YEAST–A Formal Specification Case Study in Z

Maritta Heisel
Institut für Angewandte Informatik
Technische Universität Berlin
Sekr. FR 5-6, Franklinstr. 28/29
D-10587 Berlin, Germany
heisel@cs.tu-berlin.de

Balachander Krishnamurthy
Software Engineering Research Dept
AT&T Bell Laboratories
Room 2B-140, 600 Mountain Ave
Murray Hill, NJ 07974 USA
bala@research.att.com

Abstract

A formal specification in the language Z of an event-action system called YEAST is presented. Such a specification helps *users* of event-action systems to a deeper understanding of the system's features than can be gained by natural language descriptions. *Designers* of such systems can use the formal specification as a starting point for the specification of new event-action systems. Finally, members of the formal specification community can profit from the general lessons learnt in this case study.

1 Introduction

Yeast [KR95] 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. Yeast is roughly 8400 lines of code including portions written in *lex* and *yacc* with the rest in ANSI C. Yeast is used in projects within AT&T. Yeast applications consist of a collection of specifications written to manage or automate a portion of the task.

This paper presents a Z specification of a large subset of Yeast (only a few features are omitted). We expect that designers and members of the formal specification community can benefit from such an exercise. We hope it will also be of value to Yeast application writers but this remains to be seen. A formal specification of a system like Yeast requires a new language to be learnt by

Yeast application writers who may be reluctant to do so. However, if we can point out the advantages of understanding the formal specification *and* if it is simple enough, then we have a higher chance of impacting the Yeast application writers.

The specification language Z [Spi92] has been chosen because Yeast is a sequential program that maintains an internal state. Model-based languages like Z and VDM have been especially designed for the purpose of specifying this kind of systems. In section 2, we present the specification of the Yeast system. We point out how such a specification can support users and designers of event-actions systems and report on the lessons learned on the practical application of formal specification techniques in general. We conclude with a discussion of related work.

2 The Z specification

The specification presented here can be seen as a description of the behavior of the system. It is intended to clarify the various client commands and the semantics of the event language. The intricate matching process is not presented in full detail. We do not treat error cases—their specification is straightforward.

2.1 Basic definitions

A Yeast specification has an event pattern and an action. Atomic events and action expressions are introduced as basic types. Action expressions, which can be composite, are treated as atomic here because it is usually a call to an external command and outside the scope of a Yeast specification.

$$[EVENT, ACTION\_EXP]$$

Events are the heart of the system. They can be temporal (TE) or non-temporal (NTE). Temporal events can be absolute (ATE) or relative (RTE). Non-temporal events can either be (predefined) object events (OE) or user-defined events (UE).

$$\begin{array}{l} TE, NTE : \mathbb{P}\, EVENT \\ ATE, RTE : \mathbb{P}\, EVENT \\ OE, UE : \mathbb{P}\, EVENT \\ \hline \langle ATE, RTE, OE, UE \rangle \text{ partition } EVENT \\ TE = ATE \cup RTE \\ NTE = OE \cup UE \end{array}$$

These events are also called *primitive* events. The event language of Yeast allows one to build more complex event expressions, using the connectors *then* (sequencing), *and* (conjunction) and *or* (disjunction).

$$
\begin{aligned}
EVENT\_EXP ::= \; & prim\langle\langle EVENT\rangle\rangle \\
| \; & then\langle\langle EVENT\_EXP \times EVENT\_EXP\rangle\rangle \\
| \; & and\langle\langle EVENT\_EXP \times EVENT\_EXP\rangle\rangle \\
| \; & or\langle\langle EVENT\_EXP \times EVENT\_EXP\rangle\rangle
\end{aligned}
$$

Of these connectives, *then* binds most and *or* binds least. Therefore, the following distributivity laws hold:

$$
\begin{aligned}
(A \; and \; B) \; then \; C &\simeq (A \; then \; C) \; and \; (B \; then \; C) && (1)\\
A \; then (B \; and \; C) &\simeq (A \; then \; B) \; and \; (A \; then \; C) && (2)\\
(A \; or \; B) \; then \; C &\simeq (A \; then \; C) \; or \; (B \; then \; C) && (3)\\
A \; then \; (B \; or \; C) &\simeq (A \; then \; B) \; or \; (A \; then \; C) && (4)\\
(A \; or \; B) \; and \; C &\simeq (A \; and \; C) \; or \; (B \; and \; C) && (5)\\
A \; and \; (B \; or \; C) &\simeq (A \; and \; B) \; or \; (A \; and \; C) && (6)
\end{aligned}
$$

Using these rules, each event expression can be converted into a *normal* form, similar to the disjunctive normal form of formulas of first-order predicate logic. This normal form has only *or's* as leading operators, followed by only *and's* and *then's*.

