Yeast: A case study for a practical use of formal methods

Paola Inverardi [1] Balachander Krishnamurthy[2]
Daniel Yankelevich[3]

Abstract

This paper discusses a formal semantics for an existent event-action system, Yeast, developed at AT&T Bell Laboratories. Yeast is a good case study for the use of (true-concurrent) semantic techniques since causal dependence among events, concurrency, nondeterminism and conflicting behaviour of specifications can all be modeled in Yeast. We discuss the use of the formalization in the verification of correctness of real Yeast applications with respect to various properties.

1 Introduction

In this paper we present a formal semantics for an existent event-action system, Yeast, developed at AT&T Bell Laboratories[1]. Yeast supports cooperative work in a distributed computing environment. It allows users to define event-driven specifications which when matched can trigger actions. Yeast manages events through a global space shared among all the users. Some of the events are asynchronous since they can come from the external environment. Yeast is used for a variety of applications, from deadline notification to software configuration management. A Yeast application consists of the set of specifications written to model the domain as well as the events that match the specifications.

The motivation for our work is twofold: i) From the theoretical side, Yeast represents a good case study for the use of (true-concurrent) semantic techniques [7]. Despite their conceptual simplicity, Yeast specifications exhibit a number of interesting behaviours such as causal dependence among events, concurrency, nondeterminism and conflict; ii) From the applicative side, we are interested in verifying the correctness of Yeast applications with respect to various properties. Along with creating Yeast applications we would like to be able to identify specifications that could be enabled by conflicting events, or perform a run-time ordering of specifications. A practical need thus emerges for a formal basis on which to carry out reasoning over Yeast applications.

We define the Yeast semantics in two steps: First, we provide a distributed operational semantics by means of structured inference rules. This operational semantics is distributed in the sense that in each rule only the requirements (pieces of the state) to enable it are expressed, i.e. no global information is required. The second step defines how to retrieve a Petri net representation of Yeast applications from the operational description. A Petri net allows dependency (causality), independency (concurrency) and contraposition (conflict) among a collection of Yeast specifications to be clearly represented. At this second semantic level, we formally define some properties a Yeast user may want to verify on the application.

[1]I.E.I–C.N.R., Pisa, Italy. Research partially funded by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo
[2]AT&T Bell Laboratories, Murray Hill, NJ 07974 USA
[3]Dipartimento di Informatica, Università di Pisa, Italy

Once the semantics has been provided, there are two approaches to verifying the correctness of Yeast applications. The first one is to perform analysis and verification on the Petri net. This approach relies entirely on the Petri net representation. Given the state of the art, we do not think that this approach is always practical due to the strong limitations in the ability to translate reasonable scale applications efficiently into the net semantic description. Even when the translation has been accomplished, the management of the net description both in terms of computational and space complexity can be hard.

The second approach is to build verification tools that perform the required analysis directly at the application level, i.e. over Yeast source specifications based on the semantic description we have provided. In this work we are interested in pursuing this approach. This is not a simple task and one cannot expect to be completely successful. There can be a price to be paid in terms of completeness of the verification process. In fact, in order to be practical, sometimes it may be convenient to apply heuristics or ask for user intervention.

We first describe Yeast briefly and then discuss a scenario of a Yeast application together with properties of interest. We then review operational semantics and Petri net preliminary definitions and results. The formal model section provides the operational semantics and Petri net representation of the subset of Yeast we consider. Then, this is applied on the specifications in the scenario along with an analysis of the properties of interest. In the same section, the approach we are taking to check these properties on Yeast specifications together with hints on our prototypal implementation in Prolog of two simple analysis tools are presented. We conclude with a section on current and future work.

2 Yeast

Yeast (Yet another Event-Action Specification Tool) is a general-purpose platform for constructing distributed event-action applications using high-level specifications. Yeast supports a wide variety of 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. A general-purpose event-action system makes it easier for arbitrary applications to eliminate special purpose event-action matchings.

Yeast was designed with some requirements in mind:

- The specification language must be simple, yet powerful.
- There should be no restriction on the actions that users can specify to be performed in response to the occurrence of user-specified event patterns.
- Users of the system must be able to interactively query the status of specifications they have registered with the system.

Yeast is based on a client-server architecture in which distributed clients register, manipulate and query event-action specifications, and a server performs specification matching and management. A Yeast specification consists of an event pattern along with an action; the server triggers the action whenever it detects an occurrence of the associated event pattern. The action part of a specification can perform any number of actions in response to the matching of the event pattern, including Yeast-related actions.

Event patterns consist of either temporal events or object events. Temporal events are the familiar calendar events (e.g., *at* a particular time) whereas object events are changes in attribute values of an object. Example objects that Yeast knows about are *files*, *directories* and *machines*. Example attributes of files are *size*, *owner*, etc. Yeast can match patterns that are connected by the connectives *and*, *or*, and *then*, which have straightforward semantics (e.g. both sides of an *and* pattern have to be true for the pattern to be true). The *then* connective implies that the left hand side must be true before checking is performed on the right hand side.

