Using Event Contexts and Matching Constraints to Monitor Software Processes

Naser S. Barghouti and Balachander Krishnamurthy

Software Engineering Research Department
AT&T Bell Laboratories
Murray Hill,NJ 07974 USA
naser@research.att.com, bala@research.att.com

Abstract

Automated monitoring of events that occur in a computer network is important for software process enactment. Relevant events include modification of product artifacts like source files and documents, execution and termination of tool invocations, and user actions. Monitoring these events requires capturing the context in which they occur and constraining the monitoring to relevant events only. We introduce the concept of event context, and present constructs for using event contexts to constrain event matching. The constructs have been implemented as an extension to an existing event monitoring system called Yeast, which serves as the event monitoring component in Provence. Provence is an open process-centered environment that distinguishes between the enactment of a software process model and the actual execution of a process, which takes place outside Provence, in a distributed environment. Provence monitors the actual execution of software processes, abstracted as sequences of low-level events, and maps these sequences of events into transitions in the process model enactment.

1 Introduction and Motivation

Process-centered software engineering environments (PSEEs) provide automated assistance by enacting an explicit model of a project's development processes. Most existing PSEEs are intrusive: They replace the working environment of an organization by a process-centric environment in which all interactions between the human developers and the tools and among the tools themselves are controlled by the PSEE, in accordance with the process model. Unfortunately, software engineers who have become accustomed to their old working environment resist changing to a completely new development environment, especially before the advantages have been demonstrated conclusively. A non-intrusive alternative is thus needed: mechanisms supporting this alternative include automated monitoring of development events and the ability to capture the context in which these events occur.

A non-intrusive PSEE avoids changing a project's working environment by separating the enactment of a software process model from the execution of the process. The actual execution of a project's processes takes place outside of the PSEE, within a project's current working environment. A typical working environment for a software development project comprises a loosely-integrated set of the widely-available tools that automate parts of the software development process, such as compilers, editors, debuggers, version control tools, and program building tools. Product data (e.g., source files, project documents, test suites) are often stored in files that are organized in a directory structure. Developers invoke tools to create and modify product data during the execution of the development process.

Enacting a process model requires traversing transitions that change the state of the enactment. Some of the transitions correspond to process execution activities, such as revising a design document or performing a regression test. Each of these transitions is mapped into a sequence of low-level events, such as modifications of files and invocations of tools. An event monitor detects the occurrence of such sequences of events and notifies the PSEE, which traverses the corresponding enactment transition and performs other internal transitions automatically. In this manner, software engineers continue to use a familiar working environment but benefit from the assistance provided by the process support system.

Monitoring of the execution of a project's processes requires two capabilities that are generally lacking in existing event monitoring systems (c.f. Section 2). The first is the ability to capture information about the *context* in which a event occurred. This information is needed to determine whether or not two events belong to the same sequence of events corresponding to an enactment step (for example, if the events were generated by the same user). The second capability is constraining event monitoring to the events that are relevant to a project's processes. The working environment within which a project's processes are executed is typically distributed across a computer network that generates a large number of events; it is not cost-effective to monitor all of the events that occur in the network. Contextual information provides an ideal mechanism for constraining the monitoring. For example, after a specific event has been detected, future monitoring can be limited to events that are generated on the same machine.

This paper introduces the concept of *event context*, presents a monitoring mechanism for capturing the context of relevant events, and introduces language constructs for constraining event monitoring based on contextual information. Both the monitoring mechanism and the language constructs have been implemented as an extension to an existing event-action specification system called Yeast [KR91, RK91]. The extended Yeast system is used as the event monitoring component in Provence, a non-intrusive PSEE architecture [KB93].

The paper is organized as follows. First, we discuss related work. We then present a brief overview of some of the concepts in Yeast that are needed to understand our extensions. Next, we present our extensions and show how they provide the necessary mechanisms for partially overcoming the limitations. We then describe how these extensions are being used in Provence and other Yeast applications. Next, we examine some of the trade-offs involved, and justify our choices. We conclude by discussing the benefits and limitations of our extensions.

2 Related Work

Automated monitoring of events that occur in a computer network is essential for any software system that needs to interact with and react to its environment. A monitoring system accepts event-action specifications from an application. Each specification describes an event pattern and an action. When the monitoring system detects the occurrence of the event pattern, it performs the corresponding action.

