not applied. Typically, the reduction algorithm is applied after each composition step.

The values in *italic* show combinations of components with no common interface. The numbers for these combinations are very high as would be expected. Such combinations are generally not used as intermediate steps. The values in **bold** indicate the number of states/transitions used for the above integration test strategy. The values show how the number of states/transitions can be substantially reduced as in the case of all four components being evaluated together as a complete system.

	Transmitter		Timer		Comch	
	reduced	not red.	reduced	not red.	reduced	not red.
Timer	**20/24**	20/24				
Comch	26/38	30/42	*20/55*	*20/55*		
Receiver	*144/244*	*144/244*	*56/106*	*56/106*	**36/58**	46/68

Subsystem	**reduced**	**not reduced**
Timer, Transmitter, Comch	46/64	54/74
Transmitter, Comch, Receiver	90/126	114/152
Timer, Transmitter, Comch, Receiver	**62/86**	104/134

Table 3: Size of Intermediate Results

For this example, when composing a model without the intermediate reduction steps and instead reducing it after the last composition step, the same number of states and transitions are reached. The difference, however, lies in the size of the intermediate results and the associated higher execution times. While in this case, the benefit of applying the reduction algorithm were negligible due to the size of the example, theoretically it could lead to a significant difference in execution time.

6.1.4 Generating and Executing the Test Cases

The time taken to generate the test cases for all three subsystems in this example took less than five seconds. TDE/UML generated a total of 7 test cases for all three subsystems – one test case for the subsystem *TransmitterTimer*, three test cases for subsystem *ComchReceiver* and three test cases for *ABProtocol*. In contrast, an integration approach in which all four components were tested at once with the corresponding interfaces resulted in a total of 4

tests. In this case, the incremental integration test strategy resulted in more test cases being generated than the *big-bang* approach, but smaller integration steps usually result in a more stable system and a higher percentage of detected errors. An examination of the generated test cases shows that they are not free of redundancy or multiple coverage of communication transitions, but they come relatively close to the optimum.

7. Related Work

Over the years, there have been numerous papers dedicated to the subject of test data generation [1,3,8,13,17,19,21]. Moreover, a number of tools have been developed for use within academia and the commercial market. These approaches and tools have been based on different functional testing concepts and different input languages, both graphical and textual in nature.

However, few received any widespread acceptance from the software development community at large. There are a number of reasons for this. First, many of these methods and tools required a steep learning curve and a mathematical background. Second, the modeling of larger systems beyond single components could not be supported, both theoretically and practically. Third, the design notation, which would be used as a basis for the test design, was often used only in a particular application domain, for example, SDL is used predominantly in the telecommunications and embedded systems domain.

However, with the widespread acceptance and use of UML throughout the software development community as well as the availability of suitable tools, this situation may be about to change. Apart from our approach, we know of only one other effort in this area. Offutt *et al*. [12] present an approach similar to ours in that they generate test cases from UML Statecharts. However, their approach has a different focus in that they examine different coverage requirements and are only able to generate tests for a single component. Furthermore, they do not automate the test execution step in order for developers to automatically generate *and* execute their tests. In addition, they do not specifically address the problems and issues associated with modeling distributed, component-based systems.

8. Conclusion and Future Work

In this paper, we described an approach that aims at minimizing the testing costs, time and effort associated with developing customized test drivers and test cases for validating distributed, component-based systems.

To this end, we describe and realize our test generation and test execution technology and integrate it with a UML-based visual modeling tool. We show how this approach supports both the unit and integration testing phases of the component development lifecycle and can be applied to both COM- and CORBA-based systems. We briefly outline our implementation strategy and evaluate the approach using the given example. In the following paragraphs, we focus on some of the issues resulting from this work.

Software systems, especially embedded ones, use asynchronous communication mechanisms with message queuing or shared (global) messages instead of the synchronous communication mechanism adopted by our approach. Asynchronous communication is more complex to model, because it requires the modeling of these queued messages and events. Furthermore, communication buffers must be included, when modeling and composing. Dependent on the implementation, the size of the event queue can be limited or not. If not, mechanisms have to be implemented to detect the overflow of queues. When generating test cases for asynchronously communicating systems, the complexity may quickly lead to scalability problems that would need to be examined and addressed in future work. Methods for asynchronously communicating systems are presented in [5,9, 20].

Component interaction is modeled by our approach using an event (message) exchange containing no parameters and values. Future work will result in the modeling of 'parameterized' communication. To achieve this, the model specification must be enhanced with annotations about possible data values and types as well as test requirements for these values. TDE allows test case generation using data variations with samples out of a possible range of parameter values. Pre- and post-conditions can constrain valid data values. These constraints can be checked during test execution, which extends the error detecting possibilities.

