

Complex Event Pattern Recognition for Long-Term System Monitoring

Will Fitzgerald* and R. James Firby

I/NET, Inc.

will.fitzgerald@nrc-cnrc.gc.ca, firby@inetmi.com

Aaron Phillips

Wheaton College

Aaron.B.Phillips@wheaton.edu

Jordan Kairys

Kalamazoo College

k00jk01@kzoo.edu

Abstract

One useful paradigm for monitoring autonomous systems over long periods of time is to be able to monitor and respond to complex patterns of events. The Complex Event Recognition Architecture, or CERA, is an architecture to do just this: it defines several classes of (hierarchical) event patterns and recognizers for these patterns. Software for CERA has been successfully used for a prototype monitoring system for a complex water recovery system at NASA's Johnson Space Center. We describe how CERA was used in this project, and announce the availability of open-source software versions of CERA for use in autonomous system monitoring.

Introduction

One of the difficulties of building human-machine interfaces (HMI) for autonomous systems that extend over long periods of time is the current standard pattern for HMI, the "model-view-controller" pattern (Krasner & Pope 1988) which biases HMIs that are very *thin*—the *view* is mediated by a *controller* that more or less directly reflects the underlying *model*—typically the current state of the system. The problem, of course, is that the "current state of the system," may not be all that is needed to monitor or control an autonomous system over long periods. We may also need to know the *history* of the system and predictions about the *future* of the system for understanding and control. The standard alternative—logging system events which can then be used for forensic or predictive analysis—has its own problems. The granularity of the logging data may be too fine, resulting in data streams overwhelmingly difficult to analyze. This is only exacerbated by logging data for extended periods of time.

In this paper, we describe a monitoring system called CERA that is designed to allow the implementor to create an easy to understand, hierarchical event detection and classification system. The CERA system combines a well-defined semantics for detecting extended event patterns with the ability to process the recognized patterns in arbitrary ways to generate additional synthetic events. This combination enables CERA to apply a uniform processing model to highly

idiosyncratic, numeric low-level events and highly abstract, symbolic high-level events.

In this paper, we will describe how CERA was used to monitor events that occurred over time for a complex water recovery system. We will go into some detail describing how we used CERA to define event pattern recognizers that turn raw telemetric data into increasingly useful, structured information whose semantic transparency aids long-term system monitoring.

The Water Recovery System Project

The Water Recovery System (WRS) is an advanced life support system project at NASA's Johnson Space Center (Schreckenghost *et al.* 1998b; Bonasso, Kortenkamp, & Thronesbery 2003). The WRS consists of four key subsystems: a biological water processor (BWP) for removing ammonia and organic compounds; a reverse osmosis system (RO) to remove inorganic compounds, an air evaporation system (AES) to recover additional water from the brine produced from the RO, and a final post-processing system (PPO). The BWP was first tested during a 450 day trial, and the WRS system as a whole was tested during a year long project.

The WRS's subsystems contain approximately 200 sensors and actuators. The WRS as a whole is controlled by a three-tiered architecture (Schreckenghost *et al.* 1998a): the top layer consisting of the an automated planner or commands from the human crew; the bottom layer consisting of the skill-level software for reading sensors and commanding actuators; and a middle layer for task sequencing. using the RAP System (Firby 1989).

There were a number of significant events that can be gleaned from the log data. For example, the communication channel between the WRS hardware and the RAP System would, from time to time, fail. Because the WRS hardware was no longer receiving control information, the WRS needed to safely shut down; this, of course, meant that each of the subsystems need to safely shut down. Eventually, communications would recover, and the WRS would restart; again, this meant that the subsystems would restart.

During operation of the WRS, each subsystem asynchronously logged measurements from its own sensors and actuators. These event logs were provided for a WRS monitoring demonstration. In this demonstration, the event logs

*Now at the National Research Council Canada.

