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

Natasha Sharygina
Robotics Research Group
The Univ. of Texas at Austin
Austin, TX, 78712
natali@mail.utexas.edu

James C. Browne
Computer Science Department
The Univ. Of Texas at Austin
Austin, TX, 78712
browne@cs.utexas.edu

Robert P. Kurshan
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ, 07974
k@research.bell-labs.com

Abstract. This paper and a companion paper [32] together define, present and apply a methodology for integration of formal verification by automata-based model-checking into a commercially supported object-oriented software development process. This paper defines and illustrates 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, which have enabled successful application of model-checking to verification of hardware systems. The design rules are incorporated into an extended object-oriented development process for software systems. The methodology, including the design rules was applied to a NASA robot control software. The complex robot control system was decomposed into several functional subsystems. Evaluation by model checking of one control intensive subsystem was performed. Results including identification of significant errors in the original robotic control system are demonstrated.

1. Introduction

Problem Statement. Software systems used for control of modern devices are typically both complex and concurrent. Object-oriented development methods are commonly employed to reduce the complexity of these software systems. Object-oriented development systems still largely depend on conventional testing to validate correctness of system behaviors. But validation of system behaviors by conventional testing is simply not adequate to attain the needed reliability since complete testing of systems of any degree of complexity is impossible.

Formal verification of system behavior through model checking [2], 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 the properties of hardware systems. It is natural to consider application of model checking to formal verification of software systems. Application of model checking to software systems, has, however, been much less successful. (Section 5 on Related Work gives an overview of some of the past and current research on the application of model checking to software systems.) 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 object-oriented development processes to representations to which model checking can be applied have generally resulted in very large interconnected state spaces.

A principal result reported in this paper is a set of design rules (Section 3) for development of object-oriented software systems which 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 object-oriented development processes defined in a companion paper [32].

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 robotics software system to attain high reliability. This case study motivates and demonstrates the design rules for

"design for verifiability" and the application of formal verification by model testing to a substantial software system.

The robot control subsystem was originally implemented by a conventional development process in a procedural object-oriented programming language (C++) generally following the Booch methodology. A four-step process to obtain a reliable system was planned.

1. The control subsystem was to be re-implemented as an executable specification in the form of an OOA model (in the S-M methodology) which was to be designed for testability.
2. This executable specification in the form of an OOA model was to be validated as thoroughly as possible by testing.
3. Model checking was then to be applied to OOA model which had been validated by testing to insure predictable behavior for all possible states of the system.
4. The control software was then to be generated by compilation of the validated and verified OOA model.

Commercially supported software systems are available for construction, validation and compilation of the OOA model (Steps 1., 2. and 4.). Model checking was to be accomplished by translation to S/R using the translator reported in [32] and application of the COSPAN [11] model checker.

The control subsystem was redesigned as an executable specification in the form of a Shlaer-Mellor (S-M) [30] (or xUML) OOA model. The OOA model consists of an object (class) model in which classes and their relationships are specified. A class is specified as a set of attributes, which are confined to simple types and a behavior model. The behavior model is specified as a state machine, one state machine for each active class. The state machines are simple Moore state machines. The actions associated with each state are specified in an action language. More detailed information on the xUML OOA models can be found in Section 2, in the companion paper [32] and in [30]. This executable specification can be tested and validated by execution of the OOA model. Testing is greatly simplified since the attributes are simple types and the behavior is 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 [29] 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 designed for testability 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 model-checking. The design rules of OOA methods do not enforce the logical equivalent of "spatial modularity" in software. 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, slicing, abstraction techniques[9],[10],[11] developed for hardware 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.

The major contribution of this work in addition to the development of a set of design rules for construction of OOA models to which verification by model checking can be practically applied is demonstration of the advantages of using the integrated OOA and model-checking methodology for

development of software systems over conventional development methods. Secondary contributions of the work presented here include (i) demonstration of the fact that verification can be used not only as a post facto check on the quality of design specifications but also to influence design decisions and (ii) discovery of serious logic errors in the test-case software.

Summary of Content. Section 2 describes the OOA models and the model checkers used in this research and describes how they are integrated. Section 3 gives the design rules for development of verifiable OOA models. Section 4 describes the case study for the robot control system. Section 5 discusses related work and Section 6 gives conclusions and depicts the directions of the future work.

2. Integration of Model Checking with OO Development

This paper defines, describes and applies the OOA design step in a methodology which integrates model checking into object-oriented software development. The other steps in this methodology are defined and described in a companion paper [32]. Model checking is applied to Shlaer-Mellor OOA models [29] which have executable semantics specified as state/event machines rather than as programs in conventional programming languages. An automata-based approach to model checking is used. The OOA models are automatically translated to automaton model. Predicates over the behavior of the OOA models are mapped to predicates over the automaton models and evaluated by model checker.

Use of OOA models with executable semantics is moving into the mainstream of OO software development. The Object Management Group (OMG) [25] has adopted a standard action language for the Unified Modeling Language (UML) [27]. This action language and S-M OOA semantics represented in UML notation define an executable subset of UML (xUML). The OOA representation used in this research is the S-M OOA as implemented by the capture and validation environment SES/Objectbench (OB) [29]. We are, on the recommendation of Steve Mellor, [private communication] referring to the OOA model we use as xUML.

