Electronic Notes in Theoretical Computer Science 70 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 17 pages

Monitoring, Checking, and Steering of Real-Time Systems

Moonjoo Kim [a] Insup Lee [b] Usa Sammapun [b] Jangwoo Shin [b]
Oleg Sokolsky [b]

[a] *SECUi.com, Korea*

[b] *Department of Computer and Information Science, University of Pennsylvania, U.S.A.*

Abstract

The MaC system has been developed to provide assurance that a target program is running correctly with respect to formal requirements specification. This is achieved by monitoring and checking the execution of the target program at run-time. MaC bridges the gap between formal verification, which ensures the correctness of a design rather than an implementation, and testing, which only partially validates an implementation. One weakness of the MaC system is that it can detect property violations but cannot provide any feedback to the running system. To remedy this weakness, the MaC system has been extended with a feedback capability. The resulting system is called MaCS (Monitoring and Checking with Steering). The feedback component uses the information collected during monitoring and checking to *steer* the application back to a safe state after an error occurs. We present a case study where MaCS is used in a control system that keeps an inverted pendulum upright. MaCS detects faults in controllers and performs dynamic reconfiguration of the control system using steering.

1 Introduction

The monitoring and checking (MaC) framework [5] has been designed to ensure that the execution of a real-time system is consistent with its requirements at run-time. The purpose of run-time analysis based on formal requirements is to bridge the gap left by the two traditional analysis techniques: verification and testing. On the one hand, formal verification analyzes all possible executions of the system, but the analysis is performed on the specification of the system, not its implementation. Recent advances in model extraction from software and software model checking [1,3] have improved the situation somwhat. Still, state-of-the-art verification techniques do not scale up well to handle large systems. On the other hand, testing is applied to a system implementation

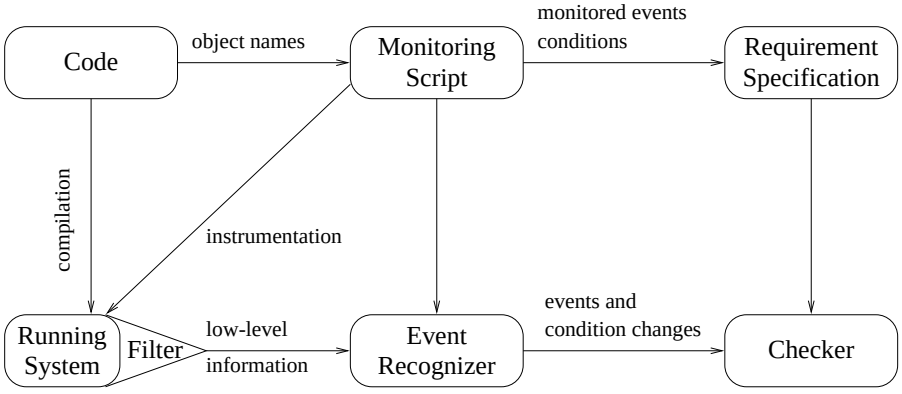

Fig. 1. MaC framework

and is routinely used on very large-scale systems, but does not guarantee that all behaviors of the implementation are explored and analyzed. In either case, run-time analysis can provide additional assurance that the current execution is correct with respect to its requirements.

Figure 1 shows the structure of the MaC framework. The user specifies the requirements of the system in a formal language. Requirements are expressed in terms of high-level events and conditions. In addition, a *monitoring script* relates these events and conditions with low-level data manipulated by the system at run-time. Based on the monitoring script, the system is *automatically* instrumented to deliver the monitored data to the *event recognizer* at run-time. The event recognizer, also generated from the monitoring script, transforms this low-level data into abstract events and delivers them to the run-time checker. The run-time checker verifies the sequence of abstract events with respect to the requirements specification and detects violations of the requirements.

The reason for keeping the monitoring script distinct from the requirements specification is to maintain a clean separation between the system itself, implemented in a certain way, and high-level system requirements, independent of a particular implementation. Implementation-dependent event recognition insulates the requirement checker from the low-level details of the system implementation. This separation also allows us to perform monitoring of heterogeneous distributed systems. A separate event recognizer may be supplied for each module in such system. Each event recognizer may process the low-level data in a different way, and all deliver high-level events to the checker in a uniform fashion.

