MODEL CHECKING OF ROBOTIC CONTROL SYSTEMS

S. Scherer[(1)], F. Lerda[(2)], and E. M. Clarke[(2)]

[(1)]*The Robotics Institute, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, U.S.A.*
[(2)]*Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, U.S.A.*
{*basti, lerda, emc*}*@cs.cmu.edu*

ABSTRACT

Reliable software is important for robotic applications. We propose a new method for the verification of control software based on Java PathFinder, a discrete model checker developed at NASA Ames Research Center. Our extension of Java PathFinder supports modeling of a real-time scheduler and a physical system, defined in terms of differential equations. This approach not only is able to detect programming errors, like null-pointer dereferences, but also enables the verification of control software whose correctness depends on the physical, real-time environment. We applied this method to the control software of a line-following robot. The verified source code, written in Java, can be executed without any modifications on the microcontroller of the actual robot. Performance evaluation and bug finding are demonstrated on this example.

Key words: Model Checking; Verification; Control Systems; Software Testing; Java.

1. INTRODUCTION

Reliability of software is key to successful robot and space missions [1]. Control software is at the foundation of complex robotic systems and higher level software relies on its correct implementation. Although logical errors in an abstract design may occur, programming bugs in the implementation of an abstract design are frequently the reason for software crashes [2].

Conceptually, there are two distinct types of methods to validate control systems: *generative methods* and *direct methods*. Generative methods [3] take an abstract representation of the system, e.g. a block diagram, and generate source code that implements the control strategy. The source code is then integrated with the control computer source code. Unless the generated code can be used unmodified and the generator of source code itself is proven correct, this method is not able to guarantee a correct implementation. Direct methods, instead, prove the correctness of the control systems itself, regardless of how it has been developed, and therefore they do not depend on the correctness of the development tools.

We chose a direct method based on software model checking [4], a verification technique that performs an exhaustive and systematic search of the reachable states of a model. Model checking can uncover subtle bugs in software and provides a trace of the execution steps that lead to a violation.

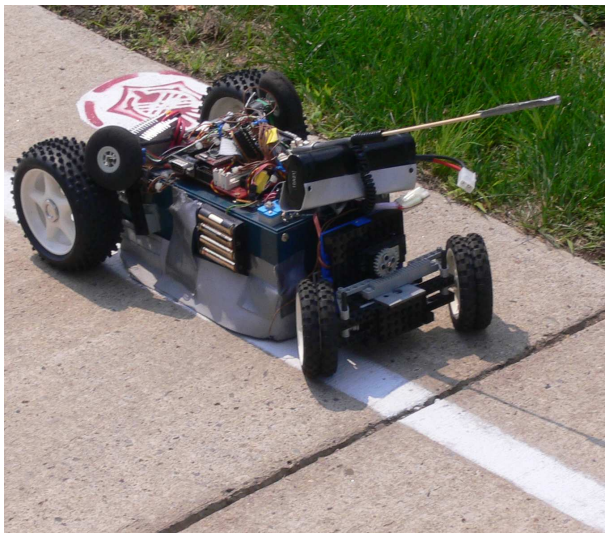

Fig. 1. A line-following robot on a white line in the initial configuration.

One of the biggest impediments in applying model checking to robotic systems is that most tools are not designed to handle the software's interaction with the physical environment. Specifically, our method explictly models the interactions with the environment, allowing the verification of more complex properties about the robotic control system. We have developed an extended version of the model checker Java PathFinder [5] that verifies models with these additional constraints.

With our method control system designers can express the controller naturally: differential equations are used to express the physical system and source code is used to describe the software. This more detailed model allows our method to verify safeness and liveness properties that refer to the controlled system as well as implicit correctness properties of the software, e.g. null-pointer dereferences, assertion violations, and deadlocks. Exploiting the exhaustiveness of the model checker, it is also possible to check the software under different scheduling choices and introduce errors on the sensors and actuators. The additional capabilities come at the price of higher computational complexity, which is well justified for mission

Proc. 'ISAIRAS 2005 Conference', Munich, Germany, 5-8 September 2005 (ESA SP-603, September 2005)

critical components.

We have applied our method to the source code of a line-following robot (Fig. 1) that won the "Mobot" race, a robot design competition at Carnegie Mellon. The robot has to follow a white line along a course in the least time possible. Two separate tasks control velocity and streering. The microcontroller of the robot executes Java code, therefore we chose Java PathFinder for our implementation. We use this robot as a case study to evaluate the effectiveness of our verification method.