COSPAN [11], an automated computer-aided verification system, has been chosen for model checking. The semantic model of COSPAN is founded on ω-automata (automata over infinite words). The input language for COSPAN, is S/R, a declarative (non-procedural), data-flow automata-language. The formalization algorithms of the xUML notation are implemented in the Objectbench-SR translator [32]. SES/Objectbench CodeGeneis system [28] is used to generate a turnkey translation of the OOA models built in Objectbench into executable code.

2.1 xUML Notation

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). Its objective is to provide a general overview of the entire domain and situate it in the organization context. 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 transition diagram* graphically represents a state/event model of an object.

The state/event model is a set of Moore state machines that consists of a fixed number of concurrently executing finite state machines. Each *state* of each machine is defined to have an action and a state transition table. 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.

2.2 COSPAN, an Automaton-based Model Checking Tool

COSPAN [11] allows symbolic analysis of the design model for user-defined behavioral traits. 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 [18]. 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 can be specified in COSPAN using 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.

2.3 xUML-to-S/R translation. The translation algorithms that map the asynchronous execution of the xUML OOA models with synchronous parallel semantics of execution of automaton models are reported in a companion paper [32]. The translator maps dynamic sets of classes instances to bounded sets of class instances, maps variables with unbounded ranges to bounded ranges and maps unbounded event queues to finite queues. Details are given in the companion paper.

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*

Executable OOA models enable validation and verification prior to implementation in procedural languages. Objects (classes) at the OOA level are abstractions of entities in the implemented system. Relationships between objects are represented as associations and not as pointers. OOA methods offer constructs – signals in UML, for example – to express state transition 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 provide means for construction of modular designs and support hierarchical representation of the system. They allow software developers to decompose a system into sub-systems, 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 sub-systems.

3.1 "Temporal Modularity" Versus "Spatial Modularity"

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. Spatial modularity is also a important system property for effective testing since strong partitioning makes testing much more effective. However, one can at least partially test a system lacking in "spatial modularity" while verification by model-checking is almost surely precluded for a poorly partitioned system.

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. "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 for OOA models that *constrain* the syntactic structure of 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.

"Spatial modularity" should not be confused with "temporal modularity" or "temporal locality." Temporal locality/modularity is realized when a system design results in the system executing in a sequence of well-defined and bounded localities. Temporal modularity is intrinsic to software systems. Temporal locality/modularity can and usually is realized by most procedural language software systems. But temporal modularity does not imply "spacial modularity". Although a block of code may contain subroutines/functions/includes, and these indeed are "modules", they are "entered and exited" in a temporal fashion that follows the flow defined by the program counter. There is no clean interface between the "calling" program block, and a function it calls: the function may have side effects all over the code (outside its boundaries). And although it may be that the function can be analyzed in isolation, generally the block that calls a function cannot be analyzed without the function. This limits the extent to which software can be broken up into pieces for analysis, and tends to defeat the pursuit of divide-and-conquer methods.

We developed a set of design rules and recommendations that describe the constraints of the structural design. The rules require the given practice to be implemented without exception. The recommendations are the practices that are to be followed unless other, more compelling, logical design or structural design reasons dictate otherwise.

These design rules for OOA models are similar to those given for development of truly object-oriented programs in object-oriented procedural languages such as C++.

3.2 Structural Design Rules

- *Design rule 1:* *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. Access to the values of a class instance should be available 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:* *Logical entities declared within a component should be defined within that component*

For a component to be reusable and verifiable it must be reasonably self-contained. A component may have dependencies on other components. But in order to avoid the situation when functionality of one component can be changed by other components, however, 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.*

A semantically meaningful definition of subtyping, along the lines of Liskov's [22]:

> *"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**"*

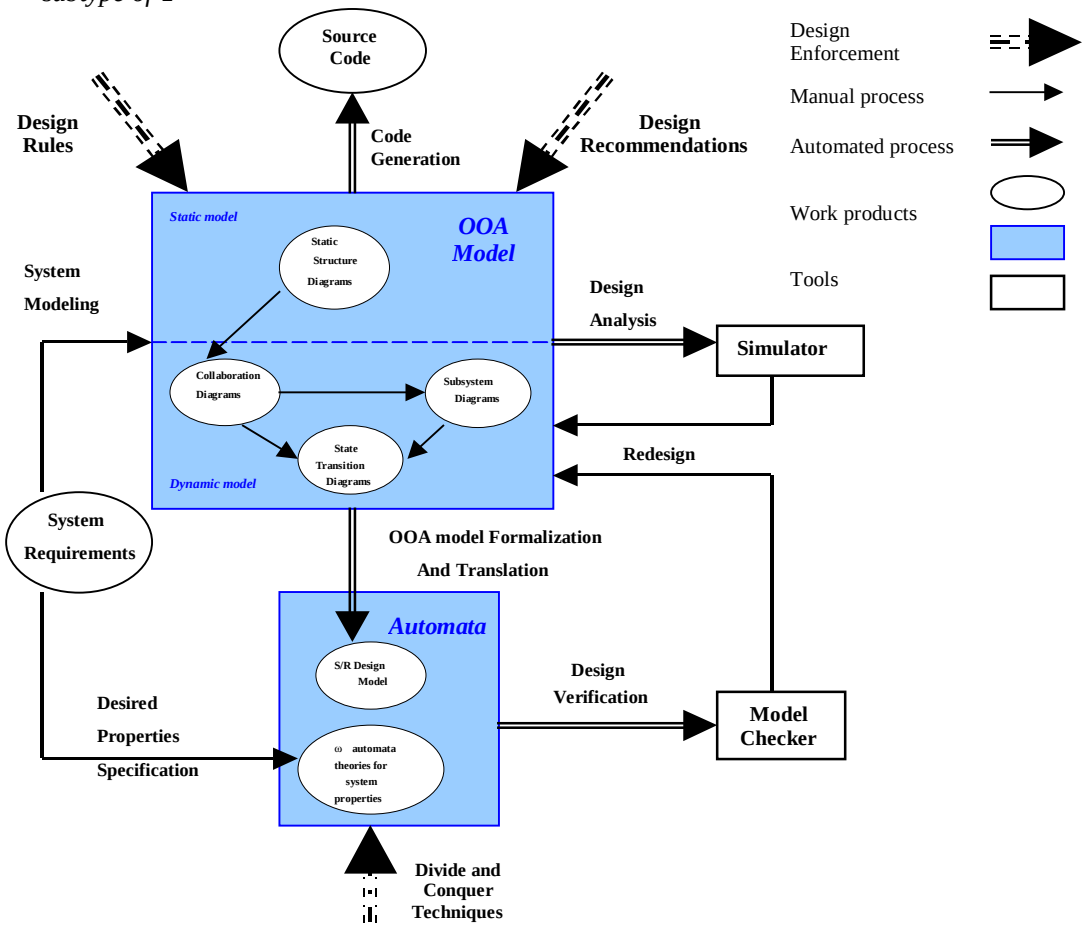

Figure 1. The OOA-based model for the spatial development of software systems

must be followed. This provides a means to infer results about subtype objects on the basis of theorems proved about a supertype. Thus, once we have proved that S is a subtype of T, we do not need to re-prove properties about programs P which rely on T, since by the intuitive definition these can not be invalidated by instances of S.

- *Recommendation: Package subsystems so as to minimize the linking to other subsystems*

A well-designed subsystem will usually contain a substantial proportion of components that do not depend on any other subsystems. Subsystems are fundamentally *open* systems but for verification must be *closed* with a definition of the environment in which they will execute. The naive environment for properties stated in universal logics is the *universal* environment, which is capable of invoking any sequence defined on the unit's interface. A minimal number of links between subsystems enables effective definition of the environment used to complete a subsystems definition for verification.

3.3 The OOA-based model for software development

We defined an OOA-based model (**Fig.1**) for software development that reduces the complexity of software systems at the design level. This model imposes the structural design rules on the software system and supports the reuse of the existing model-checking techniques developed for hardware verification. This model fulfills the need for a sound foundation in rigorous requirements modeling, design analysis, formal verification, and automated code generation. It defines a systematic formal OOA (FOOA) development process. It is a highly iterative software process that eliminates the traditional distinction between software development and maintenance. Furthermore, we claim that this iterative process serves as a development tool. A common conception is that verification is the finale of the specification process – it either shows correctness or reveals problems to be fixed. In our approach verification techniques interleave design and analysis. The complexity only gradually increases as the specification evolves, and verification at early stages is more likely tractable. In addition, analysis results can give fast feedback to designers to improve the cost-effectiveness of this technique.

4. The robot controller software study[1]

We examine a robotic software used for control of redundant robots (i.e. robots that have more than six degrees of freedom). 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 [17].

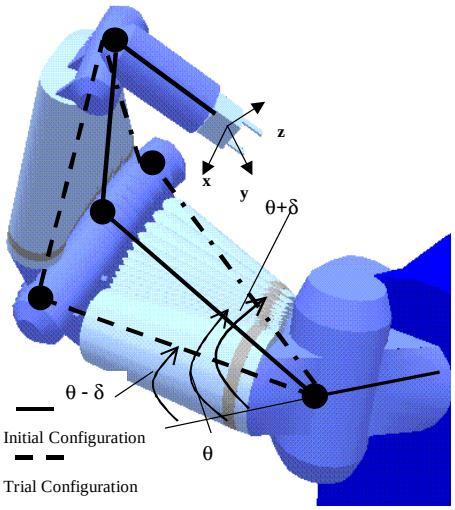

Figure 2. A part of a redundant robot. Note, there are infinite manipulator configurations for a single end-effector position. A "simple" exploration is shown for one of the joints.

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* [17]. 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 is shown in **Figure 2**, with θ - being a joint angle, and δ - being a displacement.

The original software, OSCAR [17], 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 the control subsystem as an executable specification in the form of a Shlaer-Mellor (or xUML) OOA model. An application domain perspective was used to facilitate the development of families of software systems and to implement the interface with the existing robotic computational libraries from within the OOA model. The subsequent sections present the results of the robotic domain analysis and modeling, robotic domain architectural design, the OOA model validation and verification results, and a target system development.

[1] Refer to Appendix A for robotic terms

4.1 Domain Analysis and Modeling

The OOA model includes fifteen basic classes, including their variables and associations. This executable model constitutes the formal representation of the robot controller system.

Classes. In addition to tangible objects (*Arm, Joint, End-Effector, PerformanceCriterion)*, incident objects (*Trial Configuration, SearchSpace, SimpleSearchSpace, FactorialSearchSpace*), specification objects (*Fused Criterion*), and role objects (*Decision Tree, OSCAR_interface, Checker)* were derived [29].