Run-time monitoring provides for efficient detection of violations of system requirements and raise an alarm when a violation happens. At the same time, the vast information collected during monitoring in order to detect requirement violations allows us to go one step further. The same information can be used to diagnose the problem that has lead to the violation and suggest a remedy for it. In order to apply this remedy, we need the means to provide feedback from the run-time checker back into the system. Such a feedback

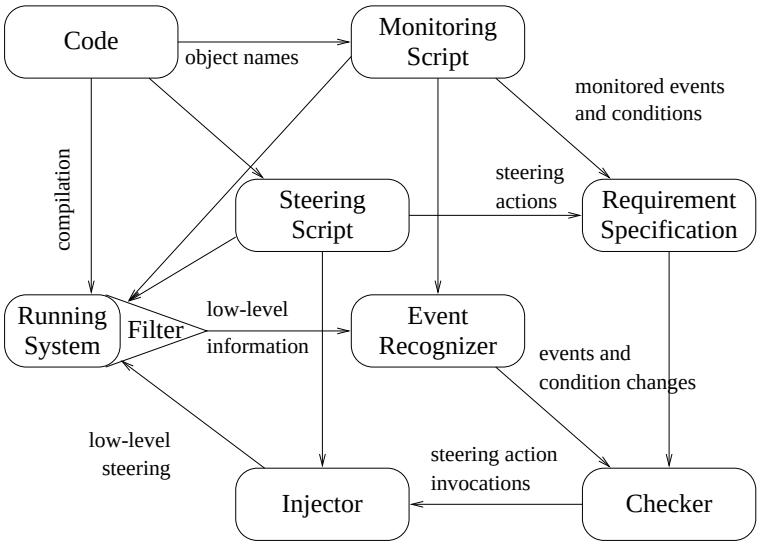

Fig. 2. The monitoring and steering framework

from the checker into the monitored system is called *steering*.

The extended framework, which we call MaCS, is shown in Figure 2. The checker has been given a capability to invoke steering actions that are delivered to the system and executed with the help of an *injector*. The injector is added to the system during the instrumentation process. The paper describes the MaCS framework and its implementation in the Java-based MaCS prototype, and presents a case study from the domain of automatic control.

The paper is organized as follows: we begin in Section 2 with the overview of the MaCS framework. In Section 2.2 we introduce the notion of steering and define a language for describing steering actions and their invocation in Section 2.3. Section 2.4 presents an implementation of steering in the MaCS prototype for monitoring, checking, and now, steering of Java programs. Section 3 describes a case study of steering-based dynamic reconfiguration of the control system for an inverted pendulum.

2 The MaCS Framework

Figure 1 shows the overall structure of the MaC architecture introduced in [5]. The architecture includes two main phases: *static phase* and *run-time phase*. During the static phase, i.e., before a target program is executed, run-time components such as a *filter*, an *event recognizer*, and a *run-time checker* are generated from a target program and a formal requirements specification. During the run-time phase, the instrumented target program is executed while being monitored and checked with respect to a requirements specification.

A *filter* is a collection of probes inserted into the target program. The essential functionality of a filter is to keep track of changes of monitored objects and send pertinent state information to the event recognizer. An *event recognizer* detects an event from the state information received from the fil-

ter. Events are recognized according to a low-level specification. Recognized events are sent to the run-time checker. Although it is conceivable to combine the event recognizer with the filter, we chose to separate them to provide flexibility in an implementation of the architecture. A *run-time checker* determines whether or not the current execution history satisfies a requirement specification. The execution history is captured from a sequence of events sent by the event recognizer.

2.1 The MaC Language

In keeping with this design philosophy, two languages have been designed for use in the MaC framework. The Meta-Event Definition Language (MEDL) is used to express requirements. It is based on an extension of a linear-time temporal logic. It allows to express a large subset of safety properties of systems, including real-time properties. Monitoring scripts are expressed in the Primitive Event Definition Language (PEDL). PEDL describes primitive high-level events and conditions in terms of system objects. PEDL is used to define what information is sent from the filter to the event recognizer, and how it is transformed into events used in high-level specification by the event recognizer. PEDL, therefore, is tied to the implementation language of the monitored system in the use of object names and types. MEDL is independent of the monitored system. High-level specifications are written in Meta Event Definition Language (MEDL). The separation between PEDL and MEDL ensures that the architecture is portable to different implementation languages and specification formalisms. Both of these languages are based on the notions of events and conditions.

Events and Conditions. *Events* occur instantaneously during the system execution, whereas *conditions* represent information that holds for a duration of time. For example, an event denoting the call to method `init` occurs at the instant the control is passed to the method, while a condition `(difference < 0.1)` holds as long as the value of the variable `difference` does not exceed `0.1`. The distinction between events and conditions is very important in terms of what the monitor can infer about the execution based on the information it gets from the filter. The monitor can conclude that an event does not occur at any moment except when it receives an update from the filter. By contrast, once the monitor receives a message from the filter that variable `difference` has been assigned the value `0.05`, we can conclude that `position` retains this value until the next update.

