The Role of Event Description in Architecting Dependable Systems

Marcio S Dias Debra J Richardson
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
{mdias,djr}@ics.uci.edu

ABSTRACT

Software monitoring is a well-suited technique to support the development of dependable systems, and has been widely applied not only for this purpose, but also for others such as debugging, security, performance, etc. Software monitoring consists of observing the dynamic behavior of programs when executed, by detecting particular events and states of interest, and analyzing this information for specific purposes.

There is an inherent gap between the levels of abstraction the information is collected (the implementation level) and the software architecture level. Unless there is an immediate one-to-one architecture to implementation mapping, we need a specification language to describe how low-level events are related to higher-level ones. Although some event specification languages for monitoring have been proposed in the literature, they do not provide support up to the software architecture level.

In this paper, we discuss the importance of event description as an integration element for architecting dependable systems. We also present how our current work in defining an interchangeable description language for events can support the development of such complex systems.

1. INTRODUCTION

As stated in the workshop call, "architectural representations of systems have shown to be effective in assisting the understanding of broader system concerns by abstracting away from details of the system". The software architecture level of abstraction helps the developer in dealing with system complexity, and is the adequate level for analysis, since components, connectors, and their configuration are better understood and intellectually tractable [16].

When building dependable systems, additional management services are required and they impose even more complexity to the system [14]. Some of these services are fault-tolerance [5] and safety, as well as security (intrusion detection) and resource management, among others. An underlying service to all these services is the software monitoring.

Software monitoring is a well-known technique for observing and understanding the dynamic behavior of programs when executed, and can provide for many different *purposes* [13][15]. Besides dependability, other purposes for applying monitoring are: testing, debugging, correctness checking, performance evaluation and enhancement, security, control, program understanding and visualization, ubiquitous user interaction and dynamic documentation.

Software monitoring consists in *collecting* information from the system execution, *detecting* particular events or states using the collected data, *analyzing* and *presenting* relevant information to the user, and possibly taking some (preventive or corrective) *actions*. As the information is collected from the execution of the program implementation, there is an inherent gap between the level of abstraction of the collected events (and states) and of the software architecture. Unless the implementation was generated from the software architectural description, or there is an easily identifiable one-to-one architecture to implementation mapping [1][10][16], we need to describe how those (primitive) events are related to higher-level (composed) events.

Many monitoring systems were developed so the user could specify composed events from primitive ones, using provided specification languages. However, in general, these specification languages are either restricted to a single monitoring system, not generic for many different purposes, or cannot associate specified events to the software architecture.

There is no monitoring system able to provide for all different purposes. One problem occurs when a user is interested in applying monitoring for more than one purpose (for instance, dependability, performance evaluation, and program visualization). In this case, he or she would probably run different monitoring systems and, consequently, need to describe the same events multiple times using different specification languages.

To put it simple, software monitoring is a well-suited technique to support the development of dependable systems, and has been widely applied for this purpose. However, monitoring systems suffer in the ability to associate collected information to software architecture level.

In this position paper, we discuss how software monitoring can be applied at the software architectural level to support dependability. In this context, we present some requirements for event description languages, and our ongoing work on xMonEve, an XML-based language for describing monitoring events.

2. EVENT MONITORING

There are basically two types of monitoring systems based on the information collection: *sampling* (time-driven) and *tracing* (event-driven). By sampling, information about the execution *state* is collected synchronously (in a specific time rate), or asynchronously (through direct request of the monitoring system). By tracing, on the other hand, information is collected when an *event* of interest occurs in the system [11].

Tracing allows a better understanding and reasoning of the system behavior than sampling. However, tracing monitoring generates a much larger volume of data than sampling. In order to reduce this data volume problem, some researchers have been working on encoding techniques [12]. A more common and straightforward way to reduce data volume is to collect interesting events only, and not all events that happen during a program execution [7][9]. This second approach may limit the analysis of events and conditions unforeseen previously to the program execution, though.

Both *state* and *event* information is important to understand and reason about the program execution [14]. Since *tracing* monitoring collects information when *events* occur, *state* information can be maintained by collecting the events associated to state changes. With a hybrid approach, the sampling monitoring can represent the action of collecting state information into an event for the tracing monitoring. Like any other event, not all events with state information should be collected, but only those events of interest. Integrating *sampling* and *tracing* monitoring and collecting the state information through events reduce the complexity of the monitoring task.

