A Combined Testing and Verification Approach for Software Reliability

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

Doron Peled
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ, 07974
doron@research.bell-labs.com

Category 1 Paper

Abstract

Automatic and manual software verification is based on applying mathematical methods to a *model* of the software. Modeling is usually done manually, thus it is prone to *modeling errors*. This means that errors found in the model may not correspond to real errors in the code, and vice versa. In particular, if the model is found to satisfy the checked properties, the actual code may still have some errors. For this reason, it is desirable to be able to perform some consistency checks between the actual code and the model. Exhaustive consistency checks are usually not possible, for the same reason that modeling is necessary. We propose a methodology for improving the throughput of software verification by performing *some* consistency checks between the original code and the model, specifically, by applying software testing. In this paper we present such a combined testing and verification methodology and demonstrate how it is applied using a set of software reliability tools. We introduce the notion of a *neighborhood* of an error trace, consisting of a tree of execution paths, where the original error trace is one of them. Our experience with the methodology shows that traversing the neighborhood of an error was extremely useful in locating its cause. This is crucial not only in understanding where the error stems from, but in getting an initial idea of how to redesign the code. We use as a case study a robot control system, and report on several design and modeling errors found during the verification and testing process.

1 Introduction

Software reliability can be enhanced by applying various different analysis methods based on mathematical theories. This includes software verification and testing. Software testing is the more commonly used technique in the software industry. It involves generating test suites and sampling the execution of the code according to them. It usually can be applied directly to the actual code. Its main disadvantage is that it is not exhaustive. Thus, although it practically helps in detecting many of the program errors, it has a high probability of missing some of them. Automatic software verification is more exhaustive, but it is usually limited to finite state systems with a 'reasonable' amount of program states due to the problem of 'state space explosion'. Because of these limitations, model checking is usually applied to a *model* of the checked code rather than to the actual code directly. This model is obtained by manual translation through a process called "modeling". The process of manual translation may induce some errors in the model. This means that errors found in the model during its verification may not correspond to real errors in the code, and vice versa. Thus, even if the model is found to satisfy the checked properties, the actual code may still have some errors.

The correspondence between the actual code and the model can be addressed using different techniques. Simulation of the model execution [17] and testing based on formal descriptions of the functional and behavioral specifications [1] can be useful for checking that the implementation is behaviorally equivalent to the design. Another approach is to develop mapping algorithms to connect the implementation and the model [11], [19], [14]. Verifying the correspondence between the code and the model can also be done formally, e.g., using theorem proving technology. In most approaches consistency check is performed on a very informal basis and it is not exhaustive. When formal methods are applied, the verification is often hard to conduct, typically not easier than applying deductive verification to the original code directly.

With current testing and verification technologies, compromises on the depth of the analysis and the development time requirements are needed. In this paper we explore the synergy of testing and verification methods. We present a hybrid methodology to software reliability that combines program analysis and verification techniques. Our approach is more pragmatic than existing methods used to eliminate discrepancies between the code and the model because it applies testing techniques. This methodology addresses the issue of minimizing the number of errors introduced into the model during the translation process. It also helps identifying the causes of the conceptual errors found during the verification along with facilitating the software redesign process.

The fundamental principals of our combined testing and verification methodology include application of the testing process both during system modeling prior to the actual verification and during evaluation of the counterexample produced by the verifier after the verification is complete.

The fact that testing methods are applied to the modeled software allows us to validate the translation. The idea is to use a testing tool for the examination of the execution paths of the model, while comparing them with the behavior of the actual code. During this process, modeling errors, as well as possible conceptual errors, may be uncovered.

The common model checking approach is to apply the verification to the model, and if a counterexample is found, to compare it with the original code in order to check whether it is indeed an error (if it is not, it is called a 'false negative'). Doing any conformance testing on the checked model with respect to the actual code increases the dependability of model checking. We introduce the notion of a neighborhood of an error trace, which consists of a tree of execution paths, where the original error trace is one of them. Our experience with this methodology shows that traversing the neighborhood of an error was extremely useful in locating its cause. This was crucial not only in understanding where the conceptual error stems from, but in getting an initial idea of how to correct the code.

We demonstrate the methodology using a case study taken from robotics, namely a robot control software. Our proposed methodology does not depend on the choice of specific tools. We have used it with the following combination of tools: as the verification tool, we used the Spin model checking system [10] and as a testing tool, we used the Pet system [7]. The errors, found in different stages of the verification process, include modeling errors and design errors. These errors led both to modification in the model and to changes in the design.

Section 2 provides the description of the combined testing and verification methodology. Section 3 describes application of the methodology to the verification of the robot control system and presents its results. Section 4 provides conclusions and describes future research.

2 The Combined Methodology: Testing and Verification

One of the main problems of software verification is that it is applied to a *model* of the software, rather than to the software directly. This stems from mathematical limitations related to dealing with infinite state systems, limitations on memory usage, and the use of different kinds of notations and programming languages in different tools. As a consequence, a discrepancy in functionality can exist between the actual code and the checked model. The possible danger is twofold: errors found in the checked model may not correspond to actual executions of the original code. On the other hand, a positive verification result may not have taken into account all the executions of the actual code.

