Development of Verifiable Programs - Application of an Approach based on Executable Object-Oriented Specifications

Natasha Sharygina[1,3], James C. Browne[2], and Delbert Tesar[3]

[1] *Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ, USA 07974*
natali@research.bell-labs.com
[2] *The University of Texas at Austin, Computer Science Department, Austin, TX, USA 78712-0050*
browne@cs.utexas.edu
[3] *Robotics Research Group, The University of Texas at Austin, Austin, TX, USA 78712-1100*
{natali,tesar}@mail.utexas.edu

Abstract

Combining validation by testing with verification by formal methods offers great potential for development of robust and reliable object-oriented software systems. However, formal verification cannot be readily applied to software developed with conventional object-oriented development methods. This paper presents the first phase of a two-phase approach for development of object-oriented software systems which combines validation of OOA models formulated in xUML (an executable subset of UML[18],[24]) with formal verification through model checking. The second phase, application of model checking to the validated OOA model has been presented separately [22]. Model checking is accomplished by translating the validated OOA model to the COSPAN, automaton-based model checking system [5].

This paper defines, describes and illustrates an OOA model construction process leading to OOA models which can be both validated and verified. This process leads to efficient designs, which minimizes complexity of the resulting computer-controlled systems. Since OOA models in xUML are executable they can be validated by testing using simulation. Finally, since the OOA model complexity level is much less than the procedural programs to which they are translated, model-checking can be applied to the validated OOA model.

We demonstrate the approach by applying it to reengineering of a NASA robotic system where testing and maintenance had been obstructed by complexity. A comparative analysis between the original robotic system, that was constructed following the Booch methodology [1], and the redesigned system is given. A number of inefficiencies and flaws in the original design which would have precluded model checking were found. An OOA model to which model checking has been successfully applied resulted from the redesign.

Keywords: Object-Oriented Design and Analysis of Complex Software Systems, Formal Verification, Design for Verification, Integration of Model Checking with Object-Oriented Techniques.

1 Introduction

Formal methods, in particular model checking, are increasingly being used to automatically establish the correctness of (and to find flaws in) finite-state systems, such as descriptions of hardware and protocols. Most software systems however are not directly amenable to finite-state verification methods. The state spaces of procedural programs are typically dynamic and unbounded. Abstract models of the programs to which model checking can be applied are not validated so that model checking is impeded by an explosion of errors. In this paper we present and apply the first phase of a two-phase OOA-based methodology (**Figure 1**) for software development based on representation of the software system as an executable OOA model, validation by simulation, application of model checking to the OOA model and translation of the validated and verified OOA model to a procedural language. The integrated environment is built on the xUML (xUML [18] is a subset of UML [19] with an executable semantics) version of the Shlaer-Mellor OOA (SM OOA) methodology [24] and the COSPAN, automaton-based model checking system [5].

The integrated methodology is based on the following premises:

- the validated and verified software system must be fully functional (must be developed in an executable form) and efficient;
- verification must be executed on an abstract model since a program which is fully functional and efficient will be far too complex for application of formal verification;
- verification is practical only when the design model has been validated by conventional testing and is largely correct;
- verification can therefore only be applied to a validatable (executable) specification;
- the abstract model must therefore be validated before verification can be applied;
- therefore, code development must be based on an executable model specification;
- xUML is an executable specification language;
- if verification is to be meaningful, the executable specification must be compilable to an efficient final program.
- xUML is compilable.

This paper reports results of a project that has been conducted at Bell Laboratories and The University of Texas at Austin (where the ideology and support tools were developed) and The Robotics Research Group (where the methodology was applied). The project has been split into two phases:

Phase 1: OOA-based modeling leading to a validated and verifiable OOA model;

Phase 2: Verification based on the validated OOA model leading to the system with proven properties.

This paper covers Phase 1 of the project. Phase 2, the OOA-based verification approach and its application are presented in [22].

The rest of the paper is organized as follows. Section 2 presents the integrated OOA and model checking environment. Section 3 defines the problem of executable OO-modeling that leads to construction of verifiable models. Section 4 demonstrates results of application of the OOA modeling as a part of application of the integrated methodology to redesign of a complex robotic system. Conclusions and related work end the paper.

2 Development Methodology

We present an OOA-based methodology that provides a formal basis for developing reliable programs. It integrates formal verification with software development, and institutes mechanisms for checking design completeness and consistency. Formal verification by model checking is applied to OOA models that have executable semantics specified as state/event machines rather than to programs in conventional programming languages.

The program development and maintenance based on our approach involves the following steps:

1. System modeling. A system is constructed in the xUML representation following the design rules defined in Section 3.

2. *The OOA model validation.* Execution behavior of the designed model is validated by simulation with a discrete event simulator in terms of state consistency, concurrency among state model instances, proper event generation and consumption, and values of attributes determining the path through the state model.

***3. Manual source to source transformation of the model to verifiable form*.** Design rules (reported in [22] that lead to the construction of the models with tractable state spaces are applied to the xUML model.

4. Specification of the system predicates. The desired properties to be verified are specified by the developer.

5. Formal Verification. The OOA-models are transformed into the syntax accepted by a model checker and formally verified against the specified properties. If errors exist, changes are made manually modifying the OB graphical model. Steps 2-5 are repeated until all desired properties hold.

