Yeast: A General Purpose Event-Action System

Balachander Krishnamurthy and David S. Rosenblum, *Member, IEEE*

Abstract—**Distributed networks of personal workstations are becoming the dominant computing environment for software development organizations. Many cooperative activities that are carried out in such environments are particularly well suited for automated support. Taking the point of view that such activities are modeled most naturally as the occurrence of events requiring actions to be performed, we have developed a system called Yeast (Yet another Event-Action Specification Tool). Yeast is a client-server system in which distributed clients register event-action specifications with a centralized server, which performs event detection and specification management. Each specification submitted by a client defines a pattern of events that is of interest to the client's application plus an action that is to be executed in response to an occurrence of the event pattern; the server triggers the action of a specification once it has detected an occurrence of the associated event pattern. Yeast provides a global space of events that is visible to and shared by all users. In particular, events generated by one user can trigger specifications registered by another user. Higher-level applications are built as collections of Yeast specifications. We use Yeast on a daily basis for a variety of applications, from deadline notification to software process automation. This paper presents an in-depth description of Yeast and an example application of Yeast, in which Yeast specifications are used to automate a software distribution process involving several interdependent software tools.**

Keywords—**Computer networks, distributed computing, event-action system, event models, software development environments, software process, specifications.**

I. Introduction

An *event-action system* is a software system in which events occurring in the environment of the system trigger actions in response to the events. The triggered actions may generate other events, which trigger other actions, and so on. A wide variety of software applications can be naturally characterized as event-action systems.

Most existing event-action systems are special-purpose systems that support a particular application domain. Good examples of such systems include *tool integration systems* (such as Field [1], and a commercial version of Field called the HP SoftBench[1] Development Environment [2]), *active databases* (such as ODE [3] and AP5 [4]), *rule-based development environments* (such as Marvel [5],[6]), and *software process monitoring systems* (such as Amadeus [7]).

Given the prevalence of event-action processing within a multitude of applications, it would be desirable to make available a general-purpose event-action component that can be easily integrated with any application that needs such a facility, thus obviating the need for each application developer to develop such a capability from scratch. However, in order for such an event-action system to be fully general, it must satisfy a number of requirements:

1. The system must be open, in the sense that it has knowledge about external events and can be used in conjunction with any other tool chosen by its users.
2. There should be no restriction on the actions that can be performed in response to the occurrence of user-specified event patterns.
3. The specification language must be simple, yet powerful.
4. The system must be able to handle both temporal and non-temporal events.
5. The system must be extensible, in the sense that users can define new kinds of events to the system.
6. The system must be reliable, with a state that persists across machine crashes.
7. Users of the system must be able to interactively query the status of specifications they have registered with the system.
8. User interactions with the system must be authenticated to ensure the security of the system and the privacy of user interactions.

A system that satisfies these requirements pro-

The authors are with the Software and Systems Research Center at AT&T Bell Laboratories, Murray Hill, NJ 07974, USA. E-mail: {bala,dsr}@research.att.com

[1]SoftBench is a trademark of the Hewlett-Packard Company.

vides a general event-action capability that can be easily and reliably integrated with applications that need its services. The key requirements for supporting ease-of-integration are the first two listed above, namely openness of the system and the absence of restriction on actions. These key requirements, as well as some of the others listed above, have not been met by previous systems.

In this paper we describe a system called **Yeast** (**Y**et another **E**vent-**A**ction **S**pecification **T**ool) that to a large degree satisfies these requirements. Yeast is a general-purpose platform for constructing distributed event-action applications using high-level event-action specifications. Yeast can support a wide variety of event-action applications, including calendar and notification systems, computer network management, software configuration management, software process automation, software process measurement, and coordination of wide-area software development. Yeast enhances and generalizes the capabilities of previous event-action systems in several ways—by supporting automatic recognition of a rich collection of predefined event classes, by providing extensibility in the form of user-defined events, and by providing a general, application-independent encapsulation of the event-action model.

We begin in Section II with a description of the architecture and operation of Yeast. Section III describes the features of Yeast in detail. Section IV describes a large-scale application of Yeast, in which a collection of Yeast specifications is used to provide automated support for a software distribution process. Section V presents implementation and performance details. Section VI discusses related work in more detail. We conclude in Section VII with a discussion of the extent to which Yeast satisfies the requirements enumerated above, and with a discussion of our future plans for Yeast.

II. Architecture and Operation of Yeast

In this section we present a high-level view of the architecture of the Yeast system and its basic operation. Figure 1 depicts the architecture of Yeast.

As shown in the figure, Yeast is a client/server system. The server is a central entity accepting client commands from a number of (possibly remote) users. The primary function of the server is to accept, match and manage *specifications* on behalf of users. A specification describes a pattern of events that the user is interested in as well as the action to be triggered by Yeast when it has detected a match for the event pattern. The user invokes client commands through the computer system's command interpreter (such as the UNIX® shell); the interaction between user and server during client command invocation is synchronous and interactive. Client commands are used to register specifications with the server and to perform various definition, query and specification management chores. The server and client programs can reside anywhere on the network. The client commands are described in further detail in Section III-C.

Each Yeast specification comprises an event pattern and an action. The event pattern is a compound pattern of *primitive event descriptors*, each of which *matches* a corresponding primitive event (i.e., a single event occurrence). An event descriptor matches an event either *transiently* or *permanently.* In particular, one can specify an event descriptor that matches during some periods of time and not during other periods of time.

For example, one can specify an event descriptor that is to be matched whenever the load on a particular computer host exceeds some threshold value.[2] Such a descriptor only matches during those intervals of time when the load actually exceeds the specified threshold value. Thus, such a descriptor is said to match transiently.

On the other hand, one can specify an event descriptor that stays matched once a matching event for the descriptor is detected. For example, one can specify an event descriptor that is to be matched whenever a specified date and time have been reached. Once the date and time have been reached, the date and time will always be in the past and thus will always have been reached at any future date and time. Thus, such a descriptor is said to match permanently.

Once the server has detected a match for the

®UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/OPEN corporation.

[2]Section III presents details on the different kinds of primitive event descriptors that can be specified.