The usual practice in this case is that a model is constructed and verified, and error traces found later during the verification are compared against the original code. If these error traces are not compatible with the code, then the model needs to be modified accordingly. Because of a possible modeling error, when model checking does not come up with an error, there is sometimes very little that we know about the correctness of the checked property.

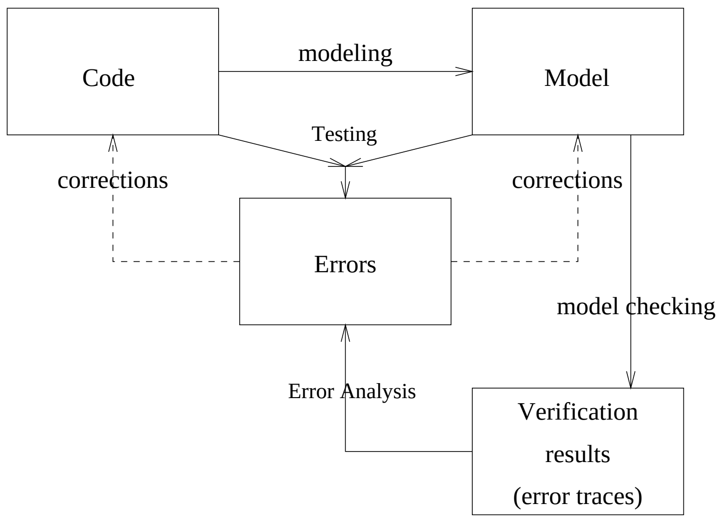

Figure 1: The combined testing and verification methodology

In order to minimize the effect of the possible discrepancy between the model and the code, we propose a methodology of testing the checked model against the actual code, as part of applying formal methods. This is done by integrating interactive 'white box testing' of the model [15] with the verification process. We examine execution paths of the model and compare them to the original code. Detected discrepancies usually reflect modeling errors and result in the modification of the model. Moreover, in some cases, the testing process can result in the discovery of errors in the original code.

Figure 1 is used to illustrate our methodology. It adds into the verification process a testing tool, which is used to analyse execution paths of the model. A testing tool that supports interactive and visual representation of the program's structure would benefit to the process of simulation and examination of the program execution paths the most.

The interactive testing approach we use is based on a flow graph notation. We remind the reader this commonly used formalism. A node in a flow graph is one of the following: *begin, end, predicate, random, wait, assign, send* or *receive*. The *begin* and *end* nodes appear as ovals, the *predicate*, *wait* and *random* nodes appear as diamonds, labeled by a condition, or the word *random* (which stands for a nondeterministic choice). All other nodes appear as boxes labeled by the assignment, send or receive statement. Each node, together with its output edge constitutes a *transition*, i.e., an atomic operation of the program, which can depend on some condition (e.g., the current program counter, an *if-then-else* or a while *condition* in the node, the non-emptiness of a communication queue) and make some changes to the program variables (including message queues and program counters). White box testing (specifically, unit testing) can be performed by examining paths in flow graphs. Different *coverage techniques* [15] suggest criteria for the appropriate coverage of a program by different paths.

We introduce the notion of a *neighborhood* to the analysis of the execution paths. Consider an execution path in the flow graph of a sequential program. A *related path* of the flow graph can be obtained from it by selecting at some point an alternative edge out of a condition node. Continuing this process of obtaining alternative paths, we obtain a tree, which we call the neighborhood of the original path. In concurrent code, a trace consists of a sequence of nodes from different processes. Projecting on the processes, the neighborhood of a path generates a single tree for each process. Figure 2 represents a path (emphasized) and its neighborhood.

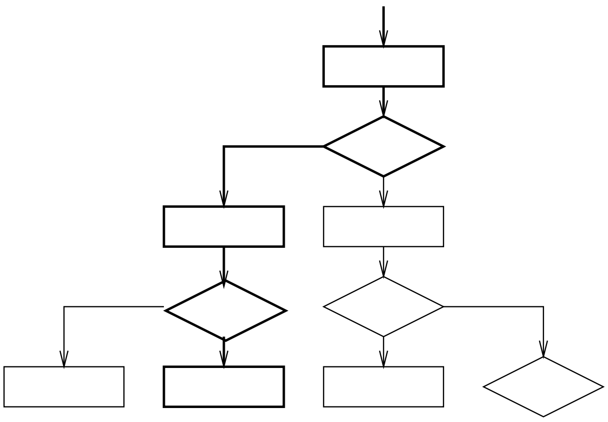

Figure 2: Representation of a neighborhood of a path

In our approach we use testing in two different ways:

White box testing of the model, while comparing it with the original code. We perform interactive software testing before performing model checking. Traversing execution paths allows us to better understand the code. Errors found in this way are usually modeling errors. They lead to changing the model, and repeating the testing process. There is no clear guidance to how much testing is needed to obtain confidence that the model reflects the code properly.

Testing the neighborhood of an error trace. After an error trace is found with a model checker, a testing tool is used to traverse the neighborhood of that trace. We explore the neighborhood until the relevant part of the code is understood, and the cause for the error is detected. The main point is that it is not only a single error trace that is used to detect an error, but also some related execution paths, namely the neighborhood. Errors that are found in this way can be both modeling or conceptual errors. Thus, a change is needed in either the model or the original code, respectively.