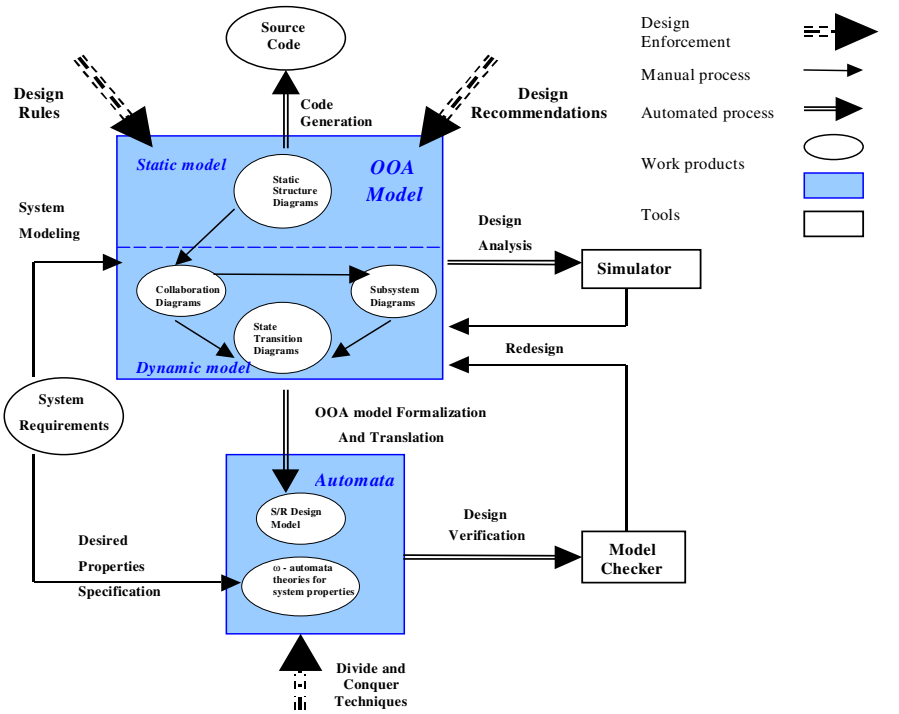

Figure 1. The OOA and Automata-based Model Checking Software Development Environment

6. Target system development. Requirements of a target system are specified. Automatic translation of the xUML model into executable code of the target application is performed.

The automation of our approach is provided though implementation of the OOA method, SES/Objectbench system [21] (OB), automatic code generator, SES/CodeGenesis [20], automata-based model checking tool, COSPAN [5], and the translator [25] that performs automatic translation of the OOA models into automaton models.

2.1 OOA Modeling with Executable Semantics

We focuse on a subset of the SM-OOA notation that defines an executable subset of UML (xUML). Use of OOA models with executable semantics is moving into the mainstream of OO software development. The Object

Management Group [18] has adopted a standard action language for the Unified Modeling Language (UML) [2]. This action language and SM-OOA semantics represented in UML notation define executable an executable subset of UML (xUML).

xUML notation is suitable for modeling of objects, subsystems, their static structure, and their dynamic behavior. *Object models* are class diagrams, showing the classes and relationships between classes. A class diagram gives the fields and methods of a class. The execution behavior of a class is described by a state machine. Transitions between states are driven by events.

The concepts used during the domain engineering are *class*, *state*, and *event*. Classes used in the design are C++ classes. A state of an object is defined as having one action and one state transition table. Since the action is considered to be associated with the receiving of an event by the object, this is most like a Moore state machine. The execution of the action occurs after receiving the event. A transition table is a list of any number of events, and "the next" states that are their result. Events have an arbitrary identifier, a target object, and associated data elements. The event class supports the event's enqueing and dispatching.

The communication between state models of the system supported by the SM methodology is asynchronous interleaved.

2.2 Automata-based Model Checking

In the methodology presented in Figure 1 the OOA modeling provides the construction of validated executable specifications and model checking assures that the executable specifications meet the desirable requirements. Model checking [3] is a procedure that checks that a given system satisfies desired behavioral property through exhaustive enumeration of all states reachable by the system. When the design fails to satisfy a desired property, a counterexample is generated, which is used to identify the cause of the error.

The semantics model of the COSPAN mode checker, used in this project, is based on ω-automata [12]. The automaton is defined in terms of a directed graph with a finite set of vertices called *states*, some of which are designated *initial*; a transition condition for each directed edge or *state transition* in the graph; and an *acceptance condition* defined in terms of sets of states and state transitions. Each transition condition is a nonempty set of elements called *letters* drawn from a fixed set called the *alphabet* of the automaton. There are several different equivalent definitions of acceptance in use. The acceptance condition of ω-automaton can capture the concept of something happening "eventually" or "repeatedly", "forever" or "finally" (forever after). The set of subsets of the alphabet forms a Boolean algebra *L* (*Language*) in which the singleton sets (*words*) are atoms. COSPAN programs are written in S/R, which is a declarative, data-flow language. The S/R language supports both synchronous and asynchronous coordination of components (including interleaving and non-interleaving semantics). The semantic model underlying S/R consists of coordinating components defined as interconnected Mealy- or Moore-like finite state machines each of whose respective outputs is a (non-deterministic) relation of its input and state. Moreover, the semantic model supports automaton acceptance conditions, giving S/R the expressive power of ω-regular languages.

A straightforward approach to automata-based verification is to follow Kurshan's methodology [12]. We represent the system's model, its components and properties to be verified as respective *L*-processes, as explained in [12]. Coordination in a system of independent *L*-processes P_1, ... P_k is modeled by behavior of the product *L*-process P

$$P = P_1 \otimes P_2 \otimes ... \otimes P_k \qquad (1)$$

Each component P_i imposes certain restrictions on the behavior of the product P, and the behavior of P is the "simultaneous solution" for behavior of the respective components. This is equivalent to the intersection of the respective language,

$$L(P) = L(P_1) \cap ... \cap L(P_k) \qquad (2)$$

The formal definition of verification of property *T* for a program *P* is the automaton language containment check

$$L(P) \subset L(T). \qquad (3)$$

In words, this says that all behaviors of the program, P are behaviors "consistent with" (i.e., "of") the property, T.

3 The OOA-based Modeling Constraints

This paper defines and describes the problem of model construction and validation during the OOA design and analysis step. We identified three types of the OOA design constraints, which reduce complexity of the application domain during the design step and enable practical application of model-checking to verification of programs. These are design *efficiency, dependability* and *verifiability*. The OOA structural design principles and rules that support implementation of these constraints were developed and integrated within the proposed development methodology.

1. Design Efficiency. In the modeling system requirements, the first step is to capture the static and

dynamic structures of the system by abstracting objects, and their relationships in collaborating to perform a task. The object models are "data intensive" while "state transition diagrams" which present objects lifecycles are control intensive. Both the data and control parts of the system have to capture the system's functionality in a manner that ensures efficient coordination of functional elements of the system.

Design principle: *A component is the appropriate design unit to capture the system's functionality (not a class).*

This principle defines how the *functionality* of the system has to be captured. The following rules enforce efficient designs:

Efficiency Design Rules:

1. *No functional operations are defined outside of the component.*

Functionality is to be fully captured by the component.