The paper is organized as follows: in Section 2 we discuss some related work; Section 3 introduces our algorithm; Section 4 presents the case study in detail and the results of the verification; lastly, Section 5 provides some conclusions and directions for future work.

2. RELATED WORK

The use of model checking in the context of robotic applications is not new. For instance, model checking has been applied to a robot arm in [6] and a servo loop controller in [7]. However, in these cases, the goal of verification was to uncover violations of safety and liveness properties in the design and the actual software implementation had been abstracted. These approaches fail to capture the software implementation in enough detail to guarantee the absence of errors and require considerable manual effort in developing a suitable abstraction of the software.

Java PathFinder itself, the model checker we extended, has been used to verify the software of a Mars rover [2]. However, the focus of that study was to uncover common fatal program errors like data races and deadlocks, which do not depend on the environment the software is executed in. Our method, an extension of Java PathFinder, is still able to find these errors, but it adds the capability of checking more complex properties that involve an interaction with the environment.

Modeling techniques for systems that include both continuous and discrete components exist. In particular, hybrid automata [8] can represent the type of systems that we consider in this paper. Moreover, model checking techniques for hybrid automata have been proposed and model checking tools like HyTech [9] have been used to verify robotic systems, such as arm control systems [10], mobile robot systems [11], and highway traffic control systems [12]. These techniques focus, however, on the continuous components of the controlled system and a coarse abstraction of the software is usually considered. Our method, instead, focuses on the software and applies directly to the source code of the controller.

3. METHOD

3.1. Modeling the System

We aim to verify control software. Typical examples are controllers for robotic systems, where a program is running on an embedded microcontroller. However, in general, it is not possible to examine the software in isolation. The correctness of the control software often relies on implicit assumptions about the system it controls. Moreover, properties are expressed in terms of the behavior of the controlled systems, not in terms of the behavior of the software itself.

In this section we present a model made of three components: *(i)* the controlled physical system; *(ii)* the control software; and *(iii)* the scheduler.

The Controlled Physical System. The physical system is modeled as a continuous or discrete time dynamical system. In the following, we assume that the system can be modeled using linear time-invariant ordinary differential equations. Specifically, we consider the dynamics of the physical system in state space form[13], i.e. $\dot{x} = Ax + Bu$, where x is a vector of state variables and u is a vector of inputs. The inputs u are the quantities that the software controls via the actuators. Let y to be the vector of outputs, determined by the equation $y = Cx + Du$. The outputs y are the quantities that can be observed by the software via the sensors. The values of A and B determine the dynamics of the system. The values of C and D determine which parts of the state are accessible by the software.

The Control Software. The software consists of a set of tasks, implemented on top of a real-time scheduling component. Each task is a concurrent thread of execution. The body of a task is a fragment of code written in a programming language. The body is executed periodically with a fixed period. The controller communicates with the controlled physical system by means of sensors and actuators. The sensors measure some physical quantity of the controlled system while the actuators change some physical quantity. The sensors and the actuators define the interface between the controller and the controlled system.

The Scheduler. The execution of the code is constrained by the scheduler which executes each task at the proper time. The main reason for including the scheduler in our model is that the behavior of the controller usually depends on the timing of the different tasks. Therefore, we maintain some of this information in order to verify the correctness of the software. Moreover, the introduction of a scheduler may even make the verification easier: by constraining the execution of the different threads, a reduced set of interleavings needs to be explored, which yields a smaller number of reachable states.

3.2. Verification Algorithm

The algorithm we propose is based on the depth-first search model checking algorithm implemented in Java PathFinder. The depth-first search looks for a violation of the properties by visiting all reachable states of the system. At each step the next state is generated; if the state has been visited before, the algorithm backtracks; otherwise, the search continues with the newly discovered state. The procedure continues until all reachable states have been visited.

Our algorithm (Fig 2) needs to keep track of the discrete variables present in the program as well as the continuous variables which define the state of the physical environ-

ment. Moreover, a new state is generated either by a discrete transition (the execution of a program statement) or by a continuous transition (the evolution of the physical system over a given amount of time).

```
var visited : set = {};
procedure HybridMC(P, S, E, ds, cs);
begin
  visited = visited ∪ {(ds, approx(cs))};
  foreach ds' in DiscreteSuccs(P, S, ds, cs) do
  begin
    cs' = ContinuousSucc(E, delta(S), ds', cs);
    if (ds', approx(cs')) ∉ visited then
      HybridMC(P, next(S), E, ds', cs');
  end;
end;
function DiscreteSuccs(P, S, ds, cs): set;
var succs: set = ;
  procedure DFS(ds);
  begin
    visited = visited ∪ {(ds, approx(cs))};
    if no_tasks_are_running(ds) then
      succs := succs ∪ {ds};
    else
      foreach ds' in ProgramSuccs(P, S, ds, cs) do
      begin
        if (ds', approx(cs)) ∉ visited then
          DFS(ds');
      end;
  end;
begin
  DFS(ds);
  return succs;
end;
```

Fig. 2. Pseudo-code of the main algorithm.