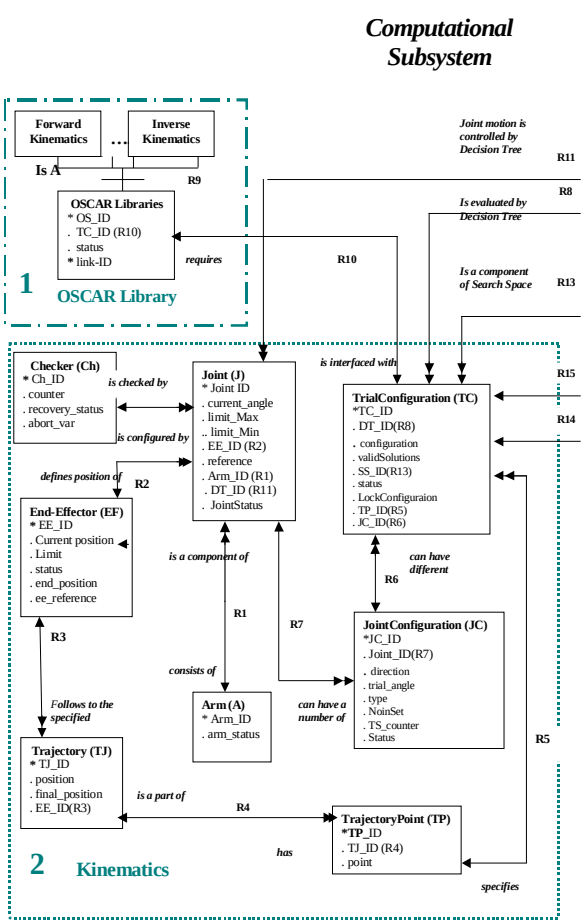

Figure 3a. OID for the Computational Subsystem of the Multi-Criteria Optimal Control domain

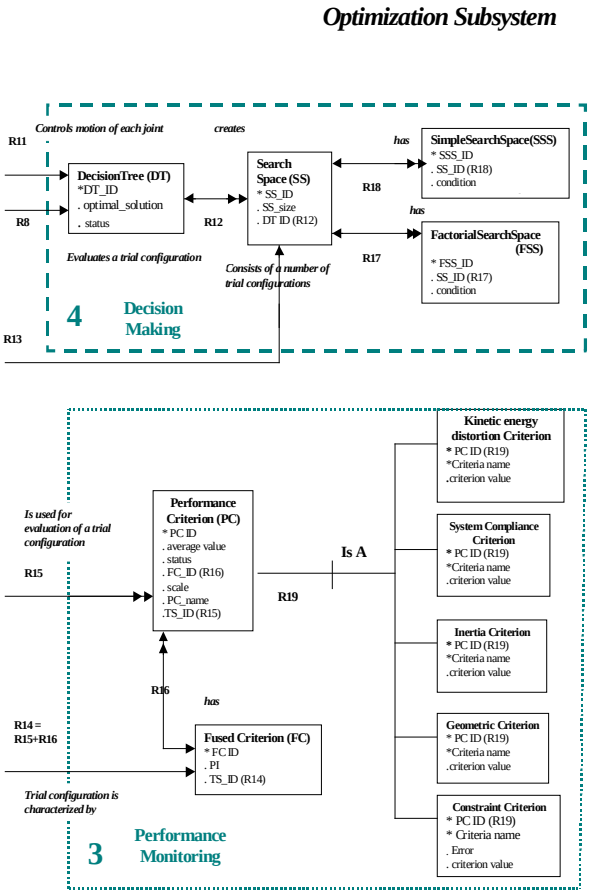

Figure 3b. OID for the Optimization Subsystem of the Multi-Criteria Optimal Control domain

Attributes. The attributes of the *JointConfiguration (JC)* object are illustrated below. *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 (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", "fact+delta", "fact-delta"}. Attribute *Type* domain is {"simple","factorial"}.

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. These two principal functions can be thought of as clusters, each representing a group of objects that are interconnected by many relationships. 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 (**Fig. 3a**) 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 *End_Effector (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 simplified *Kinematics* unit that we verified is represented in **Figure 7a.** It displays the flow of signals of this functional unit in terms of a sequence of data or control signals exchanged among the objects in the system. A transition is represented as an arrow originating from a source object to a destination object.

A robot arm consists of N joints and one end-effector. The end-effector is the last link of the robot, used to accomplish a task. The end-effector may be holding a tool, or the end-effector itself may be a tool. These physical entities are represented by the classes *Arm*, *Joint*, *EndEffector* in the software design. For each joint we specify an angle (*current_angle*), representing a rotation of the joint relative to its neighboring link. The end-effector position (*Current_Position*) is given as vector of three coordinate positions and three orientation angles.

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 [17]. 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 [17] 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 sends a command to the *Arm* class indicating its invalid state.

During the development of this software, one is mainly concerned with the satisfaction of the reliability requirements. In the presented system the ultimate goal is to ensure that the end-effector is moving only when the arm is indeed in a valid state.

The optimization subsystem (**Fig. 3b**) 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. The development of the robotic system, which structure is compliant with these rules, required a process of refining and distilling the designs over several iterations.

Rule 1: Target system requirements elicitation was performed using global data declaration. An API has been implemented which enables a dialog with the robotic system engineer, presenting the user with the optional features, available for selection for the target system.

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

Rule 3: We enforced a rule that prohibited redefining shared variables outside of the scope of the object where they were declared.