The majority of the event specification languages and event matching systems were proposed either in the tool integration area or in the active database community. These systems vary in the specification formalism they provide. Widely-used formalisms include finite state machines [GJ91], labeled petrinets [GL81], event graphs [CKAK93], and logic combinators [RK91, KR91].

Active databases react to the occurrence of events by triggering appropriate actions [Mor83]. The ability to specify events and actions is included in the query languages of active databases, typically as rules [UDL90]. The primary focus in active databases has been monitoring of individual internal database events, such as inserting a tuple in a relation or updating a database object. In contrast, the focus of our work is monitoring patterns of external events and capturing the operating system context (machine, process, user, etc.) within which these events occur.

Recently, researchers in the database community have proposed mechanisms for specifying patterns of events (also called composite events) and for capturing external events. For instance, Pissinou *et al.* presented a temporal query language that contains constructs for triggering actions based on the occurrence of patterns of both internal and external events [PSE+94]. However, none of these proposals included capturing the contexts in which events occur.

Among the active database systems, Snoop is probably closest to our work since it provides the capability to capture some contextual information (also referred to as event attributes) [CKAK93]. Snoop, however, does not provide mechanisms for using contextual information to constrain event matching. For example, one cannot constrain the matching of events that logically match various parts of the same event pattern to those events that were generated by the same user or on the same machine. This kind of constraint is often necessary for monitoring sequences of events in a multi-user distributed environment, where similar events might be generated by different users and might occur on different machines.

In the tool encapsulation domain, Forest [GI90] and Field [Rei90] provide capabilities for monitoring events generated by tools. The advantage of this approach is that the envelope can be strengthened to generate all the necessary contextual information. Internal constraint matching is also possible though it is not apparent if this has been done in either Field or Forest. The shortcoming of the approach is that it requires all the tools participating in the monitoring to be enveloped. Our approach is more generic *and* does not require tool enveloping. New tools can be transparently added to our environment without requiring any enveloping since monitoring is done at the level of operating system events.

The Hakoniwa PSEE follows a monitoring and navigation approach that is similar in some respects to the tool integration approach [LiMIT93]. In Hakoniwa, development process models are activity-centered and based on Communicating Sequential Processes (CSP). Each task is modeled as a sequence of primitive activities, which correspond to the event sequences generated by the translator in our model. Synchronization between concurrent activities is modeled explicitly in Hakoniwa in the form of message passing primitives. There are two main difference between our model and the model in Hakoniwa. First, the process model in our case can be described in any high-level formalism, such as rules or petri nets. The sequences of events for monitoring purposes are *generated* from the high-level model. In Hakoniwa, the model is lower level, and thus, we believe, more cumbersome to define. Second, it is not clear from [LiMIT93] how much Hakoniwa changes the developers' working environment. Specifically, it seems that developers execute tasks by selecting items from the menus generated by Hakoniwa. If this is the case, the developers are constrained to use Hakoniwa's interface to execute development processes, and thus we would classify Hakoniwa as an intrusive PSEE.

3 Event Context

Events in a computer network occur in a context. The context comprises many pieces of information. The specific context information needed in the case of PSEEs are: (1) the integer-valued identifier of the operating system process that generated the event (`pid`), (2) the string-valued identifier of the user who owns that process (`uid`), and (3) the name of the machine from which the event was generated (`hostname`). These three contextual elements collectively identify the exact source of an event. Other contextual information, such as the timestamp of an event, can also be captured and may be useful in other applications.

The context of an event can be captured only if the event is detected at the time of its occurrence. This can be achieved if the monitoring system is notified about the occurrence of events. A notification can be generated by a user, a program, or an instrumented operating system monitor (c.f. Section 7. In contrast, if an event is detected *after* it has occurred, the context surrounding the occurrence of the event may be lost. For example, say a user wants to be notified when a particular file that is accessible to multiple users and from multiple machines is modified. If the event monitoring system uses polling to detect the occurrence of the event based on a preset time interval (e.g., every thirty seconds), all it can determine is that the file has changed since the last time the monitoring system examined it. It is not always possible to know who modified the file, on which machine it was modified, and whether or not more than one modifications have occurred during that interval.

The event-action specification tool Yeast provides mechanisms for detecting events either immediately via announcements or by polling. We used these mechanisms and extended them to implement event concepts and matching constraints.