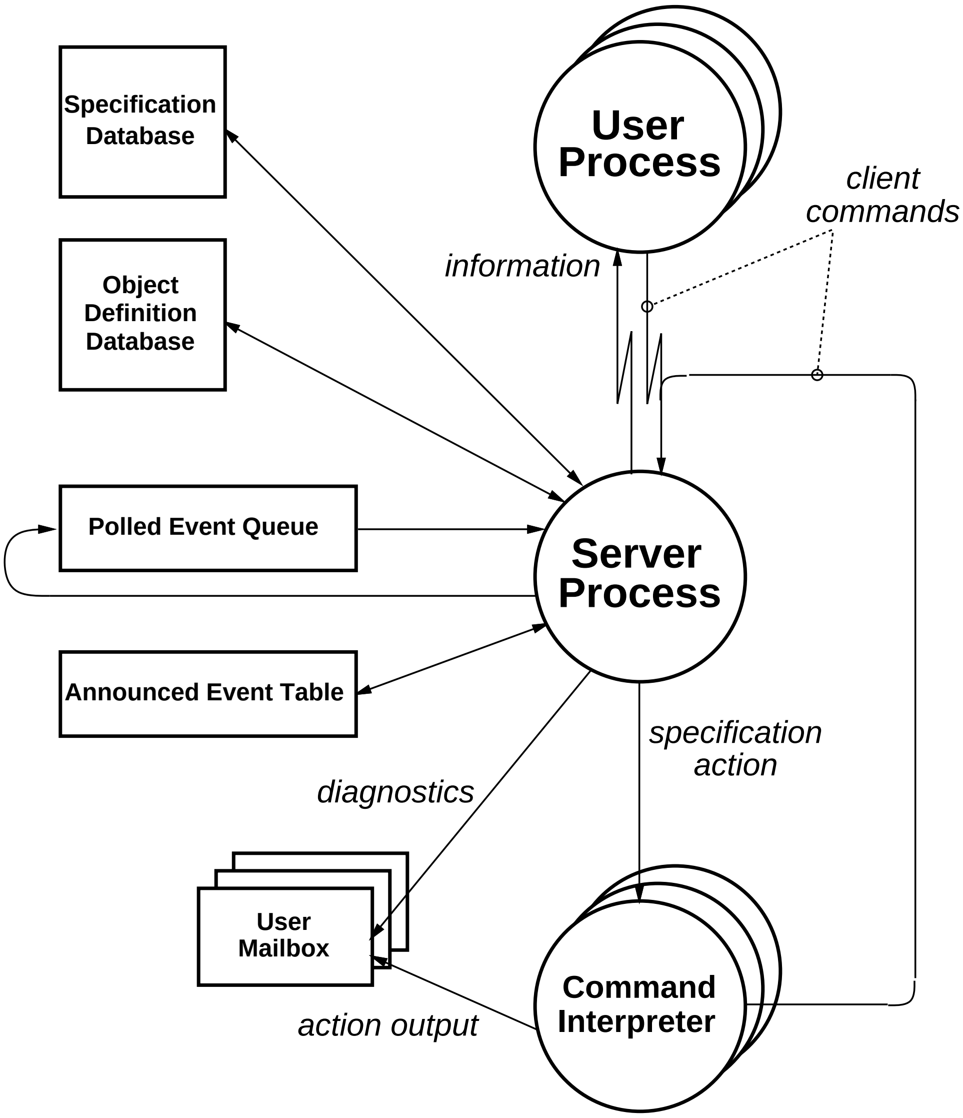

Fig. 1. Architecture of the Yeast System.

event pattern of a specification, it ceases any further matching activity on the event pattern, and it triggers the associated action. The action of a specification is any valid sequence of commands that can be executed by the computer system's command interpreter, including Yeast client commands. The server invokes the computer system's command interpreter to execute the action component of a specification at the earliest possible time after it has matched the event pattern of the specification. The action is executed on the host on which the server is running; however, the action itself can explicitly invoke remote execution commands, such as the UNIX command `rsh`, to execute all or part of the action on remote machines.

Upon termination of the action, any output that was not otherwise directed to files or piped to other commands is sent by electronic mail to the user who registered the specification. In addition, because the user interacts with the server solely through brief client command invocations, any other specification-related information the server must communicate to the user—such as problems that arise during specification matching, or output from a specification action—is also sent via electronic mail to the user who registered the specification.

In Yeast's model of events, a primitive event corresponds to a change in the value of an attribute of an object belonging to some object class. Because of the importance of time events and the special syntax for their descriptors, we divide primitive events into two classes:

- *time events*, which involve clock times, dates and time intervals; and
- *object events*, which involve changes to non-temporal objects.[3]

Primitive events are further characterized as being either *predefined* or *user-defined*. Predefined primitive events can be automatically detected by the server; in particular, the server polls the system environment for their occurrences. Predefined events are matched by invoking operating system routines and are thus closer to operating system events. User-defined primitive events must be announced to the server by a user or by some program, because the server has no information about the semantics of user-defined events in order to detect their occurrences automatically. Predefined primitive events involve predefined object classes and attributes that the server can poll; predefined events include both time events and object events. User-defined primitive events involve user-defined attributes of user-defined or predefined object classes and are thus all object events. It is possible to write specifications whose event patterns will never match once certain kinds of time event descriptors stop matching.[4] If the server detects that a specification is unmatchable, the server removes the specification, and notifies the user who registered the specification.

The server stores a specification's event descriptors for predefined primitive events in the Polled Event Queue (shown in Figure 1) in the order in which the server will next poll for a match. The server stores a specification's event descriptors for user-defined primitive events in the Announced Event Table, where the descriptors are checked as the server receives announced events.

The server stores specifications and their associated information (such as the login ID of the owner of the specification) in the Specification Database, for as long as the specifications are active. The Specification Database includes a persistent copy of the specification stored in the file system. Whenever the server is restarted after a machine crash, the server re-registers the file-system copies of specifications that were active at the time of the crash. The server stores the definitions of object classes and attributes in the Object Definition Database. This latter database contains the definitions both of object classes and attributes that are predefined at the time of starting the server, and object class and attribute definitions that are defined by users. The set of predefined object classes can vary from machine to machine and is dependent on the resources that are available in the computer system on which the server executes.

We observe in passing that Yeast only requires a rather simple object model; especially notable is the lack of inheritance in the model. After extensive study of the impact of adding inheritance to the model, we felt that adding inheritance would unnecessarily complicate the event-matching semantics, without greatly adding to the expressive needs of most applications.

III. Features of Yeast

In this section we describe the language of Yeast specifications, including a description of the predefined object classes and attributes. We then describe the client commands that are used to interact with the Yeast server. We conclude the section with a sample scenario of user interaction with Yeast.

A. The Yeast Specification Language

A Yeast specification consists of an event pattern along with an action, written with the following syntax:

event_pattern **do** *action*

The event pattern contains primitive event descriptors formed using the connectives **then** ("sequence-of"), **and** ("all-of") and **or** ("one-of").

A.1 Time Event Descriptors

A time event descriptor matches the passage of a *relative* amount of time (e.g., 20 minutes from now) or the occurrence of an *absolute* time (e.g., 7 am Monday). Some time event descriptors match permanently while others match transiently (as defined in Section II).

[3] In terms of our model of events as being changes to attributes of objects, time events involve an object class that has only one object (the system clock) with one attribute (the current time).

[4] Such time event descriptors are described more fully in Section III.

Relative time event descriptors are specified by the keywords **in** and **within**. The relative times can be specified in days, hours, minutes and seconds. An **in** event descriptor matches permanently *after* the specified time has elapsed, while a **within** event descriptor matches only *until* the specified time has elapsed (and thus matches transiently). The following are some examples of relative time event descriptors:

- **in** 2 hours 10 minutes—matches permanently after 2 hours and 10 minutes have elapsed from now.
- **within** 6 days 10 hours—matches from now until 6 days and 10 hours have elapsed.

Absolute time event descriptors are specified by the keywords **at** and **by**. An **at** event descriptor matches permanently *after* the specified time has been reached, while a **by** event descriptor matches *until* the specified time has been reached (and thus matches transiently). Absolute times must at least specify a time of day, with an optional day of week or date; the optional date can specify either a day, a month and a day, or a month, day and year. Absolute time specifications implicitly specify the next occurrence of the specified time.[5] The following are some examples of absolute time event descriptors:

- **at** 8am—matches permanently after the next occurrence of the time 8 am.
- **by** 8am—matches from now until the next occurrence of 8 am.
- **at** 8am saturday—matches permanently after the next occurrence of 8 am on a Saturday.
- **at** 8am 31—matches permanently after 8 am on the last day of the current month.
- **by** 8am august 31—matches from now until 8 am on the next occurrence of August 31.
- **at** 8am august 31 1960—matches permanently from now, since the specified time and date have already passed.
- **by** 8am august 31 1996—matches from now until 8 am on August 31, 1996.

Absolute time event descriptors can be modified by one of the following modifiers, which constrain matching to individual days:

- **daily**, **today** and **tomorrow**, which are used only with a time of day;
- **weekly**, used with a day of week;
- **monthly**, used with a day of month; and
- **yearly**, used with a month and day of month.

The modifiers have their obvious meaning, with the further constraint that **at** event descriptors match between the specified time and midnight at the end of the day on matching days, while **by** event descriptors match between midnight at the beginning of the day and the specified time on matching days. Note that modified absolute time event descriptors match transiently. The following are some examples of modified absolute time event descriptors:

- **by** 10pm **today**—matches between now and 10 pm today, or never if it is after 10 pm today.
- **at** 8am **daily**—matches between 8 am and the end of the day every day.
- **by** 8am saturday **weekly**—matches between the beginning of the day and 8 am every Saturday.
- **at** 8am 31 **monthly**—matches between 8 am and the end of the day on the last day of every month (including months with less than 31 days).
- **by** 8am dec 31 **yearly**—matches between the beginning of the day and 8 am every December 31st.

