Formal Verification of Control Software: A Case Study

Andreas Griesmayer[1], Roderick Bloem[1], Martin Hautzendorfer[2], and Franz Wotawa[1]

[1] Graz University of Technology, Austria
{agriesma,rbloem,fwotawa}@ist.tu-graz.ac.at
[2] Festo AG, Vienna, Austria
hautzendorfer@festo.at

Abstract. We present a case study of formal verification of control logic for a robotic handling system. We have implemented a system in which properties can be specified in the source code, which is then automatically converted to Java and checked using Java Path Finder. The model checker, working under the assumption of a nondeterministic environment, is able to efficiently verify critical properties of the design.

1 Introduction

Software errors can cause large amounts of damage, not only in safety-critical systems, but also in industrial applications. An error in the control program for a robot, for example, may cause damage to products and to the robot itself. In such areas as automotive engineering, both the robots and the products are very expensive. Moreover, the followup costs can be a multiple of the direct costs: the production line may have to be halted while a technician travels to the site to repair the problem.

The design of concurrent software is difficult. The environment strongly influences the order in which parts of the program are executed, which introduces a source of variation that makes testing difficult [LHS03].

To make sure that errors do not occur, formal techniques are required. Model checking [CGP99] in particular is a technique to prove adherence of a system to a given property, regardless of the behavior of the environment. Today, model checking is mainly employed in hardware [KG99], whereas research into model checking for software is still in its infancy [BR01,CDH+00,God97,VHB+03].

The benefits of model checking are

Full coverage. Unlike testing, model checking verifies all possible behavior.
Light-weight specification. Only properties of interest are stated and the specification need not be finished before the implementation is started.
Automatic proof. The user is not exposed to the mathematical details of the proof of correctness.
Reuse in testing. Properties written for formal verification and for testing can be shared.

In this paper, we present a case study of formal verification of control software for a robotic handling system. The software was built to show the capabilities of *DACS*, a

novel language for control software developed by Festo. Though small, it is a typical example of software written for an industrial handling system and can thus be used to check for the absence of typical errors.

We formulated safety properties (the robot arm does not move when blocked) as well as liveness properties (the robot does not get stuck in a given state). The model checker was able to prove absence of such bugs in the original system and to detect bugs in an altered system. In order to prove such properties for any environment behavior, we modeled a nondeterministic environment. Our system works by translating DACS code into JAVA, which is then fed to Java Path Finder [VHB+03]. Properties are specified directly in the DACS source code. We expect that the approach shown here is applicable to a large variety of control software.

Related research includes the work done by Bienmüller, Damm, and Wittke [BDW00], who verify automotive and aeronautic systems specified with state charts. State charts are a specification formalism, whereas our approach verifies the implementation directly. A specification is not always available, and verification on the implementation has the advantage that changes made to the implementation only are also verified. To our knowledge, direct verification of implementation code by translation to Java has not been attempted before.

In Section 2, we will describe the DACS language and the handling system. In Section 3, we discuss how correctness was verified, and we conclude with Section 4.

2 Handling System

The handling system is shown in Fig. 1. One robot arm takes products from the carrier and moves them to the conveyor belt and another one takes them back, forming a closed loop. The system consists of two belts and two robot arms with five actuators each to control the movements (raise and lower the arm, move the arm across, open and close the two grippers, and turn the grippers on the arm).

2.1 DACS

The control software has been implemented in DACS. The statements at hand are similar to familiar imperative languages like Java. Methods and variables regarding common objects are combined in classes. Each method is stored in its own file, and the static structure of the program is stored in a series of XML files containing the classes, their attributes and methods, and the instantiation of the objects. Each class is instantiated a finite, fixed number of times. Dynamic creation of objects and dynamic allocation of memory is not used because of time and memory constraints on an embedded system. This implies that the state space of the system is finite, which makes model checking easier.

A system consists of a hierarchical set of components, each described by a state machine. Each state machine executes one step (possibly staying in the same state) before passing control to its ancillary state machines, resulting in a program that can be run in one thread of control, but behaves like a set of parallel machines. A snipped of an state machine is given in Figure 2(a).

