A Formal Object-Oriented Analysis for Software Reliability: Design for Verification

Natasha Sharygina[1], James C. Browne[2], and Robert P. Kurshan[1]

[1] Bell Laboratories, 600 Mountain Ave.,
Murray Hill, NJ, USA 07974
{natali,k}@research.bell-labs.com

[2] The University of Texas at Austin, Computer Science Department,
Austin, TX, USA 78712
browne@cs.utexas.edu

Abstract. This paper presents the OOA design step in a methodology which integrates automata-based model checking into a commercially supported OO software development process. We define and illustrate a set of design rules for OOA models with executable semantics, which lead to automata models with tractable state spaces. The design rules yield OOA models with functionally structured designs similar to those of hardware systems. These structures support model-checking through techniques known to be feasible for hardware. The formal OOA methodology, including the design rules, was applied to the design of NASA robot control software. Serious logical design errors that had eluded prior testing, were discovered in the course of model-checking.

1 Introduction

Problem Statement. Software systems used for control of modern devices are typically both complex and concurrent. Object-Oriented (OO) development methods are increasingly employed to cope with the complexity of these software systems. OO development systems still largely depend on conventional testing to validate correctness of system behaviors, however. This is simply not adequate to attain the needed reliability, for complex systems, on account of the intrinsic incompleteness of conventional testing.

Formal verification of system behavior through model checking [4], on the other hand, formally verifies that a given system satisfies a desired behavioral property through exhaustive search of ALL states reachable by the system. Model checking has been widely and successfully applied to verification of hardware systems. Application of model checking to software systems, has, in contrast, been much less successful. To apply model checking to software systems the software systems must be translated from programming or specification languages to representations to which model checking can be applied. The resulting representation for model checking must have a tractable state space if model checking is to be successful. Translation of software systems designed by conventional development processes and even by OO development processes to representations to which model checking can be applied have generally resulted in very large interconnected state spaces.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 318–332, 2001.

A principal result reported in this paper is a set of design rules (Section 3) for development of OO software systems that when translated to representations to which model checking can be applied, yield manageable state spaces. These design rules are the critical initial step in the methodology for integration of formal verification by model checking into OO development processes.

Approach. *The validity and usefulness of design rules and the effectiveness of the integration of formal verification into object-oriented software development can be evaluated only in the context of their application.*

This paper reports a case study in re-engineering the control subsystem for a NASA robotics software system. The goal of the project was to increase the subsystem's reliability. This goal was achieved, as serious logical design errors were discovered, through the application of model-checking. This case study motivates and demonstrates the design rules for "design for verifiability" and the application of formal verification by model checking to a substantial software system.

The robot control subsystem was originally implemented by a conventional development process in a procedural OO programming language (C++) generally following the Booch methodology [1]. In the re-engineering project, a four-step process to obtain a reliable system was planned:

1. Re-implement the control subsystem as an executable specification in the form of an Object-Oriented Analysis (OOA) model (in the Shlaer-Mellor (SM) methodology [26]);

2. Validate this executable specification as thoroughly as possible by testing;

3. Apply model checking to the OOA model to validate its behavior at all possible states of the system;

4. Generate the control software by compilation of the validated and verified OOA model.

The application of the SM OOA method captures the robotic domain entities in terms of classes/objects and relate them using the relationships diagrams. Testing and evaluation of the execution behavior of the constructed system was greatly simplified due to the fact that the OOA classes were represented as a set of attributes, confined to simple types, and that the behavior of the system was state machine specified. Such an OOA model can be viewed as being *designed for testability*. This executable specification can also be translated by a code generation system to C++ code. A commercially supported software system, SES/Objectbench (OB) [25] was used in this step.

OOA models with executable semantics are representations of software system, which should be amenable to model-based verification techniques. An OOA model represents the program at a higher level of abstraction than a conventional programming language. The OOA model partitions the system into well-defined classes. But, attempts to apply model checking to these apparently highly modular OOA models led to intractably large state spaces for the robot control system model. The cause for this problem is suggested by examining hardware systems. In hardware, the "calling" module and "called" module are separated spatially and communicate through a clean interface and a specified protocol. This "spatial modularity" supports divide-and-conquer analytical techniques, as each module can be analyzed in isolation. This is essential for successful model-checking because it resolves a fundamental problem of the *state space* *explosion*. The design rules of OOA methods do not enforce the logical equivalent of “spatial modularity” in software. (In software, modularity tends to be “temporal”, in the sense that modular subroutines are invoked in succession. This “temporal modularity” does not directly support divide-and-conquer techniques). For example, accessor and mutator methods cause coupling of the states of instances of different classes. The logical equivalent of “spatial modularity” for software is the strong form of name space modularity where the name spaces modules are rigorously disjoint and all interactions among modules are across specified interfaces and follow specified protocols. “Spatial modularity” (strong name space modularity) is consistent with the intent of the OOA approaches of conceptual encapsulation but it is not explicitly considered in most OO design methods. We introduce a set of design rules that constrain the syntactic structure of OOA models to conform to “spatial modularity”. The systems become spatially modular (in the hardware sense when system elements can be analyzed in isolation) and support existing verification techniques developed for hardware systems. We applied the *design for verifiability* rules to a further redesign of the robot controller system. The results are encouraging - we were able to apply the partitioned development, model checking, assume/guarantee reasoning, abstraction techniques [14] developed for hardware systems