A.2 Object Event Descriptors, Object Classes and Attributes

Object event descriptors use a relational test to specify a change in the value of an attribute of an object. They have the following syntax:

obj_class obj_name obj_attr relational_test

The *obj_class* and *obj_attr* must either be predefined or else must have been defined to Yeast using the client commands **defobj** and **defattr** (described below). The *relational_test* of an object event descriptor is a test against the value of the specified *obj_attr* of the specified *obj_name* at the time a match of the descriptor is attempted. The special relational tests **changed** and **unchanged** are available for some predefined attributes. The *obj_attr* has a type, which is one of the predefined types:

- **boolean**;
- **integer**;
- **procstatus** (status values of operating system processes);
- **real**;

[5] In Section III-A.3 we describe in more detail what "now" and "next" mean to the Yeast server.

- **reltime** (relative times);
- **string**; and
- **systime** (unmodified absolute times).

Depending on the choice of *obj_class*, *obj_attr* and *relational_test*, an object event descriptor matches either permanently or transiently (as defined in Section II).

Table I lists the predefined object classes and their predefined attributes. The following examples illustrate some event descriptors involving the predefined object classes and attributes:

- **file** foo mtime > 8am Oct 1 1995—matches permanently once file `foo` has been modified after 8 am on October 1, 1995.
- **file** foo mtime **changed**—matches permanently after the next time file `foo` has been modified.
- **dir** foo count == 20—matches transiently whenever the number of files in directory `foo` is exactly 20.
- **filesys** /t1 capacity >= 98—matches transiently whenever 98% or more of the capacity of file system `/t1` is in use.
- **user** dsr@research loggedin == true—matches transiently whenever user `dsr` is logged in on host `research`.
- **host** research load < 2.0—matches transiently whenever the load on host `research` is less than 2.0.
- **process** emacs.bala@research size > 10000—matches transiently whenever the memory usage of any process named `emacs` running on host `research` and owned by user `bala` exceeds 10,000 kilobytes.

All **file**, **dir** and **tty** objects named without full pathnames are implicitly prefixed by the current working directory that was in effect at the time the enclosing specification was registered. Event descriptors involving predefined attributes of the object class **file** have a special semantics when the specified file is a directory. In particular, the event descriptor is matched if it matches either for the directory itself or for any of the files contained in the directory. This semantics applies only to the top-level directory, not recursively to the complete subdirectory structure.

A.3 Compound Event Descriptors

Compound event patterns are formed using three connectives, which, in order of decreasing priority, are **then**, **and** and **or**. Parentheses can be used to enforce any desired grouping. A compound event descriptor combined with **and** is matched whenever all the constituent event descriptors match at the same time. A compound event pattern combined with **or** is matched whenever any of the constituent event descriptors match. In the case of a **then** connective, the server matches the event pattern on the left side of the operator **then** before it attempts to match the event pattern on the right side; only after the right side is matched is the complete pattern considered to be matched.[6] These semantics are fairly straightforward, although there are some additional subtleties in the matching of **and** and **then** compound event descriptors.

Yeast associates a *reference time* with each specification; we alluded to the reference time informally in the above examples with phrases such as "now" and "next occurrence". The reference time is either the time the specification was registered or the last time the left-hand side of a **then** compound event descriptor was matched, whichever is later. Yeast matches time event descriptors, and object event descriptors whose relational test is **changed** or **unchanged**, relative to the reference time.

For example, consider the following specification:

file foo mtime **changed then in** 10 min **do** echo hello

The event pattern of this specification is a sequence of two primitive event descriptors. The **file** event descriptor matches the first time file **foo** is modified after the specification is registered with the Yeast server. The counting of the 10 minutes in the **in** event descriptor then begins once that modification has been detected, not at the time the specification is registered. Once those 10 minutes have elapsed, the whole event pattern is then considered to be matched, and the action is triggered.

A pair of event descriptors joined by **and** will match only if the constituent event descriptors match *at the same time*. For example, consider

[6] For the purposes of matching the right side of the operator **then**, the left side is considered to match permanently the first time a successful match is detected.

TABLE I
THE PREDEFINED OBJECT CLASSES AND ATTRIBUTES OF YEAST

Object Class	Attribute	Description
`dir` (directories in the file system)	`atime`	last access time
	`count`	number of files in the directory
	`mode`	access permissions
	`mtime`	last modification time
	`owner`	login ID of owner
`file` (files in the file system)	`atime`	last access time
	`mode`	access permissions
	`mtime`	last modification time
	`owner`	login ID of owner
	`size`	number of bytes in the file
`filesys` (mounted file systems)	`capacity`	percentage of total capacity in use
	`size`	total capacity in kilobytes
`host` (named computer hosts)	`load`	load average
	`up`	whether or not the host is operational
	`users`	number of users logged on
`process` (operating system processes)	`etime`	elapsed clock time
	`size`	kilobytes of memory used
	`status`	execution status
	`stime`	CPU time in privileged mode
	`utime`	CPU time in user mode
`tty` (terminal devices)	`mode`	access permissions
	`mtime`	last modification time
`user` (user login IDs)	`location`	tty of login session
	`loggedin`	whether or not the user is logged in

the following two compound event descriptors:

in 10 minutes **and host** research load $>$ 5.0

within 10 minutes **and host** research load $>$ 5.0

The first compound descriptor contains an **in** time event descriptor, which matches permanently after the specified time has elapsed. Thus, the first compound descriptor will match only if the load on host `research` exceeds 5.0 some time *after* 10 minutes have elapsed, regardless of whether or not the load exceeded 5.0 before the 10 minutes have elapsed. On the other hand, the second compound descriptor contains a **within** time event descriptor, which matches transiently until the specified time has elapsed. Thus, the second compound descriptor will match only if the load on host `research` exceeds 5.0 some time *before* 10 minutes have elapsed; once the 10 minutes have elapsed without the load exceeding 5.0, the second compound descriptor will never match.

B. Actions

The action portion of a Yeast specification, which is triggered by the server when the event pattern has matched, is any valid sequence of commands that can be executed by the computer system's command interpreter. The syntax and semantics of actions are thus defined by the command interpreter. Each specification is stored with the user's environment information that was in effect at the time the user registered the specification. The action is executed using this associated environment information. In a UNIX system this environment includes the user's command search path, current working directory, alias list, and so on. Note that this environment information may not be sufficient to enable successful execution of the action; for example, actions requiring interactive input will fail. If the action produces any output that is not otherwise directed to files or piped to other commands, by default it is mailed to the user who registered the specification. Note that because Yeast client commands are invoked through the command interpreter, the action of a specification can invoke Yeast client commands.

C. Client Commands

Users interact with the Yeast server through a collection of client commands that are invoked through the computer system's command interpreter. The Yeast client commands can be categorized as follows:

1. commands for registering new specifications (**addspec** and **readspec**);
2. commands for defining new object classes and attributes (**defobj** and **defattr**);
3. a command for generating events involving user-defined object classes or attributes (**announce**);
4. commands for manipulating registered specifications (**lsspec**, **rmspec**, **fgspec**, **suspspec** and **modgrp**);
5. commands for manipulating object classes and their attributes (**lsobj**, **rmobj**, **lsattr** and **rmattr**);
6. commands for controlling access to object classes and attributes (**authobj**, **authattr** and **lsauth**); and
7. commands for registering and unregistering users with Yeast (**regyeast** and **unregyeast**).

Users must initially register themselves with the Yeast server via the client command **regyeast** before they can carry out any other client interactions.

C.1 Registering Specifications

Users register specifications with Yeast via the client command **addspec**, which has the following syntax:[7]

addspec [repeat] {*+group_name*} *Yeast_spec*

The syntax of the *Yeast_spec* was described in Section III-A. The optional specifier **repeat** indicates that the specification is to be immediately re-registered with the Yeast server whenever the server matches the event pattern and triggers the action (or whenever the server removes the specification because it is unmatchable).

A specification can optionally be given one or more *group_name*s to create named, logically-related groups of specifications. Group names are used in client commands that manipulate specifications (i.e., those commands in category (4) listed above) to refer to a group of specifications with a single name and to refer to specifications from within the action component of other specifications. For example, one can use a group name to name a set of specifications dealing with a particular aspect of a project; the group name could then be used to suspend matching on the group if matching on events related to that aspect needs to be temporarily stopped. The logical naming also enables specifications to be manipulated by the actions of other specifications. Group names are illustrated more fully in Section IV.

The client command **readspec** can be used to register a collection of specifications stored in a file.