2. *No redundant operations within a component.*
3. *If an algorithm requires evaluation of multiple instances of an object, construct the design, which creates and executes the instances concurrently, assuring that all operations on these instances belong to the same functional component.*

Assuming a sequential type of evaluation of multiple choices can erroneously lead to distribution of functionality between the system's components, or lead to the construction of unnecessarily complex components.

4. *Allow minimal cyclic dependencies between objects of the component*.

Functionality of the system should be captured in a way that assures that dependencies between objects of a component are minimized.

5. *No cyclic dependencies between the components*.

Full independence of the components of the system should be provided.

2. Design Dependability. After the system design is constructed, the next step is to assure that the design is consistent. "Internal" and "External" consistency is checked. All design fragments are "internally" consistent if they contain no syntax action-language errors, action language compilation errors or link errors. "External" consistency is defined by execution behavior of the system components and it is evaluated in terms of state consistency, concurrency among state model instances, proper event generation and consumption, and values of attributes determining the path through the state model.

The system is dependable if:

- it is "internally" and "externally" validated;
- its design manages complex concurrent activities and support error resolution among multiple interacting objects.

The first statement is assured by simulating the model behavior with a comprehensive set of test scenarios. Testing of the analysis model is performed by specifying pre- and post-conditions for the actions of the system state machines. Pre- and post-conditions give us the ability to stop the simulation when unexpected conditions occur. The event tables with complete specification of allowable events are specified and allow generation of the code for catching incorrect event sequences.

The second statement is assured by implementation of the following principle:

Design Principle: *Hierarchical ownership of lower-level objects is the appropriate style of the design.*

This principle defines how the *structure* of the system has to be organized. The following rules enforce this principle:

Dependability Design Rule:

6. *Create a process that controls the execution of other running processes.*

A process that controls the processes execution is called a *controller*. It controls cooperative concurrency and implements coordinated and disciplined error recovery, and maintains the consistency of shared resources. The controller has a set of states that are activated by the participating processes which cooperate within the component. Logically, the lifecycle of the controller starts when all processes have been activated and finishes when all of them reached the end of their lifecycles. The controller actions coordinate the processes following the specified control algorithm of the system, implement an error recovery if the coordination is not possible or propagate a failure exception to the participating processes.

3. Verifiable Design. The xUML models naturally complement to the application of the model checking techniques due to the fact that their complexity level is far less than the procedural language programs to which they are translated. They provide finite state representation of the systems design, abstraction of implementation detail prior to the actual code generation and support hierarchical system representation. For example, data representation of the xUML objects is expressed in terms of fundamental types (integer, double, char, boolean) and simple data structures (arrays). No complex data structures such as stacks, binary trees, etc., are used. Instead, if some entities require complex data structures to capture their functionality, relationships among objects, which include associations, composition aggregations,

generalizations and specializations, maybe used to represent such data structures. Control operations between objects, on other hand, are expressed as signals without reference to the internal states of the objects (function calls as used in C, C++ are substituted by events between processes).

If model checking applied to the OOA models encounters the *state-space explosion problem* (which is the case for complex computer-controlled systems), a higher level of decomposition, along with abstractions and restrictions at the design level are required. Our approach initiates the use of design rules and recommendations (which are reported in [22]) for construction of the OOA models with tractable state spaces. These so called *verifiability design rules* are aimed at the development of the systems that support divide-and-conquer methods of model checking though implementing event mechanisms that allows to avoid coupling of internal states of classes and to develop fully self-contained units.

4 The Case Study

We applied the integrated OOA and model-checking methodology to reengineering of a robotic decision-support software system [10] which testing and maintenance were obstructed by its complexity.

4.1 Robot Redundancy Resolution Problem[1]

A software system used for robot redundancy resolution was examined. A robot is considered redundant if it has more independent joints than a number of independent variables specifying position and orientation of the robot's end-effector (which equals to 6). For a redundant robot an *infinite* number of the robot's joint displacements can lead to a definite end-effector position. **Figure 2** shows a redundant robot demonstrating this phenomenon. Failure recovery is one of the examples of redundancy resolution application: if an actuator fails, the controller locks the faulty joint and the redundant robot continues operation. Multiple performance criteria [11] are used to solve a decision-making problem of choosing among all possible joint configurations.

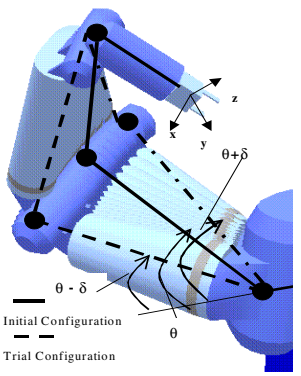

Figure 2. Plane Geometry of a Redundant Robot, Representing a *Simple* Exploration Pattern of the Direct Search Technique

A decision-making algorithm called direct search [10] has been redesigned in this project. It is based on the concept of a joint-level perturbation (or joint explorations). Perturbation at the joint level means

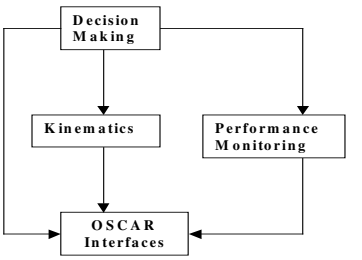

Figure 3. Functional Layout of the Robotic Decision Support System

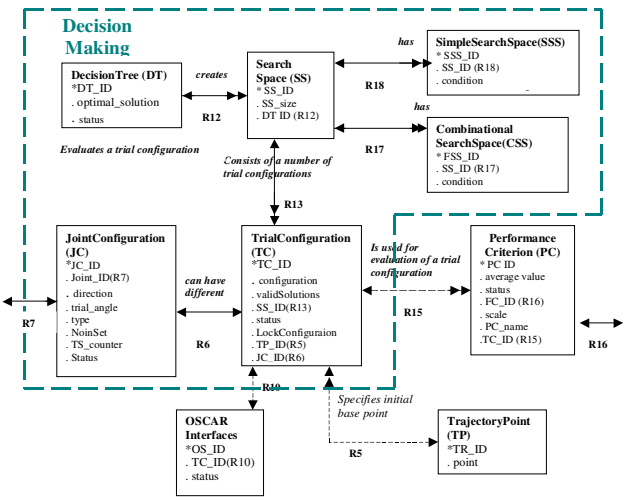

Figure 4. Object Information Diagram of the Decision Making Component

temporarily changing one or more of the joint displacements in either the positive or negative direction. This project focuses on two different exploration strategies: "simple" and "combinational". "Simple" exploration describes a strategy of perturbing only one joint displacement at a time. This exploration pattern finds the individual influence of each joint displacement on the arm configuration at any fixed end-effector position. "Combinational" perturbation pattern explores the effect of both individual and simultaneous displacements of redundant joints. **Figure 2** demonstrates a "simple" exploration for one of the robot arm joints with θ - being a joint angle and δ - being a displacement.