We use events and conditions to capture and reason about temporal behavior and data behavior of the target program execution; events are abstract representations of time and conditions are abstract representations of data. For formal semantics of events and conditions, see [5,6].

To illustrate MEDL and PEDL specification, we will describe their components as well as an example in Figure 3, which will check whether an elevator

stops at the height level with the floor.

```
ReqSpec evenLevelElevator  // MEDL script
  // imported events and conditions
  import event startElevator, currentFloor, heightAtStop;
  // auxiliary variable declarations
  var floorHeight;
  var desiredHeight;
  var difference;
  // definition of events and conditions
  condition even = difference < 0.1 && difference > -0.1;
  // safety properties and alarms
  alarm uneven = start( !even );
  // auxiliary variable updates
  startElevator ->
      {  floorHeight' = 4;  }
  currentFloor  ->
      {  desiredHeight' = value(currentFloor, 0) * floorHeight;  }
  heightAtStop  ->
      {  difference' = value(heightAtStop, 0) - desiredHeight;  }
end

MonScr evenLevelElevator  // PEDL script
  // exported events and conditions
  export event startElevator, currentFloor, heightAtStop;
  // declarations of monitored entities
  monobj  int Elevator.floor;
  monobj  int Elevator.stop().height;
  monmeth void Elevator.init(void);
  // definitions of events and conditions
  event startElevator = startM( Elevator.init(void) );
  event currentFloor = update( Elevator.floor );
  event heightAtStop  = update( Elevator.stop().height );
end
```

Fig. 3. MEDL and PEDL scripts for the elevator example

2.1.1 Meta Event Definition Language (MEDL)

The safety requirements (invariants) are written in MEDL. A MEDL specification includes the following sections:

Imported events and conditions. A list of events and conditions delivered by the event recognizer is declared. Definitions of imported events and conditions are given in a separate script discussed below. In Figure 3, the three imported events are `startElevator, currentFloor, heightAtStop`.

Definitions of events and conditions. Events and conditions are defined using imported events, imported conditions, and auxiliary variables, whose

role is explained later in this section. These events and conditions are then used to define safety properties and alarms. The condition `even` in Figure 3 uses the auxiliary variable called `difference` to define the even level of the elevator. Events can have tuples of values asociated with them. For example, an event that corresponds to the update of a variable has the single value that is the new value of the variable. An event for a method call contains values of all the parameters. Expression `value(currentFloor,0)` returns the first (in this case, the only) value associated with event `currentFloor`.

Safety properties and alarms. The correctness of the system is described in terms of safety properties and alarms. Safety properties are conditions that must be *always* true during the execution. Alarms, on the other hand, are events that must never be raised (all safety properties [7] can be described in this way). Note that alarms and safety properties are complementary ways of expressing the same thing. The reason that we have both of them is because some properties are easier to think of in terms of conditions, while others are in terms of alarms. Using the condition `even` in the elevator example from previous section, we define an alarm called `uneven`, which will alarm whenever the condition `even` becomes false.

Auxiliary variables. In order to increase expressive power of MEDL, we allow auxiliary variables to be used in MEDL specifications. Auxiliary variables can be used to define events and conditions, and their values are updated in response to events. Auxiliary variables allow us, for example, to count the number of occurrences of an event, or talk about the ith occurrence of an event. From the elevator example, the auxiliary variables `floorHeight, desiredHeight, difference` are used to keep the height of a floor, the desired stop height, and the difference between the desired stop height and the actual stop height, respectively. The `difference` variable, for example, is updated by the occurrences of the heightAtStop event. A special auxiliary variable `currentTime` is used to refer to the current time of the target program. It is set to be the last timestamp received from the filter.

When specifying how the variables are updated, we use primed and unprimed variables to refer to their new and, respectively, old value of the variable. This is necessary to ensure that the outcome of checking does not depend on the order of event evaluation. Consider the following artificial example.

```
e -> { x' = 5 }
event e1 = update(x);
event e2 = start( x == 5 );
e1 -> { x1' = x; }
e2 -> { x2' = x1'; }
```

An occurrence of event `e`, by assigning a new value to `x`, triggers both events `e1` and `e2`. By using `x1'` in the update of `x2`, we require that event `e1` be processed before `e2`. If, on the other hand, `x1` was used in

the update, the outcome would be the same for any evaluation order. The algorithm for evaluating MEDL formulas keeps the dependency graph for all events, conditions, and variable updates and evaluation proceeds according to the dependencies. Cyclical dependencies are detected and rejected by the algorithm. See [10] for more details.

2.1.2 Primitive Event Definition Language (PEDL)

PEDL is the language for writing low-level specifications that map run-time data of the system execution into high-level events and conditions. PEDL encapsulates all implementation-specific aspects of the monitoring process, and therefore, is by necessity specific to the target programming language. A PEDL specification consists of the following sections.