4 Selective Overview of Yeast

Yeast matches arbitrary patterns of events, and triggers actions specified by the user. A user writes a set of specifications, each consisting of a complex event pattern and an action. These specifications are registered with the Yeast daemon. The daemon detects several types of event that occur in a computer system, and matches these events against the event descriptors in the event pattern part of the specifications. When an event pattern is fully matched, the corresponding action, which can be any program executable from the shell, is triggered.

The Yeast daemon accepts a set of specifications via the Yeast client command *addspec*. Each specification is of the form:

addspec *event_pattern* do *action*

where *event_pattern* is a complex expression of *event descriptors* and *action* is any shell-executable command. The action part is considered external and thus is not parsed by Yeast, but is triggered as is. The three combinators *and* (conjunction), *or* (disjunction) and *then* (sequencing) are used to form the event pattern. An *event descriptor* describes an object event of the form:

class object attribute relational_test

where *class* is the name of an object class, *attribute* is the name of an attribute, and *relational_test* is a test against the value of the specified *attribute* of the specified *object* at the time a match of the descriptor is attempted.

Yeast applications can define object classes and attributes for these classes. For example,

```
defobj resource

defattr resource usage integer
```

defines a new object class called *resource*, and an integer-valued attribute *usage* for that class. The value of the attribute might reflect, for example, the usage level per day of a resource object. Keeping track of this information might be useful in planning resource management for a project. Note that the object need not correspond to a file.

Yeast predefines seven built-in object classes (`file`, `dir`, `process`, `tty`, `user`, `host`, and `filesys`), and for each of these classes, Yeast defines a set of pre-defined attributes. For example, *mtime* (modification time) is a pre-defined attribute for the class `file`. Applications can define new attributes for pre-defined object classes. For example,

```
defattr file syscall string
```

defines a new string-valued attribute called *syscall* to the pre-defined object class `file`. Users can then write specifications that involve both pre-defined attributes and application-defined attributes. For example, the event pattern of the specification

```
addspec file /bin/yeast mtime changed
  then  file /bin/yeast syscall == write
  do ...
```

is composed of two descriptors: the first is matched when the value of the pre-defined attribute *mtime* of the file /bin/yeast has been modified (*changed* is syntactic sugar for "different from previous value"), and the second is matched when the application-defined attribute *syscall* of the file is set to *write*.

The events that match the first descriptor differ in nature from those that match the second descriptor. The first type of event changes the value of a pre-defined attribute, which is maintained by the file system, external to Yeast; the second assigns a value to an application-defined attribute of an object, which is maintained by Yeast. The Yeast daemon can automatically detect the occurrence of the first type by *polling* (checking periodically) the file system to check the value of the specific attribute (*mtime* of the file /bin/yeast), and subsequently determining if it has changed since the specification was added. This type of event is called a *polled event*.

In contrast, the second type of event cannot be detected automatically; instead, Yeast must be explicitly notified about the occurrence of the event via the `announce` client command. Consequently, this type of event is called an *announced event*. The following is an example of announced event that matches the second event descriptor in the example above:

announce file /bin/yeast syscall = write

Like all announced events, this announced event involves an object (/bin/yeast), a single attribute (`syscall`), and a specific value for the attribute (`write`).

Announced events can be generated explicitly by any authenticated user, either directly from the shell or from within a program. They can also be generated automatically by a file system monitor that intercepts all operating system calls [KK92]. In this case, when a `write` system call (or any other system call) is executed on the file, the file system monitor intercepts the system call and automatically generates a corresponding announced event like the one above. The Yeast daemon can thus be notified immediately about the occurrence of the write event. This mechanism is the basis of the automated monitoring in Provence, as explained in Section 7.

The difference in the way that Yeast matches the two types of event has implications on the ability to capture the context of the events. In particular, as discussed earlier in Section 3, only the context of announced events can be captured in Yeast because polled events are detected after they occur. The discussion in the rest of the paper concerns only announced events. Therefore, we will use the term "event" in the rest of the paper to mean announced event.

5 Extending Yeast with Event Contexts

We extended the Yeast language with several constructs for capturing the context in which events occur. The first construct is naming of event descriptors as a mechanism for binding a name to the context of the event that matches the named descriptor. The syntax is:

> addspec *N*: *event descriptor* ... do *action*

where *N* is an identifier and *event descriptor* describes an event. The name *N* is used as a placeholder for the context associated with the event that matches the event descriptor. In that sense, *N* can be thought of as a variable whose value is bound to the context of an event after a successful matching.