05/23/01	00:03:28	15121	12470	247	0	...
05/23/01	00:08:36	15121	12548	246	0	...
05/23/01	00:13:38	15121	12470	247	0	...
05/23/01	00:18:33	15040	12470	246	0	...
05/23/01	00:23:35	15040	12548	246	0	...
05/23/01	00:28:30	15121	12548	246	0	...
05/23/01	00:33:32	15040	12548	246	0	...

Figure 1: Example data from the Reverse Osmosis log

were replayed and a real-time event monitor, CERA, was to watch for combinations of measurements that represented significant events, such as a loss of RAP System communication (LORC), recovery of RAP System communication (RORC), and the intermediate system and subsystem “safing” and restarting—as well as that entire event sequence.

Using CERA on the WRS Data

In this section, we provide specific details on how CERA was used in a WRS project. For more information of how CERA was used as a subsystem for human/machine collaboration, see (Martin *et al.* 2004) in this workshop. We provide these details to give an in-depth, real-life use case of complex event pattern detection.

What system developers typically *have* are telemetry data like that in Figure 1. What system developers would *like* is a way to abstract away the many details of this data, and recognize patterns of events and sub-events which comprise higher-level, more complex event patterns. For example, for the WRS data, to recognize that the four subsystems have completed safely shut down, it is useful to write something like this¹:

```
(define-recognizer (safing-complete)
  (pattern
    '(all
      (safing (system bwp) (status on))
      (safing (system ro) (status on))
      (safing (system aes) (status on))
      (safing (system pps) (status on)))
    (on-complete (st end)
      (signal-event '(all-safed) st end)))
```

That is, a recognizer named `safing-complete` that is watching for the pattern of four `safing` events occurring (one from each of the WRS subsystems; order not mattering) When all of these are seen, a synthesized `all-safed` event should be signaled.

What are the advantages of such a system? The first, and most obvious, advantage is that of *explanation*. It is much easier for a human to understand this recognition pattern than to parse megabytes of log data looking for the data relationships which indicate safing of the various sub-systems. Further, the pattern explicitly states what the relevant lower-level events are, as well as the pattern of their occurrence, to

¹The original version of CERA was written using Common Lisp, hence this Lisp-like syntax. Of course, the specific syntax chosen is not the crucial point.

the completion of this pattern. In this case, an `all-safed` event is signalled just in the case that the four `safing` events have occurred. Patterns also have explanatory power *in situ*, while the pattern has not yet completed. That is, they can indicate the *partial completion* of patterns. For example, if two `safing` events have occurred—say, the AES and RO subsystems have completed safing—then it’s reasonable to say we are in the middle of the completion of an `all-safed` event.

This leads to a second advantage, that of *prediction*. Because these patterns describe what has happened—in the previous example, two subsystems have completed safing—we can also use them to predict what might (or, perhaps, should) happen—in the current example, that the BWP and PPS subsystems may will complete safing.

Third, we can build levels of abstraction of event patterns to make complex event pattern recognition possible and understandable. The `safing-complete` recognizer above is part of the overall pattern of RAP System communication loss, safing completion, RAP System communication reacquired, and restart complete that comprise the overarching event we described earlier. In order to get an overall view of the situation at hand, the raw data events must be packaged into larger, hierarchically structured events (Thronesbery & Schreckenghost 2003).

Within this project, CERA acted as a stand-alone system to read the measurement events from a file and recognize semantically meaningful state changes, such as LORC and RORC events. When such events were recognized, CERA packaged them up in XML format and sent them to other parts of the monitoring system via CORBA connections.

The Raw Subsystem Events

The first step in the CERA event processing for the WRS monitoring system was to transform the complex, multi-valued events from each subsystem into simpler, more abstract events. To do this, four recognizers were defined, one to match the measurement events from each of the four subsystems. Each time that a measurement event was matched, three things happened: first, measured telemetry values were saved away for possible later inclusion in an abstract event². Second, a function was called to see whether the subsystem had gone into safe mode. For example, the PPS system has gone into safe mode if its PT09 value is less than 1.4, its R040 value is 0 and its FM09 value is less than 200³. Third, another function checked in a similar way to see whether the subsystem was restarting. The specific data values and predicates for each subsystem are different, but the basic recognizer logic is the same.

