Proceedings of the 2001 Winter Simulation Conference
B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, eds.

AUTOMATED OBJECT-FLOW TESTING OF DYNAMIC PROCESS INTERACTION MODELS

Levent Yilmaz

Simulation and Software Division
Trident Systems Incorporated
Fairfax, VA 22031, U.S.A.

ABSTRACT

This paper deals with the assessment of accuracy of simulation models from the perspective of dynamic object flows. Dynamic objects (also called temporary entities or transactions) move physically or logically from one model component to another and represent entities such as aircraft, data packet, passenger, and vehicle. Accurate flow (movement) of thousands or millions of dynamic objects within a complex simulation model significantly affects the overall model validity. We present a new automated testing technique for assessing the accuracy of dynamic object flows. The permissible sequence and precedence of dynamic object flows are specified using the context-free grammar formalism. The specification accuracy is assessed using a variety of verification and validation techniques. The executable model is instrumented and dynamic object flow trace data is generated. The trace data is automatically compared with respect to the specification and each dynamic object movement traced during model execution is automatically verified.

1 INTRODUCTION

The quality of a simulation model is assessed by using indicators such as accuracy, complexity, ease of use, execution efficiency, maintainability, reusability, and portability. This paper focuses on the assessment of simulation model accuracy. Verification, validation, testing, and accreditation activities primarily deal with the assessment of accuracy of simulation models.

Model verification is substantiating that the model is transformed from one form into another, as intended, with sufficient accuracy. Model verification deals with building the model right. Model validation is substantiating that the model, within its domain of applicability, behaves with satisaccuracy consistent with the simulation project objectives (Yilmaz 1998; Yilmaz and Balci 1997). Model validation deals with building the right model. Model testing is ascertaining whether inaccuracies or errors exist in the model. In model testing, the model is subjected to test data or test cases to determine if it functions properly. A test is devised and testing is conducted to perform either validation or verification or both. Some tests are devised to evaluate the behavioral accuracy (i.e., validity) of the model, and some tests are intended to judge the accuracy of model transformation from one form into another (verification). Accreditation is “the official certification that a model or simulation is acceptable for use for a specific purpose.’’ (DOD Directive 5000.59).

Many different views (e.g., control flow, data flow, structural) exist for testing model accuracy. More than 100 techniques proposed in the literature (Yilmaz and Balci 1997) describe these views. In this paper, we take the view of dynamic object flows (movements) in assessing model accuracy. A dynamic object (also called temporary entity or transaction) is an element of interest which moves physically or logically from one model component to another. It represents entities such as missile, patient, satellite, and ship. A complex simulation model may contain thousands or millions of dynamic objects and the accuracy of their movements significantly affects the overall model validity.

This paper is organized as follows. After an introduction in Section 1, Section 2 defines dynamic object flow sequence and precedence, and describes major stages of the proposed testing technique. Section 3 describes how dynamic object flows are specified based on context-free grammar formalism. Instrumentation of the executable model and generation of object flow trace data are presented in Section 4. Section 5 describes how the trace data are automatically compared with respect to the specification to judge the accuracy of dynamic object movements. Applicability of the testing technique is illustrated in Section 6 by some examples. Conclusions are stated in Section 7.

2 OVERVIEW OF THE TESTING TECHNIQUE

The proposed testing technique is intended to assess the accuracy of simulation models from the perspective of dynamic object flows. Successful application of the testing technique does not imply overall model validity but helps us increase our confidence in model accuracy in terms of dynamic object movements. Accurate movements of thousands or millions of dynamic objects within a complex simulation model significantly affects the overall model validity.

2.1 Process Interaction World-View

One of the fundamental conceptual frameworks, also called simulation strategy or formalism, of discrete-event simulation is the *process interaction* world-view. Process interaction framework enables a modeler to describe the life-cycle of an object that moves through the processes to engage in activities required by the system under study. (Balci 1988) provides a comprehensive review as well as implementation details of this simulation formalism and strategy, for which the developed testing technique is directly applicable.

2.2 Object-Flow Sequence and Precedence

Object flow sequence is defined as the sequential movement in space or time of a dynamic object throughout its lifetime in a simulation model. The sequence can be either repeating or terminating. A repeating sequence represents the flow of those dynamic objects that circulate within the model. A terminating sequence represents the flow of those dynamic objects that are destroyed at some point during model execution.