The significant difference between other event-action tools such as *cron*, and Yeast is that Yeast can be told about new events. Users can define new object classes and attributes for them. Since change in values of user defined attributes cannot be detected, they have to be *announced* to Yeast.

A simple example shows the extensibility of Yeast. The *defattr* command (part of Yeast) defines a new attribute of a particular type.

```
defattr file debugged boolean
```

Users can now make specifications that are matched when the *debugged* attribute attains a particular value. For example,

```
addspec file libx debugged == true do make system
```

Once Yeast is notified that the file *libx* has been debugged, the system is automatically rebuilt via the *make system* action. The user who has authentication over the attribute *debugged*, after debugging the *libx* file, sends the following announcement:

```
announce file libx debugged = true
```

The announcement facility makes Yeast extensible and thus applicable to a wide range of tasks.

3 A Software Development Scenario

In this section we present a small Yeast application that attempts to model a portion of a software development process. We then present a set of questions of interest which, while specific to the scenario, outline hints at the general properties of specifications that we would like to analyze.

3.1 Scenario

In a software development environment, change that needs to be done as a result of suspected problems or enhancement requests is labeled Modification Request or MR for short. Typically MRs are reviewed by an MR review board who decide which version of the software needs to be changed and which developers are responsible for each change. The MR is assigned to the responsible developers who fix the problem. After testing their changes locally, the developers *submit* the MR, upon which the modified software is rebuilt by an integrator. An approval team typically then approves the modified software, enabling it to be distributed.

For the purposes of this scenario we will consider a simplified example skipping over the MR review process and selection of the appropriate version of the software.

When a MR comes in, it is assigned a number and its status is set to *active*. Suppose it is assigned to a group of three people who have authority to fix the problem. Theoretically they are supposed to agree amongst themselves that the problem has been fixed and then notify the integrator of this fact.

The steps in the portion of the process we model via Yeast are:

- Submitting an MR,
- Rebuilding of software component by the integrator.
- Approval by the approver.

Additionally, suppose that there are temporal constraints in the process. Say, the MR has been assigned at 8AM Monday; developers have until 5PM Wednesday to fix the problem; the integrator then has until 5PM Thursday to rebuild it and the approver must approve it by 5PM Friday. The dependencies are clear: software cannot be approved until it is rebuilt, and rebuilding cannot occur until the MR has been submitted.

With this as background, let us look at a Yeast model of the above process. We consider some potential problems with this model later. For the scenario, let us assume MR23 is the MR in question on the software module m1, and that it was assigned to developers Anna, Bala, and Chico, the integrator is Dan and the approver is Eduardo. The Yeast command `defobj` defines a new object class, and `defattr` defines a new attribute for an object class.

```
defobj MR                       # models a modification request
defattr MR status string        # string attribute of  MR status

defobj module                   # module that has associated MR
defattr module rebuilt boolean  # boolean attr of module status
defattr module approved boolean # boolean attr of module status
```

After the developers test their changes, any of them can send an announcement stating that the MR has been submitted by the developer as follows:

```
announce MR MR23 status = devsub
```

The specification that the above announcement would match is made by the integrator Dan:

```
addspec MR MR23 status == devsub do rebuild m1          (s1)
```

where rebuild is presumably a script for rebuilding software modules. If the *rebuild* script succeeds in rebuilding m1, then the following announcements are generated as part of the script.

```
announce module m1 rebuilt = true
announce MR MR23 status = submitted
```

If the *rebuild* script fails then the following announcement is sent

```
announce module m1 rebuilt = false
```

Likewise, the following specification is made by the approver Eduardo

```
addspec module m1 rebuilt == true do TestAndApprove m1  (s2)
```

where TestAndApprove is a script that tests the module and if the tests are successful generates the following announcements

```
    announce module m1 approved = true
    announce MR MR23 status = approved
```

Only Dan has the necessary authentication on the attribute *rebuilt* of the object-class module. Similarly, only Eduardo has authentication on the attribute `approved`.

If the status of the MR is still active at 5PM Wednesday then the integrator needs to be notified about this. Likewise, if the module `m1` has not been rebuilt by 5PM Thursday the approver needs to be notified. These are the temporal constraints in this mini-process.

```
addspec at 5pm Wednesday and MR MR23 status == active do
    Nobuild dan m1                                          (s3)
```

where Nobuild is a script that notifies the integrator via mail and announces the failure of the rebuild:

```
    Mail -s missed_MR23_deadline dan
    announce module m1 rebuilt = false
```

```
addspec at 5pm Thursday and module m1 rebuilt == false do
    Noapprove eduardo m1                                    (s4)
```

where Noapprove is a script that notifies the approver Eduardo and announces the failure of the approval:

```
    Mail -s missed_MR23_deadline eduardo
    announce module m1 approved = false
```

3.2 Properties of interest

The Yeast modeling of the process has potential problems and some properties of interest. An analysis of the specifications should bring these out.