C.2 Defining Object Classes and Attributes, and Announcing Events

New object classes and attributes are defined to Yeast with the client commands **defobj** and **defattr**, respectively. Events involving user-defined object classes and attributes must be announced to Yeast with the command **announce**, which has the following syntax:

announce *obj_class* *obj_name* *obj_attr* = *attr_value*

For example, we can define an attribute called `debugged` for the object class **file** and then register a specification that notifies project personnel whenever file `project.c` is debugged:

defattr file debugged boolean
addspec file project.c debugged == true
  do notify project.c debugged

The above specification would be matched when the person responsible for debugging the file `project.c` generates the following announcement

announce file project.c debugged = true

In the current implementation of Yeast, announced events match permanently.

Given the importance of announcements as the fundamental mechanism for generating user-defined events, **announce** is also available in the form of a program library routine that can be linked in with application programs that need to generate Yeast announcements. As described below in Section V-C, the library routine was especially useful in integrating Yeast with the Multiple Dimensional File System to avoid polling for file events.

[7]In describing the syntax, we use the convention that square brackets denote optional tokens, while curly braces denote tokens that can appear zero or more times.

C.3 Manipulating Specifications, Object Classes and Attributes

Several client commands are available for manipulating existing specifications, object classes and attributes.

The command **lsspec** lists a user's active specifications and shows the internal number that the Yeast server has assigned to each specification. **Rmspec** is used to remove specifications. **Suspspec** is used to suspend matching of specifications, while **fgspec** is used to resume matching of specifications. All these commands operate on both specification numbers and specification groups. Users can execute all of these specification-related commands only on the specifications they have registered via **addspec**.

Modgrp is used to add specifications to and remove specifications from specification groups. As with the other commands, users can use **modgrp** only on their own specification groups.

The command **lsobj** lists all the predefined and user-defined object classes, and the command **lsattr** lists the attributes of an object class along with their types. **Rmobj** is used to remove a user-defined object class, while **rmattr** is used to remove a user-defined attribute. **Rmobj** and **rmattr** can only be used by users who have appropriate permissions, as described next.

C.4 Controlling and Determining Access

All client interactions with the Yeast server undergo authentication in order to ensure that Yeast users do not interfere with one another (accidentally or otherwise). When a user defines a new object class via **defobj**, the new object class is owned by that user; ownership is determined likewise for attributes defined via **defattr**. An owner can use the client commands **authobj** and **authattr** to give another user one of four levels of access to an object class or attribute, respectively:

- *Read access:* The user can register specifications whose event patterns involve the object class or attribute.
- *Announce access:* The user can announce events involving the object class or attribute. Announce access includes read access.
- *Write access:* The user can define and remove attributes of the object class. Write access includes announce access.
- *Owner access:* The user can delete and remove the object class itself. Owner access includes write access.

For example, the owner of an object class might give other users announce access for a particular attribute and read access for all other attributes. The commands **authobj** and **authattr** can also be used to remove a user's access privileges. The predefined object classes and attributes are owned by Yeast, and all users are given read access to them. The client command **lsauth** can be used to list the authentication information of an object class.

D. A Sample Scenario of Client Command Invocations

In order to illustrate the effects and interactions of the various client commands, we depict in Figure 2 a typical scenario of client command invocations in a UNIX system along with the output they produce. In the figure, the numbered percent signs are the command interpreter prompt.

After registering with the Yeast server (command 1), the user invokes **addspec** (2) to register a simple specification that executes an `echo` command after a minute has elapsed. Immediately afterwards, the user invokes **lsspec** (3), which lists the user's specifications and shows that Yeast has assigned the specification the number `1`, which can be used to refer to the specification in subsequent command invocations. Giving this number as an argument to **lsspec** (4) causes **lsspec** to also print out the next time at which the server will attempt to match the specification's event pattern.

The user next gives the specification number to **suspspec** (5) to suspend matching of the specification. The output from **lsspec** (6) now has a caret following the number `1` to indicate that the specification is suspended. The user resumes matching of the specification through the invocation of **fgspec** (7), as indicated by the absence of the caret in the **lsspec** listing (8). The user waits for 60 seconds (9) and then invokes **lsspec** again (10). The output from **lsspec** is now empty, indicating that the event pattern of the specification has been matched and its action triggered. The user will have received electronic mail containing the output of the `echo` command in the action of the specification.

```
1% regyeast
you have been registered with yeast
2% addspec in 1 minute do echo 1 minute elapsed
3% lsspec
 1   addspec in 1 minute do echo 1 minute elapsed
4% lsspec 1
 1   addspec in 1 minute do echo 1 minute elapsed
Will attempt match at Sun Jan 1 11:39:28 1995
5% suspspec 1
6% lsspec
 1 ^ addspec in 1 minute do echo 1 minute elapsed
7% fgspec 1
8% lsspec
 1   addspec in 1 minute do echo 1 minute elapsed
9% sleep 60
10% lsspec
11% addspec +g1 in 1 hour do echo 1 hour elapsed
12% addspec +g1 in 2 hours do echo time to go home
13% lsspec g1
 1   addspec +g1 in 1 hour do echo 1 hour elapsed
 2   addspec +g1 in 2 hours do echo time to go home
14% modgrp +g2 1
15% lsspec g2
 1   addspec +g1 in 1 hour do echo 1 hour elapsed
16% modgrp -g1 2
17% lsspec g1
 1   addspec +g1 in 1 hour do echo 1 hour elapsed
18% rmspec g1
19% lsspec
 2   addspec +g1 in 2 hours do echo time to go home
20% addspec tool yeast debugged == true do echo yeast is bug-free
unknown object class:  tool
21% defobj tool
22% defattr tool debugged boolean
23% addspec tool yeast debugged == true do echo yeast is bug-free
24% lsspec
 2   addspec +g1 in 2 hours do echo time to go home
 3   addspec tool yeast debugged == true do echo yeast is bug-free
25% lsspec 3
 3   addspec tool yeast debugged == true do echo yeast is bug-free
Specification waiting for announcement
26% announce tool yeast debugged = true
27% lsspec
 2   addspec +g1 in 2 hours do echo time to go home
```

Fig. 2. Sample Scenario of Client Command Invocations.

Next the user registers two specifications (11, 12) with the indication that they belong to a specification group called `g1`. Giving the name `g1` as an argument to **lsspec** (13) causes **lsspec** to list all specifications currently in group `g1`. The user invokes the command **modgrp** (14) to add specification `1` to another group called `g2`, in addition to group `g1`. As the output of **lsspec** on `g2` (15) shows, the other specification in group `g1` was not made a member of group `g2`. The user invokes **modgrp** again (16) to remove specification `2` from group `g1` without affecting the membership of specification `1` in group `g1`, as shown by the output of **lsspec** (17). In addition, using **rmspec** on group `g1` (18) deletes only specification `1`, as shown in the output of **lsspec** (19), since specification `2` is no longer in group `g1`.

The user registers another specification (20) involving an object class called `tool` and an attribute of `tool` called `debugged`. However, the specification is rejected by the Yeast server because object class `tool` is undefined. The user invokes **defobj** to define the object class (21) and **defattr** to define the attribute and its type. The user registers the specification again (23), this time successfully as the output of **lsspec** shows (24). Giving the number Yeast assigned to the new specification, `3`, as an argument to **lsspec**

(25) shows that Yeast cannot use polling to match the specification and must instead wait for an announcement to be made. The user invokes **announce** (26) to announce an event involving the newly-defined object class and attribute. As the output of **lsspec** shows (27), this announcement matched the event pattern of specification 3, thus causing the specification to be removed after its action was triggered.

IV. An Example Application of Yeast

In order to gain more experience with Yeast and to study the effectiveness of our event-action approach, we developed a collection of Yeast specifications that automate portions of a software distribution process. In this section we describe the process and illustrate some of the Yeast specifications.