The monitoring system needs to know what are the events of interest, i.e. *what events should be collected*. Therefore, it provides an event specification language to the user. Additionally, it needs to know *what kind of analysis* it should perform over the collected information. The user may provide a specification of the *correct behavior* of the system and the monitoring checks for its correctness, showing when the system did not perform accordingly to the specification. Another approach is to have the user specifying the *conditions of interest*, and the monitoring system identifying and notifying him/her when these conditions are detected. A third approach, not frequently used by monitoring systems, is to *characterize* (build a model of) the system behavior from the program execution, mainly for program understanding and dynamic documentation.

Since analysis is so intrinsic to the monitoring activity, it became normal to have monitoring specification languages where the user describes not only the events, but also the analysis to be performed. As a consequence, monitoring specification languages are biased to the kind of analysis performed by the monitoring system. To the best of our knowledge, there is no monitoring specification language that separates the concerns of "*what are the events of the system?*" (describing the events of interest only), "*what is(are) the purpose(s) for monitoring the system?*" (performance, reliability, etc), and "*what kinds of analysis should be performed?*" (i.e. condition detection; correctness checking or comparison; or model characterization).

In the current step of our research, we are focusing on the first question for monitoring specification languages, i.e. "*what are the interesting events of the system?*". We are defining an extendable and flexible language (xMonEve) for describing monitoring events independently of the system implementation, the purpose of analysis, and the monitoring system.

2.1 Requirements for xMonEve

Initially, we identified new requirements for event description languages. Some of the requirements that guide us through the development of xMonEve are:

- *general purpose*: need to be flexible enough to accommodate event description for multiple monitoring purposes (i.e. independent of the analysis to be performed);
- *independence of monitoring system*: must allow generic description of events, both primitive and composed, not restricted to a specific monitoring system (or environment);
- *implementation independence*: need to provide mechanisms that separate the conceptual event to the implementation mapping;
- *reusable*: event description should be reusable independently of the implementation and monitoring system;
- *extensible*: extension of event description should be supported, so more specific information can be associated to the events. For instance, one extension can be the association of monitoring events to software architectural elements.

Like most monitoring specification languages, xMonEve can represent both primitive and composed events. Primitive events are events that occur in a specific moment in time, i.e. an instantaneous occurrence. Composed events are events composed of other events (primitive or composed ones), and have a specific moment of start and end. While its starting time is defined by the first event to happen, the last event determines its ending time.

Composed events provide a higher-level abstraction for the system execution. Primitive events may be filtered out and abstracted into composed events, having unneeded details thrown away.

One important advantage of event description is that it is well suited to bridge the gap between software architecture and implementation (mapping). For multiple reasons (such as reuse, maintainability, performance, fault-tolerance, security, etc), the implementation *structure* may not exactly correspond to the conceptual architectural structure. Events imply in a *functional mapping* for associating architecture and implementation, instead of a *structural mapping*. A *functional mapping* between implementation and any previous software specification document (software architecture, requirements, etc) should be always possible. If a system functionality cannot be associated to implementation actions (independently of how hard it may be for a human being to do this association), than this functionality was not implemented at first place.

Therefore, although events play an important role in the mapping between architecture and implementation, event specification languages have often ignored this importance, and not provided any mechanism to associate these different abstraction levels.

2.2 Describing Events with xMonEve

The purpose of this paper is not to provide a complete discussion about the xMonEve language, but to give an overview of its concepts and emphasize some specific details relevant for the context of architecting dependable systems.

In xMonEve, every event type has *ID*, *name*, *description*, *attributes*, and *abstraction*. The *abstraction* field is used to associate the event to a context. For instance, while a primitive event "*open*" may be associated to the "*File*" abstraction, a composed event "*open*" may be associated to the "*CheckingAccount*" abstraction. It is important to note here that *CheckingAccount* may or may not represent a structure (e.g., class or subsystem) of the system implementation. This mechanism allows multiple levels of abstraction, from the implementation level to the requirement level, passing through design and also software architecture. In the previous example, CheckingAccount may be a component abstraction at the software architectural level.

```
<event name=open type=primitive ID=#>
  <abstraction>File</abstraction>
  <description>opening file</description>
  <attributes>
    <field name=filename type=string>
    <thread_id>
    <timestamp>
  </attributes>
  <...>
</event>
```

Figure 1. Example showing common features to every event.