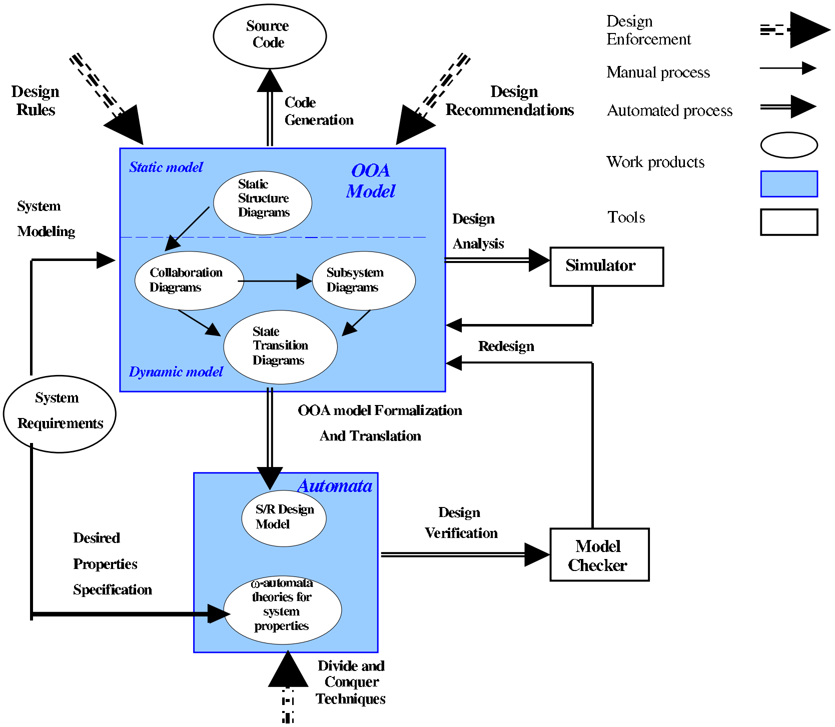


Fig. 1. The OOA-based methodology for the spatial development of software systems

to our software system. This powerful combination of techniques helped us to break the computational complexity barrier to the application of verification by model checking to OOA models of software systems. Model checking was accomplished by translation to S/R, an input language of the COSPAN [9] model checker, using the translator reported in [27].

In this paper, we develop a set of design rules for construction of OOA models to which verification by model checking can be practically applied, and we demonstrate the application of the integrated OOA and model-checking methodology for development of software systems.

2 Integration of Model Checking with OO Development

A methodology for integration of OOA and model checking shown in **Figure 1** is presented in [27]. Overall, this model fulfills the need for a sound foundation in rigorous requirements modeling, design analysis, formal verification, and automated code generation. Model checking is applied to SM OOA (xUML) models [26] that have executable semantics specified as state/event machines rather than to programs in conventional programming languages. An automata-based approach to model checking, supported by the COSPAN [9] model checker, is used. The OOA models are automatically translated to automaton models using the OB-SR translator [27]. Predicates over the behavior of the OOA models are mapped to predicates over the automaton models and evaluated by the model checker. This research imposes the structural design rules on the software system reducing its complexity at the design level and thus supports the reuse of the existing model-checking techniques developed for hardware verification.

xUML Notation. Use of OOA models with executable semantics is moving into the mainstream of OO software development. The Object Management Group (OMG) [20] has adopted a standard action language for the Unified Modeling Language (UML) [21]. This action language and SM OOA semantics represented in UML notation define an executable subset of UML (xUML). The OOA representation used in this research is the SM OOA as implemented by the capture and validation environment SES/Objectbench (OB) [25]. We are, on the recommendation of Steve Mellor [private communication] referring to the OOA model we use as xUML.

We utilize a subset of xUML notation suitable for modeling objects, subsystems, their static structure, and their dynamic behavior. *Static structure diagrams* capture conceptual entities as classes with semantics defined by attributes. *Object information diagrams* (OID) describe the classes and relationships that hold between the classes. They graphically represent a design architecture for an application domain and give an abstract description of tasks performed by cooperating objects. *Subsystem relationship diagrams* situate the application domain in relation to its scope, limits, relationships with other domains and main actors involved (scenarios). The *collaboration diagram* is used for graphical representation of the signals sent from one class to another. This representation provides a summary of asynchronous communication between *state/event models* in the system. The state/event model is a set of Moore state machines that consists of a fixed number of concurrently executing finite state machines. The *state transition diagram* graphically represents a state machine. It consists of nodes, representing states and their associated actions to be performed, and event arcs, which represent transitions between states. The execution of an action occurs after receiving the *signal or event*. A transition table is a list of signals, and "the next" states that are their result. Signals have an arbitrary identifier, a target class, and associated data elements.

Two types of concurrent model execution are supported by xUML: *simultaneous* and *interleaved*. We utilize only the asynchronous interleaved execution model in the OOA models of this research.