Direct search is a method of solving problems numerically using sets of trial solutions to guide a search. The search begins with an estimate of the solution which

[1] Refer to http://robotics.utexas.edu/rrg/glossary for robotic terms

represents an initial arm configuration and serves as an initial base point for the search. Explorations about the base point are generated to produce a set of trial configurations. The second phase of the search is application of performance criteria to the set of candidate solutions. Criteria fusion is used to assure a balance among the criteria. A decision-making process selects one trial configuration as the next base point. Explorations are then performed about this new base point. This process of explorations and decision-making continues until the search finds an acceptable solution.

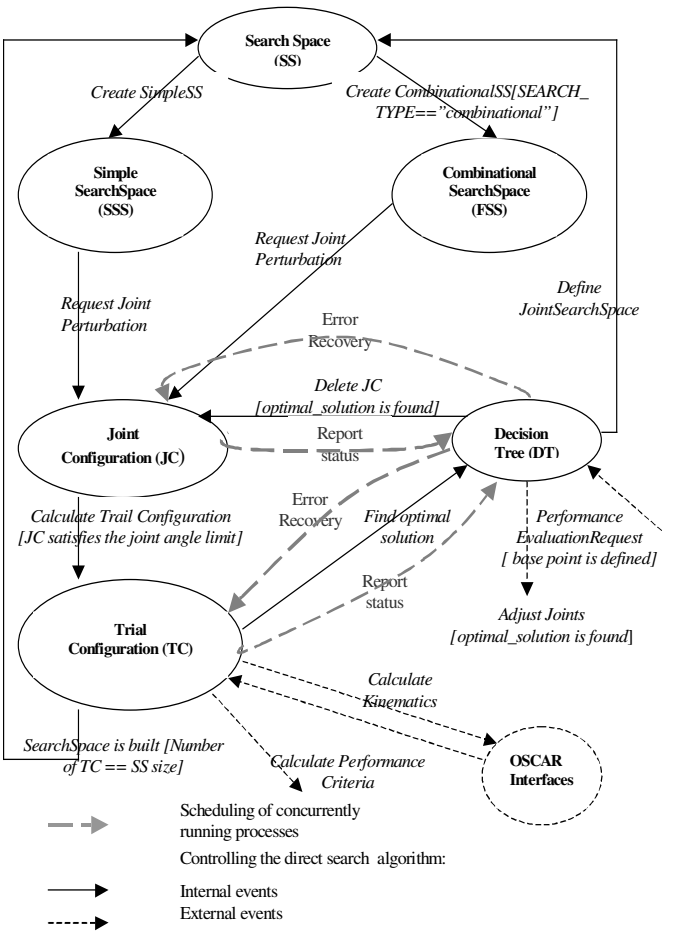

Figure 5. Collaboration Diagram of the Decision Making Component

4.2 Rigorous Specifications

The robotic decision support system was rigorously specified using the xUML notation. We present the static and dynamic structures of the robotic system, which development followed the efficiency and dependability rules defined in Section 3.

4.2.1 Static Structure. The robot system was designed as a collection of four major functional components: *OSCAR Libraries, Kinematics, Performance Monitoring* and *Decision Making*. Each component consists of a number of tightly coupled, cohesive objects. *OSCAR Libraries* unit defines interfaces to the computational libraries of the existing robotic software, OSCAR [10]. The *Kinematics* unit captures the robot motion control algorithms. The *Performance Monitoring* unit specifies the mathematical description of the robot performance criteria, and criteria scaling and fusing algorithms. The *Decision Making* component formalizes the decision-making algorithms. The conformance to *Rule 5* is illustrated in **Figure 3.** You can see that there are no cyclic dependencies among the system components.

The architectural design of the *Decision Making* component is presented in **Figure 4**. Conceptual entities of this unit are defined by the following objects: *DecisionTree (DT), SearchSpace (SS)*, *SimpleSearchSpace (SSS)*, *CombinationalSearchSpace (CSS)*, *JointConfiguration (JC)* and *TrialConfiguration (TC)*. The *Decision Making* component interfaces the *PerformanceCriterion (PC)* object of the *PerformanceMonitoring* unit and the *TrajectoryPoint (TP)* object of the *Kinematics* unit (the interfacing relationships are depicted by dashed lines in Figure 4).

Semantics of the objects is defined by attributes and relationships. Attributes of the *JC* object are illustrated below. Each instance of the *JC* object represents orientation of a joint displaced from its base position. *JC_ID* is a key attribute whose value uniquely distinguishes each instance of a *JC* object. *Joint_ID* is a referential attribute, which represents formalization of the relationship R7 and is used to tie an instance of the *JC* object to an instance of the *Joint* object which belongs to the *Kinematics* unit (if we change the value of *Joint_ID* from 1 to 2 it means that we are evaluating configuration of the second joint instead of the first one). *Trial_angle*, *NoinSet* and *TS_counter* are descriptive attributes and they provide facts intrinsic to the *JC* object. For example, *trial_angle* is used to keep the value of the joint angle that changes during exploration about the base point. *Status*, *type* and *direction* attributes are so called naming attributes which provide facts about the arbitrary labels carried by each instance of an object. The domain of the naming attributes is specified by enumeration of all possible values that the attribute can take on. *Status* represents status of the object instance, in other words the names of all states of the object's state machine. Attribute *direction* domain is {"+delta","-delta","comb+delta","comb-delta"}. Attribute *Type* domain is {"simple","combinational"}.

The executable model of the *Decision Making* unit is

defined using a binary type of relationship. An example of one-to-many binary *SearchSpace-TrialConfiguration* relationship states that a single instance of the *SearchSpace* object *consists of* many instances of the *TrailConfiguration* object.