The members of our department at AT&T Bell Laboratories distribute experimental software development tools to organizations throughout AT&T. Many of the tools are dependent on one another, thus creating a situation in which each tool owner must keep track of the activities of several other tool owners. The management of this collection of tools has been centralized under the control of a meta-user called *advsoft*, who gathers and distributes the official versions of the tools. Figure 3 depicts the process that *advsoft* manages. In the figure, the circles represent subprocesses, and the arrows represent data flow between subprocesses. As shown in the figure, tool owners submit the newest versions of their tools to *advsoft* in cycles, which currently occur twice a year. The figure depicts in detail how the process is carried out for the tool `libx` (solid lines), while showing that an identical process is carried out for all other tools T in parallel (dashed lines). The rest of the diagram should be self-explanatory.

The responsibilities of *advsoft* are basically bookkeeping activities that can be time-consuming and error-prone when performed by a human. For instance, a human could forget to notify a tool owner of a dependent tool change and then later wonder why an acceptance/rejection response had not been received from that owner. To help alleviate this situation, we have developed a collection of Yeast specifications that automate the portions of the diagram of Figure 3 that appear in boldface, which together comprise all of the automatable activities of the process.

A. Advsoft Object Classes

In order to model the *advsoft* process in Yeast and develop the specifications, it was first necessary to identify the kinds of objects that *advsoft* manages, along with their attributes. The object classes we identified include *tools* and tool *owners*, which have the following attributes:

- Tools:
  - *Rebuilt*: This is a boolean-valued attribute that becomes true for a tool T whenever a new version of T has been built successfully.
  - *Accepted*: This is a boolean-valued attribute that becomes true for a tool T whenever T has been accepted by *all* owners of tools dependent on T.
- Owners:
  - *Accepts*: This is a string-valued attribute specifying a tool T that has been accepted by an owner U.
  - *Rejects*: This is a string-valued attribute specifying a tool T that has been rejected by an owner U.

The owner of a tool is simply the person or persons who built the tool and who are responsible for making fixes and enhancements to the tool. The following commands register these object classes and attributes with Yeast:

```
defobj tool
defattr tool accepted boolean
defattr tool rebuilt boolean

defobj owner
defattr owner accepts string
defattr owner rejects string
```

Because these object classes are user-defined, their events must be announced to Yeast. As will be seen, all of the necessary announcements are generated automatically by Yeast specifications and by other programs. In other words, these user-defined events are abstractions for patterns of low-level, predefined primitive events.

B. Advsoft Specifications

To initialize the *advsoft* process, we run a command script called `genspecs` that generates all the

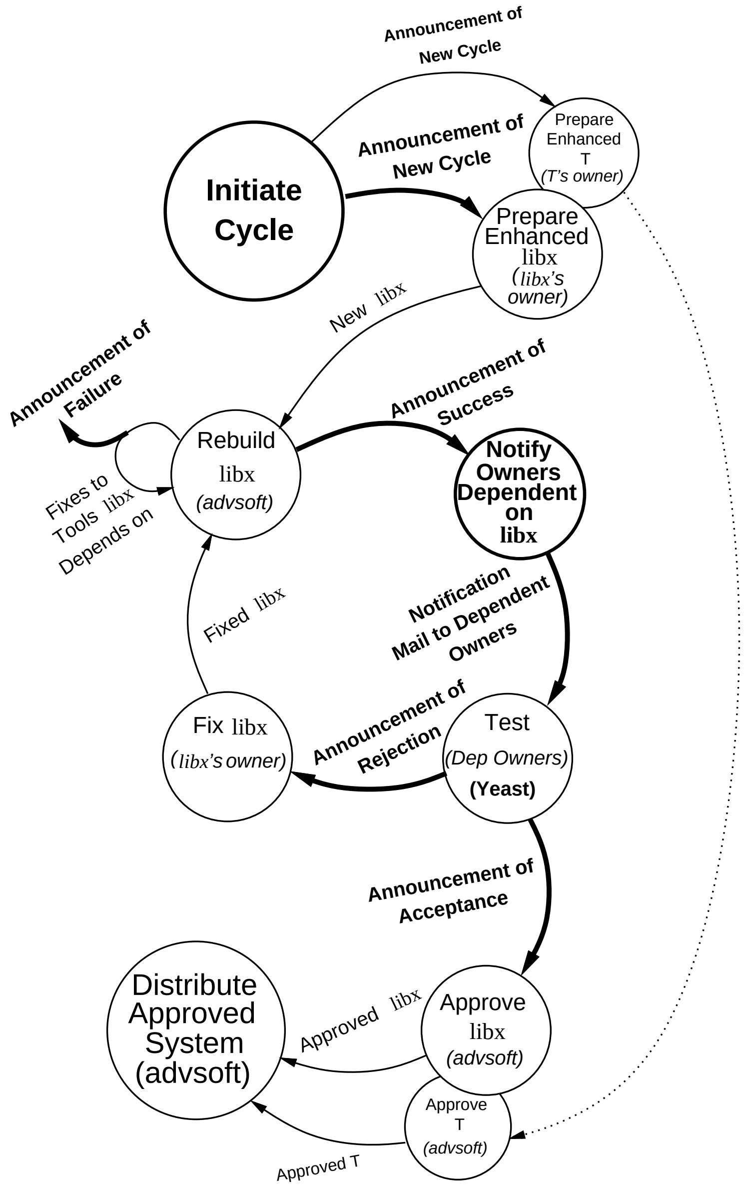

Fig. 3. The *Advsoft* Software Tool Distribution Process.

specifications that are used to automate the process. The specifications are generated for all tools and owners that are known to *advsoft*, using dependency information maintained in the *advsoft* tool database. As an example of the specifications generated by `genspecs`, the following specifications automate the subprocess *Notify Owners* of Figure 3 for the tool `libx`:

addspec repeat +advsoft +libx
  file /home/advsoft/src/libx/BUILT mtime changed
do announce tool libx rebuilt = true

addspec repeat +advsoft +libx
  file /home/advsoft/src/libx/ERROR mtime changed
do announce tool libx rebuilt = false

addspec repeat +advsoft +libx tool libx rebuilt == true
do Mail -s "libx rebuilt" U_1 U_2 ...

The first two specifications define the low-level, predefined Yeast events that determine when a new version of a tool has been built. Each tool (such as `libx`) exists in its own subdirectory of `/home/advsoft/src`. Whenever a tool is rebuilt, the file `BUILT` is created in the tool directory if the build succeeded, while the file `ERROR` is created if the build failed. The first specification says that whenever the file `/home/advsoft/src/libx/BUILT` is modified, then the high-level event "tool `libx` was successfully rebuilt" is automatically announced. The second specification says that whenever the file `/home/advsoft/src/libx/ERROR` is modified, then the high-level event "tool `libx` was unsuccessfully rebuilt" is automatically announced. Thus, the first two specifications serve to translate the low-level file system events into process-level tool-change events. The third specification illustrates one of the actions that is performed as a result of a successful rebuild; it says that whenever `libx` has been rebuilt, then electronic mail is sent to $U_1, U_2, \ldots$, who are the owners of tools that depend on `libx` (as determined by `genspecs`).

Once tool owner U_i receives a "`libx` rebuilt" message, that owner runs regression tests on his or her dependent tools and then makes one of the following two announcements, corresponding to the *Acceptance* and *Rejection* announcements of Figure 3:

announce owner U_i accepts = libx
announce owner U_i rejects = libx

In practice, these announcements can be generated automatically by the regression test scripts themselves. These announcements trigger other actions that are defined by the following specifications, which partially automate the subprocess *Test* of Figure 3; in these specifications, U_{libx} is the owner of `libx`:

addspec repeat +advsoft +libx owner U_1 accepts == libx
and owner U_2 accepts == libx
and ...
do Mail -s "libx accepted" advsoft U_{libx};
announce tool libx accepted = true

addspec repeat +advsoft +libx owner U_1 rejects == libx
or owner U_2 rejects == libx
or ...
do Mail -s "libx rejected" advsoft U_{libx};
announce tool libx accepted = false

The first specification automatically announces acceptance of `libx` once *all* dependent tool owners have announced their individual acceptance of `libx`. The second specification automatically announces rejection of `libx` once *any* dependent tool owner announces their individual rejection of `libx`.

In testing some dependent tool against a new version of `libx`, the owner of the dependent tool may find it necessary to submit a new version of the dependent tool, in order to account for interface changes and/or new features in the new version of `libx`. In such a situation, the dependent owner might withhold acceptance or rejection of `libx` until the new version of the dependent tool has itself been developed and made ready for submission to the *advsoft* process.