For example, in the clinic visual simulation shown in Figure 1, medical staff members (doctor, physician assistant, nurse, nurse practitioner) are created at time zero and the sequence of their movements in-between the rooms is repeated throughout the simulation. On the other hand, the patient flow sequences are terminating. Patients arrive randomly and after receiving service they leave the clinic (i.e., they are destroyed). Object flow precedence is defined as the flow ordering of dynamic objects within a particular model component. For example, in the clinic visual simulation in Figure 1, object flow precedence represents ordering of dynamic object movements such as (a) patient must move into an examination room before a medical staff member can enter into the room, (b) movement of patient to the receptionist must happen before the patient's movement to the check-in room that must be followed by a movement to any one of the examination rooms, and (c) a medical staff member must move to the medical staff room upon completion of examination and before moving in a new check-in or examination room to provide another service (Swisher *et al*. 1997).

2.3 Major Stages of the Testing Technique

Major stages of the automated object flow testing technique are shown in Figure 2. During the system analysis/investigation process, the system boundary is identified and the study objectives are explicitly defined. As part of this process, the testing technique requires the formal specification of permissible sequences and precedences of dynamic object movements to be included in the model representation. The formal specification is provided using the context-free grammar (CFG) formalism (Hopcroft and Ullman 1979) and is described in Section 3.1.

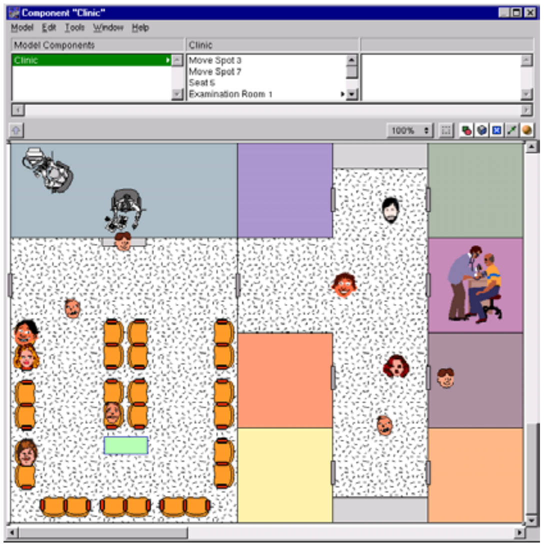

Figure 1: Dynamic Object Flows in a Clinic Model

The formal object flow specification must be verified and validated since it is the point of reference for judging the model validity with respect to the dynamic object flows. As the model development progresses, the object flow specification may be modified in view of new model representation requirements.

Once the executable simulation model is created, it is instrumented to generate dynamic object flow trace data (Section 4) in a format required by the technique. The trace data is automatically compared with respect to the specification and each dynamic object movement traced during model execution is automatically verified as described in Section 5.

3 OBJECT-FLOW SPECIFICATION

A simulation model is commonly structured as a hierarchical decomposition of components as shown in Figure 3. Components can be composite or atomic. An atomic component does not contain other components. A composite component consists of other atomic and/or composite components. A dynamic object, during its flow through the model components, drops into a deep component, interacts and engages in activities with atomic/composite objects in that component, and then ascends to the parent component.

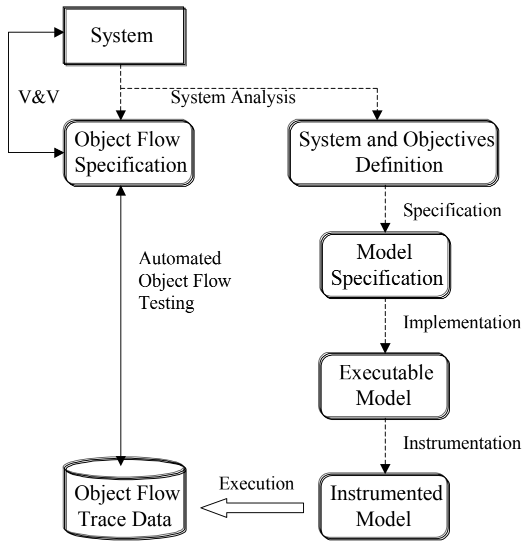

Figure 2: Automated Object Flow Testing Technique

For example, consider a simulation model of the internet in the US. The US map would be the top-level component decomposed into, e.g., virginia, texas, california. Virginia component could be decomposed into Virginia Tech campus, which is decomposed into computer science department network of host computers. An electronic mail message (a dynamic object) ascends from a host computer to the campus component, from which it flows into to Virginia state component, followed by an ascend operation to the top-level component. Then the dynamic object moves and drops into the California component, followed by a movement into the Stanford University component. In this component, the packet moves to local area network component to reach its destination host computer. This is an example of how a dynamic object flows through the model from one component to another. We propose to represent such object flows by using the Context-Free Grammar (CFG) formalism. CFG is used for a diverse set of problems in areas such as concurrent systems, databases, programming languages, and incremental compilers.