Additionally to the features that are common to every event, primitive and composed events have other distinct characteristics.

2.2.1 Primitive Events

A primitive event may be in more than one system, and with different implementations. In order to have a reusable definition for this event, multiple implementation mappings should be allowed. So, primitive events may have zero, one, or multiple mappings. These events will typically have no mapping until the programmer specify them, since he is the one with the right knowledge.

```
<event name=open type=primitive ID=#>
  <...>
  <mapping>
    <system ref=java_library/>
    <language name=java/>
    <class name=java.io.File/>
    <type name=operation>File(String pathname)
    </type>
    <when type=method_exit/>
    <assignments>
      <set field=filename parameter=pathname>
    </assignments>
  </mapping>
  <...>
</event>
```

Figure 2. Example mapping a primitive event to the implementation. In this example, the event *open* occurs when the "method" (actually the constructor) of *java.io.File* class returns, and the event field *filename* has its value assigned from the *pathname* parameter.

2.2.2 Composed Events

When defining composed events, no mapping is needed, since it is composed of other events. Besides the common event fields, composed events have three extra sections: *composition*, *correlation* and *conditions*. In *composition*, it is described what are the event types that compose this event. In *correlation*, the sequence or order of these events to generate the abstract event. The *condition* section describes the conditions that have to be satisfactory between these events so the composed event can be identified.

```
<event name=AccountTranfer type=composite ID=#>
<abstraction>Client</abstraction>
<composition>
  <alias name=before event=Bank.TransferRequest/>
  <alias name=withdraw event=Account.Withdraw/>
  <alias name=deposit event=Account.Deposit/>
  <alias name=after event=Bank.Tranfer/>
</composition>
<attributes>
  <field name=client value=before.client/>
  <field name=from value=withdraw.account/>
  <field name=to value=deposit.account/>
  <field name=amount value=withdraw.amount/>
  <timestamp start=before.timestamp.start
             end=after.timestamp.end/>
</attributes>
<correlation method=regexp>
  <sequence min=1 max=1>
    <event alias=before min=1 max=1/>
    <parallel min=1 max=1>
      <event alias=withdraw/>
      <event alias=deposit/>
    </parallel>
    <event alias=after min=1 max=1/>
  </sequence>
</correlation>
<condition>
  <and>
    <exp> before.client = withdraw.client =
          deposit.client = after.client </exp>
    <exp> withdraw.amount = deposit.amount </exp>
  </and>
</condition>
<...>
</event>
```

Figure 3. Example of the composed event "AccountTransfer". In this example we can see what events compose this one (*composition*), what is the *correlation* between these events, and what *conditions* should be satisfied between those events.

3. ARCHITECTING DEPENDABLE SYSTEMS

With xMonEve, events can be described in both *top-down* and *bottom-up* approaches, since the language is independent of the development process. However, in the context of architectural development of dependable software, a top-down approach would be more natural (but not the only possible approach). The architect would describe (incomplete composed) events at the architectural level, while the designer and/or programmer would decompose these events into lower-level events, until they could be completely defined in terms of primitive events only.

First, in this section, we discuss the role of events as the integration element for the development of dependable systems from software architecture to program execution. Afterwards, we briefly present a top-down approach for architecting such systems.

3.1 Event as the Integration Element

According to Hofmann et al. [4], both monitoring and modeling rely on a common abstraction of a system's dynamic behavior, *the event*, and therefore can be integrated to one comprehensive methodology for measurement, validation and evaluation.

When considering modeling and analysis techniques that have been applied for designing dependable (reliable) system, Markov models and simulations stand out [8]. It is important to note that the *event* abstraction is also common to these techniques. A Markov model has a state changed with the occurrence of an event, which time to occurrence is often modeled with a random exponential distribution. During simulation execution, event traces are generated, over which analyses are performed.

Therefore, the *event* abstraction can act as the basic element for integrating: reliability models, architecture designs, system implementation, and analyses. In order to have this integration, an interchangeable (shared and canonical) representation of events should be available during the whole software development process. In this context, xMonEve represents an important step towards this integration.

