
Formal Methods in System Design, 25, 241–270, 2004

Lessons Learned from Model Checking a NASA Robot Controller

NATASHA SHARYGINA nys@sei.cmu.edu
School of Computer Science and Software Engineering Institute, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213, USA

JAMES BROWNE browne@cs.cmu.edu
FEI XIE feixie@cs.cmu.edu
School of Computer Science, The University of Texas at Austin, Austin TX 78712, USA

ROBERT KURSHAN rkurshan@cadence.com
Cadence Design Systems, Inc., 571 Central Avenue, New Providence, NJ 07974, USA

VLADIMIR LEVIN vladlev@microsoft.com
Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, USA

Received August 25, 2003; Revised February 10, 2004; Accepted March 24, 2004

Abstract. This paper reports as a case study an attempt to model check the control subsystem of an operational NASA robotics system. Thirty seven properties including both safety and liveness specifications were formulated for the system. Twenty two of the thirty seven properties were successfully model checked. Several significant flaws in the original software system were identified and corrected during the model checking process. The case study presents the entire process in a semi-historical mode. The goal is to provide reusable knowledge of what worked, what did not work and why.

Keywords: software verification, component-oriented software development, abstraction, compositional reasoning, executable design specifications

1. Introduction

Motivation. Control systems ranging from smart cards to automated flight control are increasingly being implemented through software systems. Errors in these software systems can cause severe consequences. Verification by model checking has great potential for enhancing the correctness and thus the reliability and robustness of software systems. But application of model checking to software systems is still in an early stage of development. There have been few in-depth reports of case studies or systematic analyses on model checking of non-trivial software systems. There is little conventional wisdom for application of model checking to software systems although successful application of model checking to hardware systems gives guidance for system design.

This paper gives an end-to-end case study of the application of model checking to a significant software system, the control subsystem of a NASA robotics system. The goal is

to provide reusable knowledge of what worked and what did not work during model checking of a control software system and why. Model checking of a software system for a property can be accomplished only if the state space for checking the property on the software system is of tractable size. Therefore much of this case study was devoted to obtaining representations of software systems which both faithfully implement the functionality of the software system with respect to a property and at the same time have tractable state spaces.

Project overview. The goal of the project was model checking the control subsystem of the OSCAR (Operational Software Components for Advanced Robotics) [17]. OSCAR forms the basis of an operational robot control system designed for basic manipulator control extended with robot performance optimization. OSCAR incorporates kinematics, control, performance optimization and an operator interface. The subsystem of OSCAR which was model checked amounted to about 45,000 lines of C++ in the original code. The model checking system used in this case study was the OBJECTCHECK [45] system. Software systems to be model checked are specified in XUML [43], translated to the SR language (input language of the COSPAN model checker [13]) and model checked by COSPAN. The case study took over two years to complete and was effort intensive.

The project was a set of learning stages which evolved into an orderly process. The verification process which emerged from the project may be applicable to other large-scale control software systems. The presentation is "semi-historical". The presentation imposes some order on the learning process and reports on efforts that failed and from which we learned.

Three core concepts emerged from the project. The first is that software systems which are to be verified by model checking should be designed to enable model checking. This concept was enabled by defining a design template [39] for constructing model-checkable systems in XUML. A second core result was development of a structure and behavior specific state space reduction algorithm [38] targeting loop-intensive control system software. The third basic result is that use of executable design level specifications which can be tested at design level but compiled to conventional procedural programs bypass some of the sources of difficulties and errors in directly model checking programs in procedural code. The "many model" problem is avoided since all operations (i.e., property specification, finite state model extraction, abstraction, decomposition, etc.) can be made on the same (design-level) representation or result from an automated transformation of the representation. Property specification is more simple for XUML programs than for procedural programs.

Additionally, this case study provides extensive analysis of the effectiveness of various existing state space reduction techniques to software systems and provides recommendations for the systematic and effective application of integrated state space reduction during verification of software.

Paper structure. The paper is organized as follows. Section 2 presents the verification framework. Section 3 describes the robot controller system and discusses initial attempts to model check software. Section 4 defines properties of the robot controller system. Section 5 presents the verification results for the integrated state space reduction approach. Section 6 presents a loop abstraction technique and demonstrates its advantages during verification of the robot control algorithms. Section 7 gives a summary of the verification results and

reports errors found by model checking. Section 8 summarizes the lessons we learned and gives a set of recommendations on how to use model checking in the context of large-scale systems. Section 9 draws conclusions and outlines related work.

2. The verification framework

This project was executed using the capabilities provided by the OBJECTCHECK [45] system. OBJECTCHECK integrates a commercial environment [36] for development of software systems as executable design level specifications in XUML [40, 43] with the COSPAN [13] model checking system. OBJECTCHECK integrates model checking into a total software development process for systems specified in XUML. This project overlapped the development of the OBJECTCHECK system and motivated development of some of OBJECTCHECK'S capabilities and features.

2.1. The software representation

XUML [40, 43] is a dialect of UML [5] with an executable semantics. XUML representations, while they are considered design level representations, have an executable semantics and can be tested by execution and automatically translated to procedural languages to obtain a conventional implementation of the software system. A software system designed in XUML is a composition of objects where the behavior of each object is controlled by a state machine communicating asynchronously with other object state machines through non-blocking FIFO buffers. State transitions and actions are triggered by inputs from the buffers and perform variable assignments, and signal outputs. XUML programs have an asynchronous interleaving semantics. XUML is fairly widely used for implementation of embedded software systems. Commercially supported development environments including testing and validation and code generation are available [4, 18, 35, 36]. The commercial toolset OBJECTBENCH [36] developed by HyPerformix Inc. is used in the OBJECTCHECK system. It includes a graphical editor, testing and animated debugging of the model and a compiler for translation from the model to C++ code. OBJECTBENCH was used for capturing and validating XUML designs of the robot controller software, for visualizing communication sequences among the objects composing a system contained in the error traces generated by a model checking tool and for replaying these sequences on the XUML specification to ease error identification.

2.2. Translation tools

The core of OBJECTCHECK is a translator from XUML to the SR language of COSPAN. OBJECTCHECK also provides capabilities for translation of properties specified in the namespace of the XUML program, for mapping and replaying COSPAN generated error tracks in the OBJECTBENCH animated debugger and for applying source to source transformations to implement abstractions and state space reduction algorithms. The OBJECTCHECK translator, XUML2SR, converts the original graphical representation of an XUML system into

a textual form and then translates it, together with the property to be model checked, into operationally equivalent SR code. The translator may apply transformations to implement an abstraction appropriate for the property to reduce the state space of the state-transition graph specified by the XUML system including variable range bounding and static partial order reduction, SPOR [20].

2.3. Verification tools

COSPAN [13] uses the automata-theoretic approach to model checking [21]. Verification is by the automata language containment test. Language containment can be checked in COSPAN by either a symbolic (BDD-based) algorithm or an explicit state space enumeration algorithm. COSPAN implements multiple automated state space reduction algorithms including localization reduction, automated predicate abstraction, partial order reduction and also supports an assume-guarantee style of compositional reasoning. When COSPAN applies the state space reduction techniques it transforms a given SR program into a semantically equivalent one (with respect to a property or a set of properties) with a reduced state space.

2.4. Verification methodology

The methodology is described here as it finally evolved, not as we began the research. An essential element of the research is that abstractions and state space reduction algorithms are applied to both the software design model and model checkable representations.

The methodology for verification of software was devised as follows. An XUML representation of the software system which conforms to the design methods given in Sections 3, 6 is prepared. This XUML program is tested and validated just as though code were to be generated for the program without model checking. Model checking is then applied as follows:

1. The property is defined in an XUML representation of the property specification language of the COSPAN model checker [45].
2. The abstractions and state space reduction algorithms to be applied at the XUML model level are selected.
3. The model and the property are translated to SR with application of the selected abstractions and state space reduction algorithms.
4. The translated system is model checked by the COSPAN model checker with different choices of state space traversal methods and state space reduction algorithms.
5. If the property does not hold the error track provided by COSPAN is translated to an XUML representation and replayed in the development environment to diagnose the error.
6. If model checking cannot be completed (a common occurrence!), then the property and the abstraction are revisited and alternative or new state space reduction methods are applied.

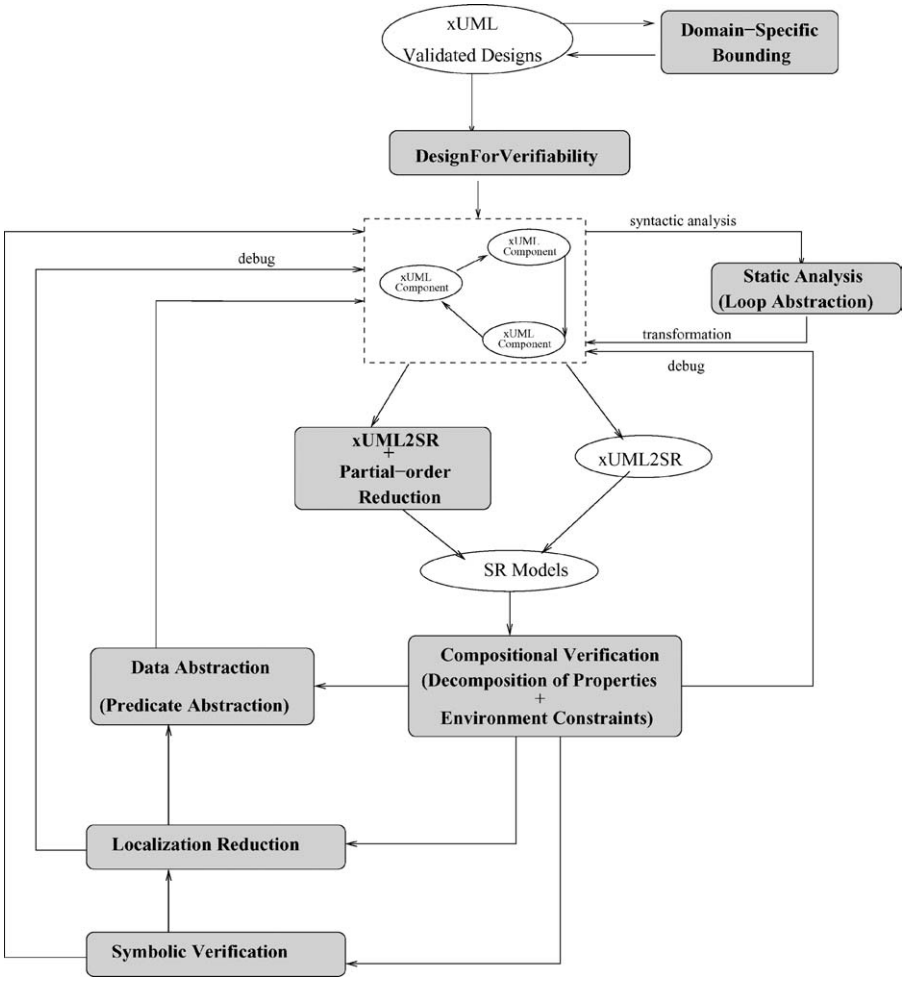


Figure 1. Integrated state space reduction approach.