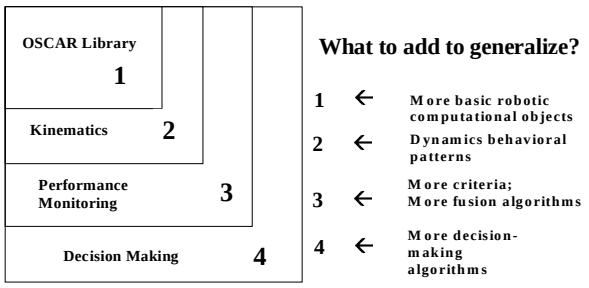

Figure 5. Functional components layout of the robotic domain

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 word, inheritance is restricted to a purely syntactic role: code reuse and sharing, and module importation.

Recommendation (System decomposition): The robotic domain is represented as a collection of basic robotics functional units, which provide an experimental framework for evaluation of the kinematics and decision support algorithms. These functional components are depicted by dashed lines in **Figure 4. Figure 5** specifies the generalization of these functional components. *OSCAR Libraries (1)* alone can be used as an element base for the manual writing of robotics applications. *OSCAR Libraries* combined with *Kinematics (1+2)* algorithms can lead to the automatic development of the robot controllers. Controllers that have an ability to perform *Performance Monitoring (1+2+3)* can be qualified as robotic intelligent controllers though an optimization problem would not be considered. All four components *(1+2+3+4)* used together allow development of controllers, which can define robot optimal configuration. As it can be seen in **Figure 4** each functional unit contains a substantial proportion of components that do not depend on other units.

4.3 OOA Model Validation and Formal Verification

The OOA model was validated by execution. 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 a Technical Report [26]. The critical observation is that application of verification by model checking would have been seriously compromised until these flaws were corrected.

Formal verification

Verification properties. We checked a collection of correctness requirements specifying the coordinated behavior of the robot control processes defined in the simplified version of the *Kinematics* unit (**Fig. 7a**). We focused on the intensive robot control algorithms and examined an instance of the robot functionality when the robot arm was moving only in the horizontal, i.e. x direction. We abstracted the *Trajectory, Trajectory point* and *JointConfiguration* classes, used for storage of the predefined trajectory paths and the possible arm configurations, used in the redundancy resolution algorithms, respectively. The requirements were encoded in a query language of COSPAN. We expressed all the formulae in terms of state predicates. We demonstrate here the properties that

did not hold during the verification. In our description we refer to the states appearing in the state transition diagrams of the *Arm*, *EndEffector* classes. **Figures 6** and **Figure 7** schematically represent the lifecycles of the *Arm* and *End-Effector* classes (some actions are omitted due to the space limitations of the paper). The state transition diagram of each process consists of nodes, representing states and their associated actions to be performed, and event arcs, which represent transitions between states.

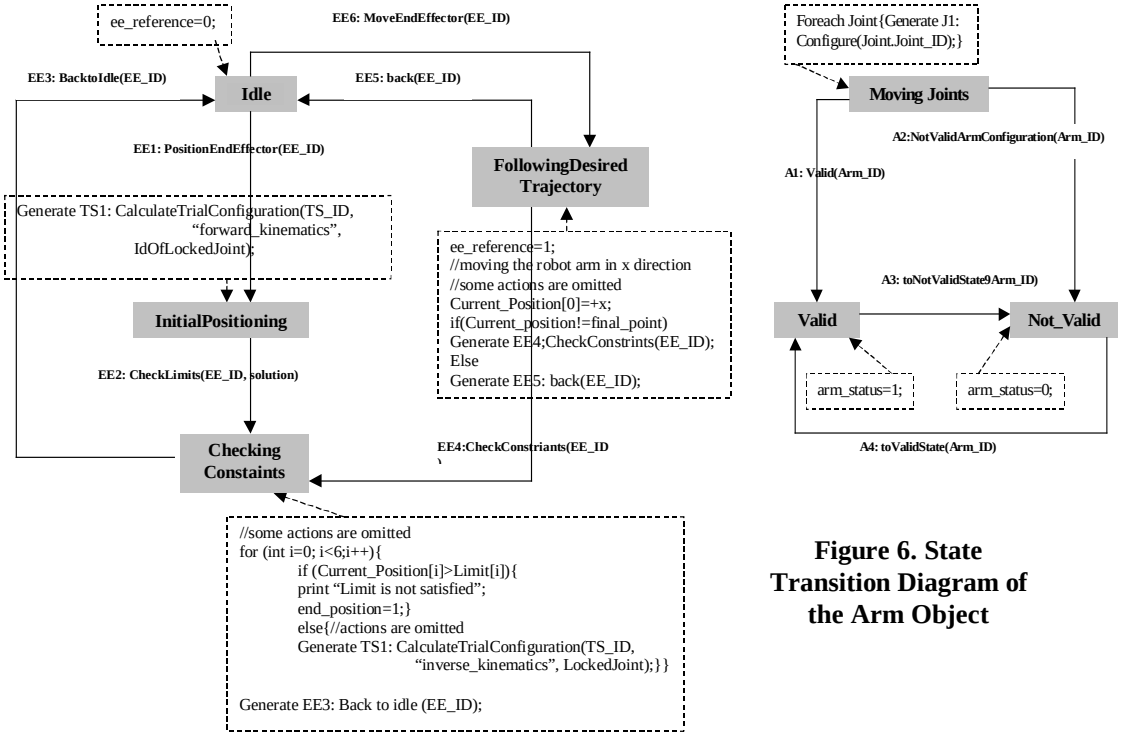

Figure 6. State Transition Diagram of the Arm Object

Figure 7. State Transition Diagram of the EndEffector Object

We present here the description of the properties that did not hold during the verification of the *Kinematics* unit. In this description we refer to the states and variables of the *Arm*, *End-Effector* and *Checker* objects. For example, the state *FollowingDesiredTrajectory* appears in **Figure 7**.