3.2 Top-Down Approach

Here, we informally and briefly describe a top-down approach for architecting dependable systems by using events as basic elements of integration.

When building Markov models for reliability analysis, architects and designers may associate information about the model to the events. In this case, the event would include the information about the state change, and also the random distribution of its occurrence. This event definition could be used for running reliability analysis prior to the system development.

```
<event name=enter_overload_state type=composite …>
  <abstraction>ComponentA</abstraction>
  <markov_model>
    <transition from="overload_state"
                to="failure_state"/>
    <distribution (...) />
    <...>
  </markov_model>
  <...>
</event>
```

Figure 4. Extension of an event description with information for the Markov model.

Independently of having or not Markov (or others) extensions to an event definition, software architects, designers and programmers may compose (or decompose) an event from (into) other events, by defining and associating these new events. Thus, multiple levels of event abstraction can be created, from requirements and software architecture abstractions to implementation primitive events.

```
<event name=overload_timeout type=composite …>
  <abstraction>ComponentA</abstraction>
  <markov_model>...</markov_model>
  <composition>
    <alias name=eos event=enter_overload_state />
    <alias name=avg event=loadAverageSampling.../>
  </composition>
  <attributes>
    <field name=status .../>
    <field name=loadaverage value=avg.la .../>
    <...>
  </attributes>
  <...>
  <condition>
    <and>
      <exp>status = "running"</exp>
      <exp>loadaverage > 10</exp>
      <exp>ellapsedtime(eos.timestamp.end)>5</exp>
    </and>
  </condition>
</event>
```

Figure 5. Event definition of Figure 4 with the information added by software architects, designers and/or programmers.

After the implementation of the application, with the event description represented in xMonEve, a monitoring system can observe the application execution and analyze its behavior at multiple abstraction levels, depending on the purpose and interest of the user. For instance, analysis can happen at the implementation level for debugging, performance, testing etc, as well as at the architectural level for dependability, performance, validation etc.

4. RELATED WORK

Many specification languages have been proposed in the literature for describing events (and states) for monitoring technique. The definition of xMonEve is influenced by characteristics present in most of them.

Some specification languages were developed based upon extended regular expressions, such as EBBA [1]. These languages put more emphasis in temporal ordering, and, in general, have limited capability to specify states, and events are assumed to occur instantaneously. These languages influenced xMonEve in the specification of the correlation of composed events, although in xMonEve we also consider non-instantaneous events.

Snodgrass [15] developed a query language for a history database, using it to specify events and states. Although this work has a large influence in monitoring techniques, the language has a limited set of operators from relational algebra with a limited representation power. One important influence of this work in ours is that, in this work, with relational algebra, the language expresses *what* derived information is desired, and not *how* it is derived.

PMMS [9] uses a specification language based on relational calculus to for description of events and user questions. A big contribution of this work is in providing an automatic technique for instrumenting the program code to collect only the events needed to answer explicit user questions. This technique removes the burden of code instrumentation from the programmer. This specification language has limitations to specify events, and this is linked to the fact that PMMS supports tracing monitoring only, and no sampling.

Shim et al. [14] proposed a language based on classical temporal logic for specifying event and states. This work influenced us in considering non-instantaneous events. However, they do not provide any mechanism to create different levels of abstraction (to associate, for instance, events to software architecture elements), neither an extensible way to associate more semantics to the event specification.

With FLEA [3], user expresses his/her requirements and assumptions for monitoring. The main idea behind it is to be able to monitor programs that were not developed with monitoring in mind, and to check software requirements though events. In a similar way, xMonEve is meant to be independent of implementation, and this also includes its structure. Additionally, we also think it is important to bridge different abstractions, such as requirements and implementation, and any other possible abstraction.

Another kind of related work is the application of software monitoring at the architecture level[1][16]. It is worth to mention that both works consider the instrumentation of connectors for collecting the information, instead of the components, and the basic element for analysis is the event at the architectural level. In these works, the mapping problem between software architecture and implementation is simplified since the implementation and software architecture design presents a one-to-one structural correspondence.

5. CONCLUSIONS AND CURRENT WORK

The event and its definition play a major role in the integration of development techniques for architecting dependable systems, since it is a common abstraction to multiples techniques. However, to have an effective integration, events also have to be described in a common way. xMonEve is an event description language for this integration purpose. We are currently working on xMonEve definition and refinement. xMonEve does not describe *how* the event is going to be collected, but *what* that event is or represents. xMonEve is not intended to be a substitute for other event specification languages, but to promote integration of techniques by providing an interchangeable description for events.