Since any of the three developers (Anna, Bala, Chico) can set the *status* attribute of `MR23`, there is a possibility that one person may set it to be *devsub* which matches specification S1 triggering a rebuild of `m1`. If another developer sets the status to *active* later, that will match S3 at 5PM Wednesday triggering a `m1 rebuilt = false` announcement. We would like to know about such possibilities *a priori.*

The following questions are of interest as it relates to analysis of the specifications:

- What is the dependency pattern on the specification/events?
- Is there a time ordering of the specification matching times?
- What are the contrary events possible that could cause a deadlock or lead to potentially erroneous conclusions?
- Which specifications will be the last to be matched?
- Which specification will be matched/attempted to be matched first?

4 The Formal Model

In this section we present the formal model we have defined for the Yeast language. Before introducing the model we recall the preliminary definitions and notions we need. In the following section we introduce the notion of Place/Transitions nets and operational semantics. For a detailed presentation of these topics the reader should refer to [5] and [6] respectively.

4.1 Preliminary Definitions

In this section, we review the main definitions of Place/Transition (P/T) nets and of operational semantics.

Definition 1 P/T Net
A net $N = (S, T, W, M_0)$ consists of disjoint sets S and T of places and transitions, the weight function $W : S \times T \cup T \times S \rightarrow \mathbb{N}$ and the initial marking $M_0 : S \rightarrow \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers.

Definition 2 Pre- and post- sets
Given a net $N = (S, T, W, M_0)$ and an element $x \in S \cup T$,

- *the pre-set of x is ${}^\bullet x = \{y \in S \cup T \mid W(y, x) \neq 0\}$*
- *the post-set of x is $x^\bullet = \{y \in S \cup T \mid W(x, y) \neq 0\}$*

Definition 3 Markings and enabling
Given a net $N = (S, T, W, M_0)$, a marking *is a function $M : S \rightarrow \mathbb{N}$. A transition $t \in T$ is enabled under a marking M, denoted $M[t >$ if for all $s \in^\bullet t$, we have $M(s) \geq W(s, t)$. An enabled transition may occur, producing a follower marking M', written $M[t > M'$, if $M[t >$ and $M'(s) = M(s) - W(s, t) + W(t, s)$ for all $s \in S$. In this case, $M[t > M'$ is called a* step.

A marking M is called *reachable* if for some sequence of steps $t_1 \ldots t_n$, $M_0[t_1 > M_1 \ldots M_{n-1}[t_n > M$.

As far as the operational semantics is concerned we take the usual SOS approach [6]. That is we describe the operational behaviour of a system in terms of inference rules which define a relation between states of the system.

Definition 4 Operational Semantics
Let S denote the states of the system under description and Act the actions produced by the system when evolving from a state to another state. Then an operational semantics OP is a set of inference rules of the form $\frac{Premises}{Conclusion}$ defining a relation $D \subseteq S \times Act \times S$; D is the least relation satisfying the rules.

In the following section we precisely define what are, in our case, states and actions.

4.2 Operational Semantics

In this section we develop the operational semantics reflecting the dynamic behaviour of the language. The operational semantics is defined starting from a syntactic presentation of the language which is slightly different from the grammar presented in the Appendix. The main differences are on the representation of events and of actions. Events are to be considered as unique. That is, two specifications which *use* the same event give rise to two distinct events. In practice, this is realised by subscripting the event with the specification identifier. As far as actions are concerned, we introduce the notion of action with parameters with the convention that action sequences are built with the ';' operator, and () denotes the empty action sequence (i.e. do nothing). When an action has more than one parameter, it indicates a branching upon a condition, and depending on the condition one among the actions (or action sequences) is executed. In this way complex *scripts*, or the intervention of a user, are modelled as a nondeterministic choice, i.e. a branch which depends on a condition that is outside of the system. For instance, the complex script described as `TestandApprove` in the specification `s2` of the SDS example in section 3.1 can be modelled as T&A ()(announce(m1.approved =true); announce(mr23.status = approved)), which reflects the fact that if the test fails, nothing is done otherwise two announcements are issued. Actually this is exactly the formalization we will use in Section 4.4 when modelling the whole SDS example.

Before introducing the rules of the operational semantics we define what is a state and an action of the system. Informally, a **state** of the system is given by a set of specifications plus the conditions that are fulfilled at a particular moment, that is, events that are true and which enable a specification.