In summary, our methodology proceeds as follows. The verification process starts with modeling. Then white box testing continues as long as modeling errors are found, and until a certain degree of confidence in the verification model is obtained. Then, we can start doing model checking. Upon finding an error, neighborhood testing is performed. Analyzing the error results in correcting the original code or the model.

3 The Methodology in Practice

As a working example for our methodology, we examine a Robot Control System (RCS). It is a subset of a multiple criterion decision support software [2], used for controlling 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. This includes space and underwater operations, nuclear cleanup, and bomb disposal. Failure recovery is one of the examples of redundancy resolution applications: if one actuator fails, the controller locks the faulty joint and the redundant robot continues operating. The robot control algorithms that support redundancy resolution combine requirements for significant computations, hard real-time responsiveness, stringent reliability requirements, and distributed and networked implementation. This combination of characteristics makes application of formal methods highly desirable. Although we analyse a simplified version of the RCS in this paper, this is not a toy example, and its study and verification is of high interest to the robotics research.

The robot control system controls the motion behavior of the robot arm and includes kinematics algorithms and interfaces to the robotic computational libraries, which are the components of the OSCAR [12] system.

In the following sections we describe the functionality of the RCS, specify properties that we formally verified using model checking, and present the combined verification and testing process in particular using the Spin verifier, and the Pet path exploration tool.

3.1 The Robot Control System

In the description of the RCS we use the convention that the names of the processes in ObjectBench representation of the RCS are in italics and start with a capital letter. The names of the variables are in italics and start with a lowercase letter.

A robot arm consists of two 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 processes *Arm*, *Joint1*, *Joint2* and *EndEffector* in the software design.

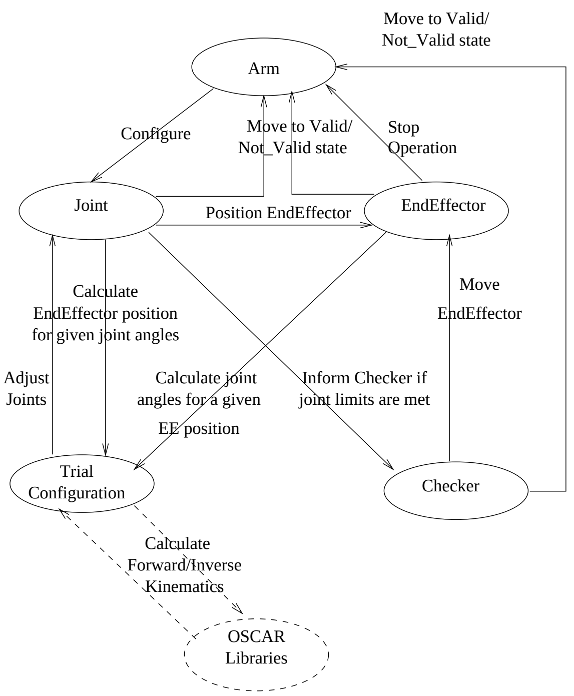

Figure 3: The object communication model for the robot control system

For each joint we specify an angle, representing a rotation of the joint relative to its neighboring link. The end-effector position (*Current_position*) is given as a vector of positions (*cp_x*, *cp_y*, *cp_z*) and orientation angles (α, β, γ). The system's task is to move the end-effector along the specified path. We examine a simplified instance, in which the end-effector moves only in the horizontal, i.e., the x direction.

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 [6]. 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 [6] 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 (*Limit_Max*, *Limit_Min*) and on the end-effector position (*max_x*,*max_y*, ...). 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* process 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* process indicating its invalid state.

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

The concurrent behavior of the robot components makes it difficult to analyze the overall behavior of the system. In particular, the calculations of the movement of different parts of the robot are done in a distributed way. A major concern is that different parameters that correspond to previous and future moves may be used incorrectly. This may result in an improper combination, leading to a faulty behavior of the arm. In order to prevent such situations, formal verification is applied.

The design of the RCS is done using the ObjectBench [20] notation. It is an object-oriented development and analysis environment, which has been used for the automatic generation of C++ code that can be used for controlling the robot. [1]

The object communication model of the RCS is presented in Figure 3. It displays the flow of events of the RCS in terms of the control signals exchanged among the objects in the system. Exchange of data in the model is represented by an arrow originating from a source object to a destination object.

The state transition diagram shown in Figure 4 specifies a lifecycle of one of the processes of the RCS, the *EndEffector* process. It consists of nodes, representing states and their associated actions to be performed, and event arcs, which represent transitions between states.

3.2 Experimental Environment

In order to verify the RCS, we have selected tools for model checking and testing, which can support our methodology. These tools are described below.

3.2.1 Model Checking in SPIN

Model checking [3, 5, 18] is a generic name for a family of algorithms aimed at the automatic verification of finite state systems. SPIN is a state-based model-

[1] Some abstraction was already used in creating the ObjectBench description. In fact, the ObjectBench code itself can actually be seen as a model for the real system. However, the fact that there are two levels of abstraction is orthogonal to our methodology, and might be misleading in presenting it. We thus treat the ObjectBench description as the 'code' and ignore the lower level of the robot control system.