COSPAN, an Automaton-based Model Checking Tool. COSPAN [9] allows symbolic analysis of the design model for user-defined behavioral properties. Each such test of task performance constitutes a mathematical proof (or disproof), derived through the symbolic analysis (not through execution or simulation). The semantic model of COSPAN is founded on ω-automata [14]. The system to be verified is specified as an ω-automaton P, the task the system is intended to perform is specified as an ω-automaton T, and verification consists of the automata language containment test $L(P) \subset L(T)$. P is typically given as the synchronous parallel composition of component processes, specified as ω-automata. Asynchronous composition is modeled through nondeterministic delay in the components.

Language containment can be checked in COSPAN using either a symbolic (BDD-based) algorithm or an explicit state-enumeration algorithm.

Systems are specified in the S/R language, which supports nondeterministic, conditional (if-then-else) variable assignments; variables of type bounded integer, enumerated, boolean, and pointer; arrays and records; and integer and bit-vector arithmetic. Modular hierarchy, scoping, parallel and sequential execution, homomorphism declaration and general ω-automata fairness are also available.

3 Design for Verification

An xUML OOA is a natural representation to which to apply model-based verification techniques. The complexity level of the executable OOA models is far less than the procedural language programs to which they are translated. In addition to the finite state representation provided by the OOA techniques, the following features of the OOA methodology reduce the complexity of the system at the design level:

Abstraction of implementation details. Relationships between objects at the OOA level are represented as associations and not as pointers. OOA constructs such as signals in UML express state transitions without reference to the internal states of objects. Separate specification of class models and behavior models separates specification of data from control.

Hierarchical system representation. OOA methods support modular designs and encourage software developers to decompose a system into subsystems, derive interfaces that summarize the behavior of each system, and then perform analysis, validation and verification, using interfaces in place of the details of the subsystems.

“Spatial Modularity” of software systems. The design property which enables verification of hardware systems by model checking is sometimes called “Spatial Modularity”. In hardware realized systems functionality is of necessity partitioned into modules which are spatially disjoint. Interaction among these spatially disjoint functional modules must take place across precisely defined interfaces and follow precisely defined protocols. The spatial partitioning of hardware modules across well-defined static interfaces supports the application of divide-and-conquer techniques, necessary to circumvent the generally infeasible computation problem inherent in model-checking.

The logical equivalent of “spatial modularity” for software is the strong form of name space modularity where the name spaces of all modules are disjoint and all interactions between functional modules are across specified interfaces and follow specified protocols. The strong name space modularity is conceptually consistent with the methodology of separation of concerns advocated by the OOA approaches. It is however is not explicitly specified in any OO design methods.

Structural design rules. We developed a set of design rules and recommendations that constrain the structural design of the OOA models to conform to “spatial modularity”. The systems become spatially modular (in the hardware sense where system elements can be analyzed in isolation) and support existing verification techniques developed for hardware systems. These design rules for OOA models are similar to those given for development of truly OO programs in OO procedural languages such as C++.

Design Rule 1. *Write access to attributes of one class by another class must be made through the event mechanism.*

The attributes of a class should be local to the class. Change of values of a class instance should be performed only through the event mechanism. This precludes coupling of internal states of classes.

Design Rule 2. *Attribute values which are shared by multiple classes should be defined in separate class and accessed only through the event mechanism.*

This design rule also avoids coupling of internal states of classes.

Design Rule 3. *Declaration and definition of functional entities must be performed within the same component.*

A component may have dependencies on other components. To prevent the situation when functionality of one component can be changed by other components any logical construct that a component declares should be defined entirely within that component.

Design Rule 4. *Inheritance must be confined to extensions of supertypes. Modification of the behavior of supertypes (overriding of supertype methods) is prohibited.*

This means to follow a meaning of subtyping, along the lines of Liskov’s [17]:

“A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for every object o1: S there is o2: T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T”.

Rule 4 enables reasoning about the correctness of newly derived subtype classes based on the verification results of previously existing subtype classes.

Recommendation. *Creating modular systems the linking between the modules/ subsystems should be minimized.*

Subsystems are fundamentally open systems but for verification must be closed with a definition of the environment in which they will execute. Simulation of the environment behavior is performed by assuming a sequence of events defined on the subsystem’s interface. A minimal number of links between subsystems enables effective definition of the environment used to complete a subsystems definition for verification.

4 The Robot Controller Study

We examine a robotic software used for control of redundant robots [1]. Redundant robots are widely used for sophisticated tasks in uncertain and dynamic environments in life-critical systems. An essential feature for a redundant robot is that an infinite number of robot's joints displacements can lead to a definite wrist (end-effector) position. Failure recovery is one of the examples of redundancy resolution applications: if one actuator fails, the controller locks the faulty joint and the redundant joint continues operating. The general task of the test-case software is to move a robot arm along a specified path given physical constraints (e.g. obstacles, joints angles and end-effector position constraints). The specific task is to choose an optimal arm configuration. This decision-making problem is solved by applying performance criteria [13].

The decision-making method is based on the local explorations and the concept of a joint-level perturbation. Perturbation at the joint level means temporarily changing one or more of the joint angles (joint angle - is the angle between two links forming this joint) either clockwise or counterclockwise. This project focuses on two different exploration strategies: simple and factorial [13]. In the simple exploration we displace one joint at a time and find how it effects the configuration of the robot arm (find all other joint angles) for a given end-effector position. The other perturbation strategy is based on 2^n factorial search. A detail of a redundant robot executing the simple exploration strategy for one of the joints is shown in **Figure 2**, with θ - being a joint angle, and δ - being a displacement.