As an example, let us consider the event expression $(e_1 \; and \; e_2) \; then \; (e_3 \; or \; e_4)$. Its normal form can be computed as follows:

$$
\begin{aligned}
& (e_1 \; and \; e_2) \; then \; (e_3 \; or \; e_4) \\
\simeq \; & ((e_1 \; and \; e_2) \; then \; e_3) \; or \; ((e_1 \; and \; e_2) \; then \; e_4)) && (by \; 4)\\
\simeq \; & ((e_1 \; then \; e_3) \; and \; (e_2 \; then \; e_3)) \; or \; ((e_1 \; then \; e_4) \; and \; (e_2 \; then \; e_4)) && (by \; 1)
\end{aligned}
$$

This normal form can be represented by a set of sets of sequences:

$$MATCH\_STATUS == \mathbb{P}(\mathbb{P}(\mathrm{seq}\, EVENT))$$

This defines the normal form of an event expression to be a set. In order to fully match the event expression, one must fully match *one* of the elements of the set. This corresponds to *or* and is thus called an *or*-set. One element of such an or-set is fully matched when *all* of its elements have been fully matched. This corresponds to *and*. Such an *and*-set consists of sequences of events, each of which has to be matched for the whole event expression be be matched. The sequences, in turn, describe the order in which primitive events have to occur to make matching successful. The normal form thus does not represent the syntax of event expressions but their semantics (according to the distributivity laws). The *MATCH_STATUS* data structure maintains a record of the partial match status of events (see Section 2.4).

The following function transforms an event expression into its normal form:

$$
\begin{array}{|l}
norm : EVENT\_EXP \longrightarrow MATCH\_STATUS \\
\hline
\forall\, e : EVENT;\ A, B : EVENT\_EXP \bullet \\
\quad norm(prim(e)) = \{\{\langle e \rangle\}\} \quad \land \\
\quad norm(or(A, B)) = norm\, A \cup norm\, B \quad \land \\
\quad norm(and(A, B)) = \{a : norm\, A;\ b : norm\, B \bullet a \cup b\} \quad \land \\
\quad norm(then(A, B)) = \{a : norm\, A;\ b : norm\, B \bullet \{s : a;\ t : b \bullet s \frown t\}\}
\end{array}
$$

For our example, the normal form looks as follows:

$$\{\{\langle e_1, e_3 \rangle, \langle e_2, e_3 \rangle\}, \{\langle e_1, e_4 \rangle, \langle e_2, e_4 \rangle\}\}$$

From the above definition it follows that the normal form of an event expression cannot contain the empty sequence.

2.2 The state space

Each Yeast specification consists of an *event part* and an *action part*. It belongs to an *owner* and has an associated *label* for easy reference. It can be *suspended*, i.e. no matching is done against the events in the specification. The users can also state that a specification is to be added again after it is completely matched and its action is triggered. The current match status of a specification has to be recorded. Specifications also get a time stamp when they are registered in the system. All this information is gathered in a schema.

$$[OWNER, TIME, LABEL]$$

$$YesNo ::= yes \mid no$$

$$
\begin{array}{|l}
\textit{Spec} \\
\hline
e : EVENT\_EXP \\
a : ACTION\_EXP \\
match\_status : MATCH\_STATUS \\
own : OWNER \\
label : LABEL \\
repeat : YesNo \\
suspended : YesNo \\
reg\_time : TIME \\
\hline
\end{array}
$$

The internal state of the Yeast system consists of three major parts:

- The specifications currently present in the system. Specifications can be grouped together, where the group names are of type $GNAME$.
- The possible events. In contrast to other event-action systems user-defined events can be added to Yeast.

- The environment in which Yeast operates. Each client command is invoked by a user on a specific machine. The environment influences the behavior of Yeast. In this paper, we only consider the user who invokes commands—this information is needed for purposes of authentication.

$[GNAME]$

SpecState

$$specs : \mathbb{F}\ Spec$$
$$specMap : LABEL \rightarrowtail Spec$$
$$groups : GNAME \leftrightarrow LABEL$$

$$specMap = \{s : specs \bullet s.label \mapsto s\}$$
$$\mathrm{ran}\ groups \subseteq \mathrm{dom}\ specMap$$

The invariant of the specification state requires that each label be unique and that each label referred to by the relation *groups* belong to an existing specification. Yeast users can define new object classes and attributes. The set of possible events depends on the defined object classes and attributes.

$[OBJECT\_CLASS, ATTRIBUTE]$

PossibleEvents

$$user\_events : \mathbb{P}\ UE$$
$$possible\_events : \mathbb{P}\ EVENT$$
$$attrs : OBJECT\_CLASS \nrightarrow \mathbb{F}\ ATTRIBUTE$$
$$class : NTE \nrightarrow OBJECT\_CLASS$$

$$possible\_events = TE \cup OE \cup user\_events$$
$$\mathrm{dom}\ class = OE \cup user\_events$$
$$\mathrm{ran}\ class = \{oc : \mathrm{dom}\ attrs \mid attrs\ oc \neq \emptyset\}$$

The predefined temporal and object events are possible events. The set of attributes belonging to an object class are recorded in the function *attrs*. The function *class* yields the object class to which a given non-temporal event belongs. Each predefined object event and each user-defined event must have a corresponding object class. If an object class has a non-empty set of attributes, then there must be some non-temporal event referring to it.