The global variable `visited` keeps track of the states that have already been explored.

The outer procedure, `HybridMC`, takes as arguments a program `P`, a scheduler `S`, an environment `E`, a discrete state `ds`, and a continuous state `cs`. `HybridMC` generates all states reachable from the state $\langle$`ds`, `cs`$\rangle$. It alternates discrete steps (`DiscreteSuccs`) and continuous steps (`ContinuousSucc`). The value of `delta(S)` is the amount of time that needs to pass before another task is scheduled. If a state that is already in the visited set is generated, the procedure backtracks. Otherwise, the search continues in a depth-first fashion from the new state $\langle$`ds'`, `cs'`$\rangle$ with an updated scheduler `next(S)`.

The procedure `DiscreteSuccs` computes the set of states that satisfy the following three conditions: *(i)* that are reachable from the given state $\langle$`ds`, `cs`$\rangle$ by means of discrete transitions; *(ii)* that do not have any outgoing discrete transition; and *(iii)* in which all the tasks are idle, waiting for the next period. The procedure uses depth-first search to enumerate the states. This is similar to the way a model checker generates the reachable states. Properties are checked within this procedure as well, but this has been left out of the pseudocode for readability.

The procedure `ProgramSuccs` computes the successors of a state that are obtained by executing a single discrete transition of the program `P`. The set of discrete transitions considered may be constrained by the scheduler `S`.

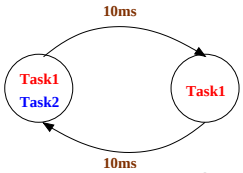

Fig. 3. A scheduler for tasks `Task1` and `Task2` with periods $10ms$ and $20ms$.

The procedure `ContinuousSucc` updates the continuous state using the differential equations given by the environment `E` and the time interval `delta`.

The scheduler `S` can be represented as a finite state machine where each node is labeled by a set of tasks that are allowed to execute and each edge is labeled by the amount of time that needs to pass before the next set of tasks can execute. For instance, a scheduler for tasks `Task1` and `Task2` with periods $10ms$ and $20ms$ respectively is depicted in Fig. 3.

The algorithm makes the simplifying assumption that the time to execute each task is zero, or very small compared to the tasks' periods. As a consequence, all inputs are read and all the controls are set at the beginning of a task's execution. This is an approximation of how the real system behaves, but it is an acceptable approximation as our case study demonstrates. In future work, we would like to explore more sophisticated ways of approximating the execution time of the code, such as use worst case execution time analysis [14].

The continuous variables can assume an infinite number of values. However, we desire a representation of the state that is finite. Since continuous variables are part of the state, we need a finite representation of the continuous variables. Such conversion is performed in the algorithm by the function `approx`. In the next section we give a possible implementation of the `approx` function which performs an approximation on the values of the continuous variables in order to reduce the set of reachable states.

3.3. Implementation

Java PathFinder [5] is an explicit state model checker that takes as input Java code. The model checker exhaustively explores all the reachable states of a given program. States are generated by a depth-first search. Computation of the successors of a state is performed by a custom Java Virtual Machine (the Java PathFinder Virtual Machine). Java PathFinder itself is written in Java, therefore, its code is executed by a standard Java Virtual Machine (or Host Virtual Machine).

One of the features of Java PathFinder that we exploited in our extension is the so-called Model Java Interface (or MJI). MJI allows the users to specify code to execute when a particular method is called, instead of the code associated with the method itself. Among the reasons for this facility are the need to specify code for native methods (methods of a Java class provided as native binary code by the runtime) as well as the need to define

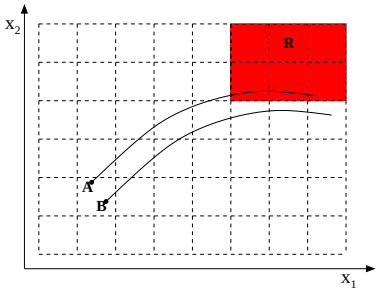

Fig. 4. Two trajectories rooted at points A and B belonging to the same approximated state. The trajectory rooted at A reaches region R, while the one rooted at B does not.

a more efficient implementation for some critical methods. Methods implemented using MJI are more efficient because their code is executed by the Host Virtual Machine instead of the Java PathFinder Virtual Machine. As a consequence, intermediate states within an MJI method are not stored by the model checker and their instructions are not interleaved with the instructions of other threads.