3.1 Object Flow Specification based on the Context-Free Grammar Formalism

In this section we explain how object flow sequences and precedences are specified using the context-free grammar formalism. The process of modular flow/precedence specification is illustrated using simple examples.

3.1.1 Object Flow Sequence Specification

A context-free grammar is a quadruple (N, T, P, S) where N is a finite set of variables, T is a finite set of terminal symbols, P is the set of rules, S is a special variable of N called the start symbol. The sets N and T are disjoint. The rules of a grammar consist of elements of the set. The rule $[R,w]$ is also written as $R \rightarrow w$. A rule of this form is also called an R rule. As a convention, the terminal symbols are denoted by lowercase letters or strings. The variable symbols (non-terminal symbols) are denoted by capital letters or strings.

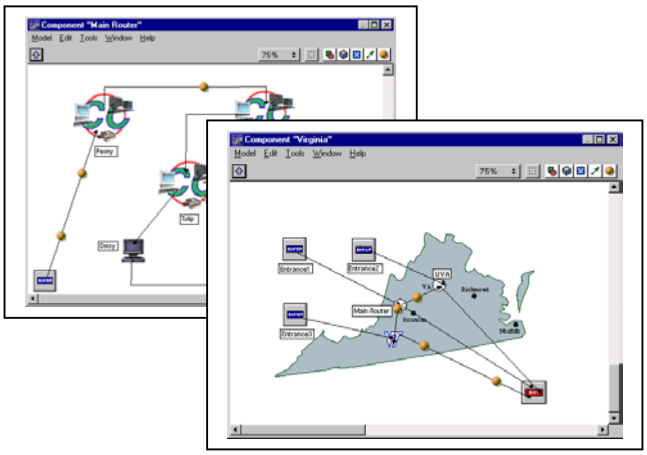

Figure 3: Hierarchical Model Decomposition

Grammars are generally used to generate properly formed structures or strings over the prescribed alphabet T. The basic step in generation of a well formed structure is the application of a rule to transform a structure. The transform operation begins with the start symbol S. Consider the application of $R \rightarrow w$ to the variable xRy where x and y are elements of the set $(N \cup T)^*$. The resultant string and the derivation operation is shown as $xRy \Rightarrow xwy$. Therefore, the transform operation replaces the variable symbol in a given string or structure with the right-hand side of the rule. We extend the CFG formalism by adding probabilities to each branch in the rule structure. That is, each rule of the form $[R,w]$ is now expressed as $[R, (w,p)]$ where w is the right-handside of the rule and p is the probability of using this rule. Suppose that R has the form $R \rightarrow w \mid \alpha$, where the probability of applying any of these two rules is 0.5. Then in our representation, R is expressed as $R \rightarrow [w, 0.5] \mid [\alpha, 0.5]$.

This extension enables us to check the accuracy of any probabilistic branching of dynamic objects during model execution. The following example illustrates the development of an object-flow specification. During the system analysis stage, the feasible flows of dynamic objects are identified using the flow circulation diagrams (FCD) (Hill 1996) In FCD two categories of objects exist. The first category represents the dynamic object flows by using arrows. Each arrow is labeled by the dynamic object name and the probability of taking this path. The other category refers to the components (deep or shallow) that manipulate and engage in activities with these flows and are represented by the circles.

Figure 4 illustrates the packet flows in a subsection of a global internet simulation model for USA using an FCD developed during system analysis. This section provides a simplified illustration of the flow of packets inside the Virginia state.