The integrated design- and model checking-level state space reduction methodology is illustrated in figure 1. Steps 1–3 are applied at the design level of software thus enabling design level property specification and program abstraction. The design level property specification abstraction is done by the software designers. Design-level abstractions can be defined and applied based on the domain knowledge without constructing the state graph of the program. Several design techniques for specification of XUML models were defined. The design level techniques include: (a) domain-specific bounding of program variables; (b) a design method which enables application of compositional model checking and (c) a loop abstraction technique that reduces the state space of the verifiable system by compressing loop transitions. These techniques were designed to create a model checkable/reduced complexity version of software. Step 4 is the model checking step, where we attempted to define

a systematic approach to application of various state space reduction techniques. Reduction techniques supported by OBJECTCHECK and COSPAN systems that were used in this work include localization reduction [21], symbolic verification[24], automated predicate abstraction [30], static partial order reduction [20], and assume-guarantee style of compositional reasoning [21]. OBJECTCHECK and COSPAN state space reduction techniques are discussed next. The design-level techniques are presented in Sections 3 and 6. The strength of the methodology is in the symbiosis of the design and model checking-level state space reduction techniques.

2.5. *State space reduction techniques*

Compositional reasoning. The compositional reasoning approach aims to establish whether for given programs M_1, M_2 and specification T, the composed system satisfies T (written $M_1 \parallel M_2 \models T$). A naive compositional approach proceeds by executing the following steps: (1) $M_1 \models T$ and (2) $M_2 \models T$, and conclude by proofs that $M_1 \parallel M_2 \models T$. Though, this rule is sound in theory, it is often not useful in practice—both M_1 and M_2 usually satisfy T only in a suitable environment. To solve this problem, the compositional principle can be strengthened to an *assume-guarantee principle*: in order to check $M \models T$, it suffices to check local properties T_1 and T_2 of local components M_1 and M_2 respectively: $M_1 \parallel T_2 \models T_1$, $M_2 \parallel T_1 \models T_2$. This obligation uses the local specification T_1 as the constraining environment with regard to the behavior of M_2 taken in isolation from M_1, and it uses T_2 to constrain M_1 from M_2. In general, for a system composed of multiple processes, assume-guarantee reasoning succeeds as long as it can be shown that each system component, M_i, satisfies a corresponding specification component, T_i, under a suitable constraining environment.

In this work we applied assume/guarantee reasoning based on the property decomposition type of compositional verification [21]. Verification proceeded through the following two steps:

1. Decomposition of global properties of a system into a set of *local* properties of the system components;
2. Verification of each property on the corresponding component. The procedure entailed using a particular environment representing an abstraction of the remaining components. Scalability of the compositional verification was addressed by specification of *abstraction constraints* of varied complexity.

Localization reduction. Given a model and a property, COSPAN automatically applies *localization reduction* [21] (also known as cone of influence reduction [8]). Localilization reduction is an iterative abstraction/refinement algorithm. Variables of the model not dependent on the variables in a property are assigned non-deterministic abstract values and the abstract model is checked for the property. If the property is satisfied on the abstracted system then the property is also satisfied for the original system.[1] If a counterexample is generated on the abstracted model and it is spurious for the concrete program, it is used to refine the set of variables to which non-deterministic values have been assigned. This

process is continued until either the property is satisfied, a counterexample is generated on the original system or the process exhausts its resources. In practice, since the set of abstract values is defined over a small domain, this process always terminates.

Partial order reduction. Under *partial order reduction* (POR) [31], the state graphs are reduced because properties are verified without exploring all interleavings of executions of independent transitions. In the framework of this report, POR was used in combination with a set of other reduction techniques (as specified in figure 1). The integrated application of POR was made possible by using *static* POR, SPOR. The SPOR transforms XUML models using the procedure specified in [20] prior to translation to SR by restricting the XUML transition structures with respect to a verifiable property.

Symbolic model checking. Represents the state transition structure of an xUML model with binary decision diagrams, which enables manipulation of entire sets of states and transitions instead of individual states and transitions. Symbolic verification has been reported to be highly effective in verification of industrial size hardware systems and it is a primary model checking approach in hardware verification.

Predicate abstraction. Introduced by Graf and Saidi [12], is a popular form of over-approximation. The basic idea of predicate abstraction is to replace a concrete variable by a Boolean variable that evaluates to a given Boolean formula (a predicate) over the original variable. This concept is easily extended to handle multiple predicates and, more interestingly, predicates over multiple variables. Replacing concrete transitions with abstract transitions can be performed automatically with the aid of decision procedures. It can take place dynamically during the state graph generation or statically before the state graph generation.

We used a prototype predicate abstraction tool supporting the predicate abstraction algorithm reported in [30]. The abstraction tool performs syntactic program transformation prior to construction of the system state graph. As a result it permits application of other state space reduction methods during model checking.

Domain-specific bounding. Model checking can be accomplished only for finite state systems. In adopting model checking for verification of infinite systems, the usual practice is to perform discretization and bounding of the program variables. That is usually done during modeling of the program specifications. For example, a continuous infinite data type, like the real type, is discretized and represented by an integer interval type. Integer variables are given bounded ranges sufficient to capture all possible values which can occur in any execution of a program. This work uses *domain-specific bounding* of program variables to reduce the variable domains to minimum sizes. The bounding is performed in the XUML models using the XUML annotation language [36].

3. Design and analysis of the robot controller system (RCS)

The test-bed software system implements the NASA robot control algorithms developed by the Robotics Research Group of the University of Texas at Austin and the NASA Johnson

space center robotics group. The algorithms were combined into the Operational Software Architecture for Advanced Robotics (OSCAR). OSCAR implements kinematic and dynamic control, robot performance evaluation and optimization, communications between peripheral devices, evaluation of sensor data, and interface with an operator. OSCAR was designed for control of redundant robots with multiple joints and multiple degrees of freedom. A redundant robot can reach a specific end-effector position through a large number (possibly infinite) set of robot joint displacements. Failure tolerance and recovery is one of the applications of redundancy: if one actuator fails, the controller locks the faulty joint but the robot continues to operate so long as sufficient joints are functioning correctly. The general task of the test-case software is to move a robot arm along a specified path to a given end effector position given physical constraints (e.g. obstacles, joint angles, etc.). Efficient operation of a redundant robot requires selection of the "optimal" arm configuration to place the end-effector in the specified position. This decision-making problem is solved by applying performance criteria and various optimization algorithms [16, 17, 29, 33]. A detail of a redundant robot executing a simple exploration strategy for reconfiguration of the robot arm is shown in figure 2, with θ—being a joint angle, and δ—being a trial displacement. The complexity of robot operation and the robot controller software increases with the number of joints and the number of degrees of freedom. The complexity of model checking also increases as the complexity of the software increases. Realistic fault-tolerance may require at least six joints. For a complete description of the robot functionality implemented in OSCAR refer to [17].

The size of the OSCAR implementation examined in this research was roughly 180 KLOC. It consisted of 120 C++ classes implementing more than 600 methods. The original implementation of the robot controller was a carefully designed and engineered C++ program. The original program, while conventionally object-oriented, had become, over time, difficult to test, maintain and modify. A dependency analysis of the original program revealed multiple hierarchical dependencies among classes. The difficulties in analysis were

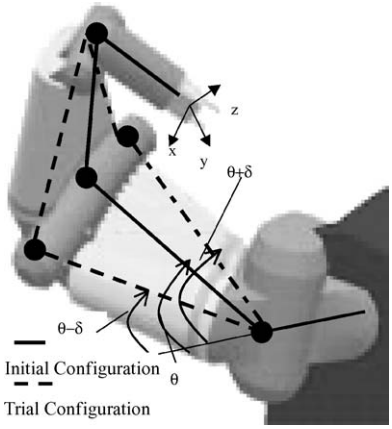


Figure 2. A part of a redundant robot, demonstrating a large number of manipulator configurations for a single end-effector position.

largely due to these complex multiple dependencies among OSCAR classes. While conceptually the existence of the derived classes simplifies the code's structure, in practice it leads to highly complex implementations. These dependencies among classes were introduced by multiple developers modifying and extending the system to meet additional requirements.

Robot control was designed to be distributed over multiple computers both for simulation and control. Distributed control complicates analysis of the system behavior. Distributed control requires careful attention to synchronization of updates for control variables. Synchronization faults may lead to inconsistent values for control parameters. This may lead to incorrect or faulty behavior of the robot. Synchronization errors are often difficult to detect by testing since the non-deterministic actions introduced by synchronization errors may be non-reproducible. The complexity of analyzing distributed algorithms, together with the stringent reliability requirements for robot functionality suggest model checking-based verification, with its complete exploration of all possible behaviors.

