Flexible Monitor Deployment for Runtime Verification of Large Scale Software

Teng Zhang[1], Gregory Eakman[2], Insup Lee[1], and Oleg Sokolsky[1](✉)

[1] University of Pennsylvania, Philadelphia, PA 19104, USA
{tengz,lee,sokolsky}@cis.upenn.edu
[2] BAE Systems, Burlington, MA 01803, USA
gregory.eakman@baesystems.com

Abstract. The paper presents a brief overview of the SMEDL monitoring system that provides flexible and scalable deployment of monitors for large-scale software. The SMEDL specification language expresses monitoring logic as a collection of monitoring objects and monitoring architecture as flows of information between the monitored system and monitoring objects. The system supports synchronous as well as asynchronous deployment of monitoring objects and dynamic instantiation of monitoring objects on demand. The application of the SMEDL system for the monitoring of a target tracking application is briefly discussed.

1 Introduction

Modern software systems affect all aspects of our lives, offering ever richer capabilities. This outsized role comes at a price: software keeps increasing in scale and complexity, requiring ever more effort to design, build, test, and deploy. Hardly any large-scale systems are designed from scratch today. Systems are integrated from separately developed modules, both vertically and horizontally. Concurrency and distributed computation are extensively used in the integration of modules. Modules are often developed by independent teams and incorporated as black boxes into the larger system. Moreover, over the life of the system, individual modules will be updated, so systems assembled and deployed in different time frames are likely to use different versions of the module. All of these factors allow incompatibilities between modules to slip in, in the form of communication protocol errors, broken assumptions made by developers of individual modules, etc. As a result, flaws in a software system are often discovered after the system is built and deployed.

Runtime monitoring can be used to detect and diagnose these flaws and alert system users and developers. In this paper, we will consider specification-based monitoring, where monitors detect deviation of system behaviors from the

This work is supported in part by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under contract FA8750-16-C-0007 and by ONR SBIR contract N00014-15-C-0126.

© Springer Nature Switzerland AG 2018
T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11247, pp. 42–50, 2018.
https://doi.org/10.1007/978-3-030-03427-6_6

specifications and raise alarms (or, potentially, trigger recovery). This kind of monitoring came to be known as *runtime verification* [3]. To be useful in monitoring of large-scale software systems, runtime verification needs to be supported by a flexible monitor deployment framework. The framework should allow us to specify our requirements, determine how each requirement should be monitored, which observations are needed by monitors to perform their job and how these observations should be extracted. In cases where dynamic instantiation of monitors or communication between monitors are needed, the framework should also allow us to specify when and how monitors need to be instantiated and removed, and what communication flows should be present.

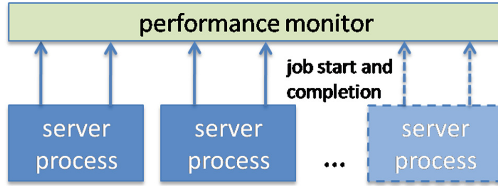

Fig. 1. Running example

Throughout the paper, we will be using a simple example of performance monitoring. Consider the setting illustrated in Fig. 1. We have a system built as a set of distributed server processes, each completing a series of jobs. A shaded process indicates that server processes can be added and shut down dynamically to satisfy demand. The monitor should calculate an average performance metric for the whole system. Such a monitor would receive timestamped observations from each process, corresponding to start and completion of each job, status of a job at completion, etc., and output alerts if performance, according to the chosen metric, falls below a threshold.

The paper is organized as follows. In Sect. 2, we give a brief introduction to runtime verification and discuss challenges of applying runtime verification techniques to large-scale software systems. Section 3 we introduce a flexible monitoring framework and discuss how it addresses the challenges. We conclude the paper with a discussion of remaining challenges and future work.

2 Background: Runtime Verification

Runtime verification is a collection of techniques for correctness monitoring of systems with respect to formally specified properties. An executable monitor is constructed for a given property and is run over a stream of observations to arrive at a conclusion, whether the property is satisfied or not. Runtime verification approaches differ in how properties are specified; how monitors are constructed; how observations are extracted; whether monitoring is performed online or over a recorded trace; if the monitoring is online, how the monitors are deployed within the running system.