4.2.2 Dynamic Structure. The Collaboration Diagram

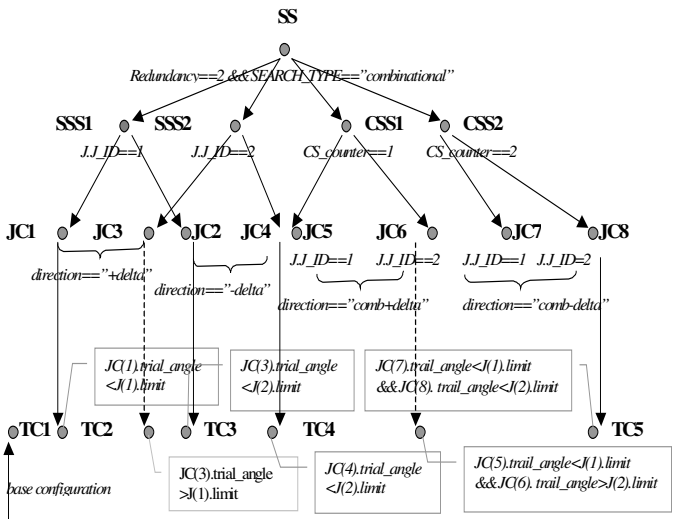

Figure 6. SearchSpace Construction for a Robot with Two Redundant Joints using *Combinational* Exploration of the Direct Search Decision-Making Technique

(CD) presented in **Figure 5** displays the flow of events and external behavior of the *Decision Making* component in terms of a sequence of data or control signals exchanged among the objects in the system. A transition on the CD is represented as an arrow originating from a source object to a destination object. Transitions are labeled in the following form:

Event[cond],

where *Event* names the transition and *cond* is a predicate for actions.

The following control algorithm of the direct search method has been designed. When request for explorations of joint displacements is scheduled, a *SS* is built. The *SS* initiated the construction of either *SSS* or *CSS* depending on the exploration strategy. During the design we revealed the fact that the algorithm for construction of the *SSS* can be reused during construction *of CSS* and combined these two exploration strategies into a single algorithm. In general there are *n SSS* where *n* is the number of joints of the robot arm and two *CSS* (one for positive and one for negative displacements of all redundant joints). The *SSS/CSS* object generates sets of *JC* around the provided base point. *JC* instances initiate calculation of *TC* instances (each *TC* basically is a vector presenting orientation of ALL joints of the robot arm) for any perturbed joint. *DT* selects the best *TC* given a set of *PC* and a number of physical constraints that are globally defined by the user. The found solution serves as the next base point for another trial of joint-level explorations. The search stops when no new solutions are found. The system returns control to the *Kinematics* unit, which continues moving the robot arm and determines a new base point for the search.

The CD illustrates how *Rule 6* was implemented. *DT* process is the action scheduler, it controls the movement of the robot arm, checks the status of the concurrently running instances of *JC* and *TC* objects, performs the error recovery operations if no solution for the robot control exists, or there is a fault in the coordination of concurrently running processes.

To demonstrate how *Rule 3* was implemented during the design of the *Decision Making* component we present a sample example. A robot arm consisting of eight joints is used in the example. We change joint angles of two redundant joints (we consider redundant joints to be counted from the base of the robot) and calculate the robot arm configuration given that an end-effector position has not changed. We use the "combinational" exploration pattern of the direct search technique.

Design Model of the Direct Search Method. The implementation of the algorithm for the *SS* construction is shown in **Figure 6**. A tree of all possible processes active during the construction of the search space is presented. Each node represents an instance of the system's object. Arrows indicate events that lead to the creation of new instances of the system's objects. Actions associated with each node represent post-conditions. Post-conditions are used to visualize the reasoning of each transaction for all objects except for the *TC* object. For the *TC* object *pre*-conditions (in squares) are used.

In our example the *SS creates* two *SSS* and two *CSS* (since there are two redundant joints) instances. Each of them generates sets of *JC* instances. For example, the *SSS1* object, which is associated with the first redundant joint, generates two instances of the *JC* object *(JC1, JC2)*: one for positive and one for negative displacements of the joint from its base angle. In our example overall eight *JC instances* are generated. Therefore, we can expect to have as much as six *TC* created (we will remind the reader that while *SSS* entities are used to investigate effects of perturbations of an individual joint on the robot arm configuration, the CSS components are used to do it for

all redundant joints). In our example only four *JC* satisfy their joints constraints and only four *TC* are created. Note that our design enforces a concurrent type of creation and existence of the system processes (each branch of the tree in Figure 6 is designed to be executed concurrently) following *Rule 3*. Even if actual implementation developed from this design is going to be sequential, the design assures that operations on all

possible joint displacements (*JC* in our design) is going to be performed within the same functional component. This reduces the complexity of the robotic system and makes testing and maintenance of the *Decision Making* component possible.

Validation of the Direct Search Model. The developed model was simulated with a scenario confirming to the specifications of our example. A snapshot of the animated analysis model execution is presented in **Figure 7**. The state model inspecting windows are presented for the *Joint*, *TC* and *JC* objects. Each window visualizes the current list of object instances for a lifecycle state model. The output includes the name of the object and its attribute names, and the attribute values for each instance. Eight instances of the *JC* object were created for two redundant joints with base angles θ_1=30° and θ_2=100° and the joint limits of 90° for both joints. Variable *delta* that specifies the displacement of the joints from their base configuration was defined to be equal to 10°. Trial angles of the 3rd and 6th *JC* instances did not satisfy the joint limit. Therefore, only 5 instances of *TC* object constitute the search space of the decision-making problem. The first instance, *TC(1)*, presents the initial (base) configuration of a robot arm, three instances (*TC(2)*, *TC(3)*, *TC(5)*) define "simple" and one instance, TC(7) defines "combinational" components of the SearchSpace as indicated by the values of *TS_counter* variable of the *JC* object.

The lower panel of Figure 7 shows the results of testing of the robot analysis model. A specified pre-condition for one of the states of the eight[th] instance of the JC object was not satisfied and a run-time error was reported.