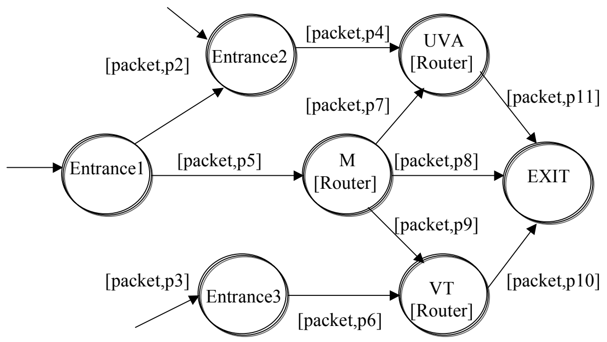

Figure 4: Flow Circulation Diagram

The packets can enter into the *VA* component from any of the three points: *Entrance1*, *Entrance2*, or *Entrance3*. The packets entering into the *VA* component from *Entrance1* flow into the component *M,* which denotes the main router. The packets ascending and showing at top of component *M* move either towards the component denoted by *VT*, to the component *UVA*, or to the *Exit* of the component. The packets dropping at *Entrance2* flow into *UVA* component and then possibly to the *Exit* of the component. Similarly, those packets dropping at *Entrance3* move into *VT* and then possibly to the *Exit* of the component unless VT is their final destination.

The flow specification of the packet class for the VA component is denoted by FS_{VA}[packet]=(*P, T, R, S*). The label *P* denotes the set of place names. A place name is in capital letters and can be a composite component, atomic component, entrance point, or an exit point. The label *T* denotes the set of terminal places which are the places of departure from the current component. The symbols in *T* are represented in lower case letters. Those symbols eventually refer to the component names visited during the flow of the dynamic object. *R* denotes the set of rules that express the flow constraints. Although each rule is associated with a probability, we do not include it in our representation by assuming that each rule has equal probability. This also increases the readbility and the clarity. However, we explain the usage of the associated probabilities when we introduce the flow sequence testing technique. Finally, *S* is the set of places where the dynamic object can start its flow in the current component. We begin developing the object-flow specification for dynamic objects belonging to the packet class, starting from the VA component, as follows:

Let FS_{VA}[packet]=(P, T, R, S) where
P={ENTRY,VA1,VA2,VA3,M,VT,UVA, EXIT},
T={m,vt,uva}
S={ENTRY} and
R is as follows:
ENTRY → VA1 | VA2 | VA3
VA1 → M
VA2 → UVA
VA3 → VT
M → (m . UVA | m . VT| m)
M → m. EXIT
VT → vt | vt . EXIT
UVA → uva | uva . EXIT

The above specification describes the permissible flow sequences allowed in the VA submodel of the US internet interconnection model. Rule 1 indicates that the dynamic objects (i.e., packets) enter into the composite component VA from three points, namely, VA1, VA2, and VA3. The components entering from the first entry flow into main router, called M. The packets are either routed to University of Virginia (UVA) or Virginia Tech (VT) model components. In this example, for simplicity and clarity, we assume all components are atomic. The packets entering from second and third entry points flow into UVA and VT, respectively, without using the main router. Packets arriving to UVA or VT components either reach their destinations or move onto the EXIT.

3.1.2 Object Flow Precedence Specification

Flow precedences can be specified similar to flow sequences. Therefore, flow sequence and flow precedence testing would be accomplished using the same testing technique. The following example illustrates the flow precedence specification for a simple hypothetical model component. Assume that we have developed an account component as part of a large bank database model. An account can receive open, withdrawal, deposit, and close requests from the user. We model each request as a separate dynamic object in the model. As the request objects arrive into appropriate account components, they engage into activity with these account components and they update the account component's state. The permissible sequence of requests are in the order of an *open* request*,* followed by any number of *withdraw and deposits*, which are terminated by a *close* request.

Let $FP_{ACCOUNT}$=(*V, DO, R, S*) be the flow precedence specification for the account component. *V* denotes the set of variable symbols that occur at the left-handside of each rule. *DO* denotes the set of dynamic objects that drop into the account component. *R* is the set of rules and *S* is the start symbol in the specification. Based on this definitions the flow precedence for the account object can be specified as follows:

V= {ACCOUNT, WD},
DO={open, deposit, withdrawal, close}
S= {ACCOUNT}, and
Rule Set R is as follows:
(1) ACCOUNT $\rightarrow$ open . WD . close
(2) WD $\rightarrow$ deposit . WD | deposit
(3) WD $\rightarrow$ withdrawal . WD | withdrawal

4 OBJECT FLOW TRACE DATA COLLECTION

In this section we describe the instrumentation process for collecting flow trace data from the model. We illustrate the instrumentation under the VSE platform. However, the ideas explained here are generalizable and applicable to other environments too. Here, we explain how the entities can be instrumented using the VSE environment which is used in substantiating the technique.

For the purpose of instrumentation we insert additional methods and modify the logic specification of the methods of dynamic objects and the components through which the objects flow. Each dynamic object keeps a list that contains the components that it visits throughout its lifetime. When it arrives to a particular component where it is destroyed or departed from the model, it writes this list to a flow trace data file. Similarly static components, through which objects flow and engage into activity, keeps a list of dynamic objects arrived.