Elements of the event's context that matched the event descriptor are accessed through the name, using the following syntax:

> *N*:*context_element*

where *context_element* is one of `pid`, `hostname`, or `uid`. The names of different event descriptors in the same specification must be different; otherwise, a syntax error results when the specification is parsed by Yeast. The specification must be corrected and resubmitted via the `addspec` client (c.f. Section 4.

5.1 Using Event Contexts in the Action

The contextual information of an event can be used in the action portion of the specification by referring to the name of the descriptor that matched the event; the specific contextual element is supplied within '%%' marks, as follows:

> addspec N1: *e1* and N2: *e2*
> do ... %N1:uid% ... %N2:hostname% ...

where both *e1* and *e2* are event descriptors that match announced events, and `N1` and `N2` are names that will be bound to the events that match *e1* and *e2*, respectively. Once an announced event matches either *e1* or *e2*, the values of the elements of the event's context are replaced in the string that represents the action part.

An anomaly can result when there is an `or` combinator in the event pattern. Consider, for example, the following specification:

> addspec N1: *e1* or N2: *e2*
> do notify ... %N1:uid% ... %N2:uid% ...

Say that an announced event that is matched by *e1* is received. Since the combinator used in the event pattern is `or`, matching *e1* is sufficient to trigger the action. In this case, *e2* would not have been matched, and thus Yeast would not be able to replace `%N2:uid%` with an actual value.

There are three options to resolve this anomaly: (1) to detect and disallow such anomalies at the time the specification is added, (2) to impose an `and` semantics on the event pattern, or (3) to abort the action when an unbounded event name occurs in the action. The first option is too restrictive since it disallows some correct specifications. The second option alters the semantics of the specification, which is undesirable. The third option ensures that incorrect results are not obtained without restricting the specification writer; thus, we have decided to use the third option in the implementation.

5.2 Attaching Other Information to Named Descriptors

We used the event descriptor names to convey two other pieces of information that are not part of the context, but that are useful nonetheless. As explained in Section 4, the content of an event in Yeast consists of an object, an attribute and its value. Basically, the content of an announced event in Yeast assigns a value to a single attribute of an object. The Yeast language lacks variables to which the object and value can be bound. Therefore, it is not possible to refer to the object and the value in the action part of a specification.

Consider, for example, the following specification:

> addspec file /bin/% syscall == write
> do notify a file in /bin was modified

The event descriptor contains a wildcard character (`%`), and thus it is matches any file modification event on any file in the directory */bin*. In particular, it matches the following two announced events:

> announce file /bin/yeast syscall = write
>
> announce file /bin/nmake syscall = write

However, Yeast cannot notify the application which file was actually modified (`yeast` in the first case and `nmake` in the second) because there is no way to refer to the actual object in the action part.

By attaching the name of the object and the value to the context, we can pass both to the action part of the specification. Consider the following improvement on the specification that monitors completion of testing of tools in the */bin* directory:

```
addspec N1: file /bin/\% syscall == write\\
      do notify Tool \%N1:object\% was modified\\
              by \%N1:uid\% on \%N1:hostname\%
```

When this specification matches a particular event, the notification program `notify` will have access to the actual name of the tool that was modified, the `uid` of the user who modified it, and the `hostname` of the machine on which the modification was made. These three pieces of information can now be used in the notification program to send a more useful notification than the generic one we had before.

6 Using Event Contexts to Constrain Matching

In addition to being used in the action part of a specification, the values of the context elements can be used to relate event descriptors in the same complex event pattern, albeit in a limited way. In particular, the values of the context element of an announced event that has already matched an event descriptor can be used to constrain matching of other unmatched event descriptors.

Consider the following example. Say that a system *db* is composed of two executables: *client* and *server*. Yeast is used to notify the users of *db* whenever a new version of the *client* and *server* have been tested on the same machine. The following specification is written for this purpose:

```
addspec file /db/client tested == true and
           file /db/server tested == true
     do notify The system db has been tested
```

The specification sends a notification when the two parts of *db* have been tested (detected via two announced events to the effect). However, it cannot guarantee that the two events were generated on the same machine. This does not satisfy the original intention, which was to make sure that the *client* and the *server* have been tested on the same machine before sending the notification. This effect can be achieved by constraining the matching of the event descriptor *file /db/server tested == true* so that it matches only the corresponding event if the event was generated from the same machine as the event that matched the first descriptor.