3.1. xUML representation of the Robot controller system

This section gives an essentially historical presentation of two redesigns and re-implementations of the control functionality of OSCAR. The goal of the project was to determine the extent to which model checking could be applied to enhance the reliability and robustness of the OSCAR system. The first step was to extract the "control functionality" of OSCAR. Control functionality is the decision procedures for managing the movements of the arm. It accepts the results of the kinematics computations as input, determines the movements to be executed and interfaces to the robot itself. From the original 180,000+ lines of C++ code, 45,000+ lines were identified as implementing control functionality. Two re-engineerings and re-implementations of the control system were required: the first was a complete redesign and re-implementation following the design principles for constructing XUML models. This redesign and re-implementation yielded a model which could be readily understood and thoroughly tested and validated. However, model checking based on this model could be completed only for systems with unacceptably small numbers of joints and degrees of freedom. A second redesign, re-implementation based upon design principles which enable application of compositional reasoning was necessary to enable successful model checking of realistic robot control algorithms. The next section reports the redesign results and the design-for-verification principles.

3.1.1. Design-for-testability. The robotic software architecture was defined as a set of communicating XUML objects. The application domain architecture was divided into *control* and *performance evaluation* subsystems (see figure 3). The input for the performance evaluation subsystem is one or more trial arm configurations from which the performance evaluation system will either select the best one or provide the control system with suggestions on what an optimal arm configuration should be. The dynamic structure of the RCS is defined by communication channels describing object interaction dynamics. Each object's behavior is defined by an XUML state machine.

The *control* subsystem includes kinematics algorithms and interfaces to the computational libraries of the OSCAR system. The control algorithm of the control subsystem starts with

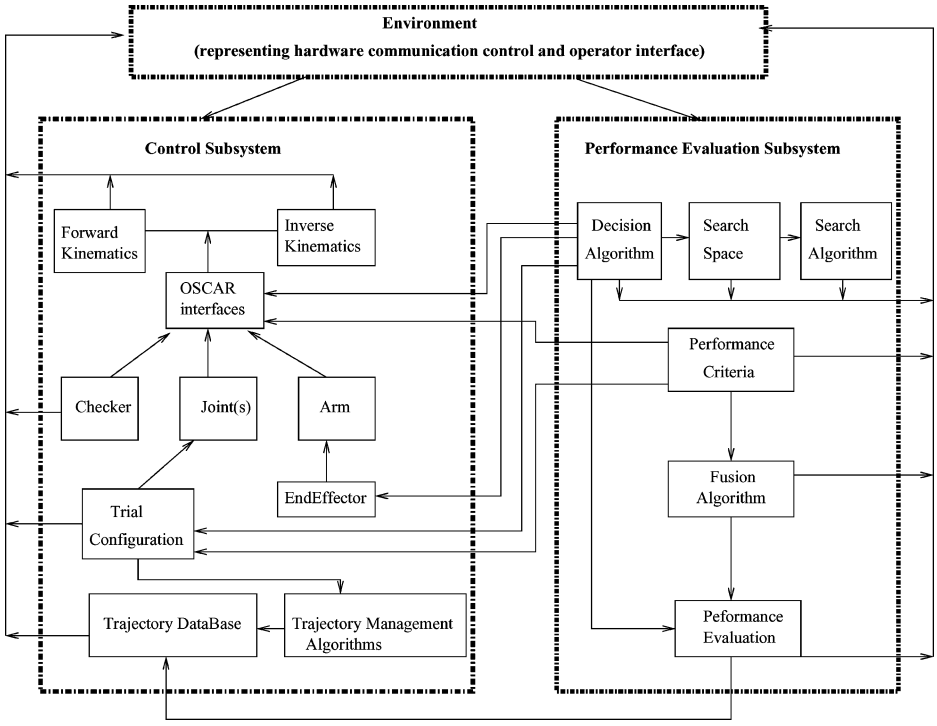


Figure 3. Architectural view of the RCS system.

defining an initial end-effector (*End-Effector* (EE)) position given the initial joint (*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 either starts a fault recovery algorithm or stops execution of the program (if fault recovery is not possible).

The *performance evaluation* subsystem implements the decision-making strategy by applying decision-making techniques identifying a solution to the multi-criteria problem. It builds a *SearchSpace*, which generates sets of *TrialConfigurations* around a base point supplied by the computational subsystem. A *DecisionAlgorithm* selects the best trial configuration given a set of *PerformanceCriteria* and a number of physical constraints that are

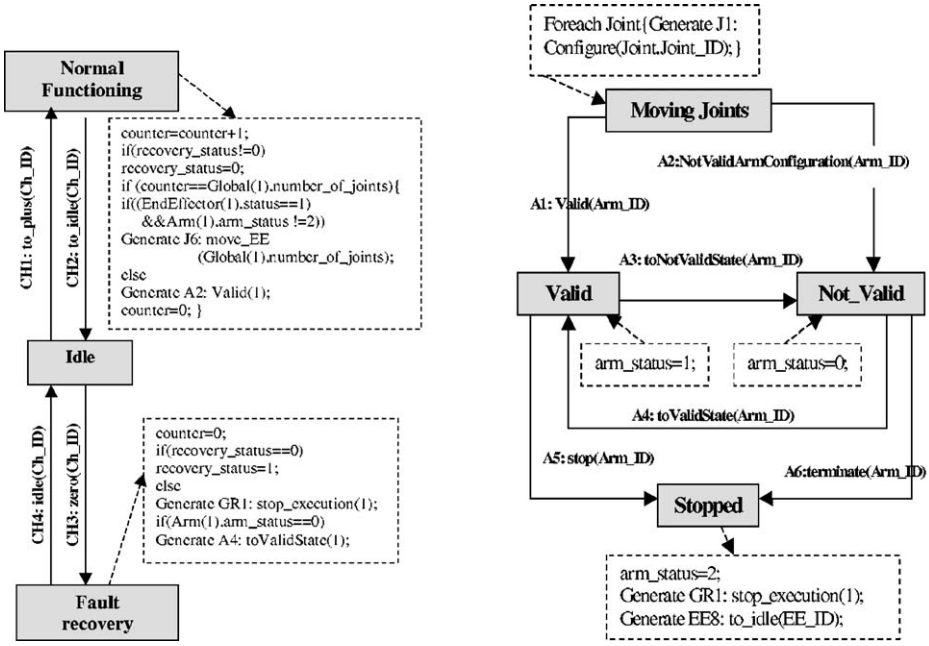


Figure 4. State transition diagram of the *Checker* (left) and *Arm* (right) objects.

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.

Figures 4 and 5 schematically represent the state machines of the *Arm*, *End-Effector* and *Checker* XUML models (some actions are omitted due to the space limitations of the paper). The overall description of the XUML RCS system can be found in [37].

3.1.2. Validation and verification. The XUML robot controller models were validated by execution of the functional scenarios of the original program. Validation of the XUML models resulted in identification of several errors and redundant computations in the original robot control algorithms.[2] These errors were corrected and the XUML code was revalidated. The re-engineered XUML code was considerably simpler in structure than the original C++ code due to the much higher level of abstraction of XUML compared to C++ and removal of the redundant computations.

An attempt was made to model check the re-engineered and validated XUML model. None of the properties (sample properties can be found in Section 4) could be verified for a system with more than 2 degrees of freedom (DOF)[3] due to state space explosion during model checking. The state space reduction techniques supported by OBJECTCHECK and COSPAN, including partial order reduction, predicate abstraction, localization reduction, and symbolic verification were applied. None of these state space techniques either taken

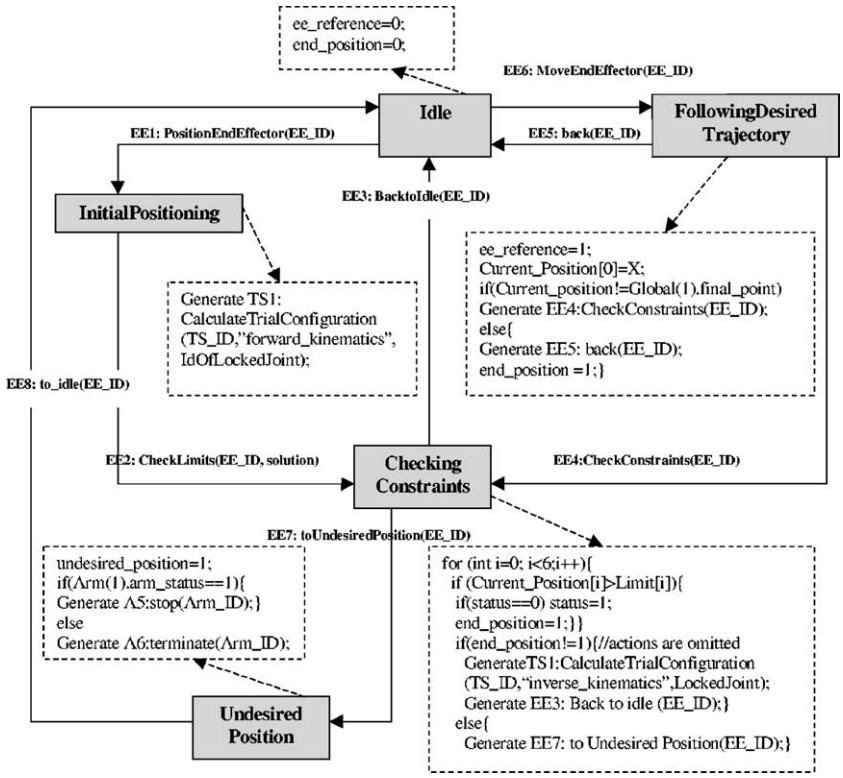


Figure 5. State transition diagram of the *EndEffector* object.

in isolation or combined with other techniques enabled completion of model checking of any property for more than two DOF.