2.1 Challenges

Application of runtime verification to large-scale software systems faces a number of challenges that stem from many of the same factors that make large-scale software hard in the first place. Below, we consider several of these challenges.

Multiple Properties with Different Criticality Levels. A large software system would have many different properties to monitor. Some of these properties relate to safety and security of the system and require fast response. Evaluation of others, for example performance properties, may be delayed or even performed off line. Monitors for these properties may rely on the same observations. And there can be dependencies between properties; for example, a security monitor may perform anomaly detection on a performance metric. Keeping track of properties along with their dependencies, and ensuring that deployment of monitors is appropriate for their criticality levels and preserves dependencies, is a challenge that quickly increases with the scale of the system.

Global vs. Local Properties. Some of the properties to monitor may be local to one module in the system, while others concern global behaviors of the system. In a distributed system, checking global properties may incur prohibitive overhead and interfere with the system operation. Local monitoring is usually preferable. Deciding where monitors should be placed and managing monitor placement in a large system is a challenge.

Multiple Variants of System Implementations. As discussed above, different system installations may utilize different implementations of system modules. This creates two challenges to be addressed. First, the same observation may have to be extracted differently in different versions of a module. For example, in one version of the module, an observation may be obtained by instrumenting a particular function call. In a subsequent version, the call may be renamed or eliminated through code refactoring, so that a different instrumentation needs to be introduced. Second, properties specific to the module may also change. For example, the property may reflect assumptions that the module makes about interactions with its environment. A new version of the module may make different assumptions. Maintaining multiple versions of the property is another challenge that is exacerbated by scale.

3 SMEDL Monitoring System

In order to address the above challenges, we have developed a prototype monitoring system [4] that aims to address challenges presented above. Below, we discuss several salient features of the SMEDL[1] system.

[1] SMEDL stands for Scenario-based Meta-Event Definition Language.

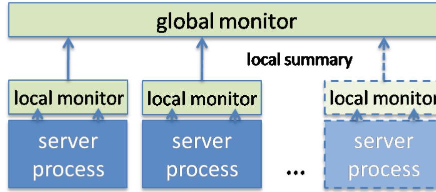

Fig. 2. Modular specification of the performance monitor

3.1 System Design

Modular Property Specification. In order to effectively monitor a property in a large-scale distributed system, SMEDL allows us to specify properties in a modular fashion. In this way, a complex property can be decomposed into a set of monitoring modules that communicate with each other and collectively implement the monitor for the overall property. A common pattern for modular specification is partitioning a global property for a distributed system into a set of locally deployed modules that operate on local observations of each process in the distributed system and convey results of local processing to the global module that computes the overall result. Consider our performance monitor example. Instead of sending all observations to a monolithic global monitor, we partition it into a set of local monitors, one for each process, and a global monitor, as shown in Fig. 2. Local monitors would calculate performance of that process and then send a summary to the global monitor that would aggregate local reports into the global value.

Monitor Coordination and Communication. Clearly, monitor modules need to communicate with each other. The flow of interactions between monitors depends on the property and how it is partitioned into modules. Specification of the monitoring architecture, described below, makes these flows explicit.

Synchronous and Asynchronous Deployment. We specify the logic of each monitor module and, separately, how this module is to be deployed. Often, the user has a choice of deploying the same module synchronously or asynchronously, so decoupling the logic of the module from its deployment strategy increases flexibility of the framework. Continuing our example, for the modular monitor shown in Fig. 2, we deploy the global monitor asynchronously, while local monitors can be deployed synchronously or asynchronously, depending on, e.g., relative overheads of the two approaches.

Dynamic Monitor Instantiation. Large-scale software systems typically contain many similar components that can be added and removed dynamically. In our example, server processes can be spun up and down to meet the demand. When this happens, local monitors are instantiated for each new server process and are connected to the global monitor.

