9-2016

SMEDL: Combining Synchronous and Asynchronous Monitoring

Teng Zhang
University of Pennsylvania, tengz@cis.upenn.edu

Peter Gebhard
University of Pennsylvania, pgeb@cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

The 16th Conference on Runtime Verification (RV 2016), pp. 482--490. Madrid, Spain, September 2016.

SMEDL: Combining Synchronous and Asynchronous Monitoring

Abstract

Two major approaches have emerged in runtime verification, based on synchronous and asynchronous monitoring. Each approach has its advantages and disadvantages and is applicable in different situations. In this paper, we explore a hybrid approach, where low-level properties are checked synchronously, while higher-level ones are checked asynchronously. We present a tool for constructing and deploying monitors based on an architecture specification. Monitor logic and patterns of communication between monitors are specified in a language SMEDL. The language and the tool are illustrated using a case study of a robotic simulator.

Keywords

monitor generation, synchronous monitoring, asynchronous monitoring

Disciplines

Computer Engineering | Computer Sciences

Comments

The 16th Conference on Runtime Verification (RV 2016), pp. 482--490. Madrid, Spain, September 2016.

SMEDL: Combining Synchronous and Asynchronous Monitoring

Teng Zhang[1], Peter Gebhard[1], and Oleg Sokolsky[1]

University of Pennsylvania, Philadelphia PA 19104, USA,
{tengz,pgeb,sokolsky}@cis.upenn.edu

Abstract. Two major approaches have emerged in runtime verification, based on synchronous and asynchronous monitoring. Each approach has its advantages and disadvantages and is applicable in different situations. In this paper, we explore a hybrid approach, where low-level properties are checked synchronously, while higher-level ones are checked asynchronously. We present a tool for constructing and deploying monitors based on an architecture specification. Monitor logic and patterns of communication between monitors are specified in a language SMEDL. The language and the tool are illustrated using a case study of a robotic simulator.

Keywords: Monitor generation, Synchronous monitoring, Asynchronous monitoring

1 Introduction

Runtime verification(RV) [1] has emerged as a powerful technique for correctness monitoring of critical systems. Numbers of approaches have been proposed among which *synchronous* monitoring [2] and *asynchronous* monitoring [3] are broadly used. Synchronous monitoring will block the execution of the system being monitored until validity of an observation is confirmed, ensuring that potentially hazardous behavior is not propagated to the system environment. This makes this method suitable for safety- and security-related contexts. However, synchronous monitoring incurs high execution overhead for the target system, and less critical properties may not require such strict guarantees. On the other hand, asynchronous monitoring may allow to check the properties with less overhead for the target system, but as the system continues its execution while checking is performed, it may not be suitable for some critical properties. Moreover, the error point is hard to locate: when the monitor reports the violation of the property, the target system may have already left the position causing the problem. Most RV tools target one of these two approaches. Furthermore, synchronous monitoring may not be suitable for distributed systems. In many practical cases, it is desirable to combine the two approaches to get the benefits of both and reduce effects of drawbacks.

The contribution of this paper is a tool for construction and deployment of hybrid monitoring. The tool uses the language SMEDL to specify monitoring

architecture and individual monitors. Properties to be checked are represented in a state-machine style in monitors. Generated monitors can be integrated with the target system or deployed separately, according to the architecture specification. Execution within a monitor is synchronous while the communication among monitors is asynchronous. This allows us to monitor properties on multiple time scales and levels of criticality.

Related work. MaC(Monitoring and Checking) [4] is an architecture for asynchronous runtime monitoring. A distributed version of MaC, DMaC [5], is proposed mainly for monitoring the properties of network protocols. MOP (Monitoring Oriented Programming) [6] is a generic framework for properties specification and the checking of the properties at runtime. Based on the work of MOP, RV-Monitor [7] can monitor hundreds of properties of Java API specifications at the same time. Using the concepts of AOP [8] and MOP, MOVEC [9] is compiler supporting the parametric runtime verification for systems written in C. [10] proposes an architecture allowing for switching between synchronous and asynchronous monitoring but it is not clear how to use synchronous and asynchronous monitoring simultaneously. [11] proposed PT-DTL, a temporal logical to describe the temporal properties of distributed system, and a decentralized monitoring algorithm for PT-DTL. However, the proposed tool, DIANA, only supports the asynchronous monitoring for distributed program with a fixed architecture. [12] presents a method for monitoring multi-threaded component-based systems described in BIP but it is not suitable for distributed systems. [13] proposes a primitive condition rule-based system RuleR which supports the hierarchy architecture of monitors but asynchronous monitoring is not supported. Thus, despite the variety of available tools, there is presently no support for combining synchronous and asynchronous monitoring.