Assume-guarantee compositional reasoning, known to be highly successful in the complexity reduction of large hardware systems, could not be applied to the XUML robot controller system because of strong coupling among the XUML objects defining the robot control system. To enable application of compositional reasoning, a software design method that enables the applicability of compositional model reasoning in model checking was developed and applied to the robot control system. The design for compositional model checking approach is presented in Section 3.1.3. Analysis of the state spaces of XUML models lead to development and application of a domain specific data abstraction for loop intensive systems. The loop abstraction is described in Section 6. Both methods are design level techniques defined for the XUML-formulated software and can be applied in combination with other state space reduction algorithms without change to the verification tools or the verification algorithms.

3.1.3. Design-for-verifiability. The design for the RCS presented in the last section conforms to accepted best practice for object-oriented design in XUML. This design, however,

contains many connections among classes including references to member variables among classes as well as event based communication. This high degree of connectivity engendered strong dependences among individual classes which coupled the state spaces of many classes and led to intractably large state spaces. It was clear that scalable model checking would require software designed with a stronger definition of modularity.

A design method based on *software spatial modularity* for generating systems with strong modularity was developed. The term "*spatial modularity*" was chosen because of a similarity to modular design principles for hardware [41]. The spatial modularity concept leads to two design principles, *functional localization* and *global data encapsulation*. We specified *design-for-verifiability* rules which implement these principles to partition a system design into a set of functionally independent "spatially disjoint" components. These rules are sketched following.

System. A system is designed as a set of interacting components.

Component. A component is a set of classes which implement some logical functionality and encapsulated by a gate class.

Gate class. The interface through which a component interacts with other components is defined by a gate class. The gate class for a component receives and routes all input events and sends all output events from the component. The behavior of a gate class is defined by a state machine that interprets events both from the component external and internal environments. Examples of the RCS components gate classes can be found in [37].

Remote data access. All attribute value updates among components are done through the event mechanism.

Global data representation. Create a separate component containing all global variables as attributes of its classes.

Inheritance. Subtype classes are required to have a semantic relationship with their supertype classes. In other words, inheritance is restricted to a purely syntactic role: code reuse and sharing, and module importation.

In a system designed following these rules, communication and interaction among components is reduced to communication among the gate classes of the components. As a result, components can be isolated from each other by disconnecting the communication events controlled by the gates. Local properties can be then defined for each component by referring to its variables. Each component's environment can be defined by making assumptions about that component's inputs. The assumptions can be derived either by simulating an event sequence at the components external interface (gate class) or by making assumptions about values of the component's gate external variables (so called "observable" variables).

The RCS system was re-engineered again, this time to conform to these design rules. Application of the *design-for-verification* rules yielded an RCS system design as a collection of spatial components. Each component was designed to implement a set of functionally independent robotic operations that may interact with other components. Figure 6 gives an

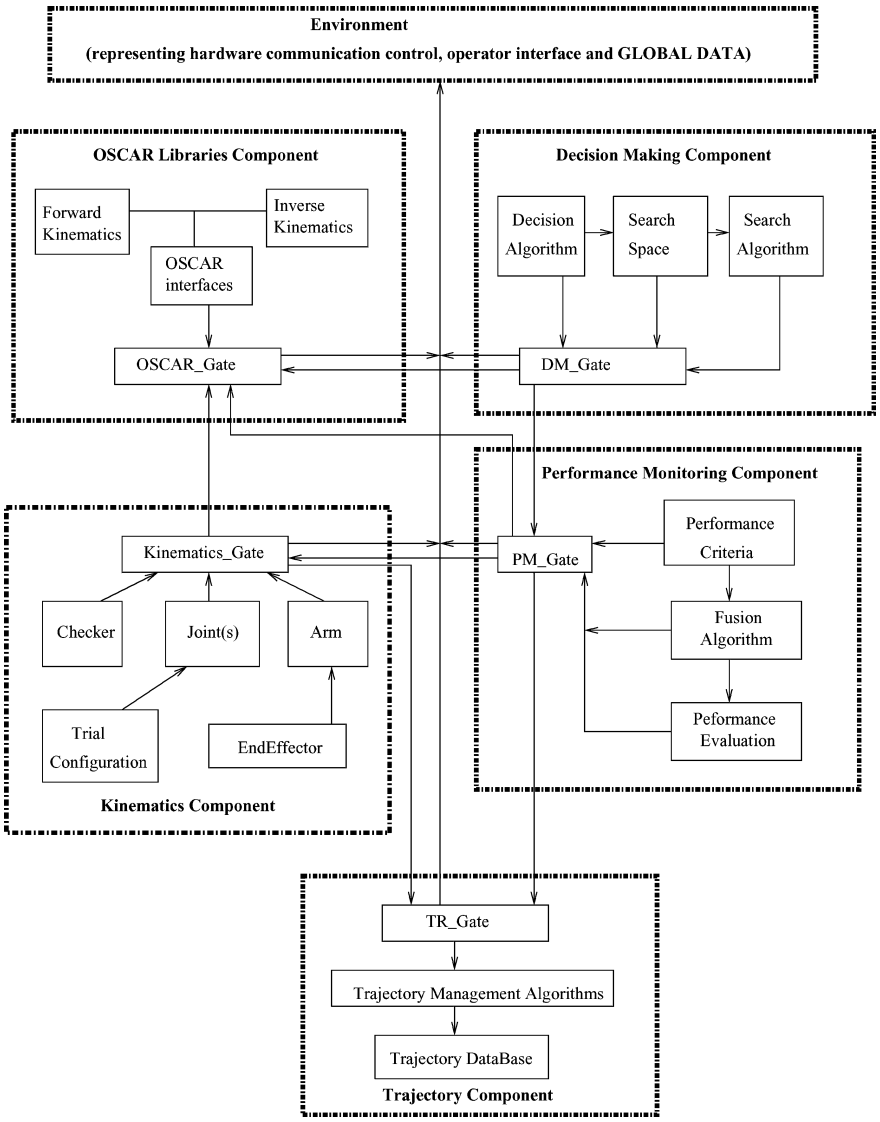


Figure 6. Architectural view of the spatially modular RCS system. Note, communication among components (including environment) is enabled via gates.

architectural layout of the RCS system redesigned following the design rules given preceding. An instance of a robot control algorithms is obtained by composing *OSCARInterfaces*, *Kinematics* and *Trajectory* components. The *OSCARInterfaces* component specifies interfaces to the OSCAR computational libraries. The *Trajectory* component defines interfaces to the robot operator or the database of predefined robot control trajectories. The components can access data stored in the *GlobalData* component to implement different robot control

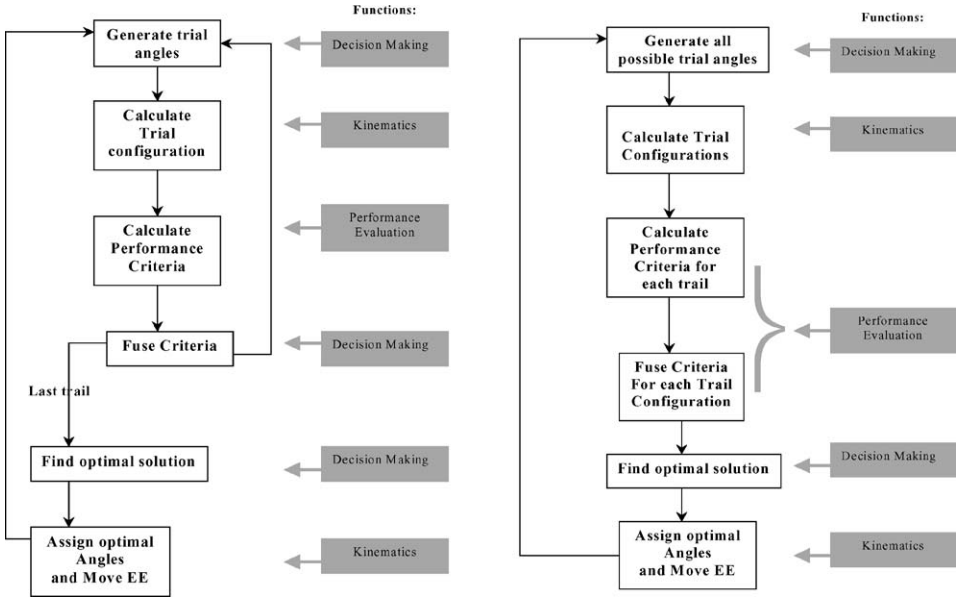


Figure 7. Implementation of the *localization* rule during re-design of the RCS system.

functions. Controllers that are able to perform *PerformanceMonitoring* can be qualified as robotic intelligent controllers though optimization problems are not considered in this design. The *Decision Making* component provides algorithms for optimizing robot control.

Figure 7 illustrates the implementation of the *functional localization* principle for the re-design of the XUML RCS components. The original design (left side of figure 7) followed a conventional design approach. *Decision-Making* and *PerformanceMonitoring* operations cannot be used independently. The right side of figure 7 illustrates the localization concept. Decision-making actions for fusion of the performance criteria were merged with the performance evaluation operations. As a result, independent components, *DecisionMaking* and *PerformanceMonitoring* were created. The components can be analyzed in isolation using the abstract representation of the neglected component for each robot control cycle.

This RCS redesign enabled the application of assume-guarantee compositional verification. This redesign did require some modifications in the algorithms for robot control. This redesigned RCS was then completely revalidated by execution of the design model. The next section discusses how the spatially modular RCS components were verified.

4. RCS properties

A set of temporal logic properties for verification of the robot controller were derived from the requirements for the robot functionality. The requirements were collected from the robot controller documentation [16, 17] and from discussions with the developers of OSCAR. A summary of the requirements is presented in Table 1.