Notice that Yeast attributes are not variables, i.e. the value of an attribute is not persistent. For example, if an announcement of the event *att1 = false* is done and after that, a specification of the form *att1 == false do mail(vladimiro)* is included, the mail to vladimiro is not sent, since the announcement of *att1 = false* has been done previously and the system does not remember it.

Definition 5 *Let A be a Yeast event and s be a Yeast specification. Then, A_s is a* state component. *Moreover, if A is of the form objectattribute = value, then $att(A_s) =$ objectattribute and $val(A_s) = value$.*

For instance, $att((mr23.status = active)_{s3}) = mr23.status$ and $val((mr23.status = active)_{s3}) = active$.

A state component is an event with a subscript identifying which specification it (partially) enables. We define the subindex *cl* for clock, and events subscripted with a *cl* are consumed by the system clock to produce new time events. Sub-indexes are omitted when they are obvious from the context, for example if the event corresponds to just one specification.

The state of the system is composed of the current set of specifications plus the set of possible enablings, i.e. the events which are valid; which leads to our next definition.

Definition 6 *The states of the system are pairs (S, E), where S is a set of specifications and E is a set of state components.*

The set $\{s_1, s_2, \ldots s_n\}$ is denoted by $s_1 \oplus s_2 \ldots \oplus s_n$. Similarly $\{e_1, e_2, \ldots e_n\}$ is denoted by $e_1 \oplus e_2 \ldots \oplus e_n$. Notice that $\oplus$ is associative and commutative. The sets

S and E have ϕ_S and ϕ_E as neutral element respectively; furthermore $\oplus$ is idempotent with respect to these neutral elements.

As far as **actions** are concerned, we just take the action part of Yeast specifications. Therefore the rules we are going to present define a relation among states of the system, i.e. $OP \subseteq (S \times E) \times Act \times (S \times E)$.

The operational semantics is defined by means of one inference rule schema. This rule schema stands for different rules which are obtained by instantiating the ACT parameter with the action parameters of the yc function and substituting the bottom left part with the corresponding output value of the yc function. The rule schema is:

$$\frac{E \vdash e, S \vdash addspec(e, ACT)}{S, E \xrightarrow{ACT} yc(ACT, S - addspec(e, ACT), E - e)}$$

where yc is recursively defined as follows:

$$\begin{array}{l}
yc : ACT \times S \times E \to S \times E \\
yc(\{a_1, \ldots, a_n\}, S, E) = yc(a_n, yc(\{a_1, \ldots, a_{n-1}\}, S, E)) \\
yc(spec, S, E) = (S \cup spec), E \\
yc(defobj, S, E) = yc(defattr, S, E) = S, E \\
yc(announce(e), S, E) = S, E \cup e \\
yc(shell\_cmd, S, E) = S, E
\end{array}$$

and

$$\frac{e \in E}{E \vdash e} \qquad \frac{s \in S}{S \vdash s}$$

$$\frac{E \vdash e}{E \vdash e \text{ or } e'} \qquad \frac{E \vdash e, e'}{E \vdash e \text{ and } e'}$$

For example, an instance of the rule schema for $ACT = addspec(e', A)$ is:

$$\frac{E \vdash e, S \vdash addspec(e, addspec(e', A))}{S, E \xrightarrow{addspec(e',A)} (S \cup \{addspec(e', A)\}), E}$$

The obvious semantics of these rules is that if the preconditions (i.e. all the expressions which appear in the upper part of the inference rule) are true then it is possible to infer the conclusions (what appears in the bottom part of the rule). In our case we say that if from the set of events E it is possible to derive the existence of the event e and similarly, from the set of specifications S it is possible to derive the existence of the specification $addspec(e, ACT)$ then it is possible to activate the specification. This amounts in executing its action part, thus resulting in a new state of the system in which the enabled specification has been removed from the set of specifications S and in the set of events E the event which has caused the enabling has been consumed.

The last four inference rules define the notion of derivability ($\vdash$) in terms of set membership ($\in$). In general, more complex notions of derivability can be defined, for example when tackling the full Yeast language we must be able to deal with more complex event patterns which may require different assumptions to be verified.

Note that the definition of yc shows that in our model the action part of a specification is atomic. This fact is justified since it reflects the actual behaviour of the Yeast system.

Since Yeast includes timing facilities, an important point is how time is modeled. The optimum would be a model where time is specified only when it is needed, and ignored otherwise. And since time in Yeast applications can be considered discrete, we model time explicitly in the operational semantics. Thus, time events are ordered, and for each pair e, e' of time events such that e' is a immediate successor of e in the (total) timing ordering, a rule of the form $S, E \oplus e \xrightarrow{clock} S, E \oplus e'$ is considered.