Exported events and conditions. Events and conditions to be sent from an event recognizer to a run-time checker are declared, and must match the imported section of the MEDL script. The exported events in the elevator example are `startElevator, currentFloor, heightAtStop`.

Declarations of monitored entities. This section is specific to the target programming language. In a typical programming language, monitored entities are variables and control locations in the program. The elevator example is designated for Java, and its monitored entities, therefore, are written closely to Java. The monitored objects are `Elevator.floor, Elevator.stop().height`, and the monitored method is `Elevator.init()`.

Definitions of events and conditions. This section uses monitored entities declared in the previous section to define events and conditions. Again, this is specific to the target programming language. Typical primitive events are updates of monitored variables and visits to monitored control locations, such as function calls. In order to make event recognition fast, we require that all events and conditions defined in a PEDL specification are in terms of a single state in the execution trace. That is, no history information needs to be stored by the event recognizer. In our elevator example, the event `startElevator` occurs when the `Elevator.init()` method starts. The event `currentFloor` and the event `heightAtStop` occur when the `Elevator.floor` and `Elevator.stop().height` are updated, respectively.

2.2 Steering

In addition to monitoring the system and detecting property violations, MaCS can provide feedback to the system based on the information collected during monitoring. This feedback can alter the execution of the system in an attempt to solve the detected problem or avoid a more serious failure. We call this feedback *steering* [9]. More generally, we can make the checker perform a complex computation and communicate the result back to the system to enhance the system's functionality.

```
steering script evenLevelElevator

   // steering entities
   steered objects
     boolean Elevator:doorLock;
     void Elevator:open();

   // steering actions
   steering action blockDoor =
     {  Elevator:doorLock = true;  } before call Elevator:open();

end
```

Fig. 4. steering script for the elevator example

Steering is accomplished by steering actions. Since the checker is executing separately from the system, steering consists of two phases. First, a steering action is *invoked* by the checker. The invoked action and its parameters are transferred to the system and the action is *executed* when the system is ready. There is by necessity a delay between an invocation and the corresponding execution of an action. Therefore, it is important that we can control when invocations as well as executions happen.

The structure of the MaC framework extended with steering is shown in Figure 2. The functionality of the runtime checker is extended with the ability to invoke *steering actions* and pass them to injector, which is incorporated into the running system during instrumentation. The injector accepts invocations of steering actions and translates them into low-level steering of the objects, i.e., calls to methods of system objects or assignments to system variables.

Steering is described in a *steering script*, discussed in the next session. The steering script compiler generates an injector from the definition of steering actions. The compiler also produces additional instrumentation directives that are applied to the system together with monitoring instrumentation. The user has the ability to control the moment when a steering action is executed in the target system. This is done by providing steering conditions with each action. Execution of a steering action is delayed until its condition is satisfied. Steering conditions are static, i.e., they are fully evaluated during instrumentation.

2.3 A Language for Steering Actions

To specify steering actions, we designed a special scripting language SADL (**S**teering **A**ction **D**efinition **L**anguage). Figure 4 shows the steering script from the elevator example.

The script consists of two main sections: declaration of steered objects (that is, system objects that are involved in steering) and definition of steering actions where the declared objects are used. Since steering is performed directly on the system objects, SADL scripts are by necessity dependent on

```
// *** imported actions ***
import action blockDoor();

// *** the invocation of the blockDoor steering action ***
uneven -> {  invoke blockDoor();  }
```

Fig. 5. Action invocation in the MEDL script

the implementation language of the target system.

Our prototype implementation of the monitoring and steering framework aims at systems implemented in Java. Because of this, SADL scripts used in the prototype are also tied to Java. The steered objects can be fields and methods of Java classes as well as local variables of methods. In the example, the steered object are the field `doorLock` and the method `open()` of the `Elevator` class. The first one is updated when the steering action is invoked, while the second one is used in the definition of the steering condition.

The second section of the steering script defines steering actions and specifies steering conditions. An action can have a set of parameters that are computed by the checker and passed to the system together with the action invocation. The body of an action is a collection of statements, each of which is either a call to a method of the system or an assignment to a system variable. In the example, the steering action assigns `true` to the `doorLock` field to lock the elevator door, and is allowed to happen before the `open()` method is called so that the door cannot be opened by the `open()` method.

In addition to a steering script, the requirement specification language, MEDL, is extended to provide for invocation of steering actions. An action is invoked in response to an occurrence of an event or an alarm.