As described above, each dynamic object keeps a list of components that it visits as an instance variable defined in its class declaration. When the dynamic object enters into the model, *EnteredModelatComponent* message is sent to the dynamic object to initialize its flow sequence list. Then the dynamic object starts its flow in the model. When the dynamic object arrives into a component, it sends the *dynamicobjectArrived* message to the component passing itself as the parameter. As the component receives the *dynamicobjectArrived* message it sends a message back to the dynamic object to insert itself into dynamic object's flow sequence list. This is handled by the *recordComponent* method of the dynamic object. The component also inserts the type of the arriving object into its flow precedence list for checking the correctness of the flow precedence sequence.

We also collect statistics to find the frequency that flow paths are traversed. For example, a priory, in the specification it is possible to associate each rule with the probability of using it. If the simulation of the system provides data that this is not the case, then it would be appropriate to examine the rules in the specification or the model implementation in more detail.

At the end of the model execution the flow trace data file contains the flow sequence paths of dynamic objects and the flow precedence list of each component. Each flow path contains a sequence of strings denoting the names of the components visited. The flow precedence list contains the names of dynamic objects that visited the component. Once the data is collected, all the flows are categorized with respect to their classes (i.e., sorted with respect to class names). The set of flow paths for a particular class is fed into the flow testing algorithm and checked for conformance to its specification.

5 AUTOMATED COMPARISON OF TRACE DATA AGAINST THE FLOW SPECIFICATION

In this section we propose a new dynamic and automated testing technique that checks the conformance of the flow paths obtained from the trace data with respect to the object flow specification. The technique tries to match the flow path to one of the feasible flow patterns that can be derived from the specification.

Given a flow specification FS_{Model} [DynamicObject] = (P, T, R, S), the label P denotes the set of place names. A place name is in capital letters and can be a composite/atomic component, entrance point, or an exit point. The symbols in T are represented in lower case letters. Those symbols eventually refer to the component names visited during the flow of the dynamic object. R denotes the set of rules that express the flows and S denotes the entry or start symbol.

In order to facilitate a basis for testing we define two types of flow paths:

- **Partial Flow Path:** A flow path, $fp \in (P \cup T)^*$, is a partial flow path if it contains at least one symbol from the set P.
- **Complete Flow Path:** A flow path, $fp \in T^*$, is a complete flow path.

Our testing technique continuously checks the compliance of the flow path obtained from the model execution with respect to the flow specification until a deviation occurs or a complete flow path is reached. Each rule in the specification transforms the initial variable, *ENTRANCE*, until a complete flow path is derived. Similar to the derivation concept in the context-free grammar formalism, if the derivation ransforms the first variable in a left-to-write reading of the partial flow path we call this *leftmost derivation.* Our technique exploits this derivation structure to provide a conformance checking procedure for flow paths collected during the model execution.

Before discussing the technique, we introduce the process of deriving a complete flow path from the given object flow specification. The flow path can be obtained from the specification by a derivation process that can be represented by a tree structure. We call the tree structure, generated by the flow path derivation process, a *flow path tree*. The flow path tree is an ordered tree and is built iteratively as follows. The root of the tree is initialized with the start variable (i.e., *ENTRY* point). If there exists a rule [R_i, $x_1x_2....x_n$] where x_i is a member of ($P \cup T$) and R_i is applied in the derivation to a partial flow path of the form uR_iv then $x_1, x_2, ..., x_n$ are added as the children of R_i. The *flow path tree* visualizes the derivation process described above.

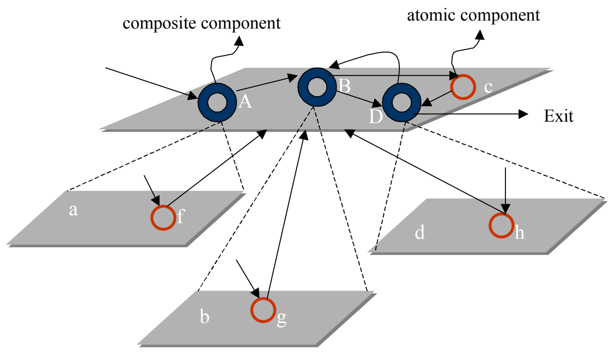

Figure 5: Hypothetical Object Flows

Existence of a flow path tree for a given flow path indicates the conformance of the flow path to the object flow specification. We demonstrate the flow path tree and flow specification graph constructs using a simplified hypothetical example. In this simple flow scenario, depicted in Figure5 , the following flow constraints are assumed:

(1) A dynamic object can flow through *A*, *B*, *C*, and *D* where *A*, *B*, and *D* are aggregate components and *C* is an atomic component. (2) A dynamic object can flow through *A*, *B*, and *D* and then continue flowing between *B* and *D* as many times as necessary. (3) A dynamic object flowing into *A* engages in activity with atomic component *f* and then leaves the component *A*. (4) In components *B* and *D* a dynamic object should flow into atomic components *g* and *h* (respectively. (5) Finally, all the flows depart from the *exit* section of the model.

Table 1: Object-Flow Constraints

(1) Entry → A	(8) G → g. ExitB
(2) A → Entry A	(9) ExitB → C | D
(3) EntryA → a . F	(10) C → c . D
(4) F → f . ExitA	(11) D → EntryD
(5) ExitA → B	(12) D → d . H
(6) B → EntryB	(13) H → h . ExitD
(7) EntryB → b . G	(14) ExitD → B | Exit (15) Exit → end.

Note that the specifications for separate components are combined to obtain aggregate flow specification for the dynamic object from its arrival to its departure. The constraints that are mentioned above are expressed using the flow specification as depicted in table 1.

Existence of a flow path tree, as shown in Figure 6, for a traced flow path indicates the correctness of the flow path with respect to the expected flow pattern depicted by the specification. However, checking the existence of a flow path tree is not trivial, since often there would be large number of applicable rules at a particular time during the derivation. When a rule reaches a dead-end, it would be necessary to backtrack to previous choices and continue rule applications until a complete flow path is achieved or until all the rules are exhausted.

To provide a solution to this non trivial conformance checking process we represent the leftmost derivations from the flow specification by a labeled directed graph, as shown in Figure 7, called the *flow specification graph* (FSG). The nodes of the graph are the partial and complete flow paths derivable from the specification starting from the initial variable that denotes the top level component. Our testing technique generates the FSG graph automatically by dynamically extending it until all feasible paths in the graph are checked or the complete flow path that is traced from the model is obtained.

Let *FS=(P, T, R, S)* be a flow specification. The FSG for this specification is a labeled graph, *FSG=(N, E, R)*, where the nodes and edges of the graph is defined as follows.

N={$fp \in (P \cup T) | S \Rightarrow fp$ by zero or more applications of the rules in R} and E={[u, v, A] $\in N$ x N x R | $u \Rightarrow v$ by the application of rule A}.

Thus the derivation of a flow path (partial or complete) *u* from the top level rule (i.e., *ENTRANCE*) is represented by the path from *ENTRANCE* to *u*. Thus, the problem of deciding the conformance of a complete flow path *fp* to the flow specification reduces to finding a path from *ENTRANCE* to *fp.*

The relationship between the flow path tree and the FSG for the hypothetical flow specification is shown in Figure 7. In particular, we show the derivation path for the sample flow pattern, ***"afbgcdh"***, where *a, b, c, d, f, g, h* are the names of the components in the model. The flow path tree for the flow constraints depicted in Table 1 is shown in Figure 6. The existence of a path from the starting rule to the final complete flow path is an indicator of the existence of a flow path tree for the given flow trace data. The proof of this fact is beyond the scope of this paper.

The FSG of a flow specification can take many forms. If there is only one derivation path for a given flow pattern, then the form of the graph is tree. However, if there exists more than one derivation path, then there exists a cycle in the graph. The existence of a cycle in the flow specification graph is an indicator of *ambiguity* in the specification.

Since the conformance checking problem is reduced to finding a path in the FSG graph of the specification, we perform a graph search operation as illustrated in Figure 8. However, note that although the graph is *locally finite*, that is, each node has finite number of incident nodes, the graph itself can have infinitely many number of nodes. This is due to the recurring patterns of flows and circular rules in the specification.

