Pioneering Technologies
for a Better Internet

Cs3, Inc.

5777 W. Century Blvd.
Suite 1185
Los Angeles, CA 90045-5600

Phone: 310-337-3013
Fax: 310-337-3012
Email: info@cs3-inc.com

License Manager Example of Monitoring using the FLEA language

Oct 11th, 1998

This document provides an example of using FLEA to define event monitors. For general information on Cs3's monitoring efforts, see http://www.cs3-inc.com/somos.html.

I. INTRODUCTION

Monitoring refers to the task of watching some system's activities so as to detect when certain combinations of events have happened. In principle, the user (who might be a person or a computer system) could observe the monitored system's events directly, and perform all the reasoning to deduce occurrencies of the combinations of events in which he/she/it is interested. The whole purpose of the Cs3's monitoring technology is to take over the burden of this reasoning. This document focuses on FLEA, Cs3's small language for expressing event combinations, and illustrates its use in monitoring a small but intuitive example. Further links point to screen snapshots illustrating how the experimental user-interface for FLEA facilitates the task of defining events, and also illustrates how event notifications are reported to the user.

Contents of this page:

LICENSE MANAGER EXAMPLE

A license manager is a system that handles requests to use a piece of licensed software so as to permit multiple, simultaneous invocations of that software provided that there are never more simultaneous invocations than licenses purchased. The license manager itself is built to ensure the requirement on limiting the number of invocations, so there is little point our monitoring this. Instead, we might wish to use monitoring to help us determine how many licenses to purchase, what limit to place on our user population, which machine to host the license manager itself on, whether to configure licenses to have some pre-set expiry time after which they must be renewed, etc. For a general discussion of monitoring and license manager issues, the interested reader is referred to [Fickas&Feather1995]. For simplicity, this document will focus on monitoring of just licenses, users, and their acquisition of licenses.

EXTERNAL EVENTS

Users may request licenses, return licenses that they have been granted, and cancel pending requests for licenses. The license manager may grant the request for a license. The systems administrator may increase or decrease the number of licenses, and add and remove registered users (people who will be allowed to request licenses). These interactions, between license manager and its environment, are what we monitor.

Each such interation is something we call an "external event" – an "event" because it is takes place at some instant in time (e.g., the time at which a user requests a license), and "external" because it is a product of the system that we are monitoring (namely the license manager and its environment of users and systems administrator). Suppose that we start by deciding to monitor just the addition and removal of users, and the addition and removal of licenses. We would declare this interest to the monitor as follows:

```
(defevent AddLicense :external ( timestamp ) )
(defevent RemoveLicense :external ( timestamp ) )
(defevent AddUser :external ( timestamp string ) )
(defevent RemoveUser :external ( timestamp string ) )
```

The first line declares an external event named AddLicense, with only a timestamp parameter, indicated by the singleton parameter list (timestamp). Similarly for RemoveLicense. The third line declares an external event named AddUser, with a two parameters, of type timestamp and String. Each occurrence of this event will have a timestamp and a string-valued parameter corresponding to the name of the user being added. Similarly for RemoveUser.

Having issued these declarations, the monitor will thereafter take notice of these events, remembering them in its database of events, and allowing the definition of further events in terms of these external events.

LOGICAL COMBINATIONS OF EVENTS

Logical combinations of events can be defined. For example,

```
(defevent AddOrRemoveLicense :definition ( ( tX ) s.t. ( or (AddLicense tX )
(RemoveLicense tX ) ) ) )
```

defines an event named AddOrRemoveLicense to be an AddLicense event **or** a RemoveLicense event, that is, whenever an AddLicense or RemoveLicense event occurs, then the monitor will record the fact that an AddOrRemoveLicense event occurred.

If an event is defined in terms of other events that themselves have parameters, then there will be a variety of ways in which those parameter values can be used. Examples:

```
( defevent AddOrRemoveMartin :definition ( ( tX ) s.t.
( or (AddUser tX "Martin") (RemoveUser tX "Martin") ) ) )
```

defines an event AddOrRemoveMartin that occurs whenever a user named "Martin" is added or removed.

```
( defevent AddOrRemoveUser :definition ( ( tX P1 ) s.t. ( or (AddUser tX P1)
(RemoveUser tX P1) ) ) )
```

defines an event AddOrRemoveUser that occurs whenever any user is added or removed, and gives that defined event the user's name as its parameter.

Our experimental user-interface for FLEA makes the task of defining events such as the above a simple matter. To see further details, click here.

SEQUENCES OF EVENTS

Sequences of events are often useful to define. For example:

```
( defevent AddThenRemoveLicense :definition ( ( tB ) s.t.
 ( E (tA ) ( then (AddLicense tA ) (RemoveLicense tB ) ) ) ) )
```

defines an event named AddThenRemoveLicense to be an AddLicense event followed later by a RemoveLicense event. Note that the RemoveLicense event need not be the very next event following AddLicense - there may be any number of other events intervening between an AddLicense event and a RemoveLicense event, and an AddThenRemoveLicense event will still be recognized as having occurred.