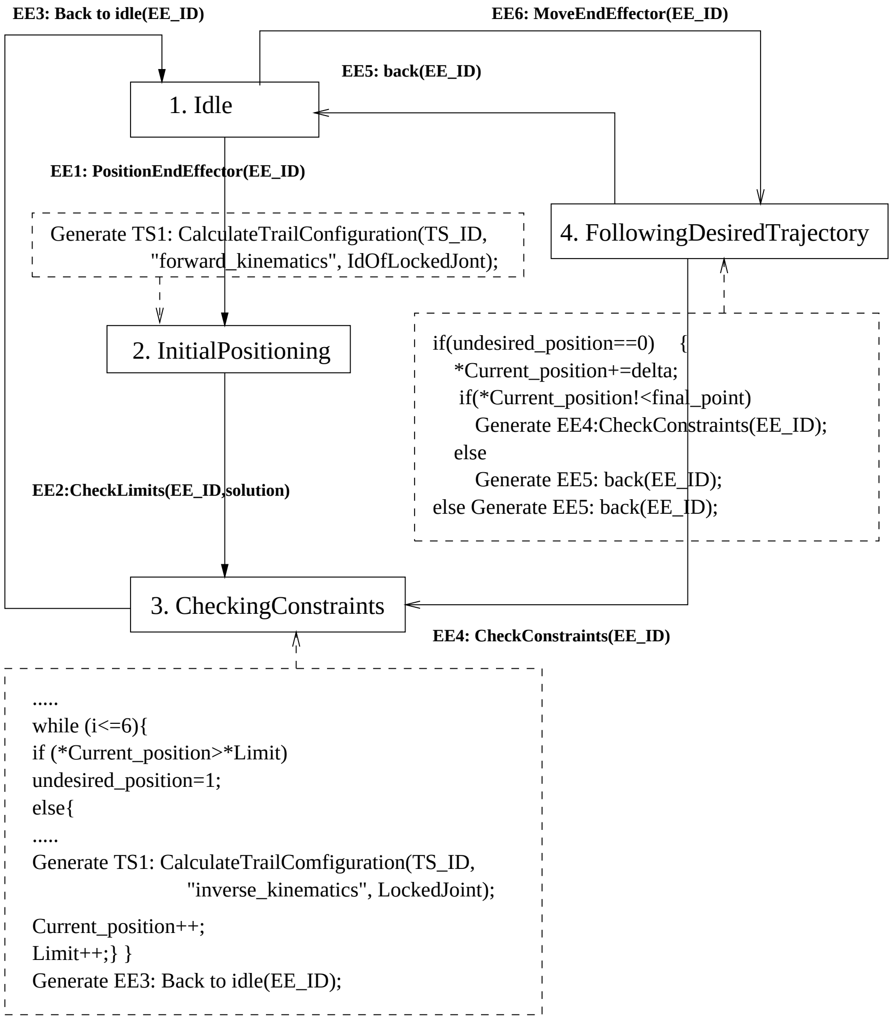

Figure 4: The ObjectBench Transition Diagram for the *EndEffector* process

checking tool designed for the efficient verification of distributed process systems.

The specification language of SPIN is called PROMELA. PROMELA is a programming language that includes sequential constructions inspired by Dijkstra's Guarded command [4], communication structures influences by Hoare's CSP [8] and expressions taken from **C** [13]. SPIN operates by an explicit enumeration of reachable states. Checking that a model of a program satisfies a property is done by performing an optimized *depth-first-search*. The basic mode of operation of SPIN is based on *exhaustive* reachability analysis.

Specific correctness properties can be expressed in the syntax of Linear Temporal Logic (LTL) [16]. This logic include *modal* operators, expressing properties that change over time. In particular, we can write $\Box\varphi$ to express that φ holds forever, $\Diamond\varphi$ to denote that φ would hold eventually, and $\varphi \,\mathrm{U}\, \psi$ to denote that φ will continue to hold until ψ holds. Combining several modalities allows us to express more complicated formulas. For example, $\Diamond\Box\varphi$ means that φ will eventually start to hold and would then continue to do so forever. The formula $\Box\Diamond\varphi$ means that φ would hold infinitely often. The formula $\Box(request \rightarrow \Diamond granted)$ can be used to assert that at any point in the execution, if a request was made, it is eventually granted.

3.2.2 Testing using PET

The PET system works with a concurrent program consisting of several processes with shared variables and a synchronous communication. Syntactically, this is an extension to the programming language PASCAL.

PET automatically generates the flow graph of the tested processes. It then allows the user to select concurrent execution paths of these processes. PET leaves the choice of the paths to the user. The user can choose a path by clicking on the appropriate nodes on the flow graph. The path can also consist of an interleaving of nodes from multiple concurrent processes. The user can also create a variant of the path by backtracking to a *predicate* (or *random*) node, and making an alternative selection. Another way to alter a path is to use the same transitions but allow a different interleaving of them. When dealing with concurrent programs, the way the execution of transitions from different nodes are interleaved is perhaps the foremost source of errors. The PET tool allows the user to flip the order of adjacent transitions on the path, when they belong to different processes.