We extended the Yeast language with *matching constraints* to achieve this limited form of relating event descriptors to each other. The simplest form of a matching constraint expression is:

> *N*: {*name*:*context_element relop* $\hat{c}$*onstant*}
> *event-descriptor ...*

where *name* is the name of an event descriptor in the specification (e.g., *N*), *context_element* is one of either the three context elements (*pid*, *uid*, *hostname*) or the attached elements (*object*, *value*), and *relop* is a relational operator (==, !=, <, >, <=, or >=). Syntactically, the name of the event descriptor followed by a ":", must always precede the constraint expression. The constant must be of the same type as the context element: integer for *pid*, string for *uid*, *hostname* and *object*, and arbitrary for *value*, depending the type of the attribute. Yeast performs type checking on all matching constraint expressions.

For an event descriptor to match an event, the values of its context must satisfy the constraint expression attached to the descriptor. For example, the matching constraint in the specification

```
addspec E1: \{E1:uid == pjw\} {\em eventdescriptor}
```

specifies that for an event to match the *event-descriptor*, the *uid* element of the context of this event must have the value *pjw*. In other words, only events generated by the user *pjw* will be considered for matching.

A slightly more complicated form is:

```
addspec N: \{ {\em name1}:{\em elt1 relop name2:elt2} \}
```

where *name1* and *name2* are the names of event descriptors in the specification; the descriptors associated with *name1* and *name2* are expected to match before this one (explained later). Elements *elt1* and *elt2* must be of the same type.

Simple matching constraints can be combined with `and` and `or` to form more complex matching constraints, for example:

> addspec E1: *e1* and E2: { E2:uid == E1:uid and E2:value $< x$ } *e2*
> do *action*

Constrained event matching provides us with a mechanism to address the shortcoming of the specification above that notified users when both parts of the system *db* were tested on a particular machine. The revised specification is:

```
addspec E1: file /db/client tested == true
      and  E2: {E2:hostname == E1:hostname}
            file /db/server tested == true
     do notify db tested on %E1:hostname%
```

The expression { `E2:hostname == E1:hostname` } is a matching constraint that achieves the desired effect of sending a notification only when both the *client* and the *server* have been tested on the same machine.

Matching constraints are associated only with primitive event descriptors; it is not possible, for example, to constrain a complex event pattern. If it were possible to associate a constraint with a complex event pattern, then Yeast would have to maintain multiple instances of partial matches of that pattern, and whenever there is a potential for a full match, the constraint would have to be checked for satisfaction. This has the potential for a combinatorial explosion in the number of partial matches that have to be maintained.

7 Application in Provence

The ability to use event contexts in the action part of specifications and in matching constraints is useful for a variety of applications, in particular ones that maintain histories of streams of related events. For example, capturing and using the value of the context element `hostname` is useful in network-based applications that need to distinguish between different machines on a network from which announcements can be generated. The `uid` element can be used in multi-user applications to identify the source of generated events and constrain matching of these events, especially when multiple users are authenticated to announce events on user-defined attributes.

The application that motivated this work is Provence, an open process-centered environment for modeling, monitoring and enacting software processes [KB93]. In Provence, a specification of a project's process model is supplied as input to the process server. The process model describes the tasks that must be performed and the dependencies among the tasks. Provence monitors the execution of a software process by translating high-level process tasks into low-level event-action specifications involving file-based events and tool invocation events that are relevant to the process. The occurrence of a sequence of low-level events indicates the performance of a particular task in the process. For example the sequence of invoking an editor, opening the file that contains the project's requirements from within the editor, modifying the contents of the file, and closing the file is equivalent to performing the "modify requirements" task.

The event-action specifications are registered with Yeast, which is responsible for notifying the process server whenever a particular sequence of events occurs. Yeast itself is notified about the occurrences of events involving files, directories or tools via announcements from the n-Dimensional File System (n-DFS) [FKR94]. One of the services provided by the *n-DFS* is generating announced events regarding opening, closing, creating and unlinking of files, and execution of tools.