The MEDL fragment shown in Figure 5 captures the extensions to the script of Figure 3 needed to prevent the elevator door from opening when the elevator stops in the wrong position. We declare the steering action `blockDoor`, imported from the steering script, which is then invoked in response to the alarm `uneven`.

2.4 Steering in the MaCS Prototype

A prototype implementation of the MaC framework has been implemented and tested on a number of examples. The prototype is targeted towards monitoring and checking of programs implemented in Java. Java has been chosen as the target implementation language because of the rich symbolic information that is contained in Java class files, the executable format of Java programs. This information allows us to perform the required instrumentation easily.

The PEDL language of the prototype allows the user to define primitive events in terms of the objects of a Java program: updates of program variables (fields of a class or local variables of a method) and method calls. Automatic

instrumentation guarantees that all relevant updates are detected and propagated to the event recognizer.

The prototype includes interpreters for PEDL and MEDL. The MEDL interpreter is the run-time checker. It accepts primitive events sent by the event recognizer and re-evaluates all events and conditions defined in the MEDL script. The PEDL interpreter is the event recognizer. It accepts the low-level data sent by the instrumented program and, based on the definitions in the monitoring script, detects occurrence of the primitive events and delivers them to the run-time checker. In addition, the PEDL interpreter produces the instrumentation data that is used to automatically instrument the system.

The MaC instrumentor is based on JTREK class library [4], which provides facilities to explore a Java class file and insert pieces of bytecode while preserving integrity of the class. During instrumentation, the instrumentor detects updates to monitored variables and calls to monitored methods and inserts code to send a message to the event recognizer. The message contains the name of the called method and its parameter values, or the name of the updated variable and its new value. Each message contains a time stamp that can be used in checking real-time properties.

In order to implement steering in the MaC prototype, we added the following components:

- A parser for SADL. The parser produces two components: 1) a list of actions together with their conditions in the form that can be used by the instrumentor; 2) a new class, `Injector`, discussed below.
- The injector is the component responsible for communication with the checker. When the system is started, the injector is loaded into the virtual machine of the monitored system. At run time, when a steering action is invoked, the injector receives a message from the checker and sets a flag to indicate that the steering action has been requested. The bodies of the steering actions are also represented in the prototype as methods of the `Injector` class.
- The functionality of the instrumentor is extended to insert the additional code at the positions prescribed by the steering conditions. The code tests the flag for action invocation and makes calls to the injector to execute the action. If multiple actions can be executed at the same location, their invocation flags are checked sequentially.
- The runtime checker is extended to handle action invocations.

3 Steering-based Dynamic Reconfiguration

In this section, we describe the case study that demonstrates the utility of the MaCS framework in dynamic flexible control systems. We used our MaCS tool to implement a fault-tolerance layer for an inverted pendulum controller. The architecture of the control system is based on the Simplex architecture

for control systems [8]. The goal of the Simplex architecture is to enable dynamic reconfiguration of the system in order to improve functionality and performance without sacrificing safety. The architecture assumes the existence of a safety controller with known properties. Users are allowed to introduce experimental controllers that may be more efficient than the safety controller, but pose a certain degree of risk of misbehavior. In that case, the system falls back to the safety controller.

The inverted pendulum (IP) system consists of a motor driven cart which is equipped with two quadrature encoders. One sensor measures the position (`track`) of the cart via a pinion which meshes with the track. The other sensor measures the angle (`angle`) of the pendulum attached to the cart. The pendulum has one degree of freedom, freely rotating around the horizontal axis that is perpendicular to the track. The purpose of the IP control system is to maintain the pendulum upright by activating an appropriate controller which transmits a correct voltage output (`volts`) to the motor. The architecture of the control system is shown in Figure 6.

The starting point for the case study was to notice that MaCS can be used to monitor the state of the system and provide the switching logic necessary to dynamically switch controllers using steering, if necessary.