In order to implement our algorithm in Java PathFinder, we first extended the representation of the state to include the continuous variables. Since the domain of the continuous variables is infinite, we need to provide a finite representation for each continuous variable. Possible representations include infinite precision rational and float-point representations, but we choose a fixed-point representation whose parameters, e.g. the precision, can be set depending on the application. This choice was made because we wanted a represention as compact as possible (to reduce the space required to store the states). A precision higher than the computational error can lead to two equivalent states having different representations.

The function `approx` introduced above converts the values of the continuous variables into the appropriate fixed-point representations. The result of this approximation can be interpreted as performing an abstraction of the continuous space which associates each approximated value with an hyper-rectangle whose size depends on the precision (Fig. 4). If two states differ only for the values of the continuous variables and these values fall in the same hyper-rectangle, then the two states have the same representation. For instance, points A and B in Fig 4 have the same representation. Note that this approximation may lead to unsoundness if an error-state (i.e. hyper-rectangle R in the figure) can be reached from some points within an hyper-rectangle (point A) but not from all (point B).

Another necessary modification to the core of Java PathFinder was to extend the scheduler to take into account discrete and continuous transitions. In our prototype implementation, we only considered periodic tasks with the same period. The scheduler will then execute the body of each task until all tasks complete. At this point, `ContinuousSucc` is executed, which updates the continuous state.

The Java PathFinder Virtual Machine is used to implement the procedure `ProgramSuccs`: given the current discrete state, it generates the next discrete state.

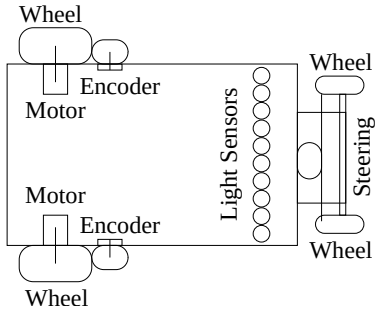

Fig. 5. Diagram of the robot showing the location of sensors and actuators.

`ContinuousSucc` is implemented as an MJI procedure. The body of this procedure is currently written manually and corresponds to a numerical algorithm for the set of differential equations that describe the physical environment. Implementing the numerical integration as a MJI procedure has the advantage of reducing the computation time as well as the state space (cf. above).

4. CASE STUDY

Below we demonstrate the feasibility of our method by applying it to the line-following robot shown in Fig. 1. We first present the problem and then explain how our robot is designed to traverse the course. After that, we outline the steps involved in model checking the software of the robot and the results we obtained. There are both general and more specific techniques, such as abstraction and system identification respectively, that enable us to verify the software using the extended Java PathFinder.

4.1. Problem Statement

The goal of this case study is to verify the correctness of the two control algorithms of the robot. These algorithms are implemented in software, running and interacting on the same microcontroller. This raises a number of issues traditionally not considered in either the model checking or control community. In the model checking context correctness is defined by invariants, many of which are implicitly defined by the environment. Moreover, the quantization steps in sensors and actuators as well as dependencies between the two control algorithms increase complexity.

A slalom course laid out on the pavement in front of the Computer Science Department at Carnegie Mellon defines the challenge for the robot. A white line connects a series of gates which the robot follows while controlling the speed. Various weather and lighting conditions challenge sensing, computing, and locomotion. The robot tries to complete the course as quickly as possible while tracking the line.

4.2. Robot

A robot designed to meet the specification requires an adequate physical platform with correct software. Therefore, we first examine the hardware underlying the software and then describe the tasks processing the sensor data and executing the control algorithms.

Hardware. The robot is 41x27x12cm in size and utilizes two battery packs with 7.2V and 12V to provide power for processing and actuators independently.

Two geared 12V motors drive the back wheels as depicted in Fig. 5. The microcontroller adjusts the pulsewidth of the signal and effectively regulates the power supplied to the motors. To actuate the rack-and-pinion steering, the microcontroller sends pulsewidth commands to a servo.

Steering and velocity commands are determined using the data read from the sensors. The robot has 12 sensors (Fig. 5): ten brightness sensors measure the reflectivity of the ground and two encoders measure the velocity and distance travelled. The brightness sensors are mounted on a line perpendicular to the direction of travel. The approximate offset from the center of the line is measured using this sensor arrangement.