Some actions are identified to be *external*, i.e. actions that a user or an external agent may perform. Such actions can occur at any moment. For instance, a user can make an announcement based on the success or failure of his/her work. Clearly it is a nondeterministic choice for the system, and this is modeled by *external* rules. So, for each external event e a rule of the form $S, E \xrightarrow{user} S, E \oplus e$ is included. To choose such a transition represents the fact that a user intervention has produced the event e.

A *computation* is just a sequence of states and rules of the form $S_1, E_1 \xrightarrow{label_1} S_2, E_2 \xrightarrow{label_2} \ldots S_n, E_n$.

4.3 A Petri Net for Yeast

The operational semantics of the language has been given in such a way that it is possible to obtain a Petri net from the rules above. In fact, in [4] it is shown that transition systems whose set of states have monoidal structure (that is, a binary operation that is associative, commutative and idempotent with respect to a zero) are in fact P/T Petri nets. We use this result here in order to derive a Petri net representation for Yeast specifications. As has been pointed out earlier, Petri nets provide a natural representation, since the properties we want to check involve *causal dependencies* among events and require a clear distinction between nondeterminism and concurrency.

Many proposals have been made for Real-time Petri nets (see for example [2]). Timed Petri nets are more complex than the model we are proposing. Here, the clock is considered a process that updates the events associated with time. Places are associated with times that are used in specifications, and the treatment of time is homogeneous since, for instance, the restrictions about time are included in the partial order associated with the net.

The Petri net corresponding to a Yeast specification is obtained directly from the operational semantic rules, since by replacing the comma with the $\oplus$ operator a transition system with monoidal structure on states is obtained.

However, only places which appear as causes of some transitions are considered in the net. In fact, no "dead places" (i.e. places with no outgoing transitions) are taken into account.

We first show how to specialize the meta-rule given in the previous section to a particular set of Yeast specifications.

Definition 7 *Let Y be a set of Yeast specifications. Let Y' be Y plus all the specifications appearing as addspec(E, A) actions in the specifications of Y. Let EY be the set of state components appearing in Y' (either as preconditions or inside actions). Then, the rules associated with Y are obtained by getting all the instances of the meta-rule choosing a specification s and an state component e for the premises such that $s \in Y'$ and $e \in EY$, and S and E are chosen to be minimal with respect to set inclusion.*

Moreover, for each external state component $e \in EY$, a rule $\phi_S, \phi_E \longrightarrow \phi_S, e$ is added, and for each pair of time state components $e_1, e_2 \in EY$ such that e_2 is the immediate *successor of e_1 in EY with respect to the time ordering, a rule $\phi_S, e_1 \longrightarrow \phi_S, e_2$ is added.*

Notice that the rules obtained are axioms of the form $S, E \xrightarrow{ACT} S', E'$. The rules of the case study given in section 4.4 are obtained in this way.

Definition 8 *Given a set of rules defining a Yeast specification Y, with EY and Y' as in the previous definition, the corresponding net N is $N = (S, T, W, M_0)$ where $S \subseteq Y' \cup EY$ is the set of state components and specifications which appear at the left of a rule, T is the set of rules, $W(s,t) = 1$ iff s appears at the left of the arrow in t and $W(t,s) = 1$ iff s appears at the right of the arrow in t, and $M_0 = Y$.*

From a brief analysis, it is easy to observe that contradictory situations may arise from a computation of the net. We call *inconsistent markings* these contradictory markings containing more than one value for an attribute.

Definition 9 *A marking $s_1 \oplus s_2 \oplus \ldots \oplus s_n$ is said to be* inconsistent *iff there exist i and j such that s_i and s_j are state components and $att(s_i) = att(s_j)$ and $val(s_i) \neq val(s_j)$.*

4.4 Case Study

In the following we show the operational semantics reflecting the dynamic behaviour the sub-processes under consideration.

In the case of our example the object under consideration are modelled as follows:

```
Event ::= mr23.status = Valstat | m1.rebuilt = Bool
      | m1.approved = Bool | 5pmWed | 5pmThu
Action ::= announce(Event) | mail(User) | rebuild(Action)(Action)
      | T&A(Action)(Action) | Action ; Action | ()
Valstat ::= devsub | submitted | active | approved
Bool ::= true | false
User ::= eduardo | dan | ...
```

We recall here that action T&A is used in the specification with two parameters, T&A ()(announce(m1.approved =true); announce(mr23.status = approved)), which reflects the fact that if the test fails, nothing is done, else two announcements are sent.

The system consists of four specifications:

(s1)	mr23.status = devsub	do	rebuild ()(announce(m1.rebuilt = true); announce(mr23.status = submitted))
(s2)	m1.rebuilt = true	do	T&A ()(announce(m1.approved = true); announce(mr23.status = approved))
(s3)	5pmW ∧ mr23.status = active	do	mail(dan); announce(m1.rebuilt = false)
(s4)	5pmT ∧ m1.rebuilt = false	do	mail(eduardo); announce(m1.approved = false)

The initial state of the system is composed of the four specifications. Formally, the initial state is the pair (s1 ⊕ s2 ⊕ s3 ⊕ s4, ϕ_E).