The following function yields the set of primitive events contained in an event expression. It is used to express the requirement that all specifications present in the system refer only to possible events.

$$events == \mathrm{ran} \circ \bigcup \circ \bigcup \circ norm$$

The schema *YeastState* relates the two sub-states.

YeastState

$$SpecState$$
$$PossibleEvents$$

$$\forall s : specs \bullet events\ s.e \subseteq possible\_events$$

Initially, no specifications are registered in Yeast, and no user events are defined.

InitYeastState

$$YeastState'$$

$$specs' = \varnothing$$
$$user\_events' = \varnothing$$

The environment is modeled by the following schema. In reality, it consists of many components but here we only model the invoker of the client command.

ClientCommand

$$owner : OWNER$$

For matching of events, it is important to know when a specification was registered and when an event occurs. For this purpose, we use use the global system clock. Since its value changes as time proceeds, it is modeled as a schema, not as a variable.

GlobalSystemClock

$$current\_time : TIME$$

Looking up the system time can now be realized as an import of the above schema.

2.3 Client commands

We present some of the heavily used Yeast client commands. First, we consider the commands changing the *SpecState* sub-state of the system. In the second part, we show how users can define new events.

addspec registers a new specification with the Yeast server. All events used in the new specification must be possible. A new internal label is associated with the new specification by the system. It is the output of this operation. If group name(s) arguments are given, the specification is added to the specification group(s) as well. Plain variables refer to the state in which an operation

is started. Variables decorated with "′" refer to the state after the operation is completed. "Δ" means that the respective state may change, whereas "Ξ" indicates that the state does not change. Inputs are decorated with "?", outputs are decorated with "!". "∃ *Spec*" is an abbreviation for an existential quantification over all variables declared in the schema *Spec*. These are combined to form an item of type *Spec* using the θ operator.

addspec

$$
\begin{array}{l}
ClientCommand \\
GlobalSystemClock \\
\Delta\, YeastState \\
\Xi PossibleEvents \\
r? : YesNo \\
gs? : \mathbb{F}\, GNAME \\
e? : EVENT\_EXP \\
a? : ACTION\_EXP \\
l! : LABEL \\
\hline
events\, e? \subseteq possible\_events \\
\exists\, Spec \bullet \\
\quad label \notin \mathrm{dom}\, specMap \land \\
\quad e = e? \land a = a? \land match\_status = norm\, e? \land \\
\quad own = owner \land repeat = r? \land suspended = no \land \\
\quad reg\_time = current\_time \land l! = label \land \\
\quad specs' = specs \cup \{\theta Spec\} \land \\
\quad groups' = groups \cup \{g : gs? \bullet g \mapsto label\}
\end{array}
$$

lsspec returns the list of specifications owned by a user.

lsspec

$$
\begin{array}{l}
ClientCommand \\
\Xi\, YeastState \\
speclist! : \mathrm{iseq}\, Spec \\
\hline
\mathrm{ran}\, speclist! = \{s : specs \mid s.own = owner\}
\end{array}
$$

The following client commands have as their input either a group name or a (nonempty) set of labels. Their disjoint sum is defined as the free type *SPECREF*.

$$SPECREF ::= lab\langle\langle \mathbb{F}_1\, LABEL\rangle\rangle \mid gr\langle\langle GNAME\rangle\rangle$$

Each of the following client commands has the same precondition. If a group name is given as an input, it must be the name of an existing group. If a set of labels is given, all its members must be labels of existing specifications. The

client executing the commands must be the owner of all specifications referred to. In order not to repeat this precondition for each client command, we express it in a schema.

Precond

$$
\begin{array}{l}
ClientCommand \\
SpecState \\
sr? : SPECREF \\
ls : \mathbb{F}\, LABEL \\
ss : \mathbb{F}\, Spec \\
\hline
ls = \textbf{if } sr? \in \operatorname{ran} gr \textbf{ then } groups(\!|\{gr^{\sim}(sr?)\}|\!) \textbf{ else } lab^{\sim}(sr?) \\
ls \neq \varnothing \land ls \subseteq \operatorname{dom} specMap \\
ss = \{s : specs \mid s.label \in ls\} \\
\forall s : ss \bullet s.own = owner
\end{array}
$$

rmspec removes one or more specifications from the system, referred to by labels or a group name.

rmspec

$$
\begin{array}{l}
ClientCommand \\
\Delta\, YeastState \\
\Xi PossibleEvents \\
sr? : SPECREF \\
\hline
\exists\, ls : \mathbb{F}\, LABEL; ss : \mathbb{F}\, Spec \mid Precond \bullet \\
\qquad specs' = specs \setminus ss \land groups' = groups \rhd ls
\end{array}
$$

We now introduce some auxiliary schemas on specifications. Most client commands only change one or two components of a specification.