1. **Deadlock Freedom** The model does not have deadlocks.

2. **Always(ee_reference=1) → (arm_status=1)**

   at any point in the execution, if *EndEffector* process is in the *FollowingDesiredTrajectory* state (*ee_reference* variable is equal to 1) than the *Arm* process is in the `'Valid" state (*arm_status* variable is equal to 1).

   Or in terms of the Kinematics domain

   The end-effector is moving only when the arm is indeed in a valid state.

3. **Eventually (abort_var = 1 Until (end_position = 1 AND (recovery_status =1 OR couner=number_of_joints))**

   In any execution of the program there is a state where a variable *abort_var* of the *Checker* process is equal to 1 and it continues to be that way until either the *end_position* variable of the *EndEffector* process is equal to 1 or both of the following are true: the *Checker* process variables *recovery_status* is equal to 1 and the variable *counter* is equal to the number of active instances of the *Joint* class.

   Or in terms of the Kinematics domain

The program terminates when it either completes the task or violates the constraints (*end_position*=1) or reaches the state where there is no solution for the fault recovery (when all joints of the robot arm violate the joint limits).

In order to reduce the complexity of the original design we had to abstract and restrict some

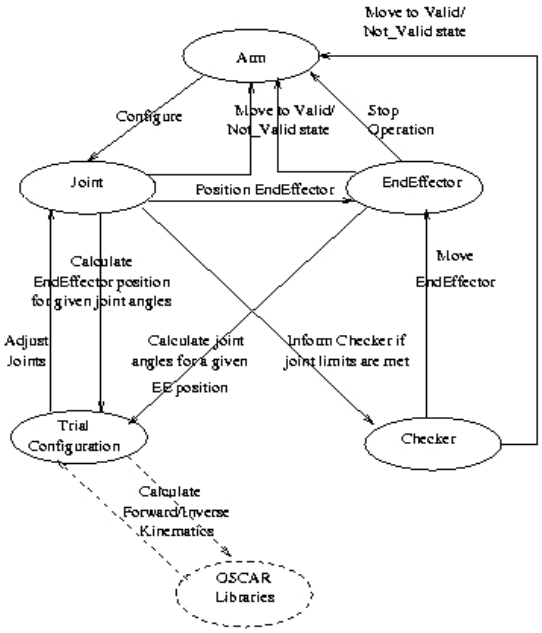

Figure 7a. Collaboration diagram of the *Kinematics* unit

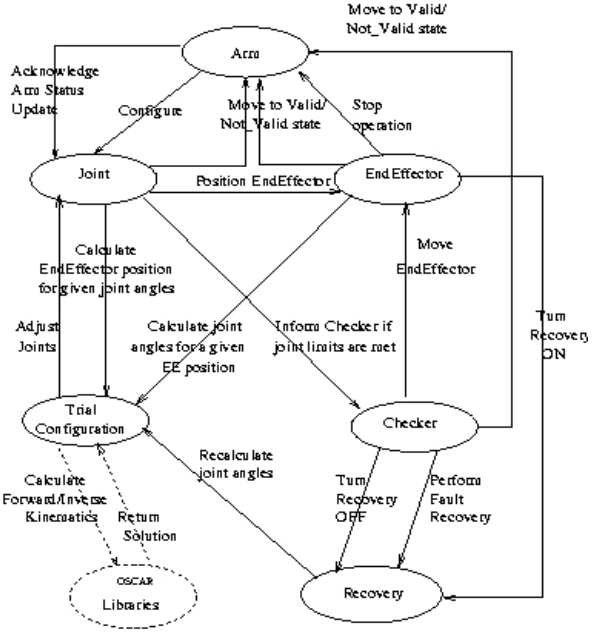

Figure 7b. Modified collaboration diagram of the *Kinematics* unit

calculations. In fact, in the OOA model the robot arm movement calculations are done through the interface with the OSCAR libraries. During verification we abstracted the actual calculations and replaced them with nondeteministic assignments of small natural numbers. Scaling of the object attribute values has been enforced in order to avoid dealing with the rational numbers that were widely used in the original code. Definitions of some parameters that were not related to the verified property were abstracted.

Verification results. The absence of the deadlocks in the model execution was checked. We found that a deadlock existed. Analysis of the error track produced by COSPAN revealed that it was due to an error in the fault recovery algorithm. The failure of the Property 3 that was aimed to check if the system terminates properly confirmed that finding. The system did not terminate in the case when there was no solution for the fault recovery. We will remind the reader that the fault recovery procedure is activated in the robot control 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. Specifically, we considered a simple example when the arm consisted of two joints. Property 3 failed since in this example requests originated from Joint1 and Joint2 to recompute the angles of these joints continued indefinitely: when Joint1 did not respect the limit then the fault recovery was called and Joint1 was locked with the angle limit value. The Joint2 angle was being recalculated for the original end-effector position. In a situation when a new angle of Joint2 did not satisfy its limit then another fault recovery procedure was called, which attempted to find a new angle for Joint1 while Joint2 angle was locked. The exhaustive search provided by COSPAN exposed the situation when there was no resolutions that satisfied the limit for Joint1 and the fault recovery was called again. This was also a confirmation of the above deadlock situation.