The input/output (I/O) ports of the microcontroller are connected to an analog-to-digital (A/D) converter, 25-tick encoders, a servo, and an amplifier. Three tasks read and write values via I/O ports and memory. The microcontroller directly executes Java, which significantly simplifies the verification using Java PathFinder. It implements Java 2 Micro Edition with the connected limited device configuration, and it has multithreading and a preemptible garbage collector. The software is written and compiled on a host computer and then downloaded to the flash memory of the microcontroller.

Software. The software implements two controllers, which regulate the steering and the speed, as two separate tasks. An additional task reads the reflectance values from the brightness sensors. All computations do not involve dynamic object creation or destruction and use only integer arithmetic to improve performance. Depending on the setup, different interleavings of tasks are possible but all tasks execute periodically with a frequency of 33Hz.

In particular, a periodic scheduler is implemented using a *piano roll timer* [15]. In this configuration, the tasks are executed in a round-robin fashion and execution of a task starts exactly when a beat occurs, allowing accurate timing of processes which need to satisfy hard real-time constraints. Priorities determine the order in which tasks are executed. The piano roll timer is configured with a beat of 1ms and a duration of 80ms. Each task has zero initial delay and a period of 10ms. The calculations of initial delays, periods, duration, beat, and priorities are not difficult but errorprone, as we will demonstrate in Section 4.3.

The A/D converter measures the voltage generated by each light sensor. The microcontroller communicates via a serial bus to read the values. Since commands are sent at the fixed frequency of 33Hz, sensor values are read at the same frequency. The two encoders are connected directly to the microcontroller and provide speed and distance travelled using interrupts.

The cached brightness values are used by the controller to calculate a steering command. After normalizing the values, the algorithm finds the left and right edge of the line. Proportional control is performed to try to keep the middle of the line centered underneath the sensors. If it is ever detected that the line is out of bounds, the last known good direction is used as the steering direction.

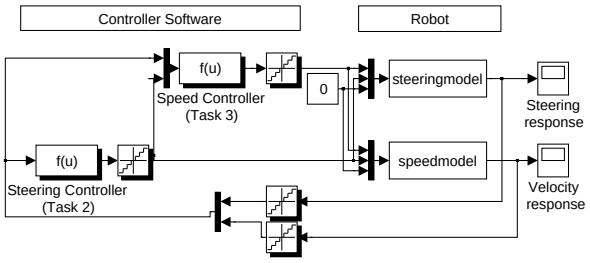

Fig. 6. A block diagram representing the control software and discrete time plant model of the robot. Inputs and outputs are quantized since fixed-point arithmetic is used. A dependency between the steering and speed control task is shown.

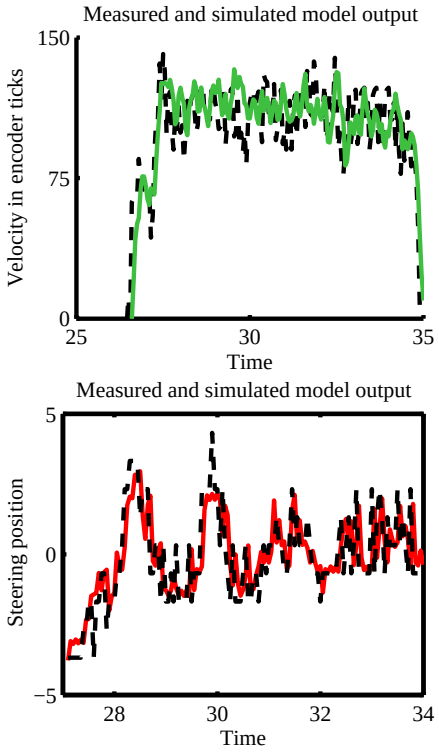

Fig. 7. The identified system response for steering and velocity data as a solid line on top of a validation dataset.

The speed of the robot is calculated from elapsed time and registered encoder ticks. The desired goal speed is determined from the the last steering command in order to adapt to curvature. The difference between the two determines the pulsewidth command sent to the motor driver.

4.3. Verification

The hardware and software are used to define assertions about the correctness. Similar to a control theoretic approach, we need an environment ("plant") to control. The system is depicted in a block diagram representation in Fig. 6. The control software is on the left and the environment is on the right. The inputs and outputs of the controller software are quantized since the implementation of the software relies on fixed point integer arithmetic.

The dependency between the steering and speed control tasks is indicated by a connection from the output of the steering controller to the input of the speed controller.