In order to make the connection between the code, the flow graph and the selected path clearer, sensitive highlighting is used. For example, when the cursor points at some predicate node in the flow graph window, the corresponding text is highlighted in the process window. The code corresponding to a predicate node can be, e.g., an *if-then-else* or a *while* condition.

Once a path is fixed, the condition to execute it is calculated, based on repeated symbolic calculation of preconditions, as in program verification [9].

The condition is calculated backwards, starting with *true*. Thus, we proceed from a *postcondition* of a transition, in order to calculate its *precondition*. In order to calculate the precondition given the transition and the postcondition, we apply various transformations to the current condition, until we arrive to the beginning of the paths. For a transition that consists of a predicate p with the 'yes' outedge, we transform the current condition from c to $c \wedge p$. The same predicate with the 'no' outedge, results in $c \wedge \neg p$. For an assignment of the form $x := e$, we replace in p every (free) occurrence of the variable x in the postcondition c by the expression e. We start the calculation with the postcondition *true* at the end of the selected path.

The main information provided by PET is the condition to execute a selected path. The meaning of the calculated path condition is different for sequential and concurrent or nondeterministic programs. In a sequential deterministic program, the condition expresses exactly the possible assignments that would *ensure* executing the selected path, starting from the first selected node. When concurrency or nondeterminism are allowed, because of possible alternative interleavings of the transitions or alternative nondeterministic choices, the condition expresses the assignments that would make the execution of the selected path *possible*.

The path condition obtained in this process is simplified using rewriting rules, based on arithmetic. Subexpressions that contain only integer arithmetic without multiplication (Pressburger arithmetic) are further simplified using a decision procedure (see [7]). In this case, we can also check algorithmically whether the path condition is equivalent to *false* (meaning that this path can never be executed), or to *true*.

3.3 Verification of the RCS

We have performed a manual translation of the ObjectBench code into a SPIN model, written in the programming language PROMELA. At the same time we translated the same code into a PET model. The target programming language of PET is only syntactically different from PROMELA. Moreover, there is a one to one correspondence in their sequential syntactic constructs (e.g., loops, conditionals) and the concurrency features (e.g., shared variables and communication). Thus, although SPIN and PET do not accept exactly the same input, we could use PET to perform the white box testing of the ObjectBench code with the SPIN code (with the obvious possibility of having introduced additional typos).

In order to reduce the complexity of the original code we had to abstract and restrict some calculations. In fact, in the ObjectBench code the robot arm movement calculations are done through the interface with the OSCAR libraries. In this example we abstracted away actual calculations and replaced them with nondeterministic 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. Figure 5 graphically represents a flow graph of the *Arm* process. The events *to_joint1!1*, *to_joint2!1* are used to initiate movements of the joints and the *arm_status* variable is used to store information about the status of the arm configuration (*Valid* vs *Not_Valid*). Below we present the PROMELA code for the *EndEffector* process. The actions associated with the events of the *EndEffector* process as specified in the Transition Diagram in Figure 4 are presented as the comments in the PROMELA code.

```
proctype endeffector (){
byte m;
byte c_p_x=0, c_p_y=0, c_p_z=0, alpha=0, beta=0, gamma=0, k;
  do
  :: c_i<2  -> {
     to_endeffector?m;
     c_i=c_i+1;
     if
     :: c_i==2 -> {
        to_trialconf!1;                         //PositionEndEffector
        to_arm!0;
        do
        :: end_position==0 ->
           ee_reference=0;
           if
           :: endeffector_status==1-> {
              to_endeffector?k;
              if
              :: abort_var==1 -> break
              ::else -> skip
              fi;
              c_p_x=c_p_x+delta;                //MoveEndEffector
              to_recovery!0,0;
              to_arm!0;
              ee_reference=1}
              ::else -> skip
              fi;
              if
              :: c_p_x<=finale -> {
              if                                   //CheckConstraints
              ::( (c_p_x<=max_x) && (c_p_y<=max_y) && (c_p_z<=max_z) &&
                  (alpha<=max_a) && (beta<=max_b) && (gamma<=max_g) ) -> {
                   if
                   ::endeffector_status==0 ->
                     to_joint2!1;
                     endeffector_status=1   }
                   ::else-> to_trialconf!0     //CalculateTrialConfiguration
```

```
                  fi }
                  ::else -> { end_position=1;
                              to_arm!1;
                              break}
                  fi     }
                  ::else ->{
                    end_position=1;
                    to_arm!1;
                    break}
                  fi    }
         od }
    ::else -> skip
    fi     }
  ::else -> break
od  }
```

3.4 Testing and Verification Results

We have formed three generations of SPIN models. *Model 0* is the first model created by translation from ObjectBench code. *Model 1* is obtained after making some corrections based on the white box testing results. *Model 2* corresponds to an intermediate version of these models. It includes several changes from *Model 1*, which where made as a result of finding modeling errors. *Model 3* is our final model of the RCS, whose implementation underwent some design changes in order to correct the conceptual errors found during the testing and model checking processes.