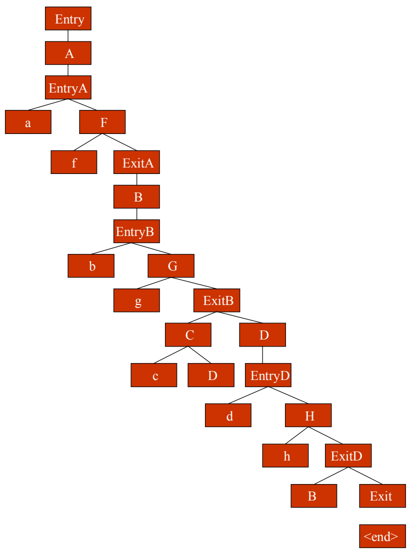

Figure 6: Flow Path Tree

Thus, the nodes of the graph are not constructed prior to the testing algorithm. The testing algorithm, as shown in Figure 8, builds the nodes of the graph as the paths are examined. The algorithm incrementally constructs the FSG and compares the flow path with respect to the paths examined in the graph. In order to facilitate backtracking in case a dead-end or infeasible path is selected, a stack mechanism is used to keep track of the choices made so far. The technique constructs paths in the graph by applying the rules to the leftmost variable in the partial flow path examined so far. The string u contains the component names and is the prefix of the partial flow path uR_iv. If the component sequence stored in u does not occur in the flow path then the algorithm reaches a dead-end and should backtrack to an earlier path and choose another rule to continue examining the paths.

The rules are applied in the order they are numbered in the flow specification.The algorithm consists of two loops. The interior loop extends the current path by applying a rule to the final node in the path. The loop terminates when the unreachable condition holds (i.e., dead-end is reached) or the most recently completed derivation obtains a complete flow path which contains only the component names.

When a dead-end is encountered the backtracking mechanism starts by popping the stack. When the stack is popped, the partial flow path at that node is restored and the next applicable rule is applied to continue path construction in the FSG. However, if the loop terminates due to the derivation of a complete flow path the algorithm return the accept message indicating that the flow trace data is correct.

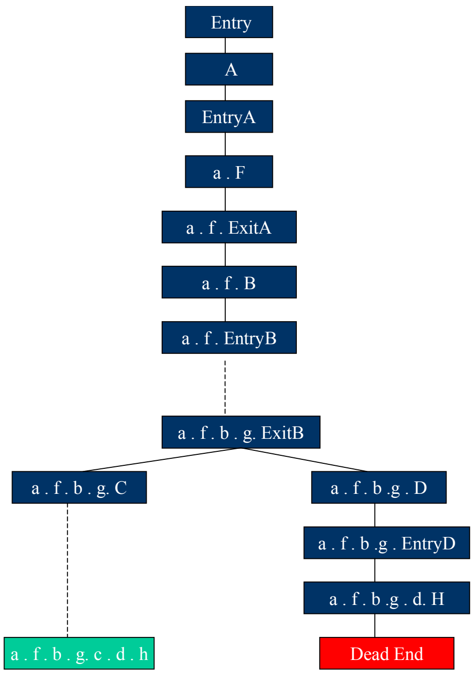

Figure 7: Derived Flow Specification Graph

```
Flow Sequence/Precedence Testing Algorithm:
 Input:  Flow specification FS=(V, C, R, TL)
         flow path fp and stack S
   ruleno=0;
   S.Push([TL,ruleno])
   repeat
      [current_path, ruleno]=S.Pop()
      unreachable=false
      repeat
         Let current_path=u$R_i$v where $R_i$ is the leftmost variable
          if u is not a prefix of fp then unreachable=true
          if there are no $R_i$ rules numbered greater than ruleno then
            unreachable=true
          if not unreachable then
            Let  $Ri \rightarrow w$ be the first rule with $id > ruleno$
                      and let j be the id of this rule
            S.Push([current_path,j])
            current_path=concatenate(u,w,v)
            ruleno=0
         endif
      until (unreachable) or (current_path is a complete flow path)
   until current_path=fp or S.EMPTY()
   if current_path=fp then accept else reject
end.
```

Figure 8: Flow Sequence/Precedence Testing Algorithm

6 ASSESSMENT OF THE TESTING TECGNIQUE

In this section we assess our testing technique by arguing about its error detecting capability. To illustrate, we show the applicability of the technique on a sample simulation model. The simulation model is seeded with specific types of errors and flow trace data is collected. Then we perform automated object flow testing on the collected data.

We consider two main types of errors:

(1) *Causal Flow Sequence Errors:* The model is said to have causal flow sequence error if there is a deviation in the causal sequence of flows with respect to the specification. For instance, the component to which the dynamic object moves might be a permissible component; however, it appears in the flow trace in the wrong causal order. Furthermore, the model is said to have causal flow sequence error if there exists flows which are not included in the flow specification. This may be due to either model implementation error or incomplete flow specification.