Table 1. Requirements for the robot control functionality.

N	Requirements
1	Conduct robot control within allowed workspace
2	Perform fault recovery (lock faulty joint and reconfigure other joints) if a joint does not satisfy the robot control restrictions
3	Implement obstacle avoidance
4(a)	Assure the robot functioning termination
4(b)	Assure the robot functioning termination in a presence of unsafe joint and end-effector configurations
5	Assure dynamic robot reconfiguration
6	Support multi-criteria optimization of the robot control
7	Implement fault detection

The specifications of the properties are defined in terms of the variables of state machines of the classes of the RCS system. The properties were encoded in an XUML level query language provided by OBJECTCHECK [45]. An example of the formulation of a safety property is given to demonstrate how a property is specified. We refer in this description to the states appearing in the state transition diagrams of the XUML RCS models in figures 4 and 5.

Robot functionality requires that control functions are performed only for safe robot arm configurations. The safety of the robot arm configuration is defined by the algorithm that checks the validity of the robot arm constraints. The safety property specifies coordination between the two algorithms: the end_effector can be moved to a new position (*FollowingDesiredTrajectory* state of the *End_Effector* XUML model) only from the confirmed valid configuration of the robot arm (*Valid* state of the *Arm* XUML model).

The property encoded in the XUML level query language of OBJECTCHECK is defined as follows:

$$\begin{aligned} &\textit{declare } r \ll \textit{End\_Effector} \gg \textit{ee\_reference} \\ &\quad \textit{declare } u \ll \textit{Arm} \gg \textit{arm\_status} \\ &\quad \textit{Always } (r = 1 \rightarrow u = 1) \end{aligned}$$

The first statement of the property defines a propositional predicate, r, that declares the variable *ee_reference* of the *End_Effector* class. The second statement defines a propositional predicate, u, that declares the variable *arm_status* of the *Arm* XUML class. The third statement declares a temporal predicate over the system: at any moment during the system execution, $r = 1$ and $u = 1$ are true at the same time. For example, the state *FollowingDesiredTrajectory* and the variables *ee_reference* appear in figure 5 and the variable *arm_status* appear in figure 4.

A selected set of the properties formulated and verified in this project is given in Table 2. Properties are given as informal English specifications. Specifications in terms of state predicates of the RCS XUML system can be found in [37]. Each property is associated with a functional requirement (Req.) defined in Table 1. A taxonomy of the properties is

Table 2. Verification properties.

N	Property robotic description	Req.	Type	Control
1	Eventually the robot control terminates	4(a)	Liveness	Yes
2	The program terminates when it reaches the state where there is no solution for the fault recovery	4(b)	Safety	No
3	If the end-effoctor reaches an undesired position than the program terminates prior to teh end-effector new move	1, 4(b)	Safety	Yes
4	Whenever the end-effector is in the "Following_Desired_Trajectory" state then the arm is in the "Valid" state	1	Safety	No
5	Fault recovery is executed when any of the joint angles does not satisfy the allowed limits	2	Safety	No
6	Fault recovery is always executed for joints that reside in their most recent base position	2	Safety	Yes
7	When fault recovery is called, the end-effector can not move to a new position until fault is resolved	7	Safety	Yes
8	If an obstacle is reached by the end-effector than the obstacle avoidance procedure is performed	3	Liveness	No
9	The robot never operates outside of allowed workspace	1	Safety	No
10	The program always performs computations for an actual robot: the general description of a robot is always reduced to that of an actual robot	5	Safety	No
11	No command to move the end-effector is scheduled before its initial position is computed	1	Safety	Yes
12	Chained fault recovery is not permitted (if the fault recovery did not complete for an instance of the robot configuration, the fault recovery for a different robot configuration instance is not allowed)	2	Liveness	No
13	End-effector is never located at some undesired locations	3	Safety	Yes
14	Only validated solutions of TrailConfigurations are used for the optimization of the robot control	6	Safety	No

presented by classifying the properties types (Type). We identified a subset of properties that refer only to the control flow of the RCS execution and are independent of the values of variables which do not determine control flow (i.e. properties that can be specified only by referring to the states of the XUML state machines). These properties are called *control properties*. For example, the safety property presented ealier is a control property since both *ee_reference* and *arm_status* variables refer to the *FollowingDesiredTrajectory* and *Valid* states of the *End-Effector* and *Arm* state machines.

5. Verification results

Presentation of verification results is separated into two subsections: the effects of designing for verifiability and a systematic comparison of the effectiveness of the state space reduction algorithms described in Section 2.5 in application to the RCS system.

Table 3. Verification statistics for the robot controller system.

DOF (Task)	P1 (Failed) (states/min:sec/MB)	P3 (Failed) (states/min:sec/MB)	P10 (Verified) (states/min:sec/MB)
2(O)	9.6e+21/550:2/1,741	1.1e+23/619:4/1,836	1.1e+19/201:3/173
2(V)	2.2e+12/350:4/735	2.3e+11/344:4/713	4.8e+6/32:3/34
3(O)	M/T exhaustion	M/T exhaustion	M/T exhaustion
3(V)	3.1e+18/415:4/1,246	2.6e+17/410:3/1,198	5.1e+9/68:50/68
4(O)	M/T exhaustion	M/T exhaustion	M/T exhaustion
4(V)	6.2e+23/592:4/1,802	6.7e+24/662:3/2,190	7.5e+14/155:1/127
5(O)	M/T exhaustion	M/T exhaustion	M/T exhaustion
5(V)	M/T exhaustion	M/T exhaustion	1.8e+24/262:4/232

5.1. Comparison of design models

This section compares model checking of the model (M_V) designed for verifiability (Section 3.1.3) with the model (M_O), a conventional object-oriented design (Section 3.1.1). Verification of the model M_V used assume-guarantee compositional reasoning.

Verification experiments were conducted on several machines. The fastest machine was an HP9000 (440 MHz) with 6144MB RAM and HP-UX11 operating system. Verification revealed that properties 5–8, 9, 10, 12–13 hold, and that properties 1–4, 11 fail.

Verification of the RCS system was conducted by assume-guarantee model checking. Verification revealed that properties 5–8, 9, 10, 12–13 hold, and that properties 1–4, 11 fail. Table 3 presents the complexity results for verification of Properties 1, 3, 10 (*P1, P3, P10*). Each entry in the table has the form $x/y/z$ where x is the number of the states reached, y is the run-time in CPU seconds, and z is the memory usage in Mbytes. The results are given for RCS models of different complexity defined by the number of DOF (number of joints) of a robot arm. The results for model checking the model M_O, are qualified with an O while the results for M_V models are qualified with a V. The differences in the results are due to the application of assume-guarantee reasoning in model checking of the V rows. The results for verification for the *Kinematics* component are given for a total number of seven $+ i$ XUML state machines, where i is the number of state machines corresponding to the number of instances of the *Joint* object. The compositional checks utilized local properties of the verifiable component. All external variables of the local properties were closed by making assumptions about the inputs of the component.

The O rows of Table 3 show that model checking all but two DOF models failed for the conventional object-oriented model, due to the memory/time (M/T) exhaustion. The V rows of Table 3 show that application of assume-guarantee reasoning to the M_V model enabled completion of model checking for all models and verification of properties that fail was completed for models up to five DOF. For five and higher DOF models, the compositional checks of properties that failed on smaller models ran out of memory. Design for verifiability made a rather dramatic difference in the complexity of systems which could be model checked.

Table 4. Verification results of property *P2* (property fails) for 2, 4 and 7 DOF models.

Reduction	2-DOF (states/min:sec/MB)	4-DOF (states/min:sec/MB)	7-DOF (states/min:sec/MB)
AG	M/T exhaustion	M/T exhaustion	M/T exhaustion
AG+POR	1.5e+18/545:34/1,718	M/T exhaustion	M/T exhaustion
AG+BDDs	M/T exhaustion	M/T exhaustion	M/T exhaustion
AG+bounds	2.3e+12/338:15/736	7.1e+17/482:1/1,424	M/T exhaustion
AG+PA	Exhaustion during abstraction	Exhaustion during abstraction	Exhaustion during abstraction
AG+bounds+POR	1.7e+10/227:3/524.9	3.4e+15/469:5/1,392	3.7e+21/964:4/2,178
AG+bounds+BDDs	M/T exhaustion	M/T exhaustion	M/T exhaustion
AG+bounds+PA	1.4e+9/174:6/244.2	5.3e+12/344:2/798.8	Exhaustion during abstraction
AG+bounds+ POR+BDDs	M/T exhaustion	M/T exhaustion	M/T exhaustion
AG+bounds+ POR+PA	1.3E+6/119:15/79.5	1.1e+9/201:40/514	Exhaustion during abstraction
AG+bounds+POR+ BDDs+PA	Exhaustion during abstraction	Exhaustion during abstraction	Exhaustion during abstraction

5.2. Evaluation of state space reduction methods

Tables 4 and 5 give an overview of a subset of state graphs we have generated using different reduction techniques. Results are given for experiments with the RCS variables bounded within the robotic specific ranges. The examples of the robotic domain types that were discretized and represented by integer types are *angles_interval*, bounded to the [−360 360] range, *coordinates_interval*, bounded to the [−1000 1000] range, and *counter_interval*, bounded to the [0 17] range. The latter type was used to control instances of the robot arm joints. We, thus, have chosen for the declaration of the *counter_interval* the largest DOF number (largest possible number of joints of the robot arm) used in the test-bed system. The results are given for verification of properties 2 and 13 (P2, P13). Both, a safety property P2 and a liveness property P13 hold during verification.

In the tables *AG* stands for the *assume-guarantee* reasoning, *POR* stands for the *partial order reduction*, *BDDs* stands for binary decision diagrams, *bounds* stands for the robotics-specific variable bounding, and *PA* stands for the *predicate abstraction* techniques. The results are given for the assume-guarantee reasoning in conjunction with the localization reduction that is automatically invoked in COSPAN for any verification effort. However, assume/guarantee reasoning served as the base technique for combination with other state space reduction methods.