4.3 Comparative Analysis of the OOA-based

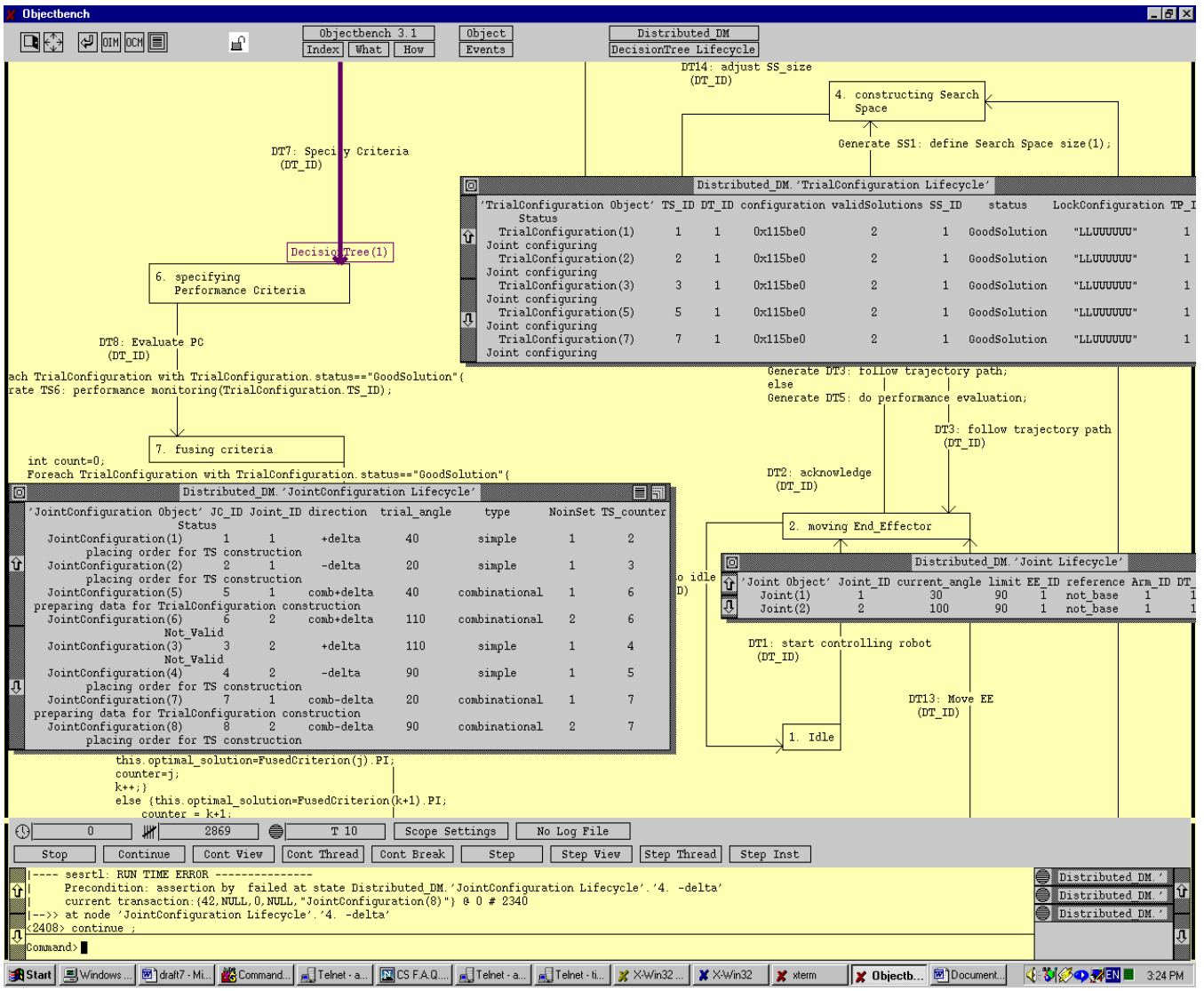

Figure 7. Demonstration of the Animated Simulation

Model with the Conventionally Developed System

The benefits of using *Rules 1* and *2,* which require to avoid redundant operations and functional declarations outside of the scope of a functional component, is demonstrated by a comparative analysis of the original and the redesigned robotic systems.

Original Design. The developers of OSCAR used the conventional approach of manual translation of design requirements to implementation in C++ programming language generally following the Booch methodology [1]. A flow chart for the direct search decision-making strategy

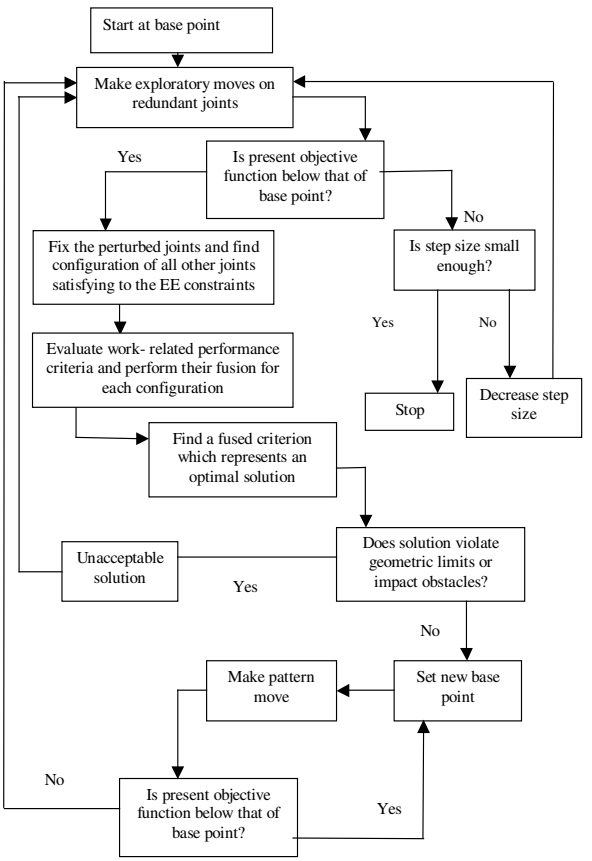

Figure 8. Flow Chart for the Direct Search Technique Implemented in the Conventional Approach

(**Figure 8**) represents the conventional implementation. Its main steps are delineated below:

1. Search begins with an estimate of the solution that serves as an initial base point;
2. A pattern of moves around the base point generates a set of trial solutions satisfying to the end-effector constraints;
3. A number of performance criteria specified by the user are applied to the set of candidate solutions;
4. Criteria fusion is used to assure no dominance of any criterion;
5. The decision-making algorithm finds the best (optimal) choice of the trial solution;
6. The optimal solution is compared against geometric constraints and singularity criteria;
7. Depending on whether the solution is valid or not, the decision-making process either selects a new base point or performs a new set of exploratory moves;
8. If the pattern move related to the new base point hits a local minima [11], the step size is reduced and the process continues;
9. Search continues until either the number of iterations exceeds the iteration limit or the step size is reduced to the step size limit.