Separation of Property Specification from Observation Extraction. A monitoring specification describes, among other things, what observations are

needed by the monitor in order to do its job. In order to deploy monitors, we also need to know how to extract these observations from the target system. Extraction of observations can be performed in many different ways, for example by instrumenting source code or binaries of system components, by snooping on the system bus, or even off line, reading from a recorded trace. Over time, the target system may evolve and offer new ways of observation extraction, or different variants of system component implementations may require different placement of instrumentation probes. It is important to accommodate these changes in the monitoring setup with as little disruption as possible. SMEDL separates monitoring logic from observation extraction using an event-based API, so that events can be raised in a specified format by an appropriate extraction technology. We have experimented with several such technologies, such as AspectC [2] for instrumenting C source code and a dynamic translation tool SySense by GrammaTech for capturing observations from binary code.

3.2 Monitoring Specification

In SMEDL, monitoring specification contains two major parts: monitoring objects and monitoring architecture.

Monitoring Objects. Monitoring objects represent logic used in checking the property. Each monitoring object has an *interface*: imported events that a monitor receives from its environment and exported events that it raises. Imported events can be observations from the target system or events sent by other monitors. Similarly, exported events can be alarms that are delivered to the system operator or used to trigger recovery, or they can be sent to other monitors for processing. The logic itself is expressed using communicating finite state machines extended with local state variables. State machines take transitions in response to imported events and can raise exported events or update state variables when a transition is taken. Monitoring modules are designed to allow the use of other formalisms to express the logic.

Monitoring objects can have *identity parameters*. Choosing different parameter values allow us to have multiple instances of monitoring objects. In our running example, a natural parameter for the local monitor is the identity of a server. We note that at the specification level, we may want to abstract from the precise nature of this identity. Depending on the system implementation, a server process may be identified by a computer name, an IP address, or maybe a virtual machine identifier where the server runs.

Monitoring Architecture. Monitoring architecture is a directed graph that represents communication between monitoring objects. Nodes of the graph have ports that correspond to events that the node can consume or produce. Nodes that represent monitoring objects have ports that match the interface of the object. Nodes can also represent components of the target system. Ports of these nodes represent observations that are obtained from this component. Edges in the graph represent communication flows from exported events of one node to

imported events of another node. Nodes in a monitoring architecture are partitioned into sets. Monitoring objects within a set are deployed together and are executed synchronously, using a single thread of control. When a monitoring object is placed into a set with a node representing a system component, monitor instances are running synchronously with the component, essentially becoming a part of component instrumentation. Objects in an architecture may be instantiated statically or dynamically. Statically instantiated objects are created at the beginning of a monitored run of the system. Dynamically instantiated objects are created when new values of identity parameters are discovered.

4 Case Studies

The SMEDL system is extensively used in the RINGS project, led by BAE Systems as part of the DARPA BRASS program. The goal of the program is to develop techniques to adapt a given application to changes in the application environment or the underlying execution platform. The RINGS project focuses on a target tracking application, developed by BAE Systems and continuously evolved over a period of over 15 years. The tracker receives data from a number of sensors, e.g., imaging devices, that supply information about observed objects, and contains algorithms that parse sensor inputs and compose observations into tracks, i.e., sequences of points representing position of an object over time.

Over time, both the application and its environment can change. For example, new sensors are introduced into the system, and parsers for new sensor inputs need to be added. Standards for the format of sensor data evolve, which also requires changes to parsers. Track processing algorithms may need to be updated to account for new sensors. The tracker uses a large number of tuning parameters to handle weather and other environmental conditions. Misconfiguration of parameter settings may lead to poor tracking results.

We use SMEDL monitors to detect when the application does not behave as intended. Alarms raised by monitors trigger adaptation modules that perform fault localization and generate patches that compensate for the detected changes. In this paper, we discuss only the detection aspect of the case study.