The paper is organized as follows. Section 2 gives a overview of SMEDL. Section 3 introduces the implementation of the SMEDL tool. Section 4 uses the case study of simple robot simulator to evaluate the performance of the tool. Section 5 concludes the paper and presents the future work.

2 Overview of SMEDL

2.1 SMEDL concepts

A SMEDL monitoring system can be divided as four parts: target system, monitoring specification, SMEDL code generator and runtime checkers, as illustrated in Fig. 1. Target system is the system to be monitored and checked. The SMEDL specification contains a set of monitoring objects and an architecture that captures patterns of communication between them. A monitoring object can be an abstraction of a system object or an abstract entity that represents interactions between multiple system objects. Objects can include a set of parameters whose values are fixed when the object is instantiated. Internal state can be maintained in an object to reflect the history of its evolution. SMEDL events are instantaneous occurrences that ultimately originate from observations of the execution of target system. Events can also be raised by monitors in response to other

events. Raised events can be delivered to other monitors for checking or serve as alarms. A SMEDL specification is independent from the system implementation so that the monitoring specification does not need to be changed as long as the specification remains the same, even if the implementation has been changed. Instead, the definition of events in terms of observations on the target system is modified. Each monitoring object is converted to executable code by the SMEDL code generator and can be instantiated multiple times with different parameters, either statically during the target system startup or dynamically at run time, in response to system events such as the creation of a new thread in the target system.

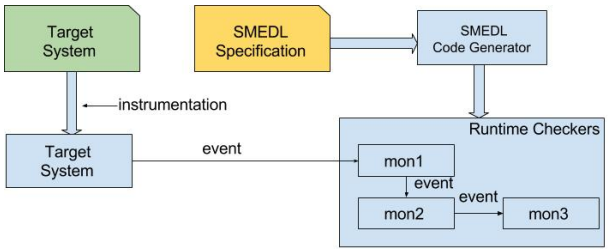

Fig. 1. SMEDL overview

2.2 Brief description of the language

The SMEDL specification contains two parts: a definition for each monitor object and a description of the monitor network that specifies monitor instances and connections between them.

Monitoring objects. A SMEDL monitoring object is a collection of extended finite state machines (EFSMs) sharing a set of internal state variables and events. More precisely, a monitoring object is a tuple $\langle interface, implementation \rangle$, where $interface$ contains the name, unchangeable identity parameters and event declarations; *implementation* contains state variables and state machines of the monitor. In SMEDL syntax, illustrated in the case study, state machines are represented as *scenarios*. Three kinds of events, *imported*, *exported* and *internal*, can be specified in the event declaration. *Imported* events of a monitor can be received and used to trigger the execution of the monitor; *exported* events are raised in the monitor and are sent to other monitors; *internal* events are processed within the monitor instance. State machines are used to define the behavior of the monitor. Transitions of the state machines are labelled with events that trigger the transition. In addition to an event, each transition may also be labeled by a guard, which is a predicate over state variables and event attributes, and a set of actions, each action is either an assignment to a local variable, or a statement that raises an event. Semantics for single monitors determines *macro-steps*, that is, synchronous compositions of individual transitions

of state machines (referred to as *micro-steps*) in response to an imported event from delivered from the environment. After finishing all enabled transitions, the monitor will output the exported events raised during the macro-step and wait for the next imported environment. Formal description of the semantics can be found in [14].

Architecture. A monitor network is a directed tree $G = \langle V, E \rangle$ where V contains the target system and a set of monitor instances which can receive or raise events; E is a set of directed edges connecting event ports from the target system to the monitors or between monitors. Monitors may receive events either from the target system or from other monitors. Monitors directly connecting with the target system will execute synchronously with the target system, while all others have their own execution threads and can be deployed locally or over a network. Events are delivered to monitor instances based on the values of instance parameters or event attributes. Thus, an architecture description language is provided for specifying event connection patterns between monitor instances. Apart from the source and target monitoring objects and events, connection patterns also specify matching rules between source and target monitor instances according to instance parameters or attributes of the event. [15] gives a detailed description of the language.