The event-action specifications generated by Provence use both named context elements and constraints. The following are typical specifications generated by Provence:

```
addspec repeat e1: file /bin/emacs syscall == exec
      do notify tool_exec /bin/emacs %e1:hostname% %e1:pid% %e1:uid%

addspec repeat e1: file /sys/require syscall == open
      then e2: { e2:hostname == e1:hostname and e2:pid == e2:pid }
                 file /sys/require syscall == write
      then e3: { e3:hostname == e1:hostname and e3:pid == e2:pid }
                  file /sys/require syscall == close
      do notify file_write /sys/require %e1:hostname% %e1:pid% %e1:uid%
```

The first specification simply monitors the invocation of the text editor Emacs. Similar specifications are generated for every tool that is deemed (by the process model) relevant to the workflow process under consideration. The second specification monitors the modification of the file `/sys/require` by a single user process. A similar specification is generated for every file that must be monitored. The *n-DFS* automatically makes the appropriate announcements to match these specifications when the corresponding events occur.

The second specification is insufficient if multiple users are expected to access the same file simultaneously. The reason is that Yeast does not support concurrent instances of the same specification; instead the "repeat" keyword causes Yeast to reinsert (resubmit) the specification only *after* it has been completely matched and its action triggered. When a user process opens the requirements file to modify it, the *n-DFS* generates an announcement of the form:

```
announce file /sys/require syscall = open
```

Thus, if more than one user process opens the requirements file for writing, a multiple of these announcements will be made. However, only the one that reaches Yeast first will match the first event descriptor of the second specification; since the event pattern of the specification will not be fully matched at that point, the rest of the announcements will be ignored. The specification will not fully match until the same user process writes and then closes the requirements file; only then will the action be triggered and the specification re-inserted. At that point, other announcements about the file being opened may match the first descriptor of the specification.

To avoid such situations, the second specification can be re-written as two specifications:

addspec repeat e1: **file** /sys/require syscall == open
do notify file_open /sys/require
    %e1:hostname% %e1:pid% %e1:uid%

addspec repeat e2: { e2:hostname == ¡hostname¿
and e2:pid == ¡pid¿ } **file**
    /sys/require syscall == write
then e3: { e3:hostname == *hostname*
    and e3:pid == *pid* }
file /sys/require syscall == close
do notify file_write /sys/require
%e1:host% %e1:pid% %e1:uid%

where the second specification is generated and registered by Provence only when it is notified via the matching of the first specification, and where <*hostname*> and <*pid*> are the hostname and the process identifier elements of the context of the open file event that matches the first specification. This will make sure that a specification similar to the second is registered every time the requirements file is opened by a user process, even if multiple user processes open the file simultaneously.

8 Discussion and Conclusions

We introduced two language constructs, event context and matching constraints, for overcoming two limitations of existing event monitoring systems: (1) the inability to capture the context in which a detected event occurred; and (2) the lack of mechanisms for relating different event descriptors in the same event pattern by common variables or constraints. We described the implementation of these constructs as an extension to an existing event monitoring system called Yeast. The extended event monitoring system is capable of capturing the context in which a matched event occurs, and using the information contained in the context to constrain further matching of events.

Extending Yeast with matching constraints introduced two interesting anomalies that have to do with the order in which events are matched. The first anomaly is that constraint satisfaction imposes an ordering on event matching even if the logic in the event pattern does not impose an ordering. The second is that ordering imposed by constraint satisfaction might contradict the ordering implied by the logic in the event pattern.

The first anomaly involves `and` and `or` combinators, which do not impose a specific ordering on event matching in a single event pattern. Adding constraints might alter that. Consider, for example, the following specification:

> addspec E1: { E1:uid == E2:uid } *e1* and
> E2: *e2* ...

The matching of the event descriptor *e1* is constrained based on the value of the `uid` element of the context of the announced event that matches *e2*. There are two steps in satisfying this constraint. First, the values of the elements of the named context `E2` must be *bound* before they can be used. Second, if the event descriptor *e1* matches the announced event, before the match is considered successful, the value of the `uid` element of the event's context must be equal to the value of the `uid` element of `E2` (which is treated as a constant in this operation). Thus, Yeast must first match the event descriptor *e2* before it can match *e1*.

The second anomaly, which is more serious, results from conflicting matching orders. As discussed earlier, the `then` sequencing combinator imposes a matching order, where the pattern before `then` had to be matched before the pattern after `then` was even considered for matching. This matching order might contradict the matching order required for constraint satisfaction. Consider the following specification:

addspec E1: { E1:uid == E2:uid } *e1* then E2: *e2* ...