The main challenge in constructing monitors for the tracking application is that there is no ground truth about tracks available at run time. Instead, monitors have to rely on indirect evidence of misbehavior. Alternatively, monitor can focus on specific faults that are known to have caused misbehavior in the past. Both approaches are imperfect: indirect monitors may not catch all violations, while fault monitors may raise false alarms if the system tolerates the fault. Below, we discuss examples of both monitoring approaches.

Track Quality Monitors. Developers of the tracking application have identified a number of metrics that characterize track output quality. These metrics, collected using a sliding window time interval, include average duration of a track observed in a time interval and the number of *unassociated detections*, i.e., observations of objects that are not associated with any track, also in a given

time interval. We can monitor these metrics at run time and raise an alarm when significant changes are observed. Note that these metrics are indirect.

Consider the design of a track duration monitor. A track is observed as a sequence of timestamped points. Each new point added to the track results in a *track report*. Each track report is delivered to the monitor as an event that carries the track identifier as attribute. We also assume that the system produces timeout events that represent boundaries of the sliding window. The monitoring architecture is very similar to the one shown in Fig. 2: there is a local monitor for each track that calculates duration of the track in the current window and, at each window boundary, sends the value to the global monitor to calculate the metric for all tracks and raise an alarm, if needed. As tracks are added by the application, new track monitors are instantiated.

To implement calculation of a quality metric over a sliding window, the window is partitioned into a series of subwindows, each represented by a separate monitor. In addition, a window manager monitor for each track handles switching of subwindows, while the aggregator monitor combined calculations from each subwindow into the overall track duration within the whole window. The architecture of the monitor is shown in Fig. 3a. Some events and auxiliary monitors are not shown for clarity. Each box represents a monitor, with types of monitor parameters shown in brackets. Edges represent events exchanged by monitors. Each edge is annotated with parameter matching that determines replication of event flows when new instances are created. Consider, for example, the `track` event raised by `WindowManager` and consumed by `Subwindow` monitor. The matching ties the first parameter of the `WindowManager` instance raising the event to the first parameter of the `Subwindow` instance receiving the event. Since `Subwindow` has the second parameter, not bound by the matching, the connection is a *fan-out*, when the `track` event is received by all instances subwindow monitors for that track. By contrast, event `metric_sub` represents a *fan-in*, when events raised by any subwindow for a track are delivered to the `Aggregator` instance for that track. Finally, `metric` events raised by any track aggregator are delivered to the same `Metric` monitor, which is not parameterized. An instance of the architecture for two tracks, and two subwindows in a window, is shown in Fig. 3b.

We illustrate a monitoring specification in SMEDL using a simplified version of the `Aggregator` monitor, shown in Fig. 4. The monitor includes a single parameter, denoted by the `identity` keyword and a number of state variables. It has two imported (input) events, one representing a report from a subwindow and the other used for initialization, and one exported (output) event, representing the track duration calculated at the window boundary. It also has a number of internal events, described below. Monitoring logic is represented by a collection of scenarios. Each scenario represents an event-driven state machine. In this example, each scenario has a single state. Each transition in a scenario is triggered by an imported or internal event and can happen only if a guard is satisfied. Guards are predicates over state variables of the monitor and attributes of the triggering events. When a transition occurs, a series of actions is executed,