3 Tool Implementation

We have developed a toolchain for deploying monitors based on SMEDL specifications, shown in Fig. 2. The tool contains two parts: *monitor generator* and *configurator*. The monitor generator generates the code for a single monitor object, while the configurator is responsible for integrating monitor instances and target program, based on the SMEDL architecture specification.

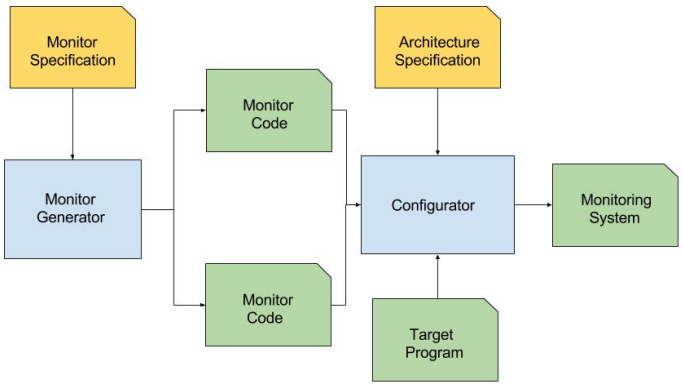

Fig. 2. SMEDL Toolchain

The monitor generator produces C code for the monitor object. The monitor API consists of a set of function calls corresponding to imported events of the monitor object. Calls to event functions trigger execution of the monitor state machines. To support the asynchronous communication between monitors, we use the publish-subscribe mechanism of the RabbitMQ middleware [16]. Each event in the architecture specification is represented as a topic. Events raised by a monitor are published to a topic according to the architecture specification. Topic names include information about names of parameters of raising monitors and event attributes. We rely on filtering provided in RabbitMQ subscriptions: monitor instances subscribing to an event can specify values of parameters and attributes that are relevant for them, according to the architecture specification.

We have developed a prototype of SMEDL toolchain which can generate the C code of single monitors. The toolchain and the case study used in this paper is available for downloading.[1]

4 Explorer: a Case Study

Explorer is a multi-threaded program for simulating robots locating and retrieving targets on a two-dimensional map. Each robot, running in its own thread, will start in a specified position on the map and has to retrieve a number of targets in a limited number of moves across the map. The goal of monitoring for this program is twofold. First, we want to check that each robot is following the search-and-retrieve protocol and, second, we collect statistics about the number of moves needed to retrieve the target. We thus define two monitor objects: one checks behavior of each robot thread and another is a statistic monitor that collects events from all behavior checking monitors. The behavior checking monitor is deployed synchronously with each new thread, while the statistic monitor is asynchronous.

Monitor specification. $ExplorerMon$, defined in Listing 1.1, directly connects to each robot for synchronous checking. The monitor has three *scenarios* $Main$, $Explore$ and $Count$. $Main$ is used to check whether robot has found the target in its view and begun to retrieve it. $Explore$ is used to describe the behavior of robots. $Count$ is used to count the number of moves of robots. There are four imported events, *view*, *drive*, *turn* and *count*. Event *view* will be sent to the monitor whenever the view of the robot has been updated. If the target is in the robot's view, the monitor will raise the internal event $found$ indicating that the robot has found the target. Event *turn* is used to check the current heading direction of the robot and update the state variable *mon_heading* accordingly. Event *drive* is triggered whenever the robot is trying to move. If the helper function *check_retrieved* returns true, the exported event *retrieved* will be raised carrying the number of moves that the robot has taken to retrieve this target. Event *count* is used to count the number of robots having taken so far. Every time *count* is triggered, the state variable *move_count* is increased by 1. Once a target has been retrieved, *move_count* will be reset.

[1] https://gitlab.precise.seas.upenn.edu/tengz/SMEDLTool

Listing 1.1. SMEDL specification for ExplorerMon

```
object ExplorerMon
identity
  opaque id;
state
  int mon_x, mon_y, mon_heading,move_count;
events
 imported view(pointer),drive(int,int,int),turn(int),count();
 internal found();
 exported retrieved(int);
scenarios
  Main:
    Explore -> found() -> Retrieve
    Retrieve -> retrieved(cnt) -> Explore
  Explore:
   Look->view(view_pointer) when contains_object(view_pointer){raise found();}->Move
                else -> Move
   Move -> turn(facing) when facing != heading{mon_heading = facing;} -> Look
      else -> Move
   Move -> drive(x, y, heading, map) when check_retrieved(map,x,y)
    { raise retrieved(move_count); mon_x = x; mon_y = y; move_count = 0; } -> Look
      else {mon_x = x;mon_y = y;} -> Look
 Count:
      Start -> count(){move_count=move_count+1;}->Start
}
```