The rules for the system are shown in Table 1. Each transition has a label, which is the set of actions performed when the transition is executed. In this way, *side effects* (as

Table 1: Transition Rules for SDS

t1)	ϕ_S , ϕ_E	$\xrightarrow{user}$	ϕ_S, mr23.status = devsub
t2)	ϕS , ϕ_E	$\xrightarrow{user}$	ϕ_S, mr23.status = active
t31)	s1 , mr23.status = devsub	$\xrightarrow{rebuild}$	ϕ_S, ϕ_E
t4)	s1 , mr23.status = devsub	$\xrightarrow{rebuild}$	ϕ_S, mr23.status = submitted $\oplus$ m1.rebuilt = true
t5)	s2, m1.rebuilt = true	$\xrightarrow{T\&A}$	ϕ_S, ϕ_E
t6)	s2, m1.rebuilt = true	$\xrightarrow{T\&A}$	ϕ_S , m1.approved = true $\oplus$ mr23.status = approved
t7)	s3, 5pmW$_{s3}$ $\oplus$ mr23.status = active	$\xrightarrow{notify}$	ϕ_S, m1.rebuilt = false
t8)	s4 , 5pmT $\oplus$ m1.rebuilt = false	$\xrightarrow{noapprove}$	ϕ_S, m1.approved = false
t9)	ϕ_S , ϕ_E	$\xrightarrow{clock}$	ϕ_S, 5pmWed$_{cl}$ $\oplus$ 5pmWed$_{s3}$
t10)	ϕ_S , 5pmWed$_{cl}$	$\xrightarrow{clock}$	ϕ_S , 5pmThu

sending a mail, or compiling a file) are modeled as labels of transitions, and hence can be easily identified in any execution.

Each rule specifies only its *requirements* (i.e. what does it need to be enabled) on the left hand side, and its *output* (i.e. what does it produce when it is enabled). Hence, the condition that enables a rule of the form $s, e \xrightarrow{label} s', e'$ in a state (S, E) is that $s \subseteq S$ and $e \subseteq E$. The *matching* of the left hand side of the rules with the current state is done modulo associativity and commutativity of the $\oplus$.

Hence, the clock has a number of rules (t9, t10 in the table) which relate the time events which appear in the specification. In the example we are considering, only 5pm Wednesday and 5pm Thursday appear, and hence only they are considered in the rules: (t9) shows that 5pm Wednesday will be eventually true and (t10) simply says that Wednesday precedes Thursday.

The initial marking is $M_0 = s1 \oplus s2 \oplus s3 \oplus s4$.

For example, suppose that a user announces *mr23.status = active*, and after that another user announces *mr23.status = devsub*. After that, at 5pm Wednesday, specification s3 will be matched, in spite of the fact that the last announcement involving the value of *mr23.status* has set it to be *devsub*. However, there is still a token on the place *mr23.status = active*. In fact, the marking reached after the execution of both events contains two values for *mr23.status*.

4.5 Properties Analysis

We now formally specify the properties we want to analyze.

- What is the dependency pattern on the events of a specification S?
  > Let N[S] be the net derived from S. Then, the dependency pattern on Yeast events corresponds to the set of partial orderings of Yeast events obtained from the partial orderings of the places in the net by substituting each place with its label. Notice that more than one pattern is possible. For instance, if the system has three specifications:

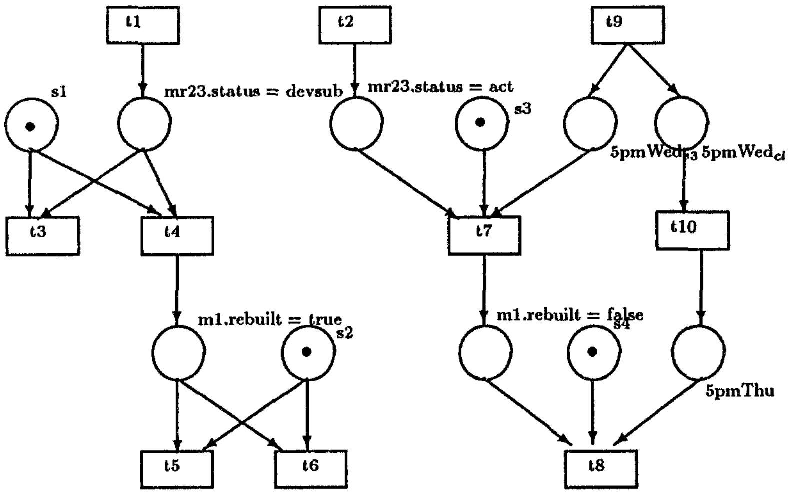

Figure 1: The Petri net for SDS

- E do script(announce(x=1))(announce(y=1))
- x=1 do announce(y=1)
- y=1 do announce(x=1)

there are two different patterns of dependencies: in one of them x=1 depends on y=1 and in the other y=1 depends on x=1.