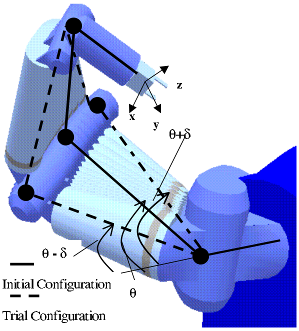


Fig. 2. A part of a redundant robot, demonstrating infinite manipulator configurations for a single end-effector position.

The original software, OSCAR [13], consisted of a set of robot control algorithms supported by numerous robotic computational libraries, was developed using a conventional approach. To obtain a reliable system we redesigned its control subsystem as an executable specification in the form of a SM OOA (xUML) model.

4.1 Domain Analysis and Modeling

The robotic OOA model includes fifteen basic classes, including their variables and associations.

Classes. In addition to tangible objects (*Arm*, *Joint*, *EndEffector*, *PerformanceCriterion*), incident objects (*TrialConfiguration*, *SearchSpace*, *SimpleSearchSpace*, *FactorialSearchSpace*), specification objects (*Fused Criterion*), and role objects (*Decision Tree*, *OSCAR_Interface*, *Checker*) were derived [26].

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

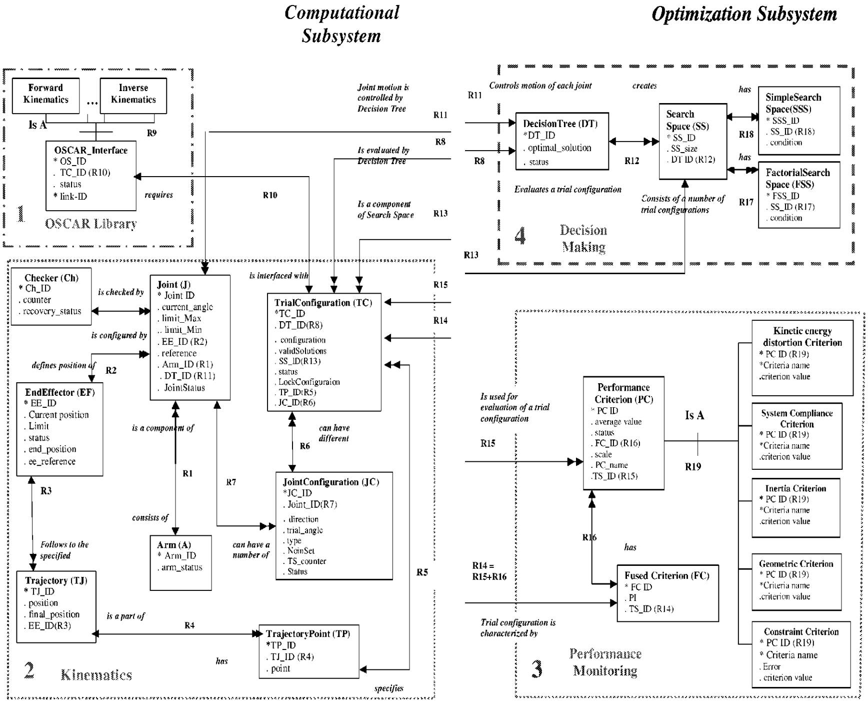


Fig. 3. OID of the Multi-Criteria Decision Support Robotic System

Attributes. The attributes of the *EE* object are illustrated below as an example. *EE_ID* is a key attribute whose value uniquely distinguishes each instance of an *EE* object. *Current_position* and *Limit* are descriptive attributes that provide facts intrinsic to the *EE* object. For example, *Current_position* is a vector specifying positions (x, y, z) and orientation angles (α, β, γ) of the *EE*. *Status*, *end_position* and *ee_reference* 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. For example, *end_position* domain is (0,1) which values reflect if the *EE* reached the final destination while moving along the specified trajectory path.

Associations. The executable model is defined using two types of the relationships: binary (those in which objects of two different types participate) and higher-order supertype-subtype (those when several objects have certain attributes in common which are placed in the supertype object). For example, one-to-many binary *Arm-Joint* relationship states that a single instance of an Arm object consists of many instances of a *Joint* object. An example of supertype-subtype relationships is a *PerformanceCriterion-*

ConstraintCriterion relationship. In this construct one real-world instance is presented by the combination of an instance of the supertype and an instance of exactly one subtype.

Robotic Decision-Support Domain Architecture. The application domain architecture was divided into *computational* and *optimization* subsystems. The input for the optimization subsystem is one or more trial arm configurations from which the optimization system will either select the best one or provide the computational system with suggestions on what an optimal arm configuration should be.

The *computational* subsystem includes kinematics algorithms and interfaces to the computational libraries of the OSCAR system. There are two methods for the computational system to define a base point of the optimization search. The first method is to calculate an *EndEffector* (*EE*) position given initial *Joint* (*J*) angles of all joints and, thus, find an initial arm configuration. The second method depicts an *EE* position as a new base point from the *Trajectory* path specified by the user.