SpecOp

$$
\begin{array}{l}
\Delta Spec \\
\hline
e' = e \land a' = a \land own' = own \land label' = label \land repeat' = repeat
\end{array}
$$

$$
\begin{array}{l}
NonMatchOp \mathrel{\widehat=} [SpecOp \mid match\_status' = match\_status \land \\
\qquad\qquad reg\_time' = reg\_time] \\
SpecSuspend \mathrel{\widehat=} [NonMatchOp \mid suspended' = yes] \\
SpecFg \mathrel{\widehat=} [NonMatchOp \mid suspended' = no]
\end{array}
$$

Specifications can be suspended and resumed by giving labels or group names.

suspspec

$ClientCommand$
$\Delta YeastState$
$\Xi PossibleEvents$
$sr? : SPECREF$

$\exists ls : \mathbb{F}\, LABEL; ss : \mathbb{F}\, Spec \mid Precond \bullet$
$\quad specs' = (specs \setminus ss) \cup \{SpecSuspend \mid \theta Spec \in ss \bullet \theta Spec'\}$
$groups' = groups$

fgspec

$ClientCommand$
$\Delta YeastState$
$\Xi PossibleEvents$
$sr? : SPECREF$

$\exists ls : \mathbb{F}\, LABEL; ss : \mathbb{F}\, Spec \mid Precond \bullet$
$\quad specs' = (specs \setminus ss) \cup \{SpecFg \mid \theta Spec \in ss \bullet \theta Spec'\}$
$groups' = groups$

defobj and *defattr* change the set of possible events. *defobj* defines a new object class and *defattr* associates new attributes with existing object classes. The function *mk_events* yields the new set of user-defined events associated with a new attribute and an object class. Note that this function is total and injective.

$mk\_events : OBJECT\_CLASS \times ATTRIBUTE \rightarrowtail \mathbb{P}\, UE$

defobj

$\Delta YeastState$
$\Xi SpecState$
$oc? : OBJECT\_CLASS$

$oc? \notin \mathrm{dom}\, attrs$
$possible\_events' = possible\_events$
$attrs' = attrs \cup \{oc? \mapsto \varnothing\}$
$class' = class$

defattr

$\Delta YeastState$
$\Xi SpecState$
$oc? : OBJECT\_CLASS$
$a? : ATTRIBUTE$

$oc? \in \mathrm{dom}\ attrs \land a? \notin attrs\ oc?$
$user\_events' = user\_events \cup mk\_events(oc?, a?)$
$attrs' = attrs \oplus \{oc? \mapsto attrs\ oc? \cup \{a?\}\}$
$class' = class \cup \{ue : mk\_events(oc?, a?) \bullet ue \mapsto oc?\}$

From the injectivity of *mk_events* and the precondition $a? \notin attrs\ oc?$ it follows that $class'$ remains a function.

2.4 Matching

Two kinds of matching semantics are applied in Yeast: For announced events or temporal events that are monotonically increasing, *sticky* matching is applied: once an event is matched it stays matched forever. For the other events, this is not possible. In an *and* expression, both parts must match at the same time. If only one part matches, it has to be re-considered until both parts match. Therefore, we need two sets of matching functions, one for sticky matching, the other for *transient* matching. Both of them are defined on the normal forms of event expressions (see Section 2.1).

The following definition introduces two predicates. The first one states that there is a binary relation on pairs of primitive events and times, called *matches*. The predicate $(e_o, t_o) matches (e_r, t_r)$ states that if event e_o occurs at time t_o, it matches the event e_r which was registered at time t_r.

The second relation holds between sets of event-time pairs. Its first argument is a set of primitive events known to have occurred at a given time. The second argument is a set of primitive events registered at a given time. The predicate is true if for each registered event there is an occurring event that matches it.

$\_matches\_ : (EVENT \times TIME) \leftrightarrow (EVENT \times TIME)$

$\_set\_matches\_ : \mathbb{P}(EVENT \times TIME) \leftrightarrow \mathbb{P}(EVENT \times TIME)$

$\forall occurs, looked\_for : \mathbb{P}(EVENT \times TIME) \bullet$
$\quad (occurs\ set\_matches\ looked\_for \Leftrightarrow$
$\quad (\forall et : looked\_for \bullet (\exists et' : occurs \bullet et'\ matches\ et)))$

We must now distinguish between events that are sticky matched and those that are matched transiently. For user-defined events sticky matching was a design decision, but transient matching is not reasonable for monotonically increasing entities like time.

$$sticky\_events : \mathbb{P}\, EVENT$$

$$TE \cup UE \subset sticky\_events$$

Apart from temporal and user-defined events, certain object events that are based on time (time stamps on file modification, for example) are also sticky matched.