We checked a collection of correctness requirements specifying the coordinated behavior of the RCS processes. The requirements were encoded as LTL formulas. We expressed all the formulae in terms of state predicates. Since SPIN prefers specifying properties over states, rather than over events, we sometimes needed to encode the occurrence of some important events by adding new state variables.

We demonstrate the advantages of using the combined testing and verification methodology using a selection of the specifications that failed the formal checking. We then discuss how the proposed methodology was used for the redesign of the original code.

Consider the following description of the checked properties. We refer in this description to the states appearing in the state transition diagrams in ObjectBench. For example, the state FollowingDesiredTrajectory appears in Figure 4.

1. $DeadlockFreedom$

   The model does not have deadlocks.

2. $\Box(ee\_reference = 1 \rightarrow arm\_status = 1)$

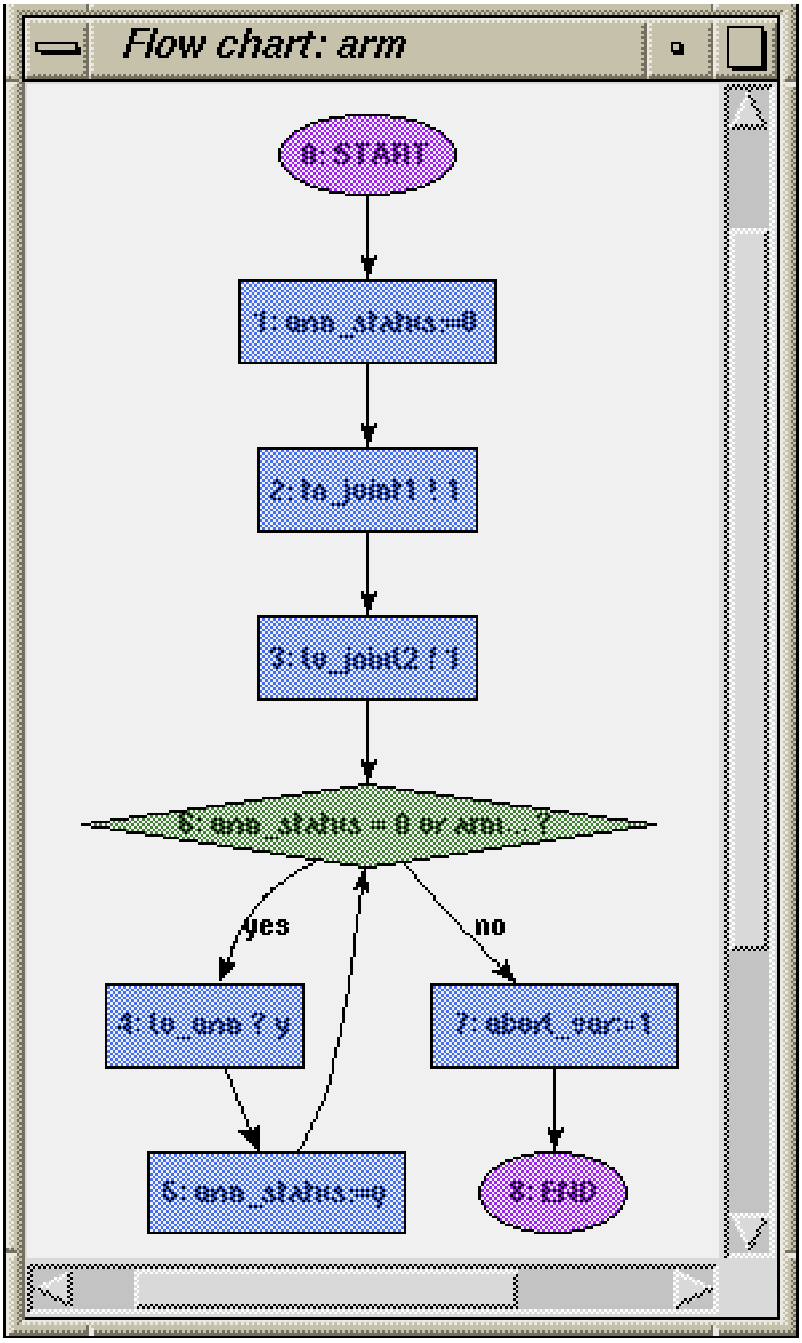

Figure 5: Flow graph representation of the *Arm* process using Pet tool

	Model 1			Model 2			Model 3	
Prop	Tools	Result	Err tp	Tools	Result	Err tp	Tools	Result
1	S +P	False	mod	S +P	False	concept	S	True
2	n/a	n/a		S +P	False	concept	S	True
3	n/a	n/a		S +P	False	concept	S	True
4	S	True		S	True		S	True

Table 1: Experimental results

Whenever the *EndEffector* process is in the FollowingDesiredTrajectory state (*ee_reference* variable is equal to 1) then the *Arm* process is in the "Valid" state (the *arm_status* variable is equal to 1).

3. $\diamond(abort\_var = 1\,\mathrm{U}\,(end\_position = 1 \vee (recovery\_status = 1 \wedge counter = number\_joints))$

   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).