C. Changes to Tools and Owners

The tool dependency specifications shown above can become obsolete as the owners of and dependencies among existing tools change, and as new tools come into existence. Other specifications are used to automatically delete obsolete tool dependency specifications and add new ones whenever such events occur. This is accomplished by combining the above specifications into several specification groups, each of which can be removed by name and reconstructed as changes in dependencies occur. For instance, the acceptance and rejection specifications shown above are in the specification group `libx`. The following specification regenerates the `libx` specifications whenever changes are made to the `libx` tool area:

```
addspec repeat +advsoft
        dir /home/advsoft/src/libx mtime changed
do rmspec libx;
   genspecs libx
```

That is, if the `libx` tool directory is modified (by the appearance of a new version of `libx`), then the specifications in group `libx` are removed using the Yeast client command **rmspec**, and `genspecs` is invoked to regenerate the `libx` specifications.

Similarly, if a brand new tool is submitted to *advsoft*, then all of the existing *advsoft* specifications are assumed to be obsolete because of the potential introduction of new or altered dependencies between tools. Therefore, the following specification is used to delete and regenerate the complete set of specifications (which are all in the specification group `advsoft`) whenever a new tool is introduced (as indicated by a modification of `/home/advsoft/src`):

```
addspec repeat dir /home/advsoft/src mtime changed
do rmspec advsoft;
   genspecs
```

D. Discussion of Advsoft

The *advsoft* process provided a real-world application for gaining experience with and identifying shortcomings of Yeast. Indeed, the specification group feature was added in response to the need for the action of one specification to manipulate other specifications.

We have illustrated just a few of the specifications we have developed to automate the *advsoft* process. In total, the *advsoft* process is used to distribute 64 tools, which require 291 Yeast specifications for automating the process. The regularity of the specifications allows the use of the simple command script `genspecs` to generate the entire set of 291 specifications.

Note that because many events in this process are represented by Yeast announcements, individual tool owners can register additional specifications that provide other kinds of automated support customized to their particular needs (such as daily reminders to test their dependent tools). Note also that the process as currently defined contains several "holes", such as a lack of enforced deadlines. Such potential refinements to the process could be easily incorporated with additional specifications.

The *advsoft* process involves another set of activities not shown in Figure 3, namely those associated with the management of distribution requests and problem reports from external tool customers. We have just begun to model and automate these activities.

V. Implementation and Performance of Yeast

The basic architecture described in Section II is sufficiently general to be adaptable to a variety of computing platforms and action command interpreters. Our current implementation of Yeast runs on UNIX platforms and uses the KornShell as command interpreter for actions [8]. In this section we describe some details of the current implementation of Yeast, including a discussion of how we avoid polling for file-related events.

A. Software and Hardware Details

Yeast is implemented as a client-server system, with all client communication sent to the server listening on a well-known TCP port. Client connections can be transparently made from any machine that has network access to the server. Each invocation of a client command is carried out as a separate, autonomous connection to the server. Users can define and change an environment variable called YEAST_OPTIONS, to specify with which server their client command invocations are to communicate; these client command invocations and servers can run anywhere on the network. Thus, it is possible to register specifications with or generate Yeast announcements to multiple servers running on several continents.

The source code for Yeast comprises roughly 8400 lines of code, of which 210 lines is a Lex specification, and 1160 lines is a Yacc specification, with the rest written in ANSI C. The current version of Yeast runs on Sun OS versions 4.0.x and 4.1.x, on Berkeley UNIX System 4.3 derivatives [9], on Hewlett-Packard 9000-series platforms running HP-UX 8.05, and on SGI Indys.

The code is quite portable. Portability of the communication substrate is provided by a locally-developed network connect-streams library. The only portion of code that requires platform-specific alteration is the implementation of the object class **process**, since the definitions of OS process data structures vary with the machine architecture.

B. *Efficiency, Fairness and Performance*

The Yeast server is a single process that must handle client connections as well as checking the Polled Event Queue (shown in Figure 1) for potential matches. If the server is busy checking event descriptors in this queue, client connections may be blocked. Likewise, if several client communications occur in a row, the server may not be able to check the Polled Event Queue, consequently delaying the triggering of actions. Fairness is guaranteed by ensuring that at least once every five event match attempts, client connection attempts are checked. Thus, there is a slight bias in favor of checking the Polled Event Queue, to ensure that specifications already registered with Yeast are promptly matched.

To determine how well the server processing can scale to large applications, we have run a number of stress tests on the server and found the performance to be quite satisfactory. The server can handle hundreds of specifications matching simultaneously without any problem. Likewise, a burst of thousands of announcements can be processed quite efficiently. Of course, it is possible to construct pathological situations that result in severely degraded server performance. In practice, however, none of the applications we have developed has ever placed an unreasonably heavy load on the server.

C. *Avoiding Polling*

File-related events, especially file creation, change and deletion events, are frequently of interest to Yeast applications; as with the other predefined object classes attributes, the Yeast server must poll for occurrences of events involving attributes of file-related objects. Of course, a key limitation of polling is that events that occur transiently between polling operations can be missed by the server. For instance, if we consider the change in the count attribute of a directory, it is conceivable that a file may be created and another deleted in between polling operations, leaving the count apparently unchanged. For applications that rely heavily on the matching of file events, there is an option of using a slightly different version of Yeast that eliminates polling of file-related events. In conjunction with the Multiple Dimensional File System (n-DFS) [10], this alternative version of Yeast is able to automatically and transparently detect occurrences of such events *without* having to poll.

The n-DFS allows users to create a logically merged view of a related group of directories and make changes to files relative to this view. For instance, a user could create a merged view of the "development" and "official" versions of a software system, with all changes made to the "development" version only. Since the n-DFS implementation traps all operating system calls that correspond to Yeast file events, we enhanced the n-DFS to announce occurrences of Yeast file events synchronously to the Yeast server. These enhancements required a simple 30-line addition to the n-DFS source. The enhancements require no change to the underlying OS kernel and no change to Yeast applications themselves. Instead, the n-DFS exploits the availability on many platforms of *dynamic shared libraries*, which can be linked dynamically with compiled programs prior to execution; the traps implemented by the n-DFS are provided through such a shared library, which essentially envelopes the operating system calls provided by the OS kernel. A generalization of this framework was implemented in a system called COLA [11].

The advantage of this scheme is that it eliminates wasteful polling for events that may never happen, and it never misses events that may be missed because of long polling intervals. Note that while the use of the n-DFS results in a more efficient implementation of matching of file-related events, it is also possible to use the version of the Yeast server that polls for these events without the aid of the n-DFS.

D. *Fault-Tolerance of the Server*

Since the Yeast server is a potentially critical component of users' applications, the Yeast server must be fault tolerant. Rather than implementing fault tolerance in the Yeast server itself, we register the Yeast server with a global fault tolerance server called Watchd. Registration is accomplished by a few lines of code that invoke an associated software fault-tolerance library called Libft [12]. If the machine on which the Yeast server is running crashes, Watchd will automatically start a new Yeast server on another machine in the local area network. Further, all subsequent Yeast client connections will be directed to the new instance of the Yeast server. All of this happens transparently to users and to Yeast applications.

VI. Related Work

In Section I we mentioned some examples of event-action systems; in this section we discuss some of the systems that had a direct influence on the development of Yeast and the ways in which Yeast improves on previous event-action systems.

Yeast was inspired by a number of earlier systems. Cron, At and Usrcron were some of the earliest event-action tools, built for the UNIX operating system [9]. These were simple scheduling tools that triggered actions at a particular time of day. Omicron [13] extended Cron with recognition of file modification events. Yeast also bears some resemblance to demand-driven systems such as Make [14] and Nmake [15] in which software configurations specified in *makefiles* are used to detect configuration changes and initiate selective build operations. The events that Make and Nmake can detect are limited to file system changes and, in the case of Nmake, changes to configuration parameters.