The OOA model. A State Transition Diagram (STD) (**Figure 9**) demonstrates the lifecycle of the *JC* object instances. Consider the different states that a *JC* object assumes after each transaction. A sample object is created when an instance of *SSS* and/or *CSS* (depending on the type of the decision-making technique) initiates event "JC1: Create". Each instance of the *JC* object receives supplemental data specifying the direction in which a *trial_angle* should be changed (+*delta*, or –*delta*). Here *delta* is a global variable specified by the user through API. After *trial_angle* is calculated its value is checked against the *limit* of a *Joint* instance (element of the *Kinematics* unit) associated with the instance of the *JC*. As a result, the system goes either into a *Valid* or *Not_Valid* state. So called "valid" *JC* instances, after performing some additional calculations, generate events which trigger creation of *TC* instances. You can see a key element in the actions of the states of considered objects. It is the **Generate** keyword, which indicates that an event is to be sent to a target object whose identity is listed as the first argument of the event's parameter list.

The lower panel of Figure 9 shows the trace of execution for the developed model. An object instance at any state receives an event, makes an appropriate transition to another state, executes the actions associated with the new state and then waits for further events. An arrow in Figure 9 indicates the execution path of the *JC* object.

Comparison Results. Development and validation of the executable specifications of the robotics decision-making domain led to discovery of design flaws and glitches in the original code. Next we discuss some representative er rors and show how implementation of the design rules helped to identify them:

- *(Rule 1)* - Non-optimal design of the direct search algorithm was discovered as a result of OOA-based

formalization. In the existing code an *optimal solution* defined by the optimization algorithm is compared to the *limits* of each joint of the robot arm as a final step in the decision-making algorithm (step 6 in the original implementation). In the OOA design these limits are checked prior to the generation of a solution. In fact, in the OOA-based model this limit check is performed by *JC* objects (see Fig. 9) since this operation is a functionality of the *JC* object.

- (*Rule 2*) - The executable architecture developed following the SM-OOA paradigm captured the design, which assures that the system is performing using only "valid" options of joint configurations (the state machine of the *JC* object explicitly defines "Valid" and "Not_Valid" states as shown in Fig. 9). In contrast, in the original software any trial configuration was considered for performance evaluation disregarding the constraints. This approach leads to situations when it is possible that none of the trial solutions represent a valid solution. In order to continue operation re-initilization of all parameters is required which is equivalent to the robot control termination. This is a faulty situation.

5 Conclusions and Related Work

This paper reports results of the model development and validation phase of a project combining validation by simulation with verification by model checking into an object-oriented software development process. The focus of this paper is OOA-based modeling leading to development of validated and *verifiable* models. Design rules for construction of efficient and dependable models were presented and implemented during the redesign of a robotic control system. Development and validation of the executable OOA model discovered serious logic flaws in the existing robotic software system. Application of verification by model checking would have been seriously compromised until the design flaws in the original implementation were corrected. The OOA model resulting from phase 1 of this project could, with some