4. $\neg MoveEndEffector\,\mathrm{U}\,(MoveEndEffector \wedge (\neg CalculateTrialConfiguration\ \mathrm{U}\,PositionEndEffector))$

   No command to move the end-effector is scheduled before defining an initial end-effector position.

We demonstrate the results of verification and testing of these properties (*Prop*) in Table 1, and follow it by a detailed discussion of the nature of the errors found. In Table 1, *S* and *P* stand for Spin and Pet, respectively, *Err tp* defines an error type, which can be either a modeling error *mod*, or a conceptual error *concept*. We use the notation *n/a* to distinguish the fact that the verification of Properties 2 and 3 was not performed for *Model 1*.

We started by examining the translated model *Model 0* and comparing it to the known behavior of the original code using Pet. By exploring several possible interprocess communications, we discovered that the variable *endeffector_status* of the *EndEffector* process was mistakenly changed in both of the joint processes (*Joint1* and *Joint2*), instead of only within the *EndEffector* process. This variable was changed unexpectedly from 0 to 1 before the calculation of an initial end-effector position was completed. Comparing this with the original code, we found that this was a modeling error. Consequently, we fixed the Pet and Spin models accordingly. In order to verify that the change was correct, we specified Property 4 (see Table 1), which was later verified to hold in *Model 1*.

After obtaining more confidence in the Spin model, by performing repeated testing using Pet, we obtained *Model 1*. We have checked *Model 1* for deadlocks (the deadlock-freedom is Property 1 in the Table 1). SPIN checks the code for several safety properties, including absence of deadlocks, if no explicit temporal formula is given. We found that a deadlock exists. Tracing the neighborhood of the deadlock using PET, enabled us to find the error. We observed that in the model, the *counter* variable of the *Checker* process is not reset to zero when it was equal to *number_joints*, as opposed to the original code.

Thus, another modeling error was identified. We have fixed this error. At this point, after these corrections to the model, we have obtained *Model 2*. We repeated the SPIN verification for Property 4 (the cause of the previous deadlock) on this model, and this check succeeded. Nevertheless, we found using SPIN that a deadlock still occurs. After examination of the error track that led to the deadlock situation, and studying its neighborhood with PET, we realized that this was due to a conceptual error in the fault recovery algorithm.

To confirm this fact we formulated and checked Property 3, which was aimed at checking whether the system terminates properly. This property did not hold for *Model 2* and the examination of the error track led us to the conclusion that the system does not terminate in the case where there is no solution for the fault recovery. We will remind the reader that the fault recovery procedure is activated in the RCS 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 SPIN 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, Property 3 failed since in our example it can be shown that requests originated from *Joint1* and *Joint2* to recompute the angles of these joints could continue indefinitely: if *Joint1* does not respect the limit then the fault recovery is called and *Joint1* is locked with the angle limit value. The *Joint2* angle is being recalculated for the original *EndEffector* position. If the a new angle of *Joint2* does not satisfy its limit then another fault recovery procedure is called, which attempts to find a new angle for *Joint1* while *Joint2* angle is locked. If there is no resolutions that satisfies the limit for *Joint1* than fault recovery is called again. This is also a confirmation of the above deadlock situation.

Another conceptual error found during verification of *Model 2* indicated a problem of coordination between the *Arm* and the *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 *ee_reference* variable changes its value from 0 to 1. An interaction between the processes led to the situation where the update of the *ee_reference* variable precedes the change of the *arm_status* value. This was the reason for Property 2 to fail.

In order to fix these conceptual errors a redesign of the original system was required. Figure 6 reflects the changes made. The corresponding model is then *Model 3.* We had to introduce a new process called *Recovery*, whose functionality

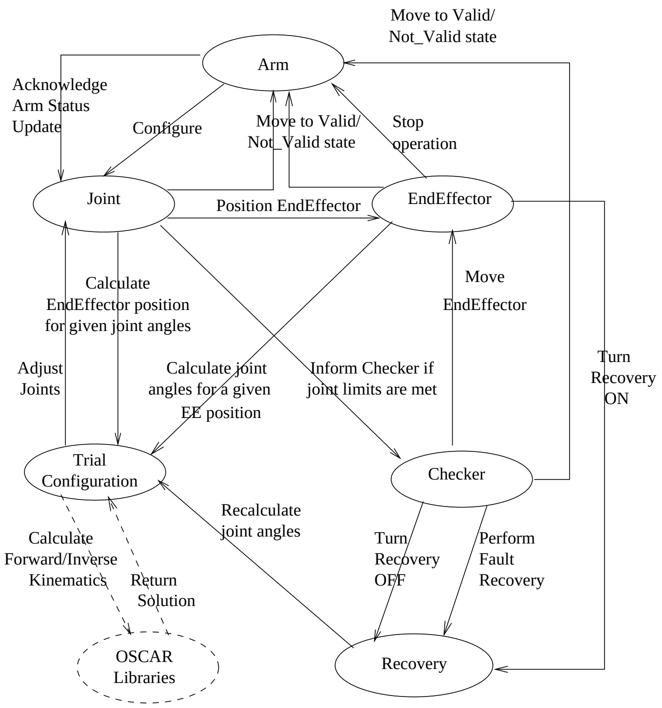

Figure 6: The modified model of the robot control system

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. Formal verification of the redesigned model confirmed that a new design satisfies all of the properties described above.