As we discussed in Section I, several systems have been built that specialize the event-action paradigm for particular application domains. These systems attain the main advantage of specialization (namely a rich set of features that support a particular application domain) by sacrificing openness and sacrificing generality in the kinds of actions that are supported. For example, Field supports tool integration with a broadcast message server (BMS) that allows tools to register interest in events that are generated by other tools. Thus, a pair of tools will be integrated in this fashion only if they both register with the BMS; in this sense Field is a closed system. Furthermore, the only action one tool can take to affect the behavior of another tool is to send the other tool a message through the BMS. In contrast with Field, Yeast is an open system, since a Yeast client can respond to events generated by another tool without the other tool even being aware that its events are affecting the client. And both the tool and the client can perform a variety of actions that affect each other's behavior. For instance, the tool can modify a file, or perform a CPU-intensive computation that increases the system load, or even send a Yeast announcement, all of which can match a client specification and trigger an arbitrary action in response.

Yeast can be used to build many of the same kinds of applications that these special-purpose systems support. For example, DSEE (DOMAIN® Software Engineering Environment[8]) is a configuration management system that incorporates event-action support in the form of *monitors* and a *monitor manager* [16,17]. The *advsoft* process described in Section IV is in many respects a configuration management application, in which Yeast specifications track inter-tool dependencies in much the same way that monitors do in DSEE. Yet DSEE is specialized to the domain of configuration management, while Yeast can be used to support a variety of other application domains.

Of the systems mentioned in the introduction, Yeast bears the greatest resemblance to Marvel [**?**, 5] A Marvel rule is analogous to a Yeast specification, with the activity of a Marvel rule being the analogue of a Yeast specification action. The *precondition* of a Marvel rule is like the event pattern of a Yeast specification, while the *effect* of a Marvel rule is like a sequence of Yeast announcements made from the action of a Yeast specification. A powerful feature of Marvel that is lacking in Yeast is Marvel's support for *opportunistic forward and backward chaining* of rules, whereby rules are automatically fired to help establish the precondition of a rule of interest (backward chaining) or to further the computation once the effect of a rule of interest has been asserted (forward chaining).

The primary difference between Marvel and Yeast is that Marvel is a closed system, in the sense that the activity of a Marvel rule can only involve tools whose effects are formally defined to Marvel via tool envelopes, whereas Yeast is an open system allowing specifications to trigger ar-

[8]DOMAIN is a registered trademark of HP-Apollo.

bitrary actions outside the control of Yeast. In this context an open system has the advantage of greater flexibility. However, it also has the drawback of making static formal analysis or opportunistic chaining of specifications more difficult, since the effect of an action is not formally defined to the system.

Another major difference between Yeast and Marvel is the way in which specifications in these systems adapt themselves to change. In Marvel, these changes can be reflected in changes to the parameters of rules. In Yeast, an event pattern that represents a change can trigger a Yeast action that automatically removes obsolete specifications and then adds replacement specifications that account for the change. Yeast and Marvel have been recently integrated into a process monitoring, visualization and analysis environment called Provence [18].

Yeast is one of the few event-action systems that permits compound event descriptors in specifications. Yeast's event pattern language was inspired by TSL [19,20], an event-based language for specifying the behavior of concurrent programs. TSL uses an event language to support specification of *constraints* on the behavior of a concurrent program, whereas Yeast uses an event language to support specification of *event triggers* for user-defined actions. Yeast's object/attribute model of events is similar to the entity-relation model used by Chen in his Network Event Manager (NEM) [21].

VII. Conclusion

We have described in detail the event-action system Yeast and some of its applications. Several projects within AT&T are using it for a variety of applications. These applications include wide area software development, requirements tracing, software tracking, security monitoring, and software process measurement. The generality and extensibility of Yeast have made it highly amenable to the differing needs of these applications.

Many of the ideas and features found in Yeast were present in previous systems. Yet a key contribution of Yeast is the way in which it combines a multitude of useful features into a single, general platform for building event-based applications. In particular, Yeast satisfies all of the requirements set forth in Section I, some to a greater degree than others:

- Yeast provides a rich specification language that supports specification and matching of patterns of both temporal and non-temporal events.
- Yeast's support for class and attribute definition and event announcement allows users to extend the event-matching capabilities of Yeast.
- Our implementation of Yeast attains reliability and fault tolerance at the software level through its incorporation of two special purpose fault tolerance components, Libft and Watchd.
- Yeast supports a rich collection of client commands that allow users to interactively query and manage the status of their specifications. Furthermore, security of all client interactions is ensured.

Most importantly, as described in Section VI, the openness of Yeast, its support for compound event patterns and its support for arbitrary actions in response to events together represent a significant improvement upon the capabilities of previous event-action systems.

While the current implementation of Yeast is able to support a wide variety of applications, it should be noted that the dependence of the current implementation upon the UNIX operating system and the KornShell command interpreter do represent limits on the openness of Yeast, the kinds of actions that can be performed, and the granularity at which event matching and action execution can take place. For instance, in order to use Yeast as the event-action component of an active database system, the Yeast server would have to be able to detect events that are not easily detectable at the UNIX level (such as a transaction being committed), and Yeast would have to send the actions of specifications to the database manager's query processor rather than to the KornShell. Providing support for different families of events and different command interpreters in a user-selectable manner is an important problem and one that we intend to study in the future.

Another weakness of the current implementation is in its support for announcements of user-defined events. Presumably users have a reasonably well-defined semantics in mind for any events they define. Yet Yeast provides no support for enforcing the semantics of user-defined events. In particular, Yeast provides no way of ensuring that every occurrence of a user-defined event is announced by the user (and thus detected by Yeast). Furthermore, the Yeast server assumes that all user-defined events have a permanent matching semantics, but only for specifications that are active at the time of a matching occurrence, not for specifications that are registered after the matching occurrence. In other words, the attribute values announced in announcements do not persist the way attribute values of predefined events persist (such as the fact that October 1, 1960, is in the past for all future specifications that are registered). For some kinds of events it may be impossible to provide such support for enforcement, yet for other kinds of events it may be possible. One way of providing detection of all instances of a user-defined event would be to allow the user to register a program with the Yeast server that could be executed during matching to determine whether an event descriptor for a user-defined event is matched. Such a program would in essence incorporate the semantics the user has assigned to the event. An even greater problem is ensuring that the intended semantics of a complete Yeast application (represented by a related collection of specifications) is implemented in a complete and consistent manner. This problem has been studied previously in the Darwin project in the context of *law-governed architectures* [22], and it is a topic for future study in Yeast.

We are making a number of enhancements to the capabilities of Yeast, including persistence of announcements, static analysis of specifications, and the addition of variables to the specification language.

Many Yeast applications manipulate objects in groups, treating each member of the group in the same manner. For example, the *advsoft* application described in Section IV manipulates several dozen software tools in a uniform manner. It must wait for events on files whose names are of the form `/home/advsoft/src/lib/`T`/BUILT`, where T is a tool name, and it must make announcements of the form "`tool` T `rebuilt = true`". Rather than require the application to register one specification per value of T, we would like the Yeast server to automatically determine all the appropriate values for T and to carry out specification matching accordingly. In the future we plan to enhance Yeast's specification language with variables, which will stand as placeholders for the variable components of such generic event descriptors and actions. In particular, variables could be used in the position of the object name of object event descriptors, as a component of the object name (i.e., pathname) in `file`, `dir` and `tty` object event descriptors, and in the position of the attribute value in object event descriptors having an equality operator. The server would automatically bind unbound variables and perform variable substitution, in effect quantifying each specification over all the values it can find for the specification's constituent variables.

As an example of the power of variables consider the following specification, which contains a variable called `%f`:

addspec repeat file %f mtime changed do spell %f

The first time a change of a file in the current directory is detected by Yeast, the above specification would be matched, and the variable `%f` would be bound to the name of the file that triggered the match. This value would then be substituted for the occurrence of `%f` in the action portion of the specification (i.e., `spell %f`, which checks the file that changed for spelling errors). Furthermore, since the specification is repeatable, this matching behavior will be repeated for *all* files in the current directory that are modified. A detailed discussion of the semantics of variables is beyond the scope of this paper, but it is clear that variables will add enormous expressive power to the language and, consequently, a great deal of semantic and implementation complexity. A prototype enhancement to Yeast supporting a limited number of predefined variables for announcements has been implemented for the Provence system [23].