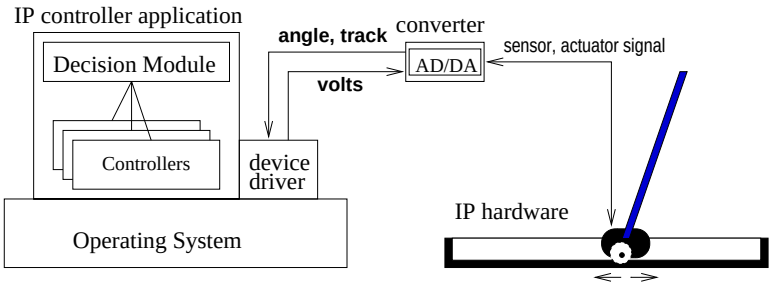

Fig. 6. Inverted Pendulum Control System

In order to use the Java-oriented MaCS prototype in the case study, we had to implement the control system in Java, which we will refer to as `IP-Java`. The class diagram is shown in Figure 7. Users can register experimental controllers at run-time using the `ECRegister` application. All controllers have to implement the `Controller` interface that has methods to set the input values and perform the computation of the control value. The `DMServerThread` class accepts the new controller requests via `KnockKnockProtocol` and replaces (or installs) the experimental controller in the decision module. The `DecisionModule` class is responsible for communicating with the inverted pendulum device. It first reads the current positioning information (`track` and `angle` values) from the device, then invokes the currently installed controller, and delivers the control value (`volts`) to the device. Since the device drivers were provided to us as C routines, we had to use Java Native Interface to access them. The `IP-Java` system also includes the safety controller that can be used as a fall-back if an experimental controller fails. However, the deci-

sion module by itself does not attempt to use the safety controller, relying on MaCS to initiate the switch at the right moment.

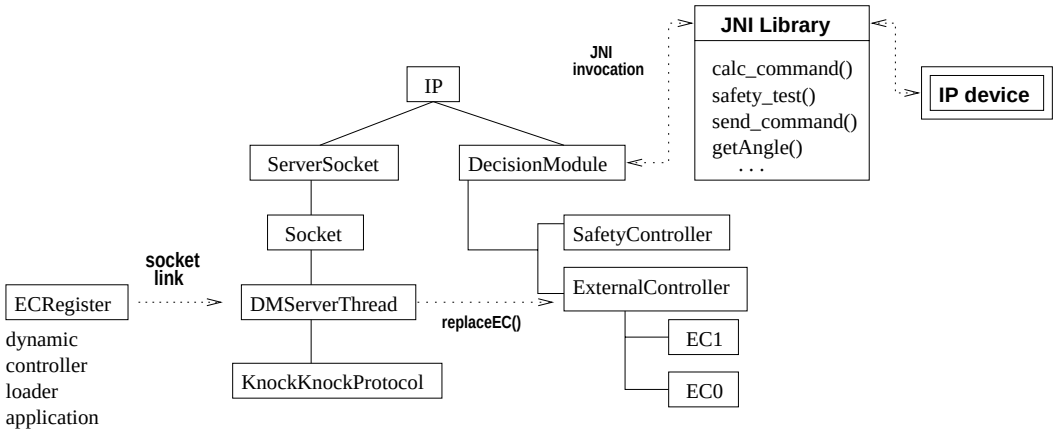

Fig. 7. `IP-Java` Schematic Diagram

Figure 8 presents the extended control system, `IP-MaCS` that uses the checker to monitor the state of the pendulum. Whenever new values are obtained from the device, the checker computes whether the state of the system is within the safety region and, if it is getting into the dangerous area, invokes the switch to the safety controller.

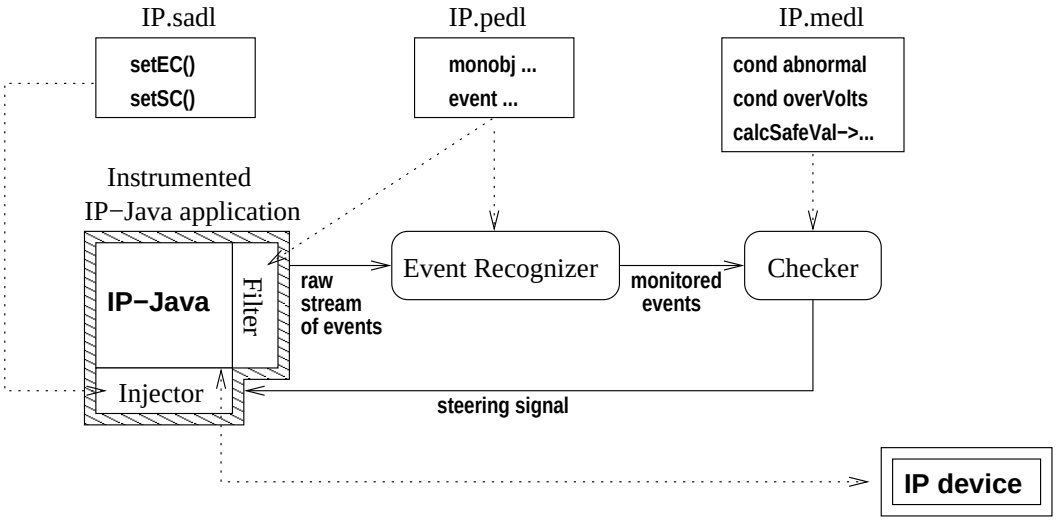

Fig. 8. `IP-MaCS` Schematic Diagram

The MaCS application is described in three scripts, `IP.pedl`, `IP.medl`, and `IP.sadl`, discussed below.