A collaboration diagram of the abstracted *Kinematics* unit which verification results we discuss in the next section is represented in Figure 6. The control algorithm starts with defining an initial end-effector position given the initial joint angles. This is done by solving a forward kinematics problem [13]. The next step is to get a new end-effector position from a predefined path. The system calculates the joint angles for this position, providing the solution of the inverse kinematics problem [13] and configures the arm. At each of the steps described above, a number of physical constraints has to be satisfied. The constraints include limits on the angles of joints. If a joint angle limit is not satisfied, a *fault recovery* is performed. The faulty joint is locked within the limit value. Then, the value of the angle of another joint is recalculated for the same end-effector position. If the end-effector position exceeds the limit, the algorithm registers the undesired position, which serves as a flag to stop the execution. A *Checker* class controls the joints that pass or fail the constraints check. If all the joints meet the constraints, the *Checker* issues the command to move the end-effector to a new position. Otherwise it either starts a fault recovery algorithm or stops execution of the program (if fault recovery is not possible).

The *optimization* subsystem implements the decision-making strategy by applying decision-making techniques identifying a solution to the multi-criteria problem. It builds a *SearchSpace* (*SS*), which generates sets of *JointConfigurations* (*JC*) around a base point supplied by the computational subsystem. *JC* instances initiate creation of *TrialConfiguration* (*TC*) instances that normalize a robot arm configuration for any perturbed joint. A *DecisionTree* (*DT*) selects the best *TC* given a set of *PerformanceCriteria* (*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 pattern of local exploration. The search stops when no new solutions are found. The system returns control to the computational subsystem which changes the position of the *EE* following the specified trajectory and determines a new base point for the search.

4.2 Compliance to the Design Rules

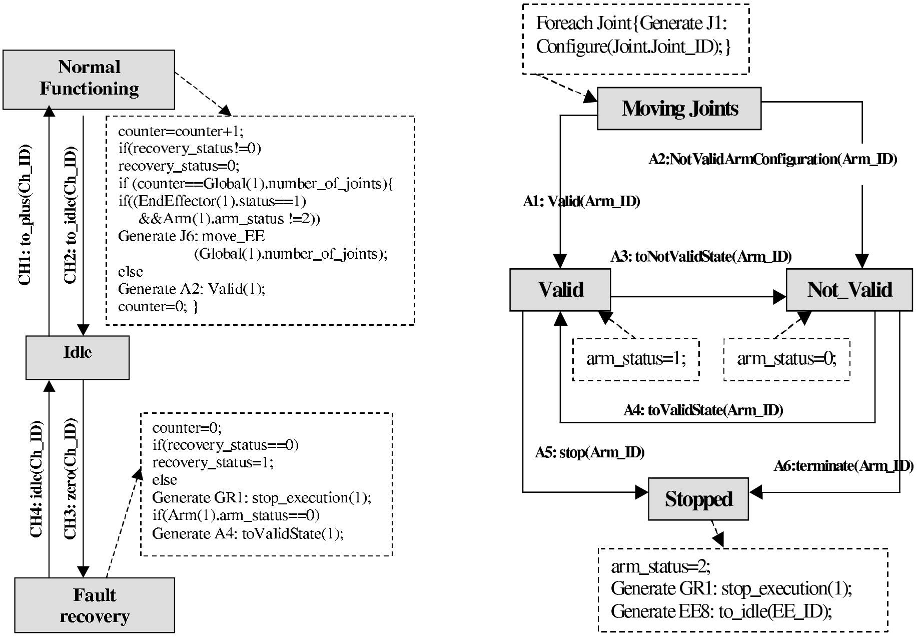
During the construction of the robot control design we followed the design rules specified in Section 3.

Rule 1: All updates of the attribute values were done through t the event mechanism.

Rule 2: After the design system was completed and validated by simulation, a separate object *Global* that contained all the global variables as its attributes was created and used during verification and code generation.

Rule 3: We restricted the design to insure that all functional components are fully self-contained.

Rule 4: The users of the developed robotic framework are allowed to add new elements to the developed architecture. Specifically, new performance criteria can be added to the architecture. These additions are subtype classes and, in order to satisfy the fourth rule, it is required that they have a semantic relationship with their supertype classes. Other words, inheritance is restricted to a purely syntactic role: code reuse and sharing, and module importation.



```
counter=counter+1;
if(recovery_status!=0)
recovery_status=0;
if (counter==Global(1).number_of_joints){
if((EndEffector(1).status==1)
    &&Arm(1).arm_status !=2))
Generate J6: move_EE
                (Global(1).number_of_joints);
else
Generate A2: Valid(1);
counter=0; }
```

```
counter=0;
if(recovery_status==0)
recovery_status=1;
else
Generate GR1: stop_execution(1);
if(Arm(1).arm_status==0)
Generate A4: toValidState(1);
```

```
Foreach Joint{Generate J1:
Configure(Joint.Joint_ID);}
```