Another error was found during verification of Property 2 indicated a problem of coordination between the *Arm* and *EndEffector* processes. The original design assumed a sequential execution pattern. In fact, it was expected that the Arm.status variable of the *Arm* process would be repeatedly updated before the *EndEffector* would switch to the FollowingDesiredTrajectory state, where the

EndEffector.status variable changes its value from 0 to 1. A concurrent interaction between the processes led to the situation where the update of the EndEffector.status variable precedes the change of the Arm.status value. This was the reason for the demonstrated property 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 7b** reflects the changes made. We introduced a new class called *Recovery*, whose functionality provides a correct resolution of the fault recovery situation described above. Additionally we added several exchanges of messages between the processes *Arm* and *Joint* in order to fix the coordination problem reported earlier.

4.4 Robotic System Engineering

When validation and verification of the analysis model was completed, the architecture was automatically compiled into C++ programming language (therefore, creating sets of C++ templates). These templates were compiled into object code and linked together to form an executable model.

Given the target system specifications, the developed architecture, and the target system configuration parameters, an instance of the target robotic system was composed. 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. Related Research

The focus of this project is integration of formal verification by model checking into a commercially supported object-oriented development process based on a subset of UML. The focus of this paper is design and development of OOA models to which model checking can be applied and a case study in the application of model checking to OOA models to a non-trivial software system.

Lilius and Paltor [20] describe a tool (vUML) for application of model checking to UML models via translation to Promela and application of SPIN [31]. No applications of the tool are given and design of OOA models to generate manageable state spaces is not discussed.

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 [14, 15] targets software systems written in C while [1], [2], and [3] focus on applying model checking on SDL programs. Havelund and Pressburger [13] apply model checking to Java programs. Corbett, et.al [6] extract finite state machines from Java programs to which to apply model checking.

Model-checking has been also applied to verification of concurrently executing state/event machines. Lind-Nielsen, et al [21] applied SMV [23] 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. [7] 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 researchers [19]. Moors [24] 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.

Finally there has been a great deal of research on formalization of object-oriented models and languages. This research is largely concerned with integration and application of verification based on theorem proving rather than model checking. There follow a few representative citations. Hubmann [16] gives a formal foundation of SSADM, the Syntropy method based on Z and statecharts. Dodani and Rupp enhance [8] the Fusion method by formal specifications written in COLD[12], Lano[19] presents a formal approach to object-oriented software development based on Z++ and VDM++. Formal semantics for interaction diagram using algebraic specification with rewriting logic is given in [33].

6. Conclusions and Future Research

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 object-oriented 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 have been proposed and applied.

Future work includes the following:

- Development of a complete set of rigorous rules that would be used as a part of the OOA methodology supporting practical application of model-checking;
- Introduction of a mechanism that enables *a designer* to reason about verifiable properties and automatically extract them from the OOA design specification for translation to SR COSPAN;
- Complete verification of the robot controller software:
  - Integration of real-time constraint into the OOA system description and it verification.
  - Collaboration with the development team for the translation system to extend it to cover additional features of the xUML and to optimize the translation process to further reduce the state space of the automaton models.
- Implementation of design techniques that facilitate the model checking:
  - We propose to support *slicing* by abstraction of parts of the control flow that are not related to a property to be verified at the design level.
  - We suggest a technique of *labeling* some parts of state machines that are not related to the verified property in order to support event/states encapsulation performed by the model checker. The labels would be interpreted by the translator and could support the translator algorithms for:
    - encapsulation of events interaction that does not effect the property;
    - substitution of a number of events that are not related to the property that is to be verified by an abstract event that when implemented will give rise to many events being sent to the same block.
- We plan to support an *assume-guarantee* style model checking to reason about correctness properties of software units. We propose to construct an environment model to close a system for verification using *state/event machines* (in contrast of using LTL and CTL specification [9]). This would allows the *designer* not only to generate all possible combinations of environmental behaviors but also to set up the priorities for some threads of control (and if needed to exclude those which are less important from the verification process) in regard to the property to be verified. This environment model is to be automatically translated as an assumption into syntax accepted by a model checker.

References

[1] Bounimova, E., Levin, V., Basbugoglu, O., and Inan, K., 1996, A Verification Engine for SDL Spec. Of Comm. Protocols, *Proc. of the First Symposium on Computer Networks*, Istanbul, Turkey, pp. 16-25.

[2] Bozga, M., Fernandez, J., Ghirvu, L., Graf, S., Krimm, J., Mounier, M., Sifakis, J., 1999, IF: An Intermediate Representation for SDL and its Applications. *Proc. of the SDL Forum*, Montreal, Canada.

[3] Bosnacki, D., Damm, D., Holenderski, L.,and Sidorova, N., 2000, Model checking SDL with Spin, *Proc. of the Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany.

[4] Clarke, E., and Emerson, E., 1982, Design and Verification of Synchronization Skeletons for Branching Time Temporal Logic, *Pr. of Log. of programs Workshop*, Yorktown, NY, Springer *LNCS* 131, pp. 52-71.

[6] Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., 2000, Bandera: Extracting finite-state models for Java source code, *Proceedings of 22nd ICSE.*

[7] Chan, W., Anderson, R., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J., 1998, Model Checking Large Software Specifications, *Proceedings of IEEE Transaction on Software Engineering*, pp. 498-519.

[8] Dodani, M., and Ropp, R., 1995, Integrating Formal Methods with Object-Oriented Methodologies. In Martin Wirsing, editor, ICSE-17 Workshop on Formal Methods Applicaiton in SE Practice, pp. 212-219.