Information extraction. The `IP.pedl` exports four primitive events to be monitored: `startPgm`, `ev_current_angle`, `ev_track_pos`, and `ev_volts`. The first one is emitted once at the start of the program, while the other three correspond to the updates of the value exchanged between the control system and the device. Each of the values is represented as a variable in

the decision module: `DecisionModule.angle`, `DecisionModule.track` and `DecisionModule.volts`, respectively. Figure 9 shows the definition of the event `ev_current_angle`. Definitions of other primitive events are similar. The value associated with each primitive event is the updated values of the variable.

```
MonScr IP

   export event ev_current_angle;

   // angle is a monitored variable
   monobj float DecisionModule.angle;

   // send event whenever the monitored variable is updated
   event ev_current_angle = update(DecisionModule.angle);

end
```

Fig. 9. Excerpts from the `IP.pedl`

Checking safety conditions. The script `IP.medl` performs the computation of the safety conditions. The safety conditions are stated in terms of the control (`volts`) and the four state variables of the system: `angle` and `track` as well as their first derivatives. The values for the derivatives are not available directly and have to be estimated by the checker. Every time the value of `angle` or `track` is updated, the checker recomputes the value for the derivative based on the new difference of the new and old value of the monitored variable:

```
var float theta, thetadot;  // variables for state values
ev_current_angle -> {
     theta' = value(ev_current_angle, 0)/52.29578;
            // convert the sensed value to radians
     thetadot' = (theta' - theta) / 0.020;
            // 0.020 is the control period in milliseconds
     }
```

Whenever the primitive event `ev_volts` happens, the checker performs the safety computation. The result of the safety computation is two conditions: `outOfTrack` and `overVolts`. The first condition is defined as

```
condition outOfTrack = track_pos' > 40 || track_pos' < -40;
```

meaning that the cart is close to the end of the track. The other condition means that the value of `volts` is out of range for the given state of the system. The computation is a complex expression that we do not show here.

This script also contains sections needed by steering actions. Figure 10 presents a fragment of the MEDL script showing the declaration of the steering action `change2SC`, imported from the steering script, the definition of the

```
// imported declarations
import action change2SC();

// event definitions
event invokeSafeController =
        start(outOfTrack || overVolts) when (controller == 1);

// steering invocation
invokeSafeController -> { controller' = 0; invoke change2SC(); }
```

Fig. 10. Action invocation in the MEDL script

```
steering script IP

steered objects
// list objects to be steered and point of steering action

  DecisionModule IP:dm;
  // the target of steering is the object dm of type DecisionModule
  // located in the class IP

  float DecisionModule:volts;
  // anchor position for the injection of steering action

  steering action change2SC =
    { call (IP:dm).setSC(); } before read DecisionModule:volts;
    // setSC() method invoked
    // at the position right before the variable volts is read

end
```

Fig. 11. The steering script for the inverted pendulum

event `invokeSafeController` that is raised by the checker when it detects an abnormal situation, and the invocation of the steering action when the event `invokeSafeController` is raised. As soon as either `outOfTrack` or `overVolts` condition becomes true, the checker invokes the steering condition `change2SC`.

Steering actions. The script in Figure 11 shows two steered objects and one steering action. These objects are the decision module object, referenced as the field `dm` of the main class `IP`, and the field `volts` of the `DecisionModule` class. The first one contains the methods that are called when the steering action is invoked, while the second one is used in the definition of the steering condition.

The steering action `change2SC` calls a method that replaces the current controller with the safety controller, and is allowed to happen whenever the variable `volts` in the decision module is about to be read.

Experiments. We have performed a number of experiments with the system, with and without MaCS. The goal of the experiments was to prove that steering can be used in dynamically reconfigurable real-time control systems. Perhaps more importantly, we wanted to use the experiments to identify the bottlenecks in our MaCS implementation.

We first ran the system with just the safety controller to make sure that MaCS does not interfere with the the operation of the controller. Visually, the pendulum successfully maintained in the upright position. We observed more jitter in the cart position in the steady state compared to the original Simplex implementation of the inverted pendulum control system, but that had more to do with the Java re-implementation of the system than with the presence of MaCS. Instrumentation added for MaCS did not increase jitter. On the other hand, detailed measurements showed that instrumentation introduced by MaCS has caused a significant slowdown of the system. Without instrumentation, the system running the safety controller took, on the average 27.7 microseconds per control cycle, with very little standard deviation. After instrumentation has been performed, the execution time rose to 176.7 microseconds on the average, and the standard deviation rose significantly from 1.44 to 7.67. However, compared to the length of the control cycle, which is 20 milliseconds, the increase is negligible.