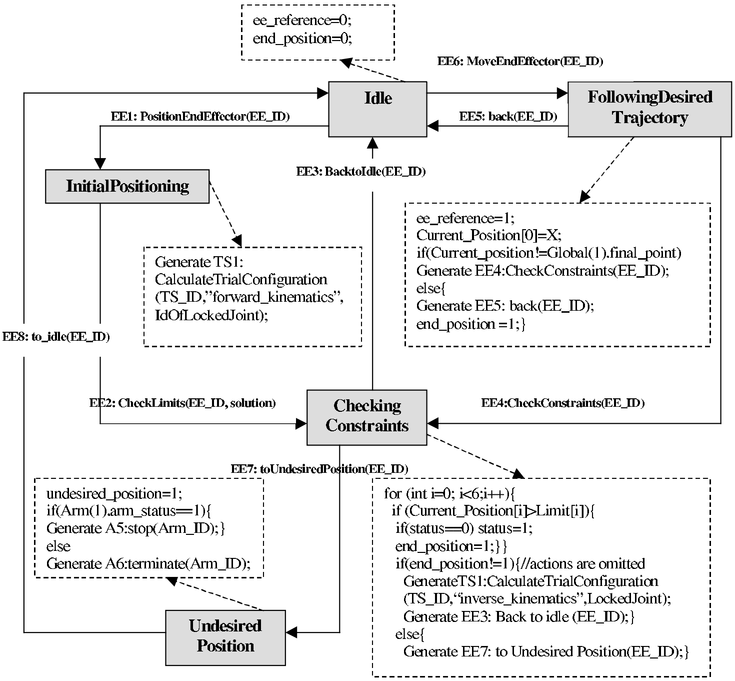
```
arm_status=1;
```

```
arm_status=0;
```

```
arm_status=2;
Generate GR1: stop_execution(1);
Generate EE8: to_idle(EE_ID);
```

Fig. 4. State Transition Diagram of the *Checker* (left) and *Arm* (right) objects

Recommendation (System decomposition): As it can be seen in **Figure 3** the architecture can be represented as a collection of basic robotics functional units. These functional components (*OSCAR Library*, *Kinematics*, *Performance Monitoring* and *Decision Making*) are depicted by dashed lines. Each functional unit contains a substantial proportion of components that do not depend on other units.


Idle
FollowingDesired Trajectory
InitialPositioning
Checking Constraints
Undesired Position
EE6: MoveEndEffector(EE_ID)
EE1: PositionEndEffector(EE_ID)
EE5: back(EE_ID)
EE3: BacktoIdle(EE_ID)
EE8: to_idle(EE_ID)
EE2: CheckLimits(EE_ID, solution)
EE4:CheckConstraints(EE_ID)
EE7: toUndesiredPosition(EE_ID)

```
ee_reference=0;
end_position=0;
```

```
Generate TS1:
CalculateTrialConfiguration
(TS_ID,"forward_kinematics",
IdOfLockedJoint);
```

```
ee_reference=1;
Current_Position[0]=X;
if(Current_position!=Global(1).final_point)
Generate EE4:CheckConstraints(EE_ID);
else{
Generate EE5: back(EE_ID);
end_position =1;}
```

```
undesired_position=1;
if(Arm(1).arm_status==1){
Generate A5:stop(Arm_ID);}
else
Generate A6:terminate(Arm_ID);
```

```
for (int i=0; i<6;i++){
  if (Current_Position[i]>Limit[i]){
   if(status==0) status=1;
   end_position=1;}}
   if(end_position!=1){//actions are omitted
     GenerateTS1:CalculateTrialConfiguration
     (TS_ID,"inverse_kinematics",LockedJoint);
     Generate EE3: Back to idle (EE_ID);}
   else{
     Generate EE7: to Undesired Position(EE_ID);}
```

Fig. 5. State Transition Diagram of the *EndEffector* object

4.3 OOA Model Validation and Formal Verification

The OOA model was validated by simulation. Several serious error or defects in the original design and in the original versions of the OOA model were identified and corrected. Space precludes us from describing the validation process and its results. Details can be found in [22].

We checked a collection of safety and guarantee requirements specifying the coordinated behavior of the robot control processes. We focused on the control intensive algorithms of the *Kinematics* unit and abstracted the calculations that were irrelevant to the actual robot control. Specifically, the *Trajectory*, *TrajectoryPoint* and *JointConfiguration* classes used for storage of the predefined trajectory paths and the possible arm configurations as well as calculations that were done through the interface with the OSCAR Libraries in the original OOA design were substituted with nondeterministic assignments of natural numbers. In fact, in this paper we present an instance of the robot functionality when the robot arm is moving only in the horizontal, i.e. x direction, which value is assigned nondeterministically in the checked model.

We define and discuss here the properties that did not hold during the verification. The properties and their descriptions are given in **Table 1**. The properties are encoded in a query language of COSPAN. The query variables are declared in terms of state predicates appearing in the state transition diagrams of the objects of the *Kinematics* unit. The following declarations are used in Table 1: p - declares the *abort_var* variable of the *Global* class; *q* - declares the *undesired_position* variable of the *EE* class; *r* - declares the *ee_reference* variable of the *EE* class; *s* - declares the *recovery_status* variable of the the *Checker* class; *t* - declares the *end_position* variable of the *EE* class; *v* - declares the *number_of_joints* variable of the *Global* class.

Figure 4, 5 schematically represent the lifecycles of the robot control processes (some actions are omitted due to the space limitations of the paper). For example, the state *UndesiredPosition* and the variables *undesired_position*, *ee_reference* of the *EE* class appear in Figure 5.