Identify the Physical System. The environment can be derived from first principles or automatically identified using a technique known as *system identification* [16]. Although the robot is simple, it is not trivial to derive the physical model from first principles because many parameters are unknown and hard to measure. Therefore, we derived the environment using system identification. We recorded seven traces of the inputs and outputs of the robot and searched for the best model fitting the relationship between the inputs and outputs.

The encoder response to a pulsewidth command applied to the motors is direct and therefore a first-order system matches well. The position relative to the line depends on the velocity of the robot and the steering angle. A fourth-order system gives a good relationship between velocity, steering pulsewidth, and position relative to the line.

We tested the response on a validation dataset to confirm the validity of our models with the identified systems. In Fig. 7 the prediction of the sensor output is plot on top of a validation dataset.

The identified system model constrains the input into the software. Since the interactions with the environment are executed at a fixed frequency of 33Hz, we express the system as a discrete state space systems with a sample time of $0.03s$. This eliminates the problem of potential numerical integration errors.

Design for Verification. While Java PathFinder can be applied to any kind of software, designing the software with verification in mind is very useful in making verification tractable. The source code of the robot, however, was not developed following this principle and some modifications to the original source code had to be made to be able to verify the software using our extended Java PathFinder. In particular, there are two unbounded variables, time and distance, that increase monotonically. This increase leads to an infinite state space: the model checker will eventually run out of memory without completing the verification. In order to make the state space finite, we abstracted the unbounded variables. If this is not possible, one can provide bounds for the variables and consider states outside the bounds to be equivalent. Moreover, the source code may contain fragments that are irrelevant for verification but increase the size of the state space. In our case study, we removed parts of the initialization code that calibrate the sensors and test the actuators.

The initial configuration of the robot defines the initial values of the state vector. Although it is possible to test ranges of initial configurations, we focused on the initial configuration depicted in Fig. 1. In this configuration the robot steers to the right since it wants to reach back to the center of the line. The robot is offset from the middle of the line by 36mm, i.e. the middle of the line is under the rightmost sensor. The robot starts with zero initial velocity.

Identify Properties. The closed system with initial conditions is fully specified and permits reasoning about the properties that are necessary to ensure the correct operation of the robot. In this context properties of safety, liveness, and correct implementation are important.

Safety properties state conditions that must hold for the robot to be able to advance and not damage the hardware. In particular, the line has to be visible by at least one light sensor to ensure that a correct command is sent. This is specified as a property which checks that the position with respect to the line is within range. The speed also has to be within a range to give the robot enough time to process and react to the available data.

- $50 < speed < 150$ (S1)
- $0 \leq \ sensor\ position\ \leq 9$ (S2)

Liveness, in our context, states that the robot is always able to make progress along the line and come closer to the middle of the line as time progresses. In the case of a straight line, we assert that the speed should be greater than zero and the robot has to attain the middle of the line after at most 3s.

- $speed > 0$ after $t > 3$ (L1)
- $4 \leq \ sensor\ position\ \leq 6$ after $t > 3$ (L2)
- $\Delta t > 0$ (L3)

It is not sufficient to define the correctness of the robot software only in terms of safety and liveness because the invariants assume a correct usage of the underlying system. For this Java microcontroller in particular, it is difficult to use the periodic scheduler and the pulsewidth module correctly because some sets of parameters are invalid and *Java Exceptions* are not thrown in all cases.

The piano roll periodic scheduler executes tasks with different periods, initial delays, and priorities by arranging them in a table with a priority mask defining which task priority is runnable. We assume that the runtime of one iteration in all tasks is always less than the period. For a complete verification one can use an analysis as in [17] and the longest time execution path to assert that the execution time is less than the period.

The piano roll timer has a *duration* and a *beat*. The implementation of the piano roll scheduler generates an array of size $s = \frac{duration}{beat}$. The array contains an entry corresponding to each of the beats that occur in a single $duration$. Each entry contains a set of bits corresponding to the task priorities. Tasks in one priority are scheduled in a round-robin fashion. A task with period p and initial delay id starts at beats multiple of p offset by id. If a task with period p, initial delay id, and priority x is added, bit x is enabled for the corresponding entries in the array. This implementation does not correspond to the intuition that one adds a task with a certain period and it is executed periodically because tasks with the same priority, initial delay, and period are executed in a round-robin fashion. Consequently, instead of executing with a period of p they execute with a period of $n \cdot p$ where n is the number of tasks. We enforce the intuition using properties derived and partly taken from [15]:

- $1 \leq beat \leq 65$ hardware limitation (PR1)

Experiment	Time	Memory
Model without Noise	0:00:07	5 Mb
Noisy Encoder Values	0:45:41	106 Mb
Light Sensor Failure	1:12:55	159 Mb
Noisy Pulsewidth Commands	0:18:18	41 Mb

Fig. 8. Computation time and memory necessary for the verification with and without different kinds of errors.