- Which specification will be the last/first to be matched/attempted to be matched?
  > Find the set of specifications corresponding to the greatest/least events in an admissible partial order.

- What are the possible contrary events that could cause a deadlock or lead to potentially erroneous conclusions?
  > As far as deadlock is concerned, some known techniques can be used (see for example [5]). For "erroneous conclusions" we assimilate them to the generation of *inconsistent markings*, thus a specification can end up with an erroneous conclusion iff its net allows a computation to reach an inconsistent marking. For instance, in the example in correspondence with the conflict situation described in the previous section, there is a step sequence, namely

$$s1 \oplus s2 \oplus s3 \oplus s4[ext2 > s1 \oplus s2 \oplus s3 \oplus s4 \oplus m23.status = active[ext1 >$$

$$s1 \oplus s2 \oplus s3 \oplus s4 \oplus m23.status = active \oplus m23.status = devsub$$

which ends up in a state (marking) containing both *m23.status = devsub* and *m23.status = active*. This conflict is propagated in the net and the step sequence above can be extended to reach a marking containing both *m1.rebuilt = true* and *m1.rebuilt = false*.

4.6 Checking Yeast Properties

In the previous section we have identified the properties of the net that correspond to the desired properties of Yeast specifications. However, as has been pointed out in the introduction, our aim is to check properties directly on the Yeast specification: we do not want to build the net and then to verify the net, since this strategy would be too hard for a real verification tool. Moreover, checking some net properties can in many cases be very difficult. For instance, checking the absence of *inconsistent states*, which is the translation in the net of "no possible erroneous conclusions exist", implies to verify that some markings are not reachable, which can be very complex. In fact, the reachability problem, even for restricted classes of nets, has been shown to have a very high complexity [3]. Thus, the strategy of building the net and subsequent checking for reachable inconsistent states is infeasible. Our strategy is to build correct tools for the system, based on the formal model but not working directly on it.

Many interesting properties can be dealt with just by using *non standard* operational semantics, i.e. by deriving from the rules of the operational semantics more abstract interpreters [10]. An abstract interpreter, based on the net description, filters some pieces of information out and presents an abstract view of the computation. In our case one possibility is to build an abstract interpreter for the construction of *dependency patterns of specifications/events.* For instance, let C be the computation of the SDS system corresponding to the conflict situation (and to the step sequence) of subsection 4.5. The (labeled) graph of specifications/events dependencies we want to obtain from our tool is the following:

$$\begin{tikzcd}
\text{mr23.status} = \text{devsub} \arrow[d, dash] & \text{5pmWed} \arrow[dr] & \text{mr23.status} = \text{act} \arrow[d, dash] \\
\text{m1.rebuilt} = \text{true} & & \text{m1.rebuilt} = \text{false}
\end{tikzcd}$$

We have tested this approach with a prototype implementation of an interpreter of the operational semantics and of an abstract interpreter for detecting dependencies among specifications/events of a given set of Yeast specifications. The implementation of the two interpreters is in Prolog whose input are a set of specifications to be analyzed obtained by directly interfacing the Yeast environment. Since Yeast specifications are dynamic (i.e. they constantly shrink as event patterns are matched and grow when new specifications are added), to generate a snapshot of the current set of specifications, we wrote a new Yeast client command called *dumpspec.* Actually, it was a trivial modification of an existing client command that was used to dump the contents of a set of specifications. The only modification needed was the format. *Dumpspec* merely has to traverse the specification data structure and output the contents.

More precisely, what is obtained is a list of the specifications, for example the input generated in correspondence of the SDS system is:

```
[addspec(object(MR, 'MR23', userdefined,tba),'rebuild m1 ',1),
 addspec(object(module, 'm1', userdefined,tba),'TestAndApprove m1 ',2),
 addspec(and(time(718477200),object(MR, 'MR23', userdefined,tba)),
         'Mail -s missed_MR23_deadline dan ',3),
addspec(and(time(718563600), object(module, 'm1', userdefined,tba)),
        'Mail -s missed_MR23_deadline eduardo ',4)]
```

A Yeast specification, as mentioned earlier, consists of an event pattern portion and an action portion. The event pattern consists of either a simple event pattern or a combination of two or more simple patterns connected with connectors *and*, *or*. A simple event is either a temporal event or an object event. The output of a temporal event simply contains "time" and the absolute time at which the event would match. Output of an object event has three components: object class, object and the attribute. If the attribute was user defined then an extra field was output to indicate the fact that the value of the attribute is to be announced ("tba"). The final element is the specification label since it is a crucial item for analysis.