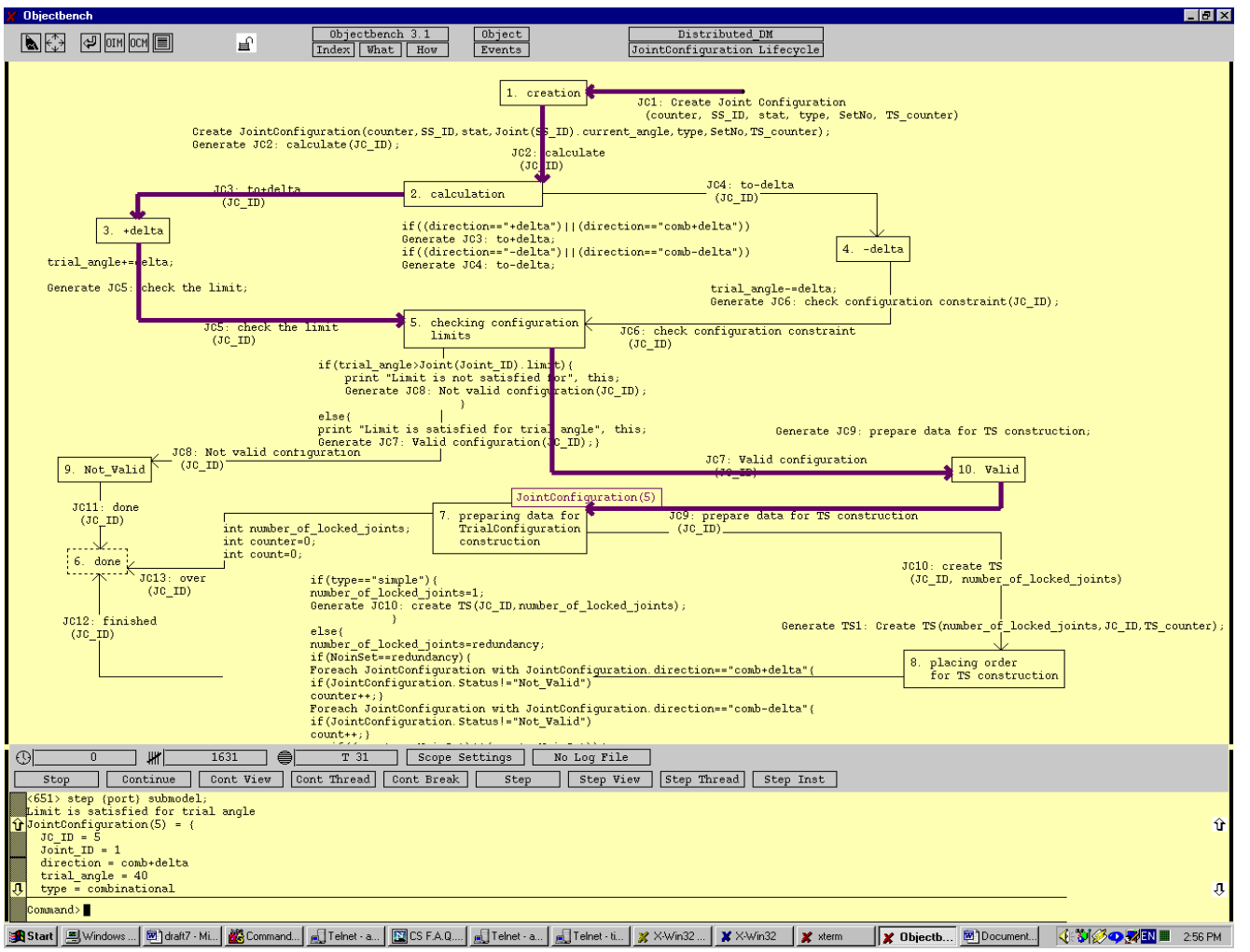

Figure 9. State Machine of the JointConfiguration Object

further modifications, be model checked. The OOA-based verification approach and its application to the robot control system are presented in [22].

There has been a great deal of research on formalization of object-oriented models and languages. This research has been largely concerned with integration and application of verification based on theorem proving rather than model checking [9],[14]. Previous work on application of model checking to software systems has mainly been either to software systems written in procedural languages or to abstract models extracted from the procedural programs. Feaver [7] targets software systems written in C. PathFinder [6] is used for model checking of Java programs.

Application of model checking to verification of systems specified using OO modeling techniques has been mainly restricted to hardware systems and communication protocols. Lind-Nielsen, et. al [15] applied SMV [16] model checker for verification of hardware systems represented by VisualState machines. Dependency analysis was used to decompose large complex systems. Chan, et.al [4] verified control algorithms of a complex aircraft collision system modeled using the UML notation. They reported that their ad hoc solutions for the manual system partitioning frequently caused invalid results. Sharygina, et.al [23] applied the SPIN model checker [8] for verification of the computer-controlled robot controller specified in the xUML notation. None approaches the issues of the system complexity management prior to model-checking.

Design guidelines for constructing testable and maintainable programs in object-oriented procedural languages have been proposed and discussed by a number of researchers. Lakos [13] presents a systematic approach for development of C++ programs. Moors [17] has proposed design rules similar to those presented in this paper for development of communication protocols. Lano [14] identified problems related to formalization of the OO techniques. There is no effort known to us, however, that would address a problem of the complexity reduction of the application domain and the state-space resolution at the design level.

Acknowledgments

Support for this work was provided by the U.S. Department of Energy (Grant DE-FG02-94EW37966).

References

[1] Booch, G., *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings (1994)

[2] Booch, G., Rumbaugh, J., and Jacobson, I., *Unified Modeling Language User Guide,* Add. Wesley (1997)

[3] Clarke, E.M., and Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic, *Workshop on Logic of Programs, Yorktown Heights, NY*, LNCS, Vol. 131, (1981) 52-71

[4] Chan, W., Anderson, R., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J., Model Checking Large Software Specification, *In Proc. of IEEE SE Trans.* (1998) 498-519

[5] Hardin R. H., Har'El, Z., and Kurshan, R.P., COSPAN, *In Proc. of CAV'96*, LNCS, Vol. 1102, (1996) 423-427

[6] Havelund, K., and Pressburger, T., Model Checking Java Programs Using Java PathFinder, *In Proc. 4'th SPIN workshop* (1998)

[7] Holzmann, G., and Smith, M., Feaver: Automating software feature verification, *Bell Labs Technical Journal*, V. 5, (2000) 72-87

[8] Holzmann, G., The Model Checker SPIN, *IEEE Trans. on Software Engineering*, Vol. 5(23), (1997) 279-295

[9] Hubman H., Formal Foundations for Pragmatic Software Engineering Methods, In B. Wolfinger, ed., Innovationen bei Rechen- und Kommunikationssystemen (1994) 1-50

[10] Kapoor, C., and Tesar, D., A Reusable Operational Software Architecture for Advanced Robotics (OSCAR), The University of Texas at Austin, Report to U.S. Dept. of Energy, Grant No. DE-FG01 94EW37966 (1996)

[11] Kapoor, C., Cetin, M., and Tesar, D., Performance based redundancy resolution with multiple criteria, *Proc. ASME Design Engin. Techn. Conf.,* Atlanta, Georgia (1998)

[12] Kurshan, R., *Computer-Aided Verification of Coordinating Processes - The Automata-Theoretic Approach*, Princeton University Press (1994)

[13] Lakos, J., *Large Scale C++ Software Design*, Addison-Wesley (1996)

[14] Lano, K., *Formal Object-Oriented Development*, Springer (1997)

[15] Lind-Nielsen, J, Andersen H., R., etc., Verification of large State/Event Systems using Compositionality and Depenedency Analysis, *In Proc. of TACAS'98*, Portugal (1998) 201-216

[16] McMillan, K., *Symbolic Model Checking*, Kluwer (1993)

[17] Moors, T., Protocol Organs: Modularity should reflect function, not timing, *In Proc. OPENARCH98*, (1998) 91-100

[18] Object Management Group (OMG), *Action Semantic for the UML*, OMG (2000)

[19] Rumbaugh, J., Jacobson, I. and Booch, G., *The Unified Modeling Language Reference Manual*, ObjectTechnology Series, Addison-Wesley (1999)

[20] SES Inc., *CodeGenesis User Reference*, SES Inc. (1998)

[21] SES Inc., *ObjectBench Techn. Reference*, SES Inc. (1998)

[22] Sharygina, N., Browne, J., and Kurshan, P., A Formal Object-Oriented Analysis for Software Reliability: Design for Verification, *In Proc. of ETAPS2001* (to appear), Genoa, Italy (2001)

[23] Sharygina, N., and Peled, D., A Combined Testing and Verification Approach for Software Reliability, *In Proc. of FME2001* (to appear), Berlin, Germany (2001)

[24] Shlaer, S., and Mellor, S., *Object Lifecycles: Modeling the World in States*, Prentice-Hall, NJ (1992)

[25] Xie, F., Levin, V., and Browne, J., Integrating Model Checking into Object-Oriented Software Development Process, Techn.Rep., University of Texas at Austin, Comp. Science Dept. (2000)