In sticky matching, to match an *or*-set, each of the *and*-sets it contains must be considered. To match an *and*-set, we must consider all the sequences it contains. The *head* of each sequence represents an event we are looking for. If the sticky event matches the head of one of the sequences, the head of the sequence is discarded due to sticky matching. If this makes the sequence empty, the whole sequence is removed from the *and*-set because the corresponding *then* expression is fully matched. These steps are defined using the function *prune*. If the head of a sequence does not match the occurring event, it is left unchanged.

An event expression is fully matched when its corresponding match status contains the empty set. This means that one *and*-set has been fully matched (all the sequences contained in the *and*-set have been removed).

$$prune : \mathbb{P}(\mathrm{seq}\, EVENT) \longrightarrow \mathbb{P}(\mathrm{seq}\, EVENT)$$

$$sticky\_match\_and : (EVENT \times TIME) \times (\mathbb{P}(\mathrm{seq}\, EVENT) \times TIME) \longrightarrow \mathbb{P}(\mathrm{seq}\, EVENT)$$

$$sticky\_match\_or : (EVENT \times TIME) \times (MATCH\_STATUS \times TIME) \longrightarrow MATCH\_STATUS$$

$$fully\_matched\_ : \mathbb{P}\, MATCH\_STATUS$$

$$\begin{aligned}
&\forall\, e : EVENT;\ as : \mathbb{P}(\mathrm{seq}\, EVENT);\ os : MATCH_STATUS;\\
&\quad t_o, t_r : TIME \bullet\\
&\quad prune\ as = tail(\!|as|\!) \setminus \{\langle\rangle\} \land\\
&\quad sticky_match_and((e, t_o), (as, t_r)) =\\
&\qquad (\mathbf{let}\ mas == \{s : as \mid (e, t_o)\ matches\ (head\ s, t_r)\} \bullet\\
&\qquad (as \setminus mas) \cup prune\ mas\ ) \land\\
&\quad sticky_match_or((e, t_o), (os, t_r))\\
&\qquad = \{as : os \bullet sticky_match_and((e, t_o), (as, t_r))\} \land\\
&\quad fully_matched\ os \Leftrightarrow \varnothing \in os
\end{aligned}$$

For our example of Section 2.1, let us assume that the time the specification was registered is t_r and that e_1 is a sticky matched event. When an event e occurs at time t_o such that (e, t_o) and (e_1, t_r) match, the resulting match status looks as follows:

$$\{\{\langle e_3\rangle, \langle e_2, e_3\rangle\}, \{\langle e_4\rangle, \langle e_2, e_4\rangle\}\}$$

In each of the two *and*-sets and each of the sequences e_1 has been removed because it has matched and stays matched. If e_2 matched next, it would also

be removed from the match status, resulting in $\{\{\langle e_3 \rangle\}, \{\langle e_4 \rangle\}\}$. This means we wait for e_3 or e_4 to be matched.

If, however, e_2 and e_3 matched simultaneously, both of them would be removed from the match status, resulting in the same match status as if only e_2 had matched: $\{\{\langle e_3 \rangle\}, \{\langle e_4 \rangle\}\}$. This means that e_3 has to be matched once more at a later time in order to fully match the event expression.

The transient matching case must match *sets* of events in order to determine if two or more events are matched at the same time (to match event pattern with the combinator *and*). Therefore, the second arguments of *match_and* and *match_or* are sets of events instead of single events.

$$
\begin{array}{l}
transient\_match\_and : (\mathbb{P}\, EVENT \times TIME) \\
\qquad \times (\mathbb{P}(\mathrm{seq}\, EVENT) \times TIME) \longrightarrow \mathbb{P}(\mathrm{seq}\, EVENT) \\
\\
transient\_match\_or : (\mathbb{P}\, EVENT \times TIME) \\
\qquad \times (MATCH\_STATUS \times TIME) \longrightarrow MATCH\_STATUS \\
\hline
\forall\, es : \mathbb{P}\, EVENT;\ as : \mathbb{P}(\mathrm{seq}\, EVENT);\ os : MATCH\_STATUS; \\
\quad t_o, t_r : TIME \bullet \\
\quad transient\_match\_and((es, t_o), (as, t_r)) = \\
\qquad (\textbf{if } \{e : es \bullet (e, t_o)\}\ set\_matches\ \{s : as \bullet (head\ s, t_r)\} \\
\qquad \textbf{then } prune\ as \textbf{ else } as) \wedge \\
\quad transient\_match\_or((es, t_o), (os, t_r)) \\
\qquad = \{as : os \bullet transient\_match\_and((es, t_o), (as, t_r))\}
\end{array}
$$

Like sticky matching, all *and*-sets contained in the match status have to be considered. To match an *and*-set, all the heads of all the sequences contained in it are considered. If *all* of these are matched by one of the occurring events they are removed. Otherwise the *and*-set stays as it is.