Verification found a number of errors in the robot control algorithms. The failure of the Property 1 indicated that in some cases the system does not terminate its execution as specified. The failure of the property 3 that was aimed to check if system terminates properly confirmed this fact. We learned that an error in the fault resolution algorithm caused this problem. We will remind the reader that the fault recovery procedure is activated if one of the robot joints does not satisfy the specified limits. In fact, if during the process of fault recovery some of the newly recalculated joint angles do not satisfy the constraints in their turn, then another fault recovery procedure is called. Analysis of the counterexample provided by COSPAN for Property 3 indicated that a mutual attempt was made for several faulty joints to recompute the joint angles of other joints while not resolving the fault situation.

Table 1. Verification properties

N	Property	Robotic Description	Formal Description
1	EventuallyAlways(*p*=1)	Eventually the robot control terminates	Eventually permanently *p*=1
2	AfterAlwaysUntil(*q*=1, *r*=1, *p*=1)	If the *EE* reaches an undesired position than the program terminates prior to a new move of the *EE*	At any point in the execution if *q*=1 than it is followed by *r*=1 until *p* is set to 1
3	AlwaysUntil(*p*=0, *t*=1 OR (*s*=1 AND *v*=1))	The program terminates when it either completes the task or reaches the state where there is no solution for the fault recovery	*p*=0 holds at any execution of the program until occurrence of either *t*=1 or the combination of *s*=1 and *v*=1

Another error that was found during verification of Property 2 indicated a problem of coordination between the *Arm* and *Checker* processes. The original design assumed sequential execution. In fact, it was expected that the *arm_status* variable of the *Arm* process would be repeatedly updated before the *Checker* process would initiate a command to move the *EE* to a new position. A concurrent interaction between the processes led to the situation where the *Checker* process could issue the command based on an out-of-date value of the *arm_status* variable. This was the reason for Property 2 to fail.

The errors found by model checking were not discovered either during the conventional testing performed by the developers of the original code or during the validation by simulation of the formalized design. In order to correct these errors a redesign of both the original system and the OOA model was required. Figure 6 provides the original and the modified state transition diagrams of the *Kinematics* unit demonstrating the design changes which we made in order to correct the found errors. We introduced a new class called Recovery, whose functionality provides a correct resolution of the fault recovery situation described above. Additionally we added exchange messages between the processes *Arm* and *Checker* that fixed the coordination problem reported earlier.

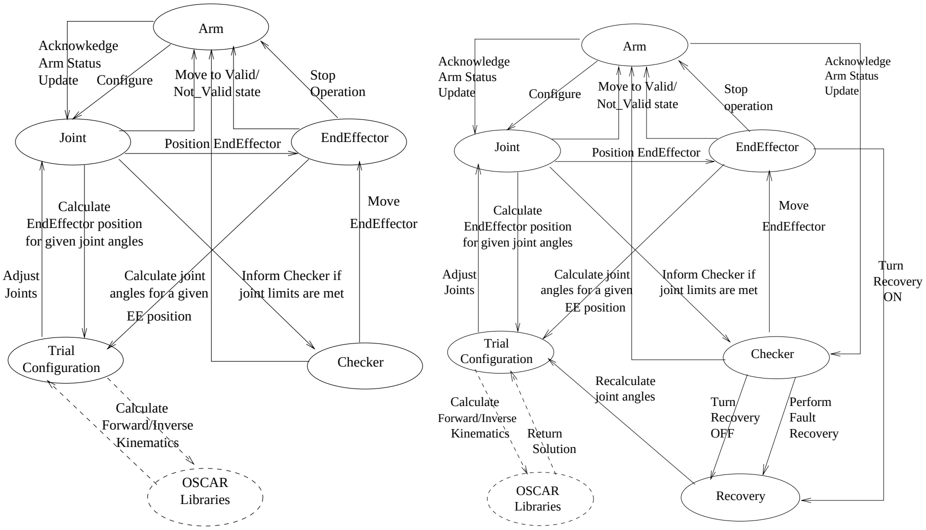


Fig. 6. Collaboration diagrams of the original (left) and modified (right) *Kinematics* unit

It is interesting to note that concurrently with this project, we examined the possibility of integrating testing into the model checking process. The SM OOA executable specification of the robot control system was used as a test-bed for that project. Specifically, an abstracted version of the *Kinematics* unit was manually translated into Promela, the input language of the SPIN [12] model checker. PET [8], an interactive testing tool that supports visual representation of data, was used to establish the conformance between the source code and the code accepted by the model checker. SPIN verification results are presented in [23]. Design errors found using SPIN were corrected and in this paper we use the corrected robot control system. The failure of the fault tolerance algorithm, however, is demonstrated in both projects. The fact that we received identical verification results using different model checking tools for property 3 confirms the validity of the verification results.

4.4 Robotic System Engineering

Given the target system specifications, the validated and verified architecture, and the target system configuration parameters, an instance of the target robotic system was automatically generated using SES/CodeGenesis system [24]. C++ source code that supports the implementation of the developed architecture can be found at www.robotics.utexas.edu/rrg/organization/dual_arm/research/ROOA/.