The following conclusions can be drawn from Tables 4 and 5:

1. When *assume/guarantee reasoning* was applied according to design-for-verification rules, it made verification of realistic RCS (with more than two DOF) feasible.

Table 5. Verification results of property *P15* (property holds) for 2, 4 and 7 DOF models.

Reduction	2-DOF (states/min:sec/MB)	4-DOF (states/min:sec/MB)	7-DOF (states/min:sec/MB)
AG	M/T exhaustion	M/T exhaustion	M/T exhaustion
AG+PO	9.4e+6/54:15/44.8	1.5e+14/165:3/148.7	3.1e+22/262:5/232.1
AG+BDDs	M/T exhaustion	M/T exhaustion	M/T exhaustion
AG+bounds	4.02e+6/38:15/36.2	7.3e+11/142:1/134.4	2.1e+19/182:4/203.7
AG+PA	Exhaustion during abstraction	Exhaustion during abstraction	Exhaustion during abstraction
AG+bounds+POR	7.5e+5/16:10/9.2	2.4e+9/62:23/65.9	5.4e+14/147:02/139.2
AG+bounds+BDDs	M/T exhaustion	M/T exhaustion	M/T exhaustion
AG+bounds+PA	4.3e+5/7:28/6.3	1.7e+9/68:50/73.3	Exhaustion during abstraction
AG+bounds+ POR+BDDs	M/T exhaustion	M/T exhaustion	M/T exhaustion
AG+bounds+ POR+PA	1.6e+5/6:48/5.2	4.8e+8/46:50/58.7	Exhaustion during abstraction
AG+bounds+ POR+BDDs+PA	Exhaustion during abstraction	Exhaustion during abstraction	Exhaustion during abstraction

2. *Domain-specific bounding* of variables was required to complete verification of even the smallest system. Domain specific bounding results in a tremendous reduction of the state space of software and, therefore, should be always used during verification.
3. Application of *partial order reduction* resulted in a significant reduction of the system state graph. When the *static POR* is available (as it was throughout this research), it should be always used during formal model generation process since static POR can always be integrated with other reduction techniques.
4. *Symbolic verification* did not succeed for any property checked on models of different complexity. The failure of the symbolic verification may have been caused by the complex ordering of the RCS computations, which prevented efficient construction of BDDs. An interesting topic for future research is to determine whether software systems may be engineered to enable symbolic verification to be effective.
5. Application of the *predicate abstraction* algorithm given in [30] was not successful for large systems. Memory exhaustion occurred on a computer with 4GB memory during the *computation* of the abstraction predicates for the concrete programs with more than five DOF. We did not try other predicate abstraction algorithms. Clearly, however, that the high computational cost of computing predicates may render complete predicate abstraction intractable for large systems. An important future research topic is to minimize a set of predicates that is computed during abstraction.

5.3. Summary of verification results

The result of combining assume-guarantee reasoning with the OBJECTCHECK-supported state space reduction techniques made reasonably complex RCS system tractable for model

checking. But model checking still was intractable for redundant and truly fault tolerant RCS systems.

6. Loop abstraction

An abstraction (loop abstraction) which enabled completion of model checking of some additional properties for RCS's with full redundancy was derived from the observations that:

- There are important safety and liveness properties (*control properties*) which are dependent only on the static control flow graph of the system. These properties are independent of the number of traversals of the loops of the control flow graph and of the values for variables not used in determining control. Therefore the control properties of the concrete program can be model checked by model checking of an abstract program with the same static control flow graph.
- The execution behaviors of control software systems, including the RCS, are typically dominated by cycles in the static control flow graph which implement feedback loops. The structure of the control flow graph is usually determined by a small set of control flow variables. The paths in the control flow graph of a program with loops are usually determined by conditional statements (guards) which depend on a subset of the control flow variables (loop variables). Model checking of such systems generates a traversal of the loops in the control flow graph for each possible value of each loop variable. Each traversal of the loop with different values of the loop variables is distinct in the state graph of the program. Additionally, each traversal of a loop will typically involve many variables ("don't care" variables) which do not participate in determination of the paths through the control flow graph. But each execution of a loop with different values for the "don't care" variables is also distinct in the state graph generated by the model checker.

The loop abstraction is an instance of an abstraction based on data independent behavior [22, 34, 44]. The loop abstraction technique generates an abstract program with the same static task graph as the concrete program from which it is derived. It differs in specifying a minimum (or nearly minimum) number of traversals of the loops of the static task graph and in freeing he values of the "don't care" variables. These abstract programs typically have orders of magnitude smaller state spaces than the concrete programs from which they are derived. The loop abstraction algorithm and a proof of its correctness are detailed in [38]. Figure 8 is a schematic of the implementation of the abstraction algorithm. The *loop_abstraction* program takes as an input results of the program behavioral analysis conducted using the discrete event simulator. The event simulator is a part of the XUML specification and validation environment. During the simulation the program is executed by traversing possible *event sequences* which can arise from the execution of interacting XUML state machines. The set of actions that are repeatedly initiated by some event are manually annotated with a *Loop Label* in the XUML specification environment. The algorithm also uses the following features provided by COSPAN to support the loop abstraction procedure:

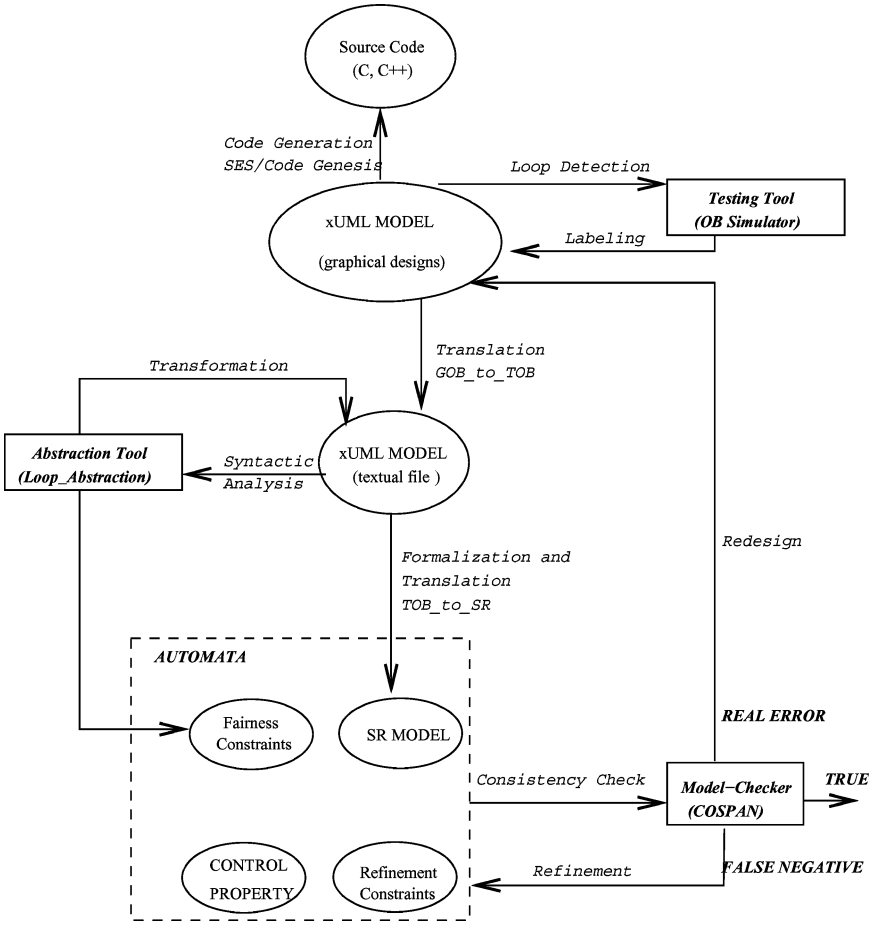


Figure 8. Implementation framework of the loop abstraction algorithm.

– the *assume/guarantee mechanism* of COSPAN is used to add fairness constraints (as required by the abstraction algorithm) and refinement assumptions to the model-checking process.
– the *localization reduction* algorithm, automatically invoked by COSPAN during model-checking, is used to eliminate from consideration variables that do not effect the verification property.

The properties of the loop abstraction algorithm [38] are:

– It is computationally simple and requires storage linear in the size of the program since it is a source to source transformation based on static analysis of the program. This property allows the loop abstraction technique to outperform the predicate abstraction techniques where computational cost impedes their application to large systems.

– It is based on syntactic manipulation of expressions, and produces a reduced program and therefore, it can be applied without change to the verification tool or the verification algorithm.
– It produces a syntactic representation of the abstract program and thus other model checking state space reduction techniques, such as symbolic model checking and partial order reduction, can be applied to the abstract program.
– It typically introduces only a small set of new behaviors so that the effort required for refinement is modest.

The application of the *loop abstraction* technique to syntactic analysis of the XUML models identified the existence of multiple loops in the robot control algorithms. The loops were abstracted by syntactic program transformation following the loop abstraction algorithm. We verified the control properties (both safety and liveness) given in Table 2. We considered several RCS variants of different complexities, defined by the number of joints i of a robot arm. We used two models to check the properties. The first model is the complete (concrete) structure of the robot arm. The second model is the abstract version of the concrete model to which the loop abstraction method has been applied.