For our example, let us now assume that e_1 is not a sticky matched event. Then, a match is only possible if both e_1 and e_2 are matched by one of the occurring events. The resulting match status would be

$$\{\{\langle e_3 \rangle\}, \{\langle e_4 \rangle\}\}$$

We are now waiting for e_3 or e_4 to occur which will fully match the event expression.

To define the operations dealing with matching, we need an auxiliary schema stating that matching leaves everything unchanged except the match status and the time of registration of a specification. Moreover, we define the effect of sticky matching on a single specification. A specification that is not suspended and is not yet fully matched is retained, however with a possibly different match status, as defined by *sticky_match_or*. A specification with a repeat-flag that is fully matched must again be registered.

$$SpecMatch \mathrel{\widehat=} [SpecOp \mid suspended' = suspended]$$

SpecSticky

$GlobalSystemClock$
$SpecMatch$
$e? : EVENT$

$suspended = yes \Rightarrow$
$\quad match\_status' = match\_status \land reg\_time' = reg\_time$

$suspended = no \Rightarrow$
$(\textbf{let}\ ms == sticky\_match\_or((e?, current\_time),$
$\qquad\qquad (match\_status, reg\_time)) \bullet$
$\quad \neg\ fully\_matched\ ms \land match\_status' = ms \land$
$\quad reg\_time' = reg\_time$
$\quad \lor$
$\quad fully\_matched\ ms \land repeat = yes \land match\_status' = norm\ e$
$\quad \land\ reg\_time' = current\_time)$

The execution of an action can cause further events to occur. The following function yields the set of events invoked by execution of an action.

$generate\_events : ACTION\_EXP \longrightarrow \mathbb{P}\ EVENT$

The next operation defines how an event announcement is treated. The output of this operation is the set of events that will be generated by the invoked actions.

ProcessStickyEvent

$GlobalSystemClock$
$\Delta\ YeastState$
$\Xi PossibleEvents$
$e? : EVENT$
$es! : \mathbb{P}\ EVENT$

$e? \in possible\_events \cap sticky\_events$
$groups' = groups$
$specs' = \{SpecOp \mid \theta Spec \in specs \land SpecSticky \bullet \theta Spec'\}$
$groups' = groups$
$es! = \bigcup \{s : specs \mid s.suspended = no \land$
$\qquad fully\_matched(sticky\_match\_or((e?, current\_time),$
$\qquad\qquad (s.match\_status, s.reg\_time))) \bullet$
$\qquad generate\_events(s.a)\}$

The client command *Announce* is just *ProcessStickyEvent*, where $e? \in UE$.

The treatment of non-announced events is similar, except that sets of occurring events have to be considered and the transient matching functions are applied.

SpecTransient

$$
\begin{array}{l}
GlobalSystemClock \\
SpecMatch \\
es? : \mathbb{P}\, EVENT \\
\hline
suspended = yes \Rightarrow \\
\quad match\_status' = match\_status \land reg\_time' = reg\_time \\
suspended = no \Rightarrow \\
(\textbf{let}\ ms == transient\_match\_or((es?, current\_time), \\
\qquad\qquad (match\_status, reg\_time)) \bullet \\
\quad \neg\ fully\_matched\ ms \land match\_status' = ms \land \\
\quad reg\_time' = reg\_time' \\
\quad \lor \\
\quad fully\_matched\ ms \land repeat = yes \land match\_status' = norm\ e \\
\quad \land\ reg\_time' = current\_time)
\end{array}
$$

ProcessTransientEvents

$$
\begin{array}{l}
GlobalSystemClock \\
\Delta\, YeastState \\
\Xi PossibleEvents \\
es? : \mathbb{P}\, EVENT \\
es! : \mathbb{P}\, EVENT \\
\hline
es? \subseteq (possible\_events \setminus sticky\_events) \\
groups' = groups \\
specs' = \{SpecOp \mid \theta Spec \in specs \land SpecTransient \bullet \theta Spec'\} \\
groups' = groups \\
es! = \bigcup \{s : specs \mid s.suspended = no \land \\
\qquad fully\_matched(transient\_match\_or((es?, current\_time), \\
\qquad\qquad (s.match\_status, s.reg\_time))) \bullet \\
\qquad generate\_events(s.a)\}
\end{array}
$$

For better performance, the system removes specifications which are known to be unmatchable. Examples of unmatchable events are those that are to be matched in the past or contradictory events that are combined with *and*. Like matching, the unmatchability of event expressions is defined inductively over its match status or normal form. The predicate $unmatchable\_prim(e, t_r, t_c)$ means that the event e which was registered at time t_r is unmatchable at time t_c. We do not fully define unmatchability for primitive events or when events are contradictory.