5 Conclusions and Related Work

This paper gives a feasibility demonstration for the application of verification by model checking to a substantial control intensive application developed in a commercially supported and widely used OO development process. The results of the demonstration are highly encouraging. Verification of significant behavioral properties of the robot control subsystem were carried out. The importance of verification to OOA model design and development has been shown. Design rules leading to xUML OOA models to which verification by model checking can be practically applied have been proposed and applied.

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 programs in procedural languages. Feaver [11] targets software systems written in C while [2], [3] focus on applying model checking on SDL programs. Havelund and Pressburger [10] apply model checking to Java programs. Corbett, et.al [5] extract finite state machines from Java programs to which to apply model checking. The results of verification of a safety critical railroad control system which complexity is comparable to our test-case study are presented in [7].

Model-checking has been also applied to verification of concurrently executing state/event machines. Lind-Nielsen, et al. [16] applied SMV [18] for verification of hardware systems represented by VisualState state machines. Dependency analysis was used to decompose a large but naturally spatially modular systems. Chan, et al. [6] verified a complex aircraft collision software. They reported that their ad hoc solutions for the manual system partitioning frequently caused invalid results. None approaches the issues of the system redesign 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 [15], [17]. Moors [19] has proposed similar design criteria for communication protocols. However, there is no an effort known to us that would address a problem of developing the OOA design rules that support resolution of the state-explosion problem at the design level.

Acknowledgement. This research was partially supported by the Robotics Research Group of the University of Texas at Austin.

References

1. Booch, G., *Object-Oriented Analysis and Design with Applications,* Benjamin/Cummings, Redwood City, CA (1994)
2. Bounimova, E., Levin, V., Basbugoglu, O., and Inan, K., A Verification Engine for SDL Spec. Of Comm. Protocols, *In Proc. of the 1st Symp. on Computer Networks,* Istanbul, Turkey, (1996) 16-25
3. Bosnacki, D., Damm, D., Holenderski, L.,and Sidorova, N., Model checking SDL with Spin, *In Proc. of TACAS2000, Berlin, Germany,* (2000) 363-377
4. 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

5. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Bandera: Extracting finite-state models for Java source code, *In Proc. of 22nd ICSE* (2000)
6. Chan, W., Anderson, R., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J., Model Checking Large Software Specifications, *In Proc. of IEEE Transaction on Software Engineering* (1998) 498-519
7. Gnesi, S., Lenzini, G., Abbaneo, C., Latella, D., Amendola, A., Marmo, P., An Automatic SPIN Validation of a Safety Critical Railway Control System, *In Proc. of Int. Conf. on Dependable Systems and Networks*, (2000) 119-124
8. Gunter, E., and Peled D., Path Exploration Tool, *In Proc. of TACAS 1999, Amsterdam, The Netherlands* (1999) 405-419
9. Hardin R., Har'El, Z., and Kurshan, R.P., COSPAN, *In Proc., CAV'96*, LNCS, Vol. 1102, (1996) 423-427
10. Havelund, K., and Pressburger, T., Model Checking Java Programs Using Java PathFinder, *In Proc. 4'th SPIN workshop* (1998)
11. Holzmann, G., and Smith, M., Feaver: Automating software feature verification, *Bell Labs Technical Journal*, Vol. 5, 2, (2000) 72-87
12. Holzmann, G., The Model Checker SPIN, *IEEE Trans. on Software Engineering*, Vol. 5(23), (1997) 279-295
13. Kapoor, C., and Tesar, D.: A Reusable Operational Software Architecture for Advanced Robotics (OSCAR), The University of Texas at Austin, Report to DOE, Grant No. DE-FG01 94EW37966 and NASA Grant No. NAG 9-809 (1998)
14. Kurshan, R., *Computer-Aided Verification of Coordinating Processes - The Automata-Theoretic Approach*, Princeton University Press, Princeton, NJ (1994)
15. Lano, K., *Formal Object-Oriented Development*, Springer (1997)
16. 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
17. Liskov, B., Data Abstraction and Hierarchy, *In Proc. of OOPSLA conference* (1987)
18. McMillan, K. *Symbolic Model Checking*, Kluwer (1993)
19. Moors, T., Protocol Organs: Modularity should reflect function, not timing, *In Proc. OPENARCH98*, (1998) 91-100
20. Object Management Group (OMG), *Action Semantic for the UML*, OMG (2000)
21. Rumbaugh, J., Jacobson, I. and Booch, G., *The Unified Modeling Language Reference Manual*, Object Technology Series, Addison-Wesley (1999)
22. Sharygina, N., and Browne, J., Automated Rob. Decision Support Software Reverse Engineering, Tech. Rep., The Univ. of Texas at Austin, Robotics Research Croup (1999)
23. Sharygina, N., and Peled, D., A Combined Testing and Verification Approach for Software Reliability, *In Proc. of FME2001* (to appear), Berlin (2001)
24. SES Inc., *CodeGenesis User Reference Manual*, SES Inc. (1998)
25. SES inc., *ObjectBench Technical Reference*, SES Inc. (1998)
26. Shlaer, S., and Mellor, S., *Object Lifecycles: Modeling the World in States*, Prentice-Hall, NJ (1992)
27. Xie, F., Levin, V., Browne, J., Integrating model checking into object-oriented software development process, Techn.Rep., University of Texas at Austin, Comp. Science Dept. (2000)