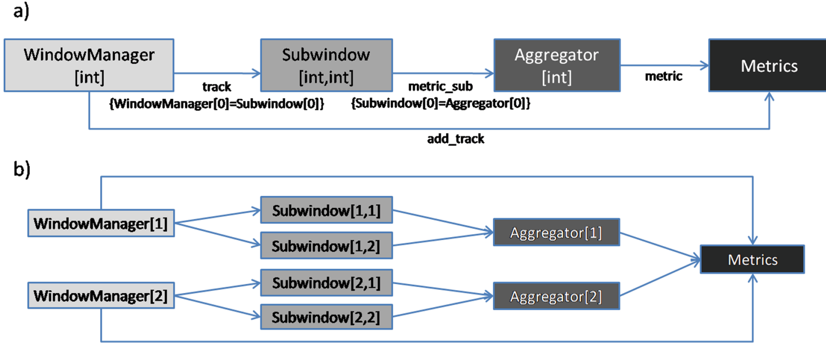

Fig. 3. Monitoring architecture for the case study

each of which either updates a state variable or raises an exported or internal event. For clarity, we do not show details of the guards and elide most of the actions. We can see that each scenario performs a certain check represented as a guard. For example, the check can determine whether the track started or was dropped within the current window, and updates the state variables accordingly. Then, an internal event is raised to trigger the next check.

```
object Aggregator

identity
    int id;

state
    int msg_cnt = 0;
    int event_cnt = 0;
    float observed_time = 0;

events
    imported initial(int, float, int, int);
    imported metric_sub(int, float, float, int);
    internal checkNum();
    internal i1( int, float, float);
    internal i2(float,float);
    internal i3(float);
    internal output();
    exported metric(int, float);

scenarios

initialization:
    init -> initial(ts, sub_w, sub_size,  prob) {...} -> init

accumulation:
    start -> metric_sub(n,ft,lt,flag) { msg_cnt ++; ...; raise i1(n,ft,lt); } -> start

chk1:
    in -> i1(n,ft,lt) when (g1) { event_cnt = event_cnt + n; raise i2(ft,lt); } -> in
    in -> i1(n,ft,lt) when (!g1) { event_cnt = event_cnt + n; raise checkNum(); } -> in

ft_chk:
    ftc -> i2(ft,lt) when (g2) { ...; raise i3(lt); } -> ftc
    ftc -> i2(ft,lt) when (!g2) { raise i3(lt); } -> ftc

lt_chk:
    ltc -> i3(lt) when (g3) { ...; raise checkNum(); } -> ltc
    ltc -> i3(lt) when (!g3) {  raise checkNum(); } -> ltc

output_chk:
    opc -> checkNum() when (g4) { observed_time = ...; raise output(); } -> opc
    opc -> checkNum() when (!g4) { observed_time = ...; raise output(); ...; } -> opc

out:
    fin -> output()  { ...; raise metric(event_cnt, observed_time); ...; } -> fin
```

Fig. 4. Specification of the `Aggregator` monitor

Sensor Format Monitors. The second case study, also motivated by past experiences, concerns data interchange formats. To facilitate independent devel-

opment of sensor devices and tracking applications, data interchange standards such as STANAG 4607 [1] have been introduced. Nonetheless, incompatibilities can still be encountered during missions, either because a sensor does not always follow the standard or because the parser module has not been updated to the latest version of the standard. The standard offers both a binary encoding of sensor messages and an XML encoding. We have developed monitors to detect deviations from the standard binary format. In the binary format, fields do not have explicit delimiters and their sizes are specified within the parser. If a field in the message has a different size than the parser expects, fields will be misaligned and the message will be parsed incorrectly. In our case study, we rely on the knowledge of acceptable ranges for a field in the message in order to detect and localize the problem.

5 Conclusions

We have presented challenges to monitoring of complex software system and briefly described a monitoring system that aims to address the challenges by offering flexible monitor specification and deployment. The monitoring system has been applied in a case study involving a large-scale target tracking application. In the case study, monitors have been used detect deviations from the original application intent and to trigger an automated search for repair.

References

1. NATO ground moving target indicator format - STANAG 4607, edition 2. https://standards.globalspec.com/std/1300603/nato-stanag-4607. Accessed 9 May 2018, September 2010
2. Coady, Y., Kiczales, G., Feeley, M., Smolyn, G.: Using aspectC to improve the modularity of path-specific customization in operating system code. SIGSOFT Softw. Eng. Notes **26**(5), 88–98 (2001)
3. Sokolsky, O., Havelund, K., Lee, I.: Introduction to the special section on runtime verification. Softw. Tools Technol. Transf. **14**(3), 243–247 (2012)
4. Zhang, T., Gebhard, P., Sokolsky, O.: SMEDL: combining synchronous and asynchronous monitoring. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 482–490. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_32