- $duration \geq beat$ (PR2)
- $beat \leq p \leq duration$ (PR3)
- $(p = 0)\ mod\ beat$ (PR4)
- $(duration = 0)\ mod\ p$ (PR4)
- $0 \leq id$ (PR5)
- $id < p$ (PR6)
- $id + p \leq duration$ (PR7)
- $p = t.p \forall\ t \in$ Tasks (PR8)
- $0 \leq x \leq 14$ hardware limitation (PR9)
- if two tasks have the same period and same initial delay then $t1.x \neq t2.x$ (PR10)
- if two tasks are enabled in the same array entry $t1.x \neq t2.x$ (PR11)

Additionally, it is important to verify that the commands for the pulsewidth module are valid in order to avoid an error condition in the system.

- The Servo has lower and upper pulsewidth-bounds: $800\mu s < t_s < 2200\mu s$ (PW1)
- 0 is not a valid command for the pulsewidth module because the output is activated constantly. The command also has to be less than the duration of 10ms: $0\mu s < t_m < 10000\mu s$ (PW2)

Implementation. The microcontroller has its own application programming interface (API) to connect to the input and output ports. Using MJI procedures (see Section 3.3), the API is emulated by the model checker, which simulates the behavior of the environment. For example, a call to *setPulseWidth* updates the continuous variables stored inside the model checker instead of sending a value to the actuator. This separates the code implementing the controller from the code handling I/O.

Once the commands have been received, the state is updated in the environment and the model checker saves this updated state. The state of the environment consists of time, the state vectors, and last inputs. Two states are considered equal if, in addition to the discrete equivalence, the countinuous state variables are approximately equal up to a given fixed-point precision. We introduce nondeterminism in various parts of the system to model errors in the commands, encoder readings, and light sensor values, in order to examine their impact on the behavior of the robot.

4.4. Results

The goal of the case study was to demonstrate our approach to verifying robot control system software. The problem was initially ill defined, because no model of the environment or formal specification was available. Considerable effort has been spent on bridging this gap.

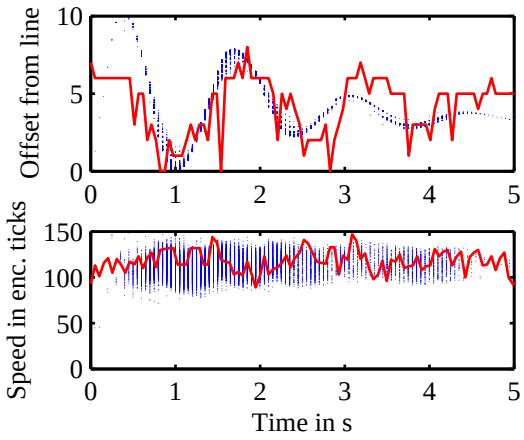

Fig. 9. Scatter plots demonstrate the range of possible outcomes for added encoder inaccuracy. The plotted line shows the result of an actual experiment.

All safety (S1, S2) and liveness properties (L1, L2, L3) were verified in a short amount of time (Fig. 8). However, model checking robot control software, like software development, is an iterative process. The environment model influences the safety and liveness properties that can be checked. For this case study, we had to refine the model of the environment multiple times.

The piano roll periodic scheduler, on the other hand, is a typical model checking problem and the existing bugs could be found within seconds. The most significant bug was that all tasks used the same period, initial delay, and priority (PR10, PR11). This configuration, as explained in Section 4.3, causes a reduced frequency of task execution. During testing of the robot this bug was never discovered because the system still behaved correctly.

In [1], a fatal type-conversion bug for the Ariane 5 rocket is analyzed. In this incident, the conversion of a 64-bit floating point value to a 16-bit signed integer caused an overflow. This occurs only for specific trajectories, making it a perfect candidate for model checking. We investigated if this type of bug could be discovered using our technique. The source code of the robot was seeded with an intermittent type-conversion bug on the number of encoder ticks per period. A signed byte, which ranges from -127 to 128, records that value. Usually, less than 128 ticks occur within one period, but occasionally this value is exceeded, e.g. when going down a hill. This bug is hard to find if the inputs and outputs are not determined using a model of the environment, but was detected using our model checker as a violation of property (L1), which asserts that the speed is always greater than zero.