The `then` combinator requires that *e1* be matched before even considering matching *e2*. However, constraint checking requires the reverse ordering. The result is that this specification will never match.

Another conflict in the matching order results from circularity of constraints. For example, the specification

addspec E1: { E1:uid == E2:uid } *e1* and E2: { E2:uid == E1:uid } *e2* ...

will never match because the matching constraint on `E1` requires that *e2* be matched before *e1* and the matching constraint on `E2` requires the reverse.

The current implementation cannot detect matching order conflicts at the time the specification is submitted (this is very difficult in the general case, especially when there are more than two constraints involved). Therefore, we decided to allow such specifications in the current implementation.

Notwithstanding these two anomalies, the two extensions we have designed and implemented enhance the expressive power of the Yeast specification language, making Yeast better suited for applications that monitor sequences of events. The process-centered environment Provence is just one such application. The extensions are also useful for applications that collect information about events in a distributed environment and applications where security is important. In future, we plan to look for additional applications for the current set of contextual elements as well as look for other contextual elements.

Acknowledgments

We thank David Rosenblum for suggesting improvements in the syntax and semantics of matching constraints. We also thank Emden Gansner for clarifying some of the concepts involved in constraints. Finally, we thank the anonymous reviewers for their excellent corrections and comments.

References

[CKAK93] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K Kim. Anatomy of a Composite Event Detector. In *Proc. of the 20th International Conference on Very Large Databases*, Santiago Chile, 1993.

[FKR94] Glenn Fowler, David Korn, and Herman Rao. *n-DFS: The Multiple Dimensional File System*, volume 2 of *Trends in Software*, pages 135–154. John Wiley & Sons, Chichester, 1994. B. Krishnamurthy (series ed.).

[GI90] D. Garlan and E. Ilias. Low-cost, adaptable tool integration policies for integrated environments. In R. N. Taylor, editor, *Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 1–10, Irvine CA, December 1990. ACM Press. Special issue of *Software Engineering Notes*, 15(6), December 1990.

[GJ91] N.H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proc. of the 17th International Conference on Very Large Data Bases*, pages 327–336, Barcelona, Spain, 1991.

[GL81] H.J. Genrich and K. Lautenbach. System modelling with high-level petri nets. *Theoretical Computer Science*, 13:109–136, 1981.

[KB93] B. Krishnamurthy and N. Barghouti. Provence: A Process Visualization and Enactment Environment. In *Proc. of the Fourth European Conference on Software Engineering*, pages 151–160, Garmisch-Partenkirchen, Germany, September 1993. Springer-Verlag. Published as *Lecture Notes in Computer Science* no. 717.

[KK92] Eduardo Krell and Balachander Krishnamurthy. COLA: Customized Overlaying. In *USENIX San Francisco Winter 1992 Conference Proceedings*, pages 3–7, 1992.

[KR91] Balachander Krishnamurthy and David S. Rosenblum. An event-action model of computer-supported cooperative work: Design and implementation. In K. Gorling and C. Sattler, editors, *International Workshop on Computer Supported Cooperative Work*, pages 132–145. IFIP TC 6/WG C.5, 1991.

[LiMIT93] Hajimu Lida, Kei ichi Mimura, Katsuro Inoue, and Koji Torii. Hakoniwa: Monitor and Navigation System for Cooperative Development Based on Activity Sequence Model. In *Proc. of the Second International Conference on the Software Process*, Berlin, Germany, February 1993. IEEE Computer Society Press.

[Mor83] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Ninth International Conference on Very Large Data Bases*, Florence Italy, October 1983. Morgan Kaufmann.

[PSE+94] N. Pissinou, R. Snodgrass, R. Elmasri, I.S. Mumick, M.T. Ozsu, B. Pernici, A. Segev, and B. Theodoulidis. Towards an infrastructure for temporal databases: Report of an invitational arpa/nsf workshop. *SIGMOD Record*, 5(1–2):1–12, June 1994.

[Rei90] S. P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57–66, July 1990.

[RK91] David S. Rosenblum and Balachander Krishnamurthy. An event-based model of software configuration management. In *3rd International Workshop on Software Configuration Management*, pages 94–97. ACM SIGSOFT, 1991.

[UDL90] M. Hsu U. Dayal and R. Ladin. Organizing long-running activities with triggers and transactions. In *ACM SIGMOD International Conference on the Management of Data*, pages 204–214, Atlantic City NJ, May 1990. ACM Press.