(2) *Missing Flow Sequence Errors:* The model is said to have missing flow error if the executable model does not generate flow traces that exercise the flow patterns specified in the flow specification. This may be due to missing flows in the model implementation. That is, some object flows that are feasible in the real world domain are not implemented in the computerized model.

Our technique catches all possible flow sequence errors in the model. However, to catch missing flow sequence errors it is necessary to enforce coverage criteria on the specifications. That is, flow trace data collection continues until the specification is covered to some certain degree by the flow traces obtained from the model execution. If the coverage criteria is not fulfilled after certain number of replications model tester analyzes the model statically to find the missing flows by checking the sections of the model that are not covered according to the coverage criteria. We define coverage criteria on the FCG and FSG representations. There are three criteria.

(1) *FCG Node Coverage*: All the nodes in the FCG should be covered. Since the nodes of FCG refer to the rules in the flow specification this coverage criteria requires the application of each rule at least once. Node coverage criteria is analogous to statement coverage in structural testing.

(2) *FCG Edge Coverage*: All the edges in the FCG should be covered. Since the edges refer to links among rules, this criteria requires all interactions among incident rules to be covered. This criteria is analogous to branch coverage criteria of structural testing.

(3) *FSG Node Coverage*: The FSG node coverage criteria requires the coverage of all possible causal sequences to be checked. Although, the specification graph is locally finite, there can be infinitely many paths in the graph due to circularity in the specification. That is, specifying recurring patterns of flows requires rules which reference themselves. Thus in a sense FSG node coverage is similar to path coverage in structural testing.

7 CONCLUSIONS

One of the fundamental conceptual frameworks, also called simulation strategy or formalism, of discrete-event simulation is the *process interaction* world-view. Process interaction framework enables a modeler to describe the life-cycle of an object that moves through the processes to engage in activities required by the system under study. The object-flow testing technique, introduced in this paper, outlines a new V&V approach that is particularly applicable to dynamic process interaction based discrete-event simulation models. The technique is based on a new innovative method used to specify and verify the expected flow patterns of the objects that engage in activities with the processes of the system under study. Therefore, the method is applicable to all models that has dynamic objects or transactions that flow through the model components. We have described the overall methodology for the application of the technique, the necessary formalisms and associated validation decision procedures, and the error detecting capability of the approach. Causal flow sequence and missing flow detection are the primary types of errors detected by the technique. Three coverage criteria are discussed to determine the conformance of the model with respect to expected/existing system flow patterns depicted by flow pattern specifications.

ACKNOWLEDGMENTS

The research leading to this paper is performed by the author while he was affiliated with the Simulation and Software Laboratory of Computer Science department at Virginia Tech. The author wishes to acknowledge the comments of Dr. Balci, whose suggestions facilitated the utilization of Visual Simulation Environment (VSE) as a testbed for this testing technique.

REFERENCES

Balci O. (1988) “The Implementation of Four Conceptual Frameworks for Simulation Modeling in High Level Languages,” *In Proceedings of the 1988 WSC.*

Hill, R. C. D. *(1996). Object-Oriented Analysis and Simulation*. First Edition, Addison Wesley.

Hopcroft and Ullman (1979). *Introduction to Automata Theory: Languages and Computation.*

Swisher, J.R., B. Jun, S.H. Jacobson, and O. Balci (1997), “Simulation of the Queston Physician Network*,” In Proceedings of the 1997 Winter Simulation Conference*, IEEE, Piscataway, NJ, pp. 1146-1154.

Yilmaz L. and O. Balci (1997). "Object-Oriented Simulation Model Verification and Validation," *In Proceedings of the 1997 Summer Computer Simulation Conference*. July 13-17, 1997.

Yilmaz L. (1998). "A Taxonomical Review of Object-Oriented Simulation Model Verification and Validation Techniques," Technical Report TR-98-6. Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Va. (March).

AUTHOR BIOGRAPHY

LEVENT YILMAZ is a senior research engineer at the Simulation and Software Division of Trident Systems Incorporated. He received a B.S degree in Computer Engineering and Information Science from Bilkent University and M.S. degree in Computer Science from Virginia Polytechnic Institute and State University, where he is currently completing his Ph.D. His research focuses on simulation model interoperability and the verification, validation and testing of object-oriented and component-based models. Mr. Yilmaz has recently worked as a PI or co-PI for several NavSea and NavAir V&V business innovation research projects and provided services as an IV&V contractor. Mr. Yilmaz is a member of ACM, IEEE Computer Society, SCS, and Upsilon Pi Epsilon.