In order to examine the effects of different kinds of sensor and actuator inaccuracy, we added nondeterministic behavior to parts of the model. Specifically, we explored the influence of a ± 5 slip in encoder ticks, a random failure of two light sensors, and a $\pm 50\mu s$ error in the output pulsewidth times. The runtime and memory usage for the different runs are shown in Fig. 8. While modeling errors

requires additional resources, it provides a better coverage of the behavior of the system.

A typical run is shown in Fig. 9 which compares the coverage of model checking to one actual trajectory. Encoder inaccuracy causes a range of trajectories to be possible. The match of the line position in the actual trajectory is not perfect because the position is inferred indirectly from the brightness sensors. In future work we want to externally track the robot to improve the identified physical model and the line tracking accuracy. Speed, on the other hand, is within the bounds of the explored state space which corresponds to a good match.

5. CONCLUSIONS AND FUTURE WORK

We presented a method, based on model checking, for the verification of robotic control systems implemented in software. This technique leverages the recent advances in software model checking but extends its applicability to systems that include a non-trivial physical component described by differential equations. The approach is based on the construction of a simulation model for the dynamical system that is used in conjuction with the actual software to perform an exhaustive state exploration.

We have studied the effectiveness of this approach by applying it to a line-following robot controller. The model of the physical system has been derived by identification and the actual code running of the robot has been checked for both safety and liveness properties.

The presented approach proved sufficient to identify bugs that depend on the interaction with the physical environment and the scheduler. It is, however, critical to have a model of the environment in order to perform formal verification. We believe that such model should be constructed together with the control software. When designing a control system control engineers already use models based on differential equations: such models should be integrated in a way that they can later be used for verification.

As for future work, we would like to extend our method to be able to model more complex systems as well as check more complex properties. Moreover, we intend to formalize our approach in terms of hybrid automata. Our final goal is to provide an integrated verification framework for control software developers.

Acknowledgments

We would like to thank the Java PathFinder team at NASA Ames Research Center for making Java PathFinder available as open source software.

Sebastian Scherer would like to thank Sanjiv Singh for his continuing support.

REFERENCES

1. Lions, J. ARIANE 5 flight 501 failure. World Wide Web, July 1996. http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf.
2. Brat, G., Drusinsky, D., Giannakopoulou, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, C., Venet, A., Washington, R., and Visser, W. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in Systems Design Journal*, 25(2-3), September 2004.
3. Sharygina, N., Browne, J., Xie, F., Kurshan, R., and Levin, V. Lessons learned from model checking a NASA robot controller. *Formal Methods in Systems Design Journal*, 25(2-3):241–270, 2004.
4. Clarke, E., Grumberg, O., and Peled, D. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
5. Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.
6. Sharygina, N. *Model checking of software control systems.* PhD thesis, University of Texas at Austin, 2002.
7. Johnson, M. E. Model checking safety properties of servo-loop control systems. In *International Conference on Dependable Systems and Networks*, 2002.
8. Henzinger, T. The theory of hybrid automata. In *Proceedings of the 11th Annual Sysmposium on Locgic in Computer Science*, 1996.
9. Henzinger, T., Ho, P., and Wong-Toi, H. HyTech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
10. Diethers, K., Fireley, T., Kröger, T., and Thomas, U. A new framework for task oriented sensor based robot programming and verification. In *IEEE International Conference on Advanced Robotics*, pages 1208–1214, June 2003.
11. Ivancic, F. Report on verification of the MoBIES vehicle-vehicle automotive OEP problem. Technical Report MS-CIS-02-02, University of Pennsylvania, March 2002.
12. Lygeros, J., Godbole, D., and Sastry, S. A verified hybrid controller for automated vehicles. *Special Issue on Hybrid Systems of the IEEE Transactions on Automatic Control*, March 1996.
13. Bay, J. S. *Fundamentals of linear state space systems*. WCB/McGraw-Hill, 1999.
14. Puschner, P. and Koza, C. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(2):159–176, September 1989.
15. Søndergaard, H. Periodic threads on aJ-100]. World Wide Web, December 2004. http://it-ingenior.vitusbering.dk/sw12801.asp.
16. Ljung, L. *System identification: theory for the user*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.
17. Bernat, G., Burns, A., and Wellings, A. Portable worst case execution time analysis using Java byte code. In *Proceedings of the 12th EUROMICRO Conference on Real-time Systems*, June 2000.