$$
\begin{array}{l}
unmatchable\_prim\_ : \mathbb{P}(EVENT \times TIME \times TIME) \\
contradictory\_ : \mathbb{P}(\mathbb{P}(\mathrm{seq}\, EVENT) \times TIME \times TIME) \\
unmatchable\_and\_ : \mathbb{P}(\mathbb{P}(\mathrm{seq}\, EVENT) \times TIME \times TIME) \\
unmatchable\_ : \mathbb{P}(MATCH\_STATUS \times TIME \times TIME) \\
\hline
\forall\, ms : MATCH\_STATUS;\ as : \mathbb{P}(\mathrm{seq}\, EVENT);\ e : EVENT; \\
\quad t_r, t_c : TIME \bullet \\
\quad (unmatchable\_and(as, t_r, t_c) \Leftrightarrow \\
\qquad (\exists\, s : as \bullet (\exists\, e' : \mathrm{ran}\, s \bullet unmatchable\_prim(e', t_r, t_c))) \vee \\
\qquad contradictory(as, t_r, t_c)) \wedge \\
\quad (unmatchable(ms, t_r, t_c) \Leftrightarrow \\
\qquad (\forall\, as' : ms \bullet unmatchable\_and(as', t_r, t_c)))
\end{array}
$$

With these definitions, the specification of the operation to remove unmatchable specifications is straightforward.

RemoveUnmatchableSpecs

$$
\begin{array}{l}
GlobalSystemClock \\
\Delta\, YeastState \\
\Delta\, SpecState \\
\Xi\, PossibleEvents \\
\hline
groups' = groups \\
specs' = specs \setminus \{s : specs \mid s.suspended = no \wedge \\
\qquad unmatchable(s.match\_status, s.reg\_time, current\_time)\}
\end{array}
$$

The above specification illustrates quite clearly how Yeast and similar event-action systems work. The matching process is fully defined once it is clear when primitive events match. The formal specification is significantly shorter than the program code. To answer questions regarding the system, it may be easier to consider the formal specification than to scan the program code.

3 Assessment of the formal specification

Apart from gaining deeper insight into principles of event-action systems the formal specification can be of further use in the future to application writers as well as designers of event-action systems. It contributes to the ongoing discussions on places where formal specifications are feasible and useful and how formal specification techniques can achieve a more wide-spread use in the future.

The specification has already been used to investigate the principles of software architectures in that an architectural style has been made concrete in the form of a formal specification [HK95]. We believe that other kinds of architectural styles can be treated similarly, with the same benefits as for event-actions systems.

3.1 Usefulness of the formal specification for users and application writers

While applications that involve a dozen or so Yeast specifications are quite manageable in a semantic sense, i.e., the interactions between the event patterns in the specifications can be easily understood, larger applications are not amenable to straightforward analysis. Often, the inner workings of Yeast are hidden from the application writers. While this is useful in many cases, there are occasions when serious application writers need to understand the precise semantics of Yeast in order to take advantage of its features and to verify their interpretation of Yeast semantics. A formal specification of Yeast yields precisely this clearer understanding in a mathematical sense.

The above specification can be used to show concrete facts about Yeast[1]. Simple facts are, for example, that *addspec* and *rmspec* are inverses of each other:

$$(addspec \fatsemi rmspec) \mid sr? = lab(\{l!\}) \vdash \Xi YeastState$$

The relation between *suspspec* and *fgspec* can be expressed similarly. More interesting facts can be derived by considering the matching operations. Here, we can show that (i) announced events cannot be withdrawn, and (ii) events are not kept. Once they occur, they are used immediately and then discarded.

Fact (i) follows from the definition of sticky matching, see *ProcessStickyEvent*. Once an event is announced all non-suspended specifications are considered to see if they are "waiting" for the event. If this is the case, the matching is performed and the corresponding part of the match status is discarded. There are no operations to undo this. Fact (ii) follows from the definition of both event processing schemas. The events that occur are only an input that is used for matching. There is no means to store the occurring events. If an event is announced and there is no specification "waiting" for it, the event is "lost", i.e. the situation is no different from the one where it had not been announced. This deduction is based on the assumption that there are no "secret" storage possibilities that we did not include in the formal specification.

These facts are of much importance to application writers and users. Let us suppose we want to use Yeast for monitoring a software maintenance process. This can be done by defining an attribute *debugged* on the object class *file* which is a boolean value. The persons who carry out the debugging will announce that their file is debugged when they consider their task as finished. If, however, some time later they discover an error and announce that the file is *not* debugged any more, Yeast cannot take notice of this new situation because of fact (i). This is important to know not only for application writers, but also for persons who only use predefined applications. For more details of this example, see [IKY93].

[1] This is based on the assumption that the formal specification indeed captures Yeast's behavior in a correct and sufficiently complete way.