The time at which the AddThenRemoveLicense event occurs is the time at which the later of its constituent events (namely, RemoveLicense) occurs. Although it might seem nice to try to define the time of the AddThenRemoveLicense event to be the time of the earlier of its constituent events, it would be contrary to the spirit of FLEA to do so (i.e., we stronglyrecommend against it!). Generally, the time of a defined event should be the earliest time at which that event can be determined to have occurred. Thus at the time of an AddLicense event, we have no way of knowing whether or not a RemoveLicense event will follow; it is only if and when a RemoveLicense event does thereafter occur that we can determine that the AddThenRemoveLicense event has occurred.

PARAMETERIZED EVENTS

Just as was the case with logical events, if an event is defined in terms of a sequence of other events that themselves have parameters, then there will be a variety of ways in which those parameter values can be used. Examples:

```
( defevent AddThenRemoveMartin :definition ( ( tB ) s.t.
  ( E (tA ) ( then (AddUser tA "Martin" ) (RemoveUser tB "Martin" ) ) ) ) )
```

The specific user named "Martin" is added, then later removed.

```
( defevent AddThenRemoveSameUser :definition ( ( tB P1 ) s.t.
  ( E (tA ) ( then (AddUser tA P1 ) (RemoveUser tB P1 ) ) ) ) )
```

A user is added, then later that same-named user is removed. The AddThenRemoveSameUser event is defined to have the name of that user as its parameter.

```
( defevent AddThenRemoveSomeSameUser :definition ( ( tB ) s.t.
  ( E (tA P1 ) ( then (AddUser tA P1 ) (RemoveUser tB P1 ) ) ) ) )
```

A user is added, then later that same-named user is removed. The AddThenRemoveSameUser event is not defined to have the name of that user as its parameter. To achieve this effect,, the variable P1 that is used to match the user name in both AddUser and RemoveUser, is declared within the E (...) list of variables (akin to a local variable in conventional programming languages).

```
( defevent AddThenRemoveUsers :definition ( ( tB PA1 PB1 ) s.t.
  ( E (tA ) ( then (ADDUSER tA PA1 ) (REMOVEUSER tB PB1 ) ) ) ) )
```

A user is added, then later that some user - not necessarily of the same name as the added user - is removed. The AddThenRemoveSameUser event is defined to have the names of both users as its two parameters.

```
( defevent AddThenRemoveSomeUsers :definition ( ( tB ) s.t.
  ( E (tA P1 P2) ( then (AddUser tA P1) (RemoveUser tB P2) ) ) ) )
```

Similar to AddThenRemoveUsers, except does not have parameters corresponding to either user's name.

```
( defevent AddMartinThenRemoveSomeUser :definition ( ( tB P2 ) s.t.
  ( E (tA ) ( then (AddUser tA "Martin") (RemoveUser tB P2) ) ) ) )
```

A user named "Martin" is added, then later some user (regardless of name) is removed. The event being defined is parameterized by that removed user's name.

SEQUENCES EXCLUDING EVENTS

Sequences of events that exclude certain event during those sequences are definable. For example:

```
( defevent LicenseChangeWithoutUserChange :definition ( ( tB ) s.t.
  ( E (tA ) ( then-excluding (AddOrRemoveLicense tA )
                             (AddOrRemoveLicense tB )
                             (AddOrRemoveUser $ $ ) ) ) ) ) )
```

defines an event named LicenseChangeWithoutUserChange to be an AddOrRemoveLicense event followed later by another AddOrRemoveLicense event without any intervening AddOrRemoveUser event. Note that the then-excluding form takes three event descriptions, the first two (as was the case for the plain then form) defining the start and end events of the sequence respectively, and the third defining the events that are to be excluded, i.e., must not occur between the start event and the end event for the sequence to be recognized as an instance of the defined event.

COUNTING EVENTS

Events can be counted. For example:

```
( defrel CountAddLicenses :definition ( ( x ) s.t. ( countof0 (AddLicense * ) x ) ) )
```

defines a count of the number of AddLicense events that have occurred. Since events, once they have occurred, never "unoccur", this count may increase in value, but will never decrease in value. Notice that this count is a defrel not a defevent,that is, defines a relation (something to hold an arbitrary value), not an event. This is because a count is not an event - it is merely a value reflecting something about the current state of the system. As we will see shortly, changes to count values can be made into events, but the counts themselves are not events.

```
( defrel CountLicenses :definition ( ( x ) s.t.
  ( - ( countof0 (AddLicense * ) = ) ( countof0 (RemoveLicense * ) = ) x ) ) )
```

defines a count of the number of AddLicense events minus the number of RemoveLicense events, i.e., keeps count of how many licenses there are. Similarly:

```
( defrel CountUsers :definition ( ( x ) s.t.
  ( - ( countof0 (AddUser * * ) = ) ( countof0 (RemoveUser * * ) = ) x ) ) )
```

defines a count of the number of AddUser events minus the number of RemoveUser events, i.e., keeps count of how many users there are.

As with other definitions, the experimental user-interface for FLEA makes the task of defining counts such as the above a simple matter. To see further details, click here.