To check if all robots retrieve all targets in the map and calculate the average number of moves the robots have used, asynchronous monitor *ExplorerStat* is defined in Listing 1.2. In the system, there is only one instance of *ExplorerStat* which will receive events *retrieved* from instances of *ExplorerMon*. Whenever *retrieved* is delivered into the monitor, state variable *sum* will be increased by 1 and the number of moves will also be added to the variable *count*. If the sum is equal to the overall number of targets, the exported event *output* will be raised with the average number of moves of a robot as an attribute.

Listing 1.2. SMEDL specification for ExplorerStat

```
object ExplorerStat
state
    int sum,count,targetNum;
events
    imported retrieved(int);
    internal reachNum();
    exported output(float);
scenarios
    stat:
        Start -> retrieved(move_count){sum=sum+1;count=count+move_count
                                                    ;raise reachNum();}-> Start
    check:
        CheckSum -> reachNum() when (sum < targetNum) -> CheckSum
                  else{raise output(count/sum);sum=0;count=0;}->CheckSum
```

Fig. 3, a, shows the corresponding runtime architecture. The two monitoring objects communicate via a single event *retrieved*. Fig. 3, b, shows a runtime view of the architecture, where $ex\_mon\_1, \ldots, ex\_mon\_k$ are instances of *ExplorerMon* associated with threads $robot\_1, ldots, robot\_k$ simulating k robots. The single instance of *ExplorerMon* receive events from all instances ex_mon_i. The implementation introduces an additional thread *sender* that publishes events from all instances to the broker of RabbitMQ.

Performance evaluation. The experiment is done on a single core virtual machine with CPU of speed 2.5GHz and memory of 4GB. The operating system is Ubuntu 14.04LTS and RabbitMQ is used as the communication middleware API. Overhead is one of the most important measurements that can show the performance of the monitoring system. There are three sources of overhead for synchronous monitoring: instantiation of monitors, checking of observations, and communication. Communication overhead is incurred only when asynchronous

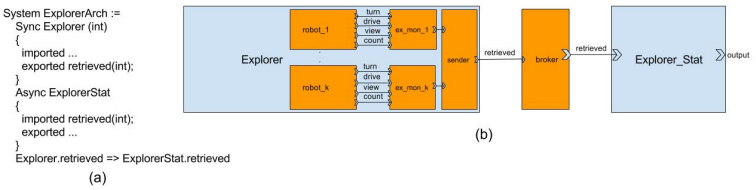

Fig. 3. Monitor Network of Explorer

monitors are present and events need to be sent to the asynchronous monitors via the middleware. The overhead of publishing events depends on the choice of middleware. Note that the initialization of connection in the main thread is incorporated into the communication overhead. Checking overhead increases with the number of observations produced by the target system. In our case study, the number of observations depend on two tunable factors: the number of robot threads in the system and the size of the map. We expect overhead to increase linearly with the number of threads, since the number of observations from each thread is independent of others. Increasing the size of the map tends to reduce overhead, since robots tend to move straight over longer distances on a larger map, without generating observations. This reduces the frequency of events, on average. In this experiment, the size of map is fixed to 40×80 and there are 5 targets in the map.

We describe two experiments that consider these factors separately. The first experiment varies the number of threads, with the size of map is fixed to 40×80 and 5 targets on the map. Fig. 4 shows that the overhead is approximately linear with the number of threads. The overall relative overhead of monitor instantiation and synchronous checking is about 2%, while communication overhead is approximately 3%, respectively. In absolute terms, processing of an average event with and without communication overhead is 2.11μs vs. 1.93 μs.

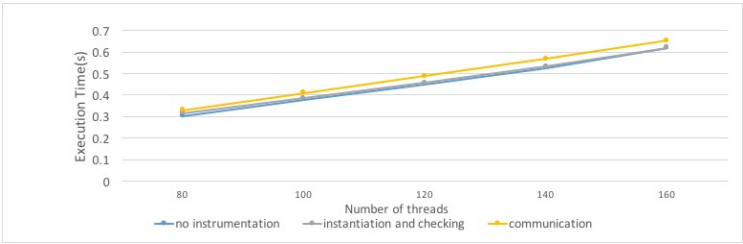

Fig. 4. Execution time vs. number of threads

The second experiment considers the overhead as a function of the map size. Table 1 shows that overhead quickly becomes negligible with the size of map

increasing. However, communication overhead remains about twice as high as checking overhead.