UML allows users to model Statecharts with hierarchical state machines and concurrent states. While the global behavioral model presented in this paper can model components with nested states and hierarchical state machines, the internal data conditions of these state machines (meaning the global state machine variables) influencing the transition behavior are not supported. Concurrent states are also not supported as yet.

In future work, we hope to support the developer with an optimal integration test strategy. By examining the type and extent of the interactions between components, our environment could provide suggestions to the developer as to the order in which components need to be integrated. This could include analyses of the intermediate composition steps as well as an initial graphical depiction of the systems and its interfaces. Such an approach could significantly influence the effectiveness, efficiency and quality of the test design.

When modeling real-time systems, timing aspects and constraints become essential. In future work, we hope to analyze real-time modeling and testing requirements. For instance, test cases could be annotated with real-time constraints. Assertions or post-conditions within the model could also contain such information which could be checked during test execution.

9. Acknowledgements

We would like to thank Tom Murphy, the Head of the Software Engineering Department at Siemens Corporate Research as well as Professor Manfred Broy and Heiko Lötzbeyer at the Technical University, Munich.

10. References

[1] Balcer M., Hasling W., Ostrand T., *Automatic Generation of Test Scripts from Formal Test Specifications*, Proceedings of ACM SIGSOFT'89 - Third Symposium on Software Testing, Verification, and Analysis (TAVS-3), ACM Press, pp. 257-71, June 1990.

[2] Beizer, Boris, *Software Testing Techniques*, Second Edition. Van Nostrand Reinhold, 1990.

[3] Derrick J., Boiten E.A.: *Testing Refinements by Refining Tests*, ZUM'98 - The Z Formal Specification Notation, Springer-Verlag, pp. 265-83, Sept. 1998.

[4] Grasso Max, *Distributed Component Systems: The New Computing Model*, Application Development Trends, pp. 43-51, Nov. 1999.

[5] Henniger O., *One test case generation from asynchronously communicating state machines*, in: Testing of Communicating Systems Vol. 10, Chapman & Hall, Sept. 1997.

[6] Hoare C. A. R, *Communicating Sequential Processes*. Prentice Hall, 1987.

[7] Hopcroft J., Ullman J., *Introduction to Automata Theory, Languages and Computation*, 3rd Edition, Addison-Wesley, 1994.

[8] Ince D.C., *The Automatic Generation of Test Data*, The Computer Journal, vol. 30, no. 1, pp. 62-9, February 1987.

[9] Kim M., Shin J., Chanson S.T., Kang S.: *An Approach for Testing Asynchronous Communicating Systems*, IEICE Transactions or Communications, Vol. E82-B, No. 1, Jan. 1999.

[10] Matena V., Hapner M., *Enterprise Java Beans Specification, Version 1.1*, Sun Microsystems, Dec. 1999.

[11] Milner R., *Communication and Concurrency*, Prentice-Hall, 1st Edition, 1995.

[12] Offutt J., Abdurazik A., *Generating Test Cases from UML Specifications*. Proceedings of 2nd International Confererence on UML'99, Oct. 1999.

[13] Poston R., *T: The Automatic Test Case Data Generator*, Proceedings of 4th Annual Pacific Northwest Software Quality Assurance Conference, pp. 168-76, Sept. 1986.

[14] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W.: *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[15] Rumbaugh J., Jacobson I., Booch G.: *The Unified Modeling Language Reference Manual*, Addision-Wesley, 1999.

[16] Sabnani K. K., Lapone Aleta M., Uyar M. Ümit: *An Algorithmic Procedure for Checking Safety Properties of Protocols*. IEEE Transactions on Communications, Vol. 37, No. 9, Sept. 1989.

[17] Sarikaya B., *Protocol Test Generation, Trace Analysis, and Verification Techniques*, Proceedings of Second Workshop on Software Testing, Verification, and Analysis (TAVS-2), IEEE Computer Society Press, pp. 123-30, July 1988.

[18] Szyperski Clemens: *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley Longman Ltd., 1998.

[19] Tai K.C., *Predicate-Based Test Generation for Computer Programs*, Proceedings of 15th International Conference on Software Engineering (ICSE), IEEE Computer Society Press, pp. 267-76, May 1993.

[20] Waeselynck H. and Thevenod-Fosse P., *A Case Study in Statistical Testing of Reuseable Concurrent Objects*, Proceedings of 3rd European Dependable Computing Conference (EDCC-3), LNCS 1667, pp. 401-418, 1999.

[21] White L.J., and Sahay P.N., *A Computer System for Generating Test Data using the Domain Strategy*, Proceedings of SOFTFAIRII - 2nd Conference on Software Development Tools, Techniques and Alternatives, IEEE Computer Society Press, pp. 38-45, Dec. 1985.