In this position paper, we present the problem of mapping implementation to software architecture; discuss the importance of the event description in the context of developing complex and reliable systems; present requirements for event description languages; presented our current work in xMonEve; show how xMonEve can support integration of reliability techniques and software architectures; propose a top-down approach for reliability; and compare our work with other specification languages from the literature.

Inside this paper, in many occasions we say "*the developer would describe the event*", or similar. However, this is a hard task by itself and should be supported by tools. Event definitions could and should be generated from other system documents, such as requirement specifications, architectural and design models, testing documents, etc. This type of tool support is also an important step towards the usefulness and success of monitoring techniques, as well as such event specification languages.

At this step, we have not gotten yet to analysis description, i.e., how to describe what types of analyses a monitoring system should perform, and for what purpose. Now, it is important for us to understand better how each different purpose may affect monitoring systems. Since a major part of the functionality of monitoring systems is the same in multiple occasions, we probably need a family of monitoring systems with customizable components, so the configuration of a monitoring systems could go one step forward. Instead of configuring sensors and probes, configuration would represent the tailoring of the whole monitoring system to attend specific developer needs.

6. REFERENCES

[1] R. Balzer, "Instrumenting, Monitoring, & Debugging Software Architectures", 1997. [http://citeseer.nj.nec.com/411425.html]

[2] P. C. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior", ACM Trans Computer System, vol. 13, n. 1, Feb. 1995, pp. 1 – 31

[3] D. Cohen, M. Feather, K. Narayanaswamy, and S. Fickas, "Automatic Monitoring of Software Requirements", Proc Int'l Conf Software Engineering (ICSE) 1997, pp. 602-603.

[4] R. Hofmann, R. Klar, B.Mohr, A. Quick, and M. Siegle, "Distributed Performance Monitoring: Methods, Tools, and Applications", IEEE Trans. Parallel and Distributed Systems, vol. 5, n. 6, June 1994, pp.585-598.

[5] Y. Huang and C. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience", Proc. 23rd Int'l Symp on Fault Tolerance Computing, 1993, pp. 2-9.

[6] C. Jeffery, "Program Monitoring and Visualization: An Exploratory Approach", Springer-Verlag, 1999.

[7] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring Distributed Systems. ACM Transactions on Computer Systems, vol. 5, no. 2, May 1987.

[8] J. F. Kitchin, "Practical Markov Modeling for Reliability Analysis", Proc Annual Reliability and Maintainability Symposium, 1988, Jan. 1988, pp. 290-296.

[9] Y. Liao and D. Cohen, "A Specification Approach to High Level Program Monitoring and Measuring", IEEE Trans. Software Engineering, vol. 18, n. 11, Nov. 1992.

[10] N. Medvidovic, D. Rosenblum, and R. Taylor, "A Language and Environment for Architecture-Based Software Development and Evolution", Proc Int'l Conf on Software Engineering, May 1999, pp. 44 -53

[11] D. M. Ogle, K. Schwan, and R. Snodgrass, "Application-Dependent Dynamic Monitoring of Distributed and Parallel Systems", IEEE Trans. Parallel and Distributed Systems, vol. 4, n. 7, July 1993, pp. 762-778.

[12] S. Reiss, and M. Renieris, "Encoding Program Executions", Proc Int'l Conf Software Engineering, May 2001.

[13] B. Schroeder, "On-Line Monitoring: A Tutorial", IEEE Computer, vol. 28, n. 6, June 1995, pp.72-77.

[14] Y. C. Shim and C.V. Ramamoorthy, "Monitoring and Control of Distributed Systems", Proc. 1st Int'l Conf on System Integration, Apr. 1990, pp. 672-681.

[15] R. Snodgrass, "A Relational Approach to Monitoring Complex Systems", ACM Trans. Computer Systems, vol. 6, n. 2, May 1988, pp.156-196.

[16] M. Vieira, M. Dias, D. Richardson, "Analyzing Software Architecture with Argus-I", Proc Int'l Conf on Software Engineering, June 2000, pp. 758 –761.