These CERA raw data event recognizers illustrate the importance of combining pattern recognition with the ability to take arbitrary action. In this case, the patterns are relatively simple, consisting of a single event (although the event holds

²This is to allow the highest granularity of data for later inspection; of course, this needs to be limited as to the amount of data retained. This is essentially an internal logging function.

³These details and rules were, of course, supplied by our NASA collaborators (Martin *et al.* 2004).

a lot of data). However, in response to recognizing the pattern a number of things must be done:

- the data values must be saved for future use;
- if the data values indicate a transition into or out of safe mode, a signal must be generated;
- if the data values indicate the system starting or stopping, a signal must be generated.

In our experience, this sort of idiosyncratic behavior in response to *changes* in specific data values occurs frequently when processing low-level telemetry. Thus, it must be simple to include arbitrary code for storing, comparing, and combining data values as appropriate.

The LORC Event

In addition to the raw measurement data from the various subsystems, CERA was also monitoring events detailing communications problems. In the WRS demonstration, external events were played back from the log that specifically mention when the RAP system loses contact with the various subsystems. A sketch of the recognizer that was used to process these events is shown in Figure 2.

Recognizer: RAPS Communication Loss
Pattern:
repeat-match:
RAPS Skills Comms Out *seconds* ← *seconds*
If RAPS Communication \neq *off-nominal*:
Record RAPS Communication state ← *off-nominal*;
Begin saving telemetry values;
Calculate start of loss from *seconds*;
Signal RAPS Communication status: *off nominal*.

Recognizer: LORC
Pattern:
one:
RAPS Communication status: *off nominal*
On complete:
Signal a LORC event.

Figure 2: Recognizer for the LORC Event

A `repeat-match` pattern is used here to watch for RAPS-SKILLS-COMMS-OUT events. When each occurs, the attached code checks to see whether the event means that the communications status has changed. When a change to `:off-nominal` occurs, it is recorded for future use, and a synthetic STATUS-CHANGE event is signalled to CERA. In addition, the recording of telemetry events from all measurement recognizers is begun and the real start time of the communications loss is computed (actual loss began the specified number of seconds before the communication loss event is generated).

Again, a number of very idiosyncratic things need to be done in response to the raw RAPS-SKILLS-COMMS-OUT event. However, once that processing is done, the recognizer for the LORC event, a sketch for which is also shown in Figure 2, can be much simpler.

The LORC event recognizer uses a simple one pattern looking for a single instance of the STATUS-CHANGE event generated by the communications loss recognizer. Then is pattern is recognized, the LORC event is saved away for future processing and also signalled to the CERA system.

The processing steps in the LORC recognizer could easily have been put into the communications loss recognizer. However, we chose to separate them to keep the low-level event processing separate from the more abstract events needed by the WRS monitor. Saving the LORC event away is the first step in building up a more complex event description to send to other parts of the monitoring system.

The LORC Situation Event

The first situation of interest to the larger WRS monitoring system is when a LORC occurs followed by all four subsystems moving into safe mode. A sketch for the recognizer for this situation is shown in Figure 3. The pattern for this recognizer is a LORC event followed by the safing status of all subsystems changing to `safe` in any order.

Recognizer: LORC Situation
Pattern:
in-order:
LORC;
all:
PBBWP status: *safe*;
RO status: *safe*;
AES status: *safe*;
PPS status: *safe*
On completion:
Get saved LORC data;
Publish it to CORBA;
Signal a LORC Situation event.

Figure 3: Recognizer for the LORC Situation Event

When the pattern is recognized, the recognizer finds the triggering LORC event saved away previously, and then publishes an XML version of the LORC situation event to the CORBA interface, and hence to the rest of the WRS monitoring system. In addition, a `lorc-situation` event is signalled to CERA. The creation of the appropriate XML for the LORC situation depends on the triggering event and the telemetry data values saved away by the recognition of previous events.