4 Conclusions

Model checking and testing techniques provide two complimentary ways of improving the reliability of software, however, traditionally they have been used separately. In fact, they are usually being advocated by different communities. We have proposed a methodology that combines the two techniques. This is done by applying testing to check the model that is created for the purpose of model checking. Further, testing is used to examine the results of model checking and locating the possible causes for the detected error.

We have demonstrated our methodology on a real application, that of a robot control system. Several algorithms that are in current use are known to have design errors. Several of these errors were identified using our methodology. We used a collection of formal methods tools, namely, the PET interactive testing system, and the model checking system SPIN. One of the major achievements of this experiment is that we could find conceptual errors and correct them quite quickly, namely within a month of work of one person. This even included the learning of the tools that were used.

We introduced the notion of *neighborhood* of an error into formal verification. This is a collection of execution sequences that are simple variants of the error trace found during the automatic verification. The PET tool was helpful in testing neighborhoods.

Model checking is often used to verify the control part of a system. It is less effective in debugging the data dependent (sometimes called 'data path') part of the system. The data dependent part often provides a conceptually infinite state space, or at least one that is too big to be automatically verified using current tools. In our methodology, we can extend the testing process to deal with the data dependent part of the system, which are not handled by finite state model checking techniques.

For example, we can extend the RCS model, to deal with 'painting' a surface. The painting is controlled by the RCS. A mathematical equation is used to limit the painted area. For example, to make sure that we are within a radius r from some origin, we can check that the relation between the radius and the x and y coordinates position is $x^2 + y^2 \leq r^2$. We can use PET to generate the necessary path conditions for executions that include the painting. This can be used in testing the behavior of the extended model.

As a consequence of our experiment with the tools SPIN and PET, and with the presented methodology, we suggest a new tool that combines the verification and testing process described in this paper. Along with the automatic verification process, the tool will have the ability to display an error trace and the capability of tracing the neighborhood of an error. The tracing will be connected visually with the code and with its graphical representation as a flow graph. We have found such a combination (by joining the capabilities of the above mentioned tools) useful in a rapid process of verification and redesign of the example software.

Acknowledgement

We would like to thank Nina Amla for reading the manuscript and making many useful suggestions for improvements.

References

[1] K. Bogdanov, M. Holcombe, H. Singh, Automated Test Set Generation for Statecharts, in D. Hutter, W. Stephan, P. Traverso and M. Ulmann, editors, *Applied Formal Methods - FM - Trends 98*, Vol. 1641 of Lect. Not. in CS, 107-121, Spring Verlag, 1999.

[2] M. Cetin, C. Kapoor, D. Tesar, Performance based robot redundancy resolution with multiple criteria, Proc. of ASME Design Engineering Technical Conference, 1998, Atlanta, Georgia.

[3] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic. Workshop on Logic of Programs, Yorktown Heights, NY, 1981, LNCS 131, Springer-Verlag.

[4] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Comm. ACM, 18(8), 1975, 453–457.

[5] E.A. Emerson, E.M. Clarke, Characterizing correctness properties of parallel programs using fixpoints, LNCS 85, Springer Verlag, Automata, Languages and Programming, July 1980, 169–181.

[6] J. J. Graig, *Introduction to Robotics: mechanics and control*, Addison-Wesley, 1989.

[7] E.L. Gunter, D. Peled, Path Exploration Tool, TACAS 1999, Amsterdam, The Netherlands, 405–419.

[8] C.A.R. Hoare, Communicating Sequential Processes, Comm. ACM, 21(8), 1978, 666–677.

[9] C.A.R. Hoare, An axiomatic basis for computer programming, Communication of the ACM 12, 1969, 576-580.

[10] G.J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall Software Series, 1992.

[11] D. Jackson. Aspect: Detecting Bugs with Abstract Dependencies. ACM Transactions on Software Engineering and Methodology, Vol. 4, No.2, April 1995, 279-295.

[12] C. Kapoor, 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.

[13] B. Kernighan, D. Ritchie, *The C programming Language*, Prentice Hall, 1988.

[14] G. Murphy, D. Notkin, Leighweight Source Model Extraction. Proc. of the ACM SIGSOFT 95 Symposium on the Foundations of Software Engineering.

[15] G.J. Myers, *The Art of Software Testing*, Wiley, 1979.

[16] A. Pnueli, The temporal logic of programs, Proc. of the 18*th* IEEE Symp. on Foundation of Computer Science, 1977, 46-57.

[17] *Precise UML Group*, http:/www.cs.york.ac.uk/36/.

[18] J.P. Quielle, J. Sifakis, Specification and verification of concurrent systems in CESAR, Proceedings of the 5th International Symposium on Programming, 1981, 337–350.

[19] K. Rustan, M. Leino, Toward reliable modular programs, Technical Report CS-TR-95-03, Califirnia Institute of Technology, January 1995.

[20] SES inc., ObjectBench Technical Reference.