[9] Dwyer, M., and Pasareanu, C., 1998, Filter-based model checking container implementations, *In Proceedings of the 6th ACM SIGSOFT.*

[10] Granberg, O., and Long, D., Model Checking and modular verification, *ACM Transactions on Program languages and Systems*, V. 16, pp. 843-872
[11] Hardin R., Har'El, Z., and Kurshan, R.P., 1996, COSPAN, *Proc., CAV'96, LNCS, Vol. 1102*, pp. 423-427.
[12] Hans, B., Jonkers, 1989, An Introduction to COLD-K. In Martin Wirsing and Johannes A. Bergstra, Algebratic Methods: Theory, Tools and Applications, v.394 of lect. Notes on Comp.Science, pp.139-206.
[13] Havelund, K., and Pressburger, T., 2000, Model Checking Java Programs Using Java PathFinder, Int. Journ. on Soft. Tools for Techn. Transf. (STTT) 2(4), 1998, 4th SPIN workshop.
[14] Holzmann, G., and Smith, M., 2000, Feaver: Automating software feature verification, Bell Labs Technical Journal, Vol. 5, 2, pp. 72-87.
[15] Holzmann, G., and Smith, M., 1999, A Practical Method for Verifying Event-Driven Software, *Proceedings of the 21st ICSE*, p. 597-607.
[16] Hubman H., 1994, Formal Foundations for Pragmatic Software Engineering Methods. In Bernd Wolfinger, editor, Innovationen bei Rechen- und Kommunikationssystemen, pp.1-50.
[17] Kapoor, C., and Tesar, D., 1996, "A Reusable Operational Software Arch. for Adv. Robotics (OSCAR)", Un.of Texas at Austin, Rep.to DOE, Grant No. DE-FG01 94EW37966 and NASA Grant No. NAG 9-809.
[18] Kurshan, R., 1994, *Computer-Aided Verification of Coordinating Processes – The Automata-Theoretic Approach*, Princeton University Press, Princeton, NJ.
[19] Lano, K., 1997, *Formal Object-Oriented Development*, Springer.
[20] Lilis, J., and Porres, I., 1999, vUML: a tool for Verifying UML Models, *Proceedings of ASEC*, Fl.
[21] Lind-Nielsen, J, Andersen H., R., etc., Verification of large State/Event Systems using Compositionality and Depenedency Analysis, *Proceedings of TACAS'98*, Portugal.
[22] Liskov, B., Data Abstraction and Hierarchy, 1987, *Proceedings of OOPSLA conference.*
[23] McMillan, K. *Symbolic Model Checking*, 1993, Kluwer.
[24] Moors, T., 1998, Protocol Organs: Modularity should reflect function, not timing, *Proc. OPENARCH98*, pp. 91-100.
[25] Object Management Group (OMG), 2000, *Action Semantic for theUML*, OMG.
[26] Sharygina, N., and Browne, J., 1999, Automated Rob. Decision Support Software Reverse Engineering, Tech. Rep., The Univ. Of Texas at Austin, Robotics Research Croup.
[27] Rumbaugh, J., Jacobson, I and Booch, G, 1999, *The Unified Modeling Language Reference Manual*, Object Technology Series, Addison-Wesley.
[28] SES Inc., CodeGenesis User Reference Manual.
[29] SES Inc., ObjectBench Technical Reference.
[30] Shlaer, S., and Mellor, S., 1992, *Object Lifecycles: Modeling the World in States*, Prentice-Hall, NJ.
[31] *The Spin Model Checker*, IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295.
[32] Xie, F., Levin, V., and Browne, J., 2000, "Integrating model checking into object-oriented software development process", submitted to FASE2001, copy available from browne@cs.utexas.edu
[33] Wirsing, M., Knapp, A., 1996, A formal approach to Object-Oriented Software Engineering, In José Meseguer, editor, *Proc. 1st Int. Wsh. Rewriting Logic and Its Applications*, volume 4 of Electr. Notes Theo. Comp. Sci., pages 321-359. Elsevier.

Appendix A: Summary of basic robotics definitions used in the paper

Degrees of Freedom (DOF) is the number of independent position variables which would have to be specified in order to locate all parts of the mechanism.
End-effector is the robot's last link. The robor uses the end-effector to accomplish a task. The end-effector may be holding a tool, or the end-effector itself may be a tool.
Forward kinematics problem is the determination of the position and orientation of the end-effector given the joints parameters.
Constraint is a restriction that limits the value of a dependent or independent variable. Inequality constraints limit the robot's joint travel (joint limits), joint speeds (speed limits), and torques (torque limits).
Inverse kinematics problem is to find the robot's joint states given position and orientation state values for the robot's end-effector.
Performance criterion is a measure based on kinematic and dynamic models of the robot useful for evaluating the state of the robot.
Redundant robot is a robot with more independent joints than equality constraints on the placement of the end-effector.
Redundancy resolution is a decision making problem of finding the trajectory $u(t) \in U$ for a given trajectory $x_e(t)$ in the task space such that u(t)

- solves the corresponding Inverse Kinematics problem, i.e., there is $\phi(t) \subseteq u(t)$ for which $f(\phi(t)) = x_e(t)$ for $t \in [t_0, t_f]$;
- enhances the robot's performance through optimization of a set of performance criteria;
- satisfies the robot's configuration limits and end-effector constraints.