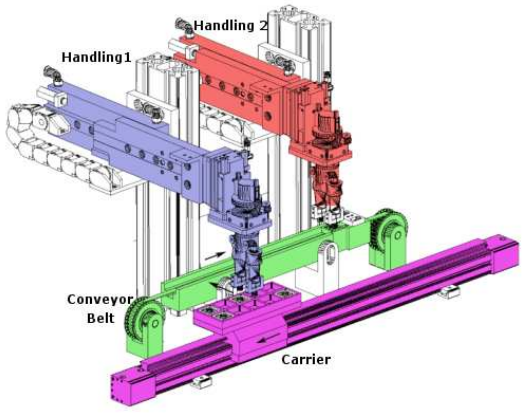

Fig. 1. the handling system

2.2 Properties

We checked two properties, each representative of a class.

safety As an example for a safety-property we checked that the robot arms do not move horizontally while they are in their down position (because they might crash with a belt).

liveness To provoke a liveness-failure, one of the conveyor-belts was "blocked" causing the system to wait infinitely to be allowed to proceed. This does not provoke a deadlock in the sense that no more events can be executed — the system loops through a couple of states — but it does not make a real progress either.

Fig. 2(a) shows the state machine controlling a single arm. *Vert* and *Hori* control the air-pressured cylinders that move the arm horizontally and vertically, respectively. When the robot arm is in its *Base* position, both are contracted, i.e., the arm is over the conveyor belt in the down position. The correct code first expands the *Vert* cylinder, moving the arm up and away from the belt, and then expands the *Hori* cylinder, moving the arm across, thus avoiding a crash with the carrier.

In the faulty version, shown in Fig. 2(b), we switched this order to provoke a crash. For the simulation of the liveness property we changed the implementation of the stepper motor of the carrier, which is part of the environment, to never report to reach its destination (not shown).

3 Verification

3.1 Translating the Handling System

For the case study, we developed a compiler which translates the DACS source code together with a set of properties to Java.

```
SEQUENCE Handling

 STEP A.Base_0:
  Hold1.Base();
  Hold2.Base();
  NEXT_STEP;


 STEP A.Base_1:
  IF Hold1.InBase() AND
     Hold2.InBase() THEN
    Vert.Work();
    NEXT_STEP;
  END_IF;


 STEP Base_2:
  IF Vert.InWork() THEN
    Hori.Work();
    Turn.Work();
    NEXT_STEP;
  END_IF;
```

(a) original DACS code

```
SEQUENCE Handling

 STEP A.Base_0:
  Hold1.Base();
  Hold2.Base();
  NEXT_STEP;


 STEP A.Base_1:
  IF Hold1.InBase() AND
     Hold2.InBase() THEN
    Hori.Work(); // error
    NEXT_STEP;
  END_IF;


 STEP Base_2:
  IF Hori.InWork() THEN
    Vert.Work(); //error
    Turn.Work();
    NEXT_STEP;
  END_IF;
```

(b) safety fault introduced

```
switch(pos){

 case 1: //Base_0
  Hold1.Base();
  Hold2.Base();
  pos=pos+1;
 break;

 case 2: //Base_1
  if(Hold1.InBase()&&
     Hold2.InBase()){
    Vert.Work();
    pos=pos+1;
  }
 break;

 case 3://Base_2
  if(Vert.InWork()){
    Hori.Work();
    Turn.Work();
   pos=pos+1;
  }
 break;
}
```

(c) Java code

Fig. 2. In the correct code, STEP *A.Base_0* gives the command to open both grippers (*Hold*-cylinders). In *A.Base_1*, the vertical cylinder is moved to its top position when both Holds reached their base position. Finally, in *Base_2*, horizontal and turn cylinders are started. In the faulty version, we switched Vert and Hori to provoke a crash.

The Java Path Finder model checker (JPF) [VHB+03] is based on a backtracking Java virtual machine. It searches the entire state space of the Java program, which in our case is finite. JPF provides assertion methods, and properties can be included in the Java sources as calls to these methods.

Most statements, such as the `IF`-statement, function calls and assignments, are translated to Java in an obvious way — the corresponding Java statements provide the same functionality. State machines (`SEQUENCE`) are translated to a `switch-case`-statement and an extra member-variable keeping the current state. The structure of a DACS-program stored in its XML files is translated to a set of Java classes, one per DACS class. The instantiation of the objects is translated to a main() function and a set of default constructors, which instantiate the main class and the ancillary classes. The main() function also contains the code to drive the main state machine. Fig 2(c) gives the Java code corresponding to Fig. 2(a).

As JPF requires a closed system in order to do model checking, we implemented the environment directly in Java. The environment of the system was modeled using JPF's features for nondeterministic choice: hardware responds to a request in time that is finite but not predetermined.