Table 6 compares run-time and memory usage for Properties 1, 3 (P1 and P3). Experiments were conducted using application of assume-guarantee rules to verification of the RCS *Kinematics* component. The verification results demonstrate significant reduction in both time and space for the abstract model, as compared to the concrete model. The reduction becomes more pronounced for larger values of DOF. Verification for the robot configurations possessing more than four joints could not be completed for the concrete model due to the memory/time exhaustion (denoted as *M/T exhaustion* in Table 6), but COSPAN succeeded for the abstracted model. Verification of control properties succeeded for the most complex configuration of the RCS system (up to seventeen DOF systems). Notably, these systems were model checked without application of any state space reduction algorithms except the combined use of assume-guarantee reasoning, localization reduction and loop abstraction. Model checking performance might be even better if other state space reduction had been applied. That was not needed, however, during verification of the RCS state space explosion was not encountered.

It would be expected that a selective and limited scope abstraction such the loop abstraction would introduce fewer unrealistic behaviors into the abstract program than more

Table 6. Comparison of verification of the concrete and abstract robotic systems.

DOF	P1:Concrete (st./min:sec/MB)	P1:Abstract (st./min:sec/MB)	P3:Concrete (st./min:sec/MB)	P3:Abstract (st./min:sec/MB)
2	2.2e+12/350:4/735	26K/0:28/4.03	2.3e+11/344:4/713	17K/0:17/3.38
3	3.1e+18/415:4/1,246	63K/3:10/4.9	2.6e+17/410:3/1,19	63K/3:10/4.9
4	6.2e+23/592:4/1,802	145K/11:28/8.4	6.7e+24/662:3/2,190	116K/7:03/7.1
5	M/T exhaustion	688K/28:10/23.9	M/T exhaustion	554K/13:40/19.1
6	M/T exhaustion	1.1M/42:17/96.5	M/T exhaustion	715K/33:17/36.2

comprehensive abstractions. This proved to be the case for the robot control system. Only a few refinements were needed. This occurred where false negatives in model checking were identified in model checking the abstract program. Then refinements were manually implemented.

In summary, the loop abstraction appears to hold particular promise for model checking of control software systems, nearly all of which implement feedback loops.

7. Summary of the RCS verification

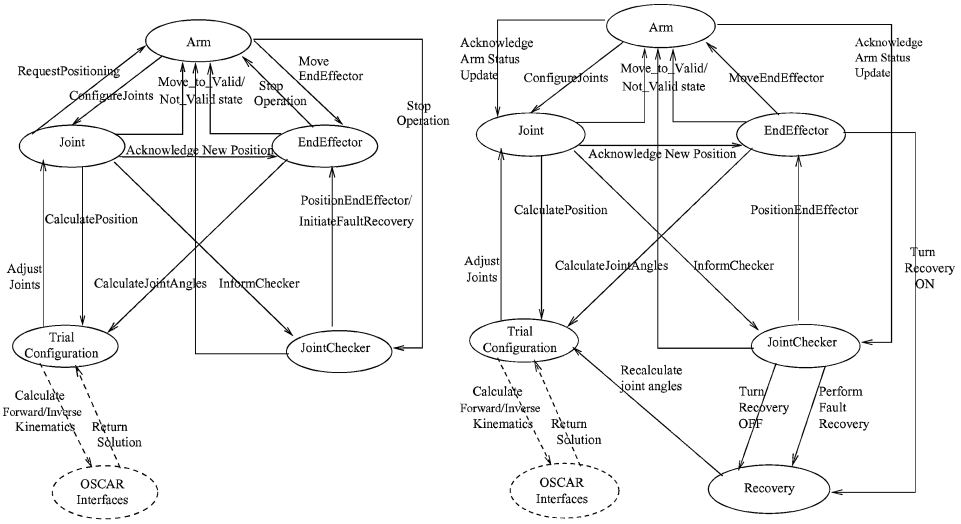
Attempts were made to apply the integrated state space reduction process to verification of 37 properties. Model checking was eventually successfully completed for 22 of these properties. Verification of control properties succeeded for the most complex robot configurations. Verification of data-dependent properties of large DOF systems (more than five DOF) could not be verified but was completed for systems with less than five DOF. In almost every case multiple attempts were necessary to model check a property. The case study was successful in that model checking of this complex system revealed six serious logical errors [37] which had not been detected by conventional testing.

7.1. Identification of errors

Failure of Properties 1, 2 and 3 upon verification revealed errors in the robot control algorithms. The failure of Property 1 indicated that the system does not always terminate its execution as expected. Property 3 designed to check the correct system termination confirmed that the system would not always terminate properly. It was found that an error in the fault resolution algorithm caused the problem. Recall that failure of one of the robot joints to satisfy the specified limits activates the fault recovery procedure. If during the fault recovery process some of the newly recalculated joint angles do not satisfy the constraints in their turn, another fault recovery procedure is called. Analysis of the counterexample provided by COSPAN for Property 3 indicated that, for several faulty joints a mutual attempt was made, to recompute the joint angles of other joints without first resolving the fault situation.

Another failure occurred during verification of Property 2. This error reflected a coordination problem between the *Arm* and *JointChecker* processes. The original design assumed sequential execution. It was expected that at each step, the *arm_status* variable of the *Arm* process would be updated before the *JointChecker* process issued a command to move the *EndEffector* to its next position. Concurrent execution of the processes, however, led to a situation where the *JointChecker* process could issue the command based on an out-of-date value of the *arm_status* variable.

The errors found by model checking were not discovered either during the conventional testing performed by the original code developers or during the validation by simulation of the formalized XUML design. In order to correct these errors a redesign of both the original system and the XUML model was required. Figure 9 provides both the original and the modified class collaboration diagrams of the *Kinematics* component. The latter demonstrates the design changes we made in order to correct the found errors. We introduced a new class called *Recovery*, whose functionality provides a correct resolution of the above-described



a. Original *Kinematics* Component b. Modified *Kinematics* Component

Figure 9. Collaboration diagrams of the original and modified *Kinematics Components.*

fault recovery situation. Additionally, we added exchange messages between the processes *Arm* and *JointChecker* that corrected the coordination problem.

8. Lessons learned

The lessons learned are implicit in the process through which we went to model check the robot controller system. Some of what we learned, we clearly should have known before we started. But there is an old saying "Too soon old, too late smart." While we don't assume that others will miss the obvious as we sometimes did, it seems worthwhile to summarize some of what we now know.

8.1. Representation issues

8.1.1. Design level representation.

Executable design level representations such as XUML *offer advantages over conventional procedural language representations for development of model checkable software systems.*

– Property formulation is facilitated by control being expressed in an explicit design model.
– Use of executable design level specifications which can be tested at design level but compiled to conventional procedural programs bypass some of the sources of difficulties and errors in directly model checking programs in procedural code. All operations (i.e., finite state model extraction, property specification, abstraction, decomposition, etc.) can

be made on the same representation or result from an automated transformation of the representation.
– Much less abstraction is necessary since data definitions are more abstract than for procedural programs.
– Source to source transformations to create property specific abstractions for state space reduction are straightforward.

8.1.2. Design for verifiability.

Design of the software system should take cognizance of the requirements of model checking.

Design of software has a strong impact on its "ability" properties: testability, maintainability and evolvability. Design for "model checkability" was an essential step successfully completing verification of RCS systems of non-trivial complexity.

8.2. Property formulation

Effective property formulation requires a systematic derivation process.

Property formulation is one of the most difficult tasks in model checking of complex software, perhaps even more difficult than test generation. The steps in property formulation followed in this case study were:

– The requirements specifications for the system were reviewed and made precise through a collaboration among the software developers and the model checkers.
– Properties were identified from the requirements specification for the software system.
– Properties were formulated in the name space of the design model.

8.3. Interaction of system structure and state space reduction algorithms

The effectiveness of state space reduction algorithms is largely determined by the structure and execution behavior of the software system.

A very strong correlation between the structure and execution behaviors of the software system and the effectiveness of state space reduction algorithms was found. The structure and execution behaviors of the RCS which we observed to impact state space reduction algorithms are:

1. Strict functional partitioning and strict name space locality.
2. Multiple paths among states.
3. Little concurrent/parallel execution in a correctly synchronized system.
4. The initial state of the robot is usually a single assignment of values to state variables.
5. The execution behavior of the RCS is dominated by traversal of feedback loops.

- Item (1) suggests assume/guarantee reasoning should be effective for properties which are defined locally on a subset of components. This was confirmed.

- Item (2) suggests partial order reduction in the format of static partial order reduction should be effective and this was confirmed.
- Item (4) suggests symbolic verification may not offer significant reduction in state space. This was confirmed.
- Item (5) motivated the loop abstraction.

8.4. Structure, execution behavior and property specific abstractions

Structure specific and execution behavior specific abstractions may be required for model checking of complex software systems.

8.5. Knowledge integration

Model checking of software systems requires integration of knowledge of the discipline area of the software, the program itself and model checking.

9. Conclusions and related work

This article has presented a case study of verification of a non-trivial software system. This case study motivated development of new software verification techniques and employed them in conjunction with a wide range of previously-developed, time-proven methods to successfully model check twenty two significant properties of a robot control system.

The approach used in this work uses an executable design level representation of a software system as a basis. The executable design is validated by testing and then verified by model checking. This approach is different from the majority of software model checking projects that implement model checking based on application of advanced static analysis algorithms supported by theorem proving to actual software code, such as Java or C. The BLAST [14], CMC [28], FEAVER [15], MAGIC [7], and SLAM [3] projects are examples of C verifiers. BANDERA [10] and JAVA PATHFINDER [6] project are examples of JAVA verifiers. They have many case studies on model checking of software systems in the context of these projects. The CMC toolkit has been applied to model check three different implementations of the AODV routing protocols [28]. The FEAVER toolkit has been applied to model check the call processing code of an industrial telephone switch [15]. The SLAM toolkit has been applied to model check a large number of Windows device drivers [3]. The JAVA PATHFINDER has been applied to model check the implementation of an AI-based spacecraft controller [6].