The structure of the Prolog programs is very simple since they straighforwardly reflect the structure of the operational semantics rules. The abstract interpreter is of course more abstract: it does not perform an execution of the Yeast program i.e. it is not concerned with the modification of the state S, E. The control flow among specifications is simulated in order to capture the dependency relationships.

However, not all properties can be checked in this way. Checking for reachability of inconsistent states seems to be as difficult as in the net. It is still possible to use our tools to approach the problem in a more flexible and interactive way. In general, the Yeast specifier has a certain degree of knowledge of which can be weak points of his/her set of specifications. In this context the OS interpreter we have realized can be used to implement the following strategy:

- Construct a set of hypothetical conflicting states, either by proper knowledge or via an algorithm that checks all possible pairs of conflicting attribute/values that appear in the specification.
- Use the *two ways* property of logical variables to run the OS interpreter with a partially instantiated goal where the output state contains a conflicting condition. In this way we can check if there exists a final state containing this condition.
- If the partially instantiated goal succedes then a possible conflict has been detected. Otherwise, the system is safe.

This is just an outline of the technique that we expect to use in order to perform verification of complex properties. Notice that (1) it works directly on specifications using the standard operational rules, and (2) the use of heuristics in the search can help in dealing with an otherwise intractable problem.

5 Conclusions

We have examined a way to formally study a practical event-action system, Yeast, which is presently being used for a variety of applications. We have presented a formal model of Yeast applications, and through the model we were able to answer several interesting properties about the specifications, which would otherwise be hard. Further, the model has already provided insights into the operation of the system pointing out potential conflicts in the construction of future Yeast applications.

We have also started the experimentation of building verification tools based on the formal model, the choice of Prolog as the prototypal language has been mainly of convenience since we wanted high flexibility in experimenting different verification strategies.

The next step will be moving towards ML as implementation language in order to achieve a better integration with the AT&T software development environment in which Yeast is largely used.

References

[1] Balachander Krishnamurthy and David Rosenblum. An Event-Action Model of Computer-Supported Cooperative Work: Design and Implementation In International Workshop on Computer Supported Cooperative Work, Berlin, April 1991

[2] C. Ghezzi, D. Mandrioli, S. Morasca, M.Pezze. A General Way To Put Time In Petri Nets In 5th International Workshop on Software Specification and Design, IEEE, May 1989.

[3] N.D. Jones, L.H. Landweber and Y.E. Lien. Complexity of some Problems in Petri Nets Theoret. Comp. Sci.,4,(1977), 277-299

[4] J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.

[5] W. Reisig. *Petri nets – an introduction.* EATCS Monographs on Theoretical Computer Science, Volume 4. Springer-Verlag, 1985.

[6] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[7] J. W. de Bakker, W.P. de Roever and G. Rozenmberg (Eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency Lecture Notes in Computer Science 354, Springer -Verlag, June 1988

[8] V. S. Alagar and G. Ramanathan. Functional Specifications and Proof of Correctness for Time Dependent Behaviour of Reactive Systems. *Formal Aspects of Computing*, 3:253–283, 1991.

[9] F. Jahanian and A. K. Mok. Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, 1986.

[10] N. De Francesco and P. Inverardi. Proving Fineteness of CCS Processes by Non-Standard Semantics In Proc. CAV 91 Workshop Lecture Notes in Computer Science 575, Springer -Verlag, pp. 266–276, 1992.

Appendix A: Yeast Grammar

Following is the grammar of the subset of the Yeast language we consider.

```
yeast_prog ::= yeast_prog  yeast_cmd
yeast_cmd ::= spec
| DEFOBJ object_class
| DEFATTR object_class obj_attribute type
| ANNOUNCE object_class obj_attribute ASSIGN value
type ::= CHARSTRING | INTEGER | BOOLEAN
spec   ::= ADDSPEC event_pat DO action
action ::= shell_cmd | yeast_cmd
```

```
event_pat ::= event_pat AND event_pat
| event_pat OR event_pat | simple_event
simple_event ::= time_event | object_event
time_event ::= AT absolute_time
absolute_time ::= time_of_day
| time_of_day day_of_week /* 3p mon */
| time_of_day day_number /* 3p 3 */
| time_of_day month day_number  /* 3p jan 2 */
time_of_day ::= hour AM | hour minute AM | hour PM | hour minute PM
hour ::= 1 .. 12
minute ::= 0 .. 59
day_of_week ::= sun .. sat
day_number ::= 1 .. 31
month ::= 1 .. 12
object_event ::= object_class object_name obj_attribute rel_test
object_class ::= string
object_name ::= string
obj_attribute ::= string
rel_test ::= rel_op attr_value
rel_op ::= EQUAL | NE | GT | GE | LT | LE
attr_value ::= string | integer | boolean
```