TRANSITION EVENTS

Events can be defined to occur whenever some particular transition occurs. Examples:

```
( defevent MoreThan5Users :transition ( ( tx ) s.t.
  ( start ( > ( CountUsers = ) 5 ) ) ) )
```

- the count of users starts to exceed 5.

```
( defevent MoreLicensesThanUsers :transition ( ( tx ) s.t.
  ( start ( > (CountLicenses = ) (CountUsers = ) ) ) ) )
```

- the count of licenses starts to exceed the number of users. This could occur either because the number of licenses increases, or the number of users decreases. Either way, it might be indicative of a surfeit of licenses, and hence something that the systems administrator might want to know of, so as to be able to return licenses for a refund (if that's possible), or at least not renew as many licenses when the time comes to do so.

```
( defevent UsersMoreThanTwiceLicenses :transition ( ( tx ) s.t.
  ( start ( > (CountUsers = ) (* 2 (CountLicenses= ) = ) ) ) ) )
```

- the count of users starts to exceed twice the number of licenses. This could occur either because the number of licenses decreases, or the number of users increases. Either way, it might be indicative of a shortage of licenses, and hence something that the systems administrator might want to know of, so as to be able to purchase extra licenses (or throw some users off the system!) so as to avoid too many occasions where too many users are competing for too few licenses.

The experimental user-interface for FLEA for defining transition events such as the above can be see by clicking here.

FURTHER DECLARATIONS OF EXTERNAL EVENTS

So far all our definitions have been in terms of the four external events AddUser, RemoveUser, AddLicense and RemoveLicense. Remember, however, that there are other interactions between license manager and its environment in which we might take an interest, namely users' requesting, returning and canceling requests for licenses, and the license manager granting license requests. It is possible to provide declarations of these external events as we go along (of course, until we issue such declarations, the monitor will have been ignoring occurrences of these events, and we cannot have defined any other events in terms of them). The declarations would be in the same style as the originally declared external events, namely:

```
(defevent RequestLicense :external ( timestamp String ) )
(defevent GrantLicense :external ( timestamp String ) )
(defevent ReturnLicense :external ( timestamp String ) )
(defevent CancelRequest :external ( timestamp String ) )
```

where the string valued parameter of each of these is the name of the user respectively requesting / being granted / returning / cancelling a request for the license.

TIME-SENSITIVE EVENTS

In monitoring the license manager, we might wish to know whether or not the monitor is fulfilling user's requests for licenses in a timely manner. Two related forms of declarations allow for the definition of such time-sensitive events:

```
( defevent Speedy :definition ( ( tB P1 ) s.t.
  ( E (tA ) ( in-time (RequestLicense tA P1 ) (GrantLicense tB P1 ) 10 ) ) ) )
```

defines an event called Speedy which will occur whenever a RequestLicense event is followed by a GrantLicense event (with the same parameter value) within the next 10 seconds. This is very much like the definition of event sequences, which used the then construct, except here there's also the need to also provide a time bound as part of the definition. The time at which such a Speedy event occurs is the time of its second event, the GrantLicense event. In a related manner:

```
( defevent tardy :definition ( ( lim P1 ) s.t.
  ( E (tA ) ( too-late (RequestLicense tA P1 ) (GrantLicense $ P1) 10 lim ) ) ) )
```

defines an event called Tardy which will occur whenever a RequestLicense event occurs and is not followed by a GrantLicense event (with the same parameter value) within the next 10 seconds. The time at which such a Tardy event occurs is the time limit, in this case, 10 seconds after the RequestLicense event.

The experimental user-interface for FLEA for defining time-sensitive events such as the above can be see by clicking here.

CONCLUDING EXAMPLE

We finish with an example that computes a measure of "user happiness", defined as the percentage of the number of Speedy events out of the total number of Speedy and Tardy events, provided the number of Speedy and Tardy events totals at least 20 (since we do not want to compute this percentage on too few statistics to be meaningful). The definition is:

```
( defrel UserHappiness :definition ( ( x ) s.t.
  ( and ( >= ( + ( countof0 (Speedy * * ) = )
               ( countof0 (Tardy * * ) = ) = ) 20 )
      ( / ( * 100.0 ( countof0 (Speedy * * ) = ) = )
         ( + ( countof0 (Speedy * * ) = )
            ( countof0 (Tardy * * ) = ) = ) x ) ) ) )
```

As with the counts we defined earlier, it is possible to define transition events in terms of this value, for example:

```
( defevent TooHappy :transition ( ( tx ) s.t. ( start ( > (UserHappiness = ) 90 ) ) ) )
( defevent TooUnHappy :transition ( ( tx ) s.t. ( start ( < (UserHappiness = ) 60 ) ) ) )
```

define two events, TooHappy - to watch for UserHappiness rising to above 90%, and TooUnHappy - to watch for UserHappiness dropping to below 60%. The former might be indicative of a surfeit of license relative to the user population's usage patterns, while the latter might be indicative of a shortage. The systems administrator will likely want to keep both users who request licenses happy, and the finance department that pays for licenses happy, which might necessitate buying enough, but not too many, licenses as indicated by these measures.

The experimental user-interface for FLEA for defining percentages such as the above can be see by clicking here.