We then introduced a faulty experimental controller into the system. The experimental controller would produce the same outputs as the safety controller for a fixed period of time and then "get stuck," sending a constant random value to the motor. In all experiments, the MaCS checker successfully detected the fault and invoked the switch to the safety controller in time for it to keep the pendulum from falling. The constant value sent to the motor by the faulty controller causes the cart to start moving in one direction, however in all experiments the cart stayed on track. To experiment with longer executions, we added a second steering action, `switch2EC`, which was invoked within a fixed period of time after `switch2SC`. This action restored the experimental controller, giving us a sequence of switches between the two controllers. The pendulum was successfully stabilized after each fault in all executions.

This MaCS-based controller switching operates in such a way that faulty values are delivered to the motor until the checker detects a problem and the switch takes effect. The "round trip" time of the monitoring/checking/steering cycle turned out to be 320 milliseconds on the average, which means that about 16 faulty values have been delivered to the motor. The standard deviation of the round trip time was almost 84000.

The large delay is attributed to the communication between the filter and the event recognizer. The filter has been optimized for large volumes of monitored data. It uses a buffered stream to send data to the event recognizer. When there is much data to send, buffered communication offers significant savings over the non-buffered communication, because of the additional cost

associated with each transmission. However, for the inverted pendulum case study it does not make sense, since the amount of monitored data transmitted in one control cycle is less than the buffer size. The obvious solution to this problem is to let the user decide whether to use buffering and specify the buffer size. This feature will be incorporated into the next MaCS release. Preliminary results show that, without buffering, steering can be performed withing one control cycle.

4 Conclusions and Future Work

We have described an approach to perform run-time correction of system behavior by supporting the notion of steering actions. Steering is embedded into a monitoring and checking framework, which detects violations of formal requirements in the observed execution of the monitored system. Monitoring in conjunction with steering can be used to provide an additional layer of fault tolerance in the system.

The case study described in this paper demonstrates that it is possible to use checking and steering features of the MaCS framework in dynamic control applications. The advantages of using MaCS is the increased flexibility of implementing control algorithms and the ability to completely separate design of controllers for a particular control problems and switching logic for dynamically replacing controllers.

Synchronous steering. Results of the case study encouraged us to explore an alternative approach to steering, which we call the *synchronous* steering. In the current implementation, the checker is sent the computed data for correctness checking asynchronously, and the system is allowed to proceed with its computation while the checking is performed. In this case, incorrectly computed values may be delivered by the system to its environment before a steering action takes effect to correct the problem. While this has not been a problem in the inverted pendulum case study, in general, it may not be acceptable. Alternatively, we can pause the system while checking is performed, and let it proceed with its computation only if the checker confirms correctness. In this way, we can guarantee that incorrect values can never be output. We note that a similar notion has been suggested as the "before-store" placement of events in [2].

We have originally settled on the asynchronous scheme since our initial estimates of MaCS performance suggested that steering delays will be unacceptably long in the steering scheme. However, recent aggressive improvements in the MaCS implementation have made it significantly more efficient. At the same time the inverted pendulum case study shows that for some systems with relatively slow dynamics the synchronous steering may be a viable alternative, which would deliver more stringent guarantees of the system correctness. We are currently extending to support the notion of synchronous steering.

Acknowledgements.

We thank Prof. Lui Sha and Kihwal Lee at the University of Illinois at Urbana-Champaign for many fruitful discussions on the Simplex architecture and for help in the experimental setup of the inverted pendulum case study. This research was supported in part by ONR N00014-97-1-0505, NSF CCR-9988409, NSF CCR-0086147, NSF CISE-9703220, and ARO DAAD19-01-1-0473 grants. The paper is dedicated to late Anirban Majumdar.

References

[1] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[2] A. Gates, S. Roach, O. Mondragon, and N. Delgado. DynaMICs: Comprehensive support for run-time monitoring. In *Proceedings of the First Workshop on Run-Time Verification*, July 2001.

[3] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, Apr. 2000.

[4] Java Technology Center, Compaq Corp. *Compaq JTrek* . Online documentation: http://www.digital.com/java/download/jtrek/.

[5] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Proceedings of the European Conference on Real-Time Systems - ECRTS'99*, pages 114–121, June 1999.

[6] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[7] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[8] L. Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, July/August 2001.

[9] O. Sokolsky, S. Kannan, M. Kim, I. Lee, and M. Viswanathan. Steering of real-time systems based on monitoring and checking. In *Proceedings of WORDS'99F, Fifth International Workshop on Object-Oriented Real-time Dependable Systems*, Nov. 1999.

[10] M. Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, CIS Dept. Univ. of Pennsylvania, 2000.