3.2 Usefulness of the formal specification for designers

The formal specification makes the underlying design principles of event-action systems more explicit. The important points to be considered while designing and implementing such systems are identified. Designers can easily locate the criteria of interest in the formal specification which should guide them in their implementation. The specification can be used as a "template" for future event-action systems. Each design decision that was taken for Yeast (e.g. to have a mixture of sticky and transient matching) can be questioned, and the specification can be modified accordingly. Our formal specification helps to disseminate and make explicit the design knowledge that has been assembled during several years. It will continue to be useful for designers by serving as a verification mechanism after completion of the realization.

The specification can also help in the understanding of other event-action systems since they are based on similar concepts. Besides making those concepts explicit, it can also be used to generate test cases that help discover the properties of the system to be analyzed.

3.3 General lessons learned on formal specification

The formal specification community can benefit from looking at our specification of a practical usable tool in use in an industrial setting. Being able to locate properties quickly for future realizations of event-actions systems gives an impetus to create formal specifications of applications where none exist.

Several points can be made concerning the application of formal specification techniques in general:

- It is useful to specify already existing systems. Such a specification cannot guide the implementation but gives a documentation of the system that can be used further in many ways. This is demonstrated by the points made in Sections 3.1 and 3.2.
- Formal specification techniques should be applied in a pragmatic way: if the specification cannot be expressed at a higher level of abstraction than the code then there is no point in writing it, at least if the code already exists.
- The formal specification is much more concise than the program text. Partially, this may be due to the fact that we did not insist on including every detail in the formal specification. The conciseness refutes one of the arguments occasionally made against formal specifications, namely that they are as long and as unreadable as the program source.
- To set up the formal specification, considerably less time and effort was needed than for the implementation. This shows that formal specification is not only interesting for safety-critical systems where costs are not a

problem. For non-critical systems like Yeast their application is feasible because the costs are not exceedingly high, at least if the persons applying the formal methods are sufficiently trained.

- For Yeast, we have the ideal case that the program and its documentation were written by the same persons. This is not true for the majority of software systems. Quite often the user manuals are not written by the implementors. In this case, the formal specification is a much better basis for the user manual than the program code[2] because it is not only much shorter but also more abstract: the formal specification concentrates on those aspects of the system that are relevant for its behavior; implementation details are not shown. For the future, one might even consider to store a formal specification in addition to manual pages. If on-line explanations can be generated from the formal specification we may be able to reduce the effort to create manual pages.

We modeled some of the information the program needs to work correctly from the environment not by inputs but by a schema (*ClientCommand*) of which only the undecorated version is imported. We deliberately did not specify any changes to the environment. From a logical point of view, this is equivalent to modeling with inputs. From a pragmatic point of view, this indicates that no user input should be required for the components of the environment schema. Instead, the system is to use information that is available internally.

4 Related work and conclusions

The specification language Z we used for this case study has gained considerable acceptance in academia as well as in industry. Of the twelve industrial projects reviewed by Craigan, Gerhart and Ralston [CGR93], four used Z, all within the commercial cluster. In several of the projects, including the work presented here, an already existing system was specified. Mostly this was done for the purpose of re-engineering the system. It appears that the usefulness of formal specifications is seen only when there is some trouble in maintaining and developing the existing system further.

The re-engineering aspect is also of importance to Yeast. Currently, a concurrent version of Yeast is under development. The formal specification of Section 2 will be used to contrast the sequential and the concurrent version and to support design decisions for the latter.

In contrast to the industrial case studies described in [CGR93], our specification does not only support re-engineering and maintenance, but is a representation of general design and architectural knowledge. As far as design knowledge

[2] User manuals will make use of informal specification and design documents which can be ambiguous; the clarification can be provided via the formal specification or the program code.

is concerned, Jacky [Jac95] has similar aims. His Z specification of a medical device was written to serve as a starting point for the specification of other safety-critical control systems. Our investigation of architectural styles [HK95], which was a strong motivation to conduct this case study, goes much further.

Acknowledgments

We thank Emden Gansner, Peter Mataga and Thomas Santen for their useful suggestions.

References

[CGR93] Dan Craigan, Susan Gerhart, and Ted Ralston. An international survey of industrial applications of formal methods. Technical Report NISTGCR 93/626, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, 1993.

[HK95] Bi-directional approach to modeling architectures, July 1995. Maritta Heisel and Balachander Krishnamurthy, Submitted to ICSE-18.

[IKY93] 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*, pages 105–120. Springer-Verlag, April 1993. Published as *Lecture Notes in Computer Science* no. 668.

[Jac95] Jonathan Jacky. Specifying a safety-critical control system in Z. *IEEE Transactions on Software Engineering*, 21(2):99–106, February 1995.

[KR95] Balachander Krishnamurthy and David Rosenblum. Yeast: A general purpose event-action system. *IEEE Transaction on Software Engineering*, 1995. To appear.

[Spi92] J. M. Spivey. *The Z Notation: A reference manual.* Prentice Hall, Englewood Cliffs, NJ, 1992.