Persistence of announced attribute values is another needed enhancement. As was discussed above, persistence is already implicitly supported for the predefined attributes and is thus a natural addition to the semantics of user-defined attributes. Yet when an announcement is received by the server, it is matched against all active specifications and then discarded. It is possible, however, that applications may require some announcements to persist; that is, the values of the attributes specified in the announcements should be saved for matching against specifications registered after the announcement is received by the server.

Many Yeast users requested a graphical front end to Yeast in addition to the textual one described in Section III-C. Recently, a graphical front end called FEY [24] was constructed to display dynamic information about the collection of specifications owned by a Yeast user. Each specification is displayed as a graph, with the nodes representing the connectors **and**, **or** and **then**, and the primitive event descriptors. The graph also displays the next time a match will be attempted on the primitive event descriptors. As the specification is matched, the portions of the graph corresponding to the matched event descriptors change color, thus providing a dynamic view of the specification state. Once the specification is completely matched, its graph disappears from the display. Other changes that affect the display include user actions such as adding specifications, removing specifications, and suspending or resuming matching of specifications.

Finally, in the area of semantic analysis of Yeast specifications, we would like to determine if and when a related set of specifications will all match, whether there are circular dependencies or deadlocks, and so on. An initial attempt in this direction has been made whereby the current set of a user's specifications is input to C-Prolog programs that perform a number of inter-specification semantic consistency checks [25].

Acknowledgments

We thank the members of our department at AT&T Bell Laboratories, who were the initial users of Yeast. Terry Anderson and Sultan Alam gave us valuable comments on early versions of Yeast, and Valerie Torres helped us understand the *advsoft* process. Alex Wolf and Dewayne Perry gave us many helpful comments on earlier drafts of this paper. We also thank the anonymous referees for their detailed comments.

References

[1] Steven P. Reiss, "Connecting tools using message passing in the Field environment", *IEEE Software*, vol. 7, no. 4, pp. 57–66, July 1990.

[2] Colin Gerety, "HP SoftBench: A new generation of software development tools", Tech. Rep. SESD-89-25, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, Nov. 1989.

[3] N. H. Gehani and H. V. Jagadish, "Ode as an active database: Constraints and triggers", in *VLDB 91: Proceedings of the 17th International Conference on Very Large Data Bases*. IEEE Computer Society, Sept. 1991, pp. 327–336.

[4] Donald Cohen, "Compiling complex database transition triggers", in *SIGMOD '89: Proceedings of the 1989 International Conference on Management of Data*. ACM SIGMOD, May–June 1989, pp. 225–234.

[5] Gail E. Kaiser, Naser S. Barghouti, Peter H. Feiler, and Robert W. Schwanke, "Database support for knowledge-based engineering environments", *IEEE Expert*, pp. 18–32, Summer 1988.

[6] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich, "Intelligent assistance for software development and maintenance", *IEEE Software*, vol. 5, no. 3, pp. 40–49, May 1988.

[7] Richard W. Selby, Adam A. Porter, Doug C. Schmidt, and Jim Berney, "Metric-driven analysis and feedback systems for enabling empirically guided software development", in *Proceedings of the 13th International Conference on Software Engineering*. IEEE Computer Society, May 1991, pp. 288–298.

[8] M. I. Bolsky and D. G. Korn, *The New KornShell Command and Programming Language*, Prentice-Hall, 1995.

[9] Computer Science Division, University of California, Berkeley, *UNIX Programmer's Manual, 4.3 Berkeley Software Distribution*, April 1986.

[10] Glenn Fowler, David Korn, Stephen North, Herman Rao, and Kiem-Phong Vo, "Libraries and file system architecture", in *Practical Reusable UNIX Software*, Balachander Krishnamurthy, Ed., chapter 2, pp. 78–90. Wiley, 1995.

[11] Eduardo Krell and Balachander Krishnamurthy, "COLA: Customized overlaying", in *Proceedings of the USENIX Winter 1992 Conference*, 1992, pp. 3–7.

[12] Yennun Huang and Chandra Kintala, "A software fault tolerance platform", in *Practical Reusable UNIX Software*, Balachander Krishnamurthy, Ed., chapter 8. Wiley, 1995.

[13] Balachander Krishnamurthy and Craig E. Wills, "Omicron: Events ⇒ Actions", Tech. Rep. CSD TR 594, Department of Computer Sciences, Purdue University, April 1986.

[14] Stuart I. Feldman, "Make—A Program for Maintaining Computer Programs", *Software—Practice and Experience*, vol. 9, no. 3, pp. 255–265, March 1979.

[15] G. S. Fowler, "The Fourth Generation Make", in *USENIX Portland 1985 Summer Conference Proceedings*, 1985, pp. 159–174.

[16] David B. Leblang and Robert B. Chase, Jr., "Computer-aided software engineering in a distributed workstation environment", in *Proceedings of the Software Engineering Symposium on on Practical Software Development Environments*. ACM SIGSOFT/SIGPLAN, Apr. 1984, pp. 104–112.

[17] David B. Leblang and Robert B. Chase, Jr., "Parallel software configuration management in a network environment", *IEEE Software*, vol. 4, no. 6, pp. 28–35, Nov. 1987.

[18] Balachander Krishnamurthy and Naser Barghouti, "Provence: A Process Visualization and Enactment Environment", in *Proceed-*

ings of the 4th European Software Engineering Conference (ESEC4), Garmisch-Partenkirchen, Germany, Sept. 1993, pp. 151–160, Springer-Verlag, *Lecture Notes in Computer Science* no. 717.

[19] David C. Luckham, David P. Helmbold, Sigurd Meldal, Douglas L. Bryan, and Michael A. Haberler, "Task Sequencing Language for specifying distributed Ada systems", in *System Development and Ada: Proceedings of the CRAI Workshop on Software Factories and Ada*, Habermann and Montanari, Eds. 1987, pp. 249–305, Springer-Verlag (Lecture Notes in Computer Science No. 275).

[20] David S. Rosenblum, "Specifying concurrent systems with TSL", *IEEE Software*, vol. 8, no. 3, pp. 52–61, May 1991.

[21] Yih-Farn Chen, *Event Management in Computer Networks*, PhD thesis, Computer Science Division, Electrical Engineering and Computer Science Department, University of California at Berkeley, December 1987.

[22] Naftaly H. Minsky, "The imposition of protocols over open distributed systems", *IEEE Transactions on Software Engineering*, vol. 17, no. 2, pp. 183–195, Feb. 1991.

[23] Naser S. Barghouti and Balachander Krishnamurthy, "Using event contexts and matching constraints to monitor software processes effectively", in *Proceedings of the 17th International Conference on Software Engineering (ICSE-17)*, Seattle, WA, May 1995, IEEE Computer Society.

[24] Eleftherios Koutsofios and Balachander Krishnamurthy, "Combining interactive tools", Presented at CSCW '92: ACM Workshop on Tools & Technologies for CSCW, Oct. 1992.

[25] Paola Inverardi, Balachander Krishnamurthy, and Daniel Yankelevich, "Yeast: A case study for a practical use of formal methods", in *TAPSOFT '93: Proceedings of the 5th International Joint Conference on Theory and Practice of Software Development*. Apr. 1993, pp. 105–120, Springer-Verlag, Published as *Lecture Notes in Computer Science* no. 668.

Balachander Krishnamurthy is a member of the technical staff in the Software Engineering Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey.

David S. Rosenblum (S'83–M'87) is a member of the technical staff in the Software Engineering Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests include software testing and analysis, software process, formal specification languages, and specification-based software development tools. Dr. Rosenblum received a PhD in Electrical Engineering in 1988 from Stanford University, where he participated in the Anna and TSL specification language projects. He also received an MS in Electrical Engineering in 1987 from Stanford University, and a BS *summa cum laude* in 1982 and MS in 1983, both in Computer Science, from North Texas State University, in Denton. He has served on the program committees of the *International Symposium on Software Testing and Analysis* (ISSTA), the *International Workshop on Software Specification and Design* (IWSSD) and the *International Conference on Distributed Computing Systems* (ICDCS). He is a member of ACM, ACM SIGAda, ACM SIGPLAN, ACM SIGSOFT, IEEE, IEEE Computer Society, and IEEE Computer Society Technical Committee on Software Engineering.