Models for complex applications might exceed the size we can cope with. As the range of a variable has a strong impact on the size of the model, data abstraction tech-

Table 1. results of model checking the control software

system	DFS				BFS			
	mem (MB)	**time (s)**	**states**	**trace**	**mem (MB)**	**time (s)**	**states**	**trace**
correct	72	18	161,023	N/A	85	1405	161,023	N/A
safety error	34	24	91,470	45,430	7.8	11	11,097	3,121
liveness error	13	4	4,257	4,256	37	4	74,790	3,992

niques like those used in the Bandera framework [CDH+00] replace it by a small number of tokens. If, for example, the rotation of the grippers is stored in degrees as integer, we could use *range abstraction* to replace all values smaller than zero and greater than 359 by *invalid*, thus reducing the range to a fraction. Because we are only interested in two special positions, we may even increase the savings by *set abstraction*, which replaces the range by the tokens $\{$*in_base*, *in_between*, *in_work*, *invalid*$\}$. Further abstraction modes include *modulo-k abstraction*, which we can use to equate, for example, numbers that have the same remainder when divided by 360, and *point abstraction*, which drops all information by using a single token *unknown*.

3.2 Modeling Properties

JPF only offers checks for invariants. We translated liveness properties to invariants by the addition of a monitor which counts the time of no recognizable progress. Progress is perceived when the top-level state machine changes state. If the counter exceeds a fixed value, an assertion is violated and the error is reported. This value is easy to set: if the value is too small, a counterexample is shown in which progress still occurs, which is easily recognized by the designer. If the value is too large, a deadlock will still be caught, but the counterexample will be longer than strictly necessary.

We need to check the truth of an invariant between each two consecutive statements. To achieve this behavior, we exploit a characteristic of model checking concurrent threads: execution of events (statements) is interleaved nondeterministically. Hence, we perform the check of the safety-condition in a separate monitoring thread, which moves to an error state when the condition is violated. If a violation ever happens, there is a thread interleaving in which the monitoring thread moves to the error condition, and the model checker finds this interleaving.

Safety properties are specified by a new keyword, ASSERTGLOBAL, which takes a DACS-expression as argument. A second keyword, ASSERT, acts like the assert statement in the C language by ensuring that the given expression is satisfied at the respective position.

3.3 Case Study

The DACS sources of the control software consist of 1300 lines of code. Conversion to Java and implementation of the environment led to 12 Java classes with a total of 1340 lines.

Table 1 gives the results of our checks, for the three cases of the correct system, the system with the safety error, and the system with the liveness error. The memory (time) column gives the amount of memory (time) needed, the states column gives the number of states traversed, and the trace column give the length of the error trace, if applicable. Experiments were performed on a Linux machine with a 2.8GHz Pentium IV and 2GB of RAM.

JPF uses Depth First Search (DFS) as its standard search order. The correct system had about 160,000 states and was checked in 18 seconds. DFS needs less memory and is far faster then Breadth First Search. The remaining test cases justify the use of BFS:

When an fault is found, JPF dumps an error trace consisting of executed statements and stops searching. BFS guarantees the shortest possible trace to an error by examining all states which are reachable in a given number of steps before increasing this number. This enhances the readability of the error trace and can significantly decrease the number of searched states, and thus amount of memory.

4 Conclusions

We have shown how control software for a robotic handling system can be verified automatically. Our example is typical of an industrial application. Though experiments with larger systems remain necessary, we have shown that translation to Java and verification by a general Java model checker leads to satisfactory results at a reasonable effort.

Robotic control software is hard to test off-site because of its concurrent behavior. Faults on-site, however, may be expensive. We believe that model checking can fill an important gap in securing the reliability of such systems.

References

[BDW00] T. Bienmüller, W. Damm, and H. Wittke. The Statemate verification environment, making it real. In E. A. Emmerson and A. P. Sistla, editors, *proceedings of the twelfth International Conference on Computer Aided Verification (CAV00)*, pages 561–567. Springer-Verlag, 2000. LNCS 1855.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In M.B. Dwyer, editor, *8th International SPIN Workshop*, pages 103–122, Toronto, May 2001. Springer-Verlag. LNCS 2057.

[CDH+00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering (ICSE'00)*, pages 439–448, 2000.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[God97] P. Godefroid. Model checking for programming languages using verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[KG99] C. Kern and M. R. Greenstreet. Formal verification in hardware design: A survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, April 1999.

[LHS03] B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent java components. *IEEE Transactions on Software Engineering*, 29:555–566, 2003.

[VHB+03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10:203–232, 2003.