The actual code based approach validates the actual code by testing and typically manually or automatically generates an abstraction of the program which is then model checked. Generation and resolution of abstractions can be a major problem. The approach based on an executable design level representation bases both validation and verification on the same representation of the software. When validation and verification are complete then the executable design may be compiled to obtain the actual code in C, C++ or Java [4, 18, 35].

Ameliorating the state space explosion is the principal requirement for effective model checking of software systems. Many efficient state space reduction techniques have been developed and successfully applied in the verification of hardware. The key state space reduction techniques are *compositional* reasoning [1, 2, 9, 21, 25–27, 32, 42] and *data abstraction* [3, 8, 11, 19, 21, 23].

This article develops two independent techniques that address the state space explosion problem. The first technique is a *methodological* approach to software design that enables the applicability of assume-guarantee compositional reasoning. The second method is an original data abstraction *algorithm*. Both methods are design-level techniques defined for software systems formulated in XUML. These design-level techniques can be applied without changes in the verification tools or the verification algorithms. This enables low cost integration with existing model checking tools. Additionally, since they are the syntactic program transformations, they can be used in integration with other state space reduction techniques.

We developed a set of design principles and rules that structure an entire software system as a set of components. The components are designed to minimize interactions with other components. Then each component can be verified separately in the context of a definable execution environment. Existing assume-guarantee compositional rules then become applicable to software components.

We also present an original data abstraction *algorithm*. The abstraction algorithm minimizes the contribution of loop executions to the program state space. The loop abstraction generates an abstract program possessing the same static task graph as the concrete program from which it derives but having a minimum (or nearly minimum) number of loop traversals. These abstract programs have state spaces smaller by orders of magnitude than the concrete programs from which they derive. We prove [38] the correctness of the loop abstraction algorithm. The correctness result implies that a control specification holds for the "concrete" program if it holds for the "abstract" program. Some loss of the abstraction's data computation precision is traded for the ability to conduct verification of control properties in practice. The abstraction algorithm is computationally simple and requires storage only linear in the size of the program since it is a source to source transformation based on static analysis of the program. We implemented the loop abstraction technique in the front-end of the model checking tool, COSPAN. The implementation uses the XUML2SR translator.

Another notable aspect of this work is that our re-engineering study of the robot controller system generalized the framework for the resulting integrated software design and verification methodology. Throughout the report we presented a two-phase approach for development of OO software systems which combines validation of OO models formulated in XUML with formal verification through model checking. The first phase resulted in the development of rigorously specified robot controller XUML models that were thoroughly validated by conventional testing. In the second phase, model verification with the aid of the techniques developed in this work was used to check the consistency of the robotic specifications. The results of both phases are encouraging. Not only were we able to verify a large set of functional and performance properties of the robot control but we also uncovered some design flaws. We verified both safety and liveness describing the safety-critical properties of robot control. These included the fault recovery, fault tolerance, generalized obstacle avoidance and proper robot control termination properties. While model checking

confirmed that most properties hold, some did not pass verification. Examination of the failed properties revealed the most common cause of robot control errors. It was an incorrect assumption that programs would execute sequentially. Instead, distributed interaction caused the failure of the safety and liveness checks. Since distributed control is widespread in robot control, the identification of these errors is of major importance. These errors would have been extremely difficult to identify by conventional testing due to the non-reproducible nature of the non-deterministic actions of a distributed system.

Notes

1. This is an existential abstraction technique that produces *over-approximation* of the original program. That is that some unrealistic behaviors can be introduced by the abstraction process. The existential abstraction approach provides a popular class of weakly preserving abstraction for universally quantified path properties (for example, LTL properties). Weak preservation in this case follows trivially: if more behaviors (i.e. more execution paths) are added and a property is true for all paths then it is true for any subset of those paths, including the behavior of the concrete system.
2. Validation of the re-engineered model and the results of the validation are covered in detail in [37].
3. Realistic fault tolerance requires at least six degrees of freedom.

References

1. M. Abadi and L. Lamport, "Conjoining specifications," *ACM Transitions on Programming Languages and Systems (TOPLAS)*, 1995.
2. R. Alur and T.A. Henzinger, "Reactive modules," in *LICS*, 1996, pp. 207–218.
3. T. Ball and S. Rajamani, "The SLAM toolkit," in *Computer-Aided Verification (CAV)*, 2001, pp. 260–264.
4. BidgePoint, *Project Technologies*, http://www.projtech.com.
5. G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin-Cummings: Redwood City, CA, 1994.
6. G. Brat, K. Havelund, S. Park, and W. Visser, "Java pathfinder—A second generation of a java model checker," in *Workshop on Advances in Verification*, 2000.
7. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," in *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, 2003, pp. 385–395.
8. E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 1999.
9. E.M. Clarke, D.E. Long, and K.L. McMillan, *Compositional Model Checking*, 1989.
10. J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, R.C.S. Pasareanu, and H. Zheng, "Bandera: Extracting finite-state models from java source code," in *ICSE*, 2000.
11. D. Dams, O. Grumberg, and R. Gerth, "Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL*, and CTL," in *PROCOMET 94: Programming Concepts, Methods, and Calculi*, 1994, pp. 561–581.
12. S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in O. Grumberg (Ed.), *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, Vol. 1254, 1997, pp. 72–83.
13. R. Hardin, Z. Har'EL, and R.P. Kurshan, "COSPAN," in *Computer-Aided Verification (CAV)*, 1996, pp. 423–427.
14. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Symposium on Principles of Programming Languages*, 2002, pp. 58–70.
15. G.J. Holzmann and M.H. Smith, "Software model checking: Extracting verification models from source code," *Software Testing, Verification, and Reliability*, 2001.
16. R. Hooper and D. Tesar, "Multicriteria inverse kinematics for serial robots," The University of Texas at Austin, Report to U.S. Dept. of Energy, Grant No. DE-FG02 86NE37966 and NASA Grant No. NAG 9-411, 1994.

17. C. Kapoor and D. Tesar, "A reusable operational software architecture for advanced robotics (OSCAR)," The University of Texas at Austin, Report to U.S. Dept. of Energy, Grant No. DE-FG01 94EW37966 and NASA Grant No. NAG 9-809, 1998.
18. Kennedy Carter tool sets, *Kennedy-Carter Corp.* http://www.kc.com/MDA/xuml.html.
19. Y. Kesten and A. Pnueli, "Control and data abstraction: Cornerstones of the practical formal verification," *Software Tools and Technology Transfer*, Vol. 2, No. 4, pp. 328–342, 2000.
20. R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun, "Static partial order reduction," in B. Steffen (Ed.), *Proc. of TACAS'98, LNCS 1384*, 1998, pp. 335–357.
21. R.P. Kurshan, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, Princeton University Press, 1994.
22. R.S. Lazic, "A semantic study of data independence with applications to model checking," PhD thesis, Oxford University, 1999.
23. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajani, and S. Bensalem, "Property preserving abstractions for the verification of concurrent systems," *Formal Methods in System Design*, Vol. 6, No. 1, pp. 11–44, 1995.
24. K.L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic Publishers, 1993.
25. K.L. McMillan, "A compositional rule for hardware design refinement," in *Computer Aided Verification*, 1997.
26. K.L. McMillan, "Verification of an implementation of tomasulo's algorithm by compositional model checking," in *Computer Aided Verification*, 1998.
27. J. Misra and K.M. Chandy, "Proofs of networks of processes," *IEEE Transactions on Software Engineering*, Vol. 7, No. 4, 1981.
28. M. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler, and D.L. Dill, "CMC: A pragmatic approach to model checking real code," in *OSDI*, 2002.
29. Y. Nakamura, *Advanced Robotics: Redundancy and Optimization*, Addison-Wesley, 1991.
30. K.S. Namjoshi and R.P. Kurshan, "Syntactic program transformations for automatic abstraction," in *Computer Aided Verification*, 2000, pp. 435–449.
31. D. Peled, "Combining partial order reduction with on-the-fly model-checking," *Formal Methods in System Design*, 1996.
32. A. Pnueli, "In transition from global to modular temporal reasoning about programs," *Logics and Models of Concurrent Systems*, 1985.
33. M. Pryor, "Task-based resource allocation for improving the reusability of redundant robots," PhD thesis, The University of Texas at Austin, 2002.
34. B. Sarna-Starosta and C.R. Ramakrishnan, "Constraint-based model checking of data-independent systems," in *ICFEM*, 2003.
35. SES: CodeGenesis, "CodeGenesis technical reference," SES Inc, 1998.
36. SES: ObjectBench, "ObjectBench technical reference," SES Inc, 1998.
37. N. Sharygina, "Model checking of software control systems," PhD thesis, The University of Texas at Austin, 2002.
38. N. Sharygina and J.C. Browne, "Model checking software via abstraction of loop transitions," in *ETAPS: Fundamental Approaches to Software Engineering (FASE)*, Vol. 2621 of *LNCS*, 2003, pp. 325–340.
39. N. Sharygina, J.C. Browne, and R. Kurshan, "A formal object-oriented analysis for software reliability: Design for verification," in *ETAPS: Fundamental Approaches to Software Engineering (FASE)*, Vol. 2029 of *LNCS*, 2001, pp. 318–332.
40. S. Shlaer and S. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice-Hall, 1992.
41. J.E. Smith, "Decoupled access/execute computer architectures," *ACM Transitions on Computer Systems*, 1984.
42. E. Stark, "A proof technique for rely/guarantee properties," in *FST&TCS*, Vol. 206 of *LNCS*, 1985.
43. L. Starr, *Executable UML: The Models that Are the Code*, Model Integration, LLC, 2001.
44. P. Wolper, "Expressing interesting properties of programs in propositional temporal logic," in *POPL*, 1987.
45. F. Xie, V. Levin, and J.C. Browne, "Objectcheck: A model checking tool for executable object-oriented software system designs," in *ETAPS: Fundamental Approaches to Software Engineering (FASE)*, Vol. 2306 of *LNCS*, 2002, pp. 331–335.