Table 1. Relation between input size and overhead

input size #	avg moves	checking	communication
30×60	43306	3.2%	6%
40×80	75205	1.9%	3.3%
50×100	112943	$< 1\%$	2%
60×120	161918	$< 1\%$	2%

We discuss results of the case study in the next section.

5 Discussion and Conclusions

We presented a tool to support generation and deployment of hybrid, i.e., synchronous and asynchronous, monitors specified in the language SMEDL. The SMEDL specification describes a network of monitors. Within the single monitor, the execution is synchronous while the communication between monitors is asynchronous. A prototype of the tool has been implemented. The paper describes evaluation of the tool using the case study of a robot simulator.

We first discuss some of our design decisions. We implement asynchronous communication using middleware, which allows us to exchange events across the network. This restricts synchronous monitoring to a single computing node. It is possible that some security-critical applications may require synchronous monitoring of multiple nodes. However, in our experience, such configurations are subject to high overhead and should be avoided when possible. In our tool, extension to synchronous monitoring over a network would be a simple extension to consider in the future. We assume that each monitor object in the architecture is deployed either synchronously or asynchronously. That is, either all imported events are supplied by the target system, or all are supplied by other monitors through the middleware. Potentially, deployments could be mixed, however implementation of the monitor becomes substantially more complicated.

From the case study, we note the choice balance between synchronous and asynchronous monitors in an architecture is not straightforward. The most surprising lesson from the case study, for us, was that the overhead of sending an event to a separate monitor can be larger than checking the event synchronously within the same monitor. Thus, intuitively, delegating checking to an asynchronous monitor makes sense only if it involves complex computation.

The presented toolset remains in active development. We are working on automatic instrumentation of C code, using an approach similar to [9]. We are also improving automatic deployment of asynchronous monitors, as well as reducing both checking and communication overheads.

References

1. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming **78**(5) (2009) 293–303
2. Francalanza, A.: A theory of monitors. In: Foundations of Software Science and Computation Structures, Springer (2016) 145–161
3. Francalanza, A., Gauci, A., Pace, G.J.: Distributed system contract monitoring. J. Log. Algebr. Program. **82**(5-7) (2013) 186–215
4. Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan, M.: Java-MaC: a run-time assurance approach for Java programs. Formal Methods in Systems Design **24**(2) (March 2004) 129–155
5. Zhou, W., Sokolsky, O., Loo, B.T., Lee, I.: DMaC: Distributed monitoring and checking. In: Runtime Verification, Springer (2009) 184–201
6. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. International Journal on Software Tools for Technology Transfer **14**(3) (2012) 249–289
7. Luo, Q., Zhang, Y., Lee, C., Jin, D., Meredith, P.O., Şerbănuţă, T.F., Roşu, G.: RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In: Runtime Verification, Springer (2014) 285–300
8. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP'97Object-oriented programming. Springer (1997) 220–242
9. Chen, Z., Wang, Z., Zhu, Y., Xi, H., Yang, Z.: Parametric runtime verification of C programs. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2016) 299–315
10. Colombo, C., Pace, G.J., Abela, P.: Compensation-aware runtime monitoring. In: Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. (2010) 214–228
11. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society (2004) 418–427
12. Nazarpour, H., Falcone, Y., Bensalem, S., Bozga, M., Combaz, J.: Monitoring multi-threaded component-based systems. Technical Report TR-2015-5, Verimag Research Report (2015)
13. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from Eagle to RuleR. Journal of Logic and Computation **20**(3) (2010) 675–706
14. Zhang, T., Gebhard, P., Sokolsky, O.: Semantics of SMEDL monitor objects. Technical Report MS-CIS-16-02, University of Pennsylvania (2016) `http://repository.upenn.edu/cis_reports/`.
15. Zhang, T., Gebhard, P., Sokolsky, O.: Architecture description language for SMEDL. Technical Report MS-CIS-16-06, University of Pennsylvania (2016) `http://repository.upenn.edu/cis_reports/`.
16. Videla, A., Williams, J.J.: RabbitMQ in action. Manning (2012)