The LORC-RORC Situation

Finally, the WRS is interested in knowing when RAPS communication is restored and all of the subsystems have restarted. The recognizer of Figure 4 watches for that situation. The pattern looks for a LORC event followed by a LORC situation followed by all subsystem restarting. The LORC is not really necessary as it will be included within the LORC situation. However, it simplifies extracting the trigger event for inclusion in the published XML description.

Recognizer: **LORC-RORC Situation**

Pattern:

in-order:

LORC;

LORC Situation;

all:

PBBWP status: start;

RO status: start;

AES status: start;

PPS status: start

On completion:

Get related events;

Get saved LORC data;

Publish it to CORBA;

Signal a *LORC-RORC Situation* event.

Stop saving telemetry values.

Figure 4: Recognizer for the LORC-RORC Situation

When this recognizer completes, it again publishes a composite XML LORC-RORC situation event to the WRS monitor via CORBA. It then signals a similar event to CERA and stops the recording of telemetry values from the subsystems.

Meanwhile, all of the recognizers continue to process new events so that the next LORC and RORC events will be detected.

These patterns (with minor editing due to space considerations) are part of a set of patterns used to recognize significant events occurring over time in the WRS. On a 700 MHz. Windows 2000 machine with 256 Mb of memory running Allegro Common Lisp 6.1, The CERA software processes an event in approximately 1.5 ms. (2346 base events, representing two days of data, in 3.55 seconds), and thus, we believe, provides a useful system that can be used for complex event detection in real time.

An Example Transcript

The sample transcript in Figure 5 is taken from one run of the WRS monitor demonstration data. It shows the events signalled by the CERA recognizers described above. Each line in the transcript begins with the event signalled and is followed by the time covered by the event. Point events show only a single time. The LORC, LORC Situation, and LORC-RORC Situation events are highlighted. In particular, note that, in contrast to the raw data transcript of Figure 1, these events can occur over time; they have internal structure, and they have semantically meaningful names.

The transcript begins with the RO subsystem stopping at 00:49 and then restarting at 01:14. The PPS subsystem also goes down from 00:55 to 01:14. Then the AES subsystem can be seen to go down at 02:10 and restart at 02:14.

Next, at 04:02, a loss of communications event is recognized that began at 03:56. This completes the pattern for the LORC recognizer and a LORC event is signalled immediately as 04:02:01.

(safing-status (system ro) (status safe)) at 00:49
(startup-status (system ro) (status stop)) at 00:49
(startup-status (system pps) (status stop)) at 00:55
(safing-status (system pps) (status safe)) at 00:59
(safing-status (system ro) (status nominal)) at 01:14
(startup-status (system ro) (status start)) at 01:14
(safing-status (system pps) (status nominal)) at 01:14
(startup-status (system pps) (status start)) at 01:14
(safing-status (system aes) (status safe)) at 02:10
(startup-status (system aes) (status stop)) at 02:10
(safing-status (system aes) (status nominal)) at 02:14
(startup-status (system aes) (status start)) at 02:14
(status-change (measure raps-comm)
(status off-nominal) (reason lost-contact))
from 03:56 to 04:02
(lorc) from 03:56 to 04:02
(safing-status (system ro) (status safe)) at 04:04
(startup-status (system ro) (status stop)) at 04:04
(safing-status (system aes) (status safe)) at 04:04
(startup-status (system aes) (status stop)) at 04:04
(startup-status (system pps) (status stop)) at 04:04
(safing-status (system pbbwp) (status safe)) at 04:05
(startup-status (system pbbwp) (status stop)) at 04:05
(safing-status (system pps) (status safe)) at 04:09
(lorc-situation) from 03:56 to 04:09
(safing-status (system aes) (status nominal)) at 14:32
(startup-status (system aes) (status start)) at 14:32
(status-change (measure raps-comm)
(status nominal) (reason aes)) at 14:32
(safing-status (system pbbwp) (status nominal)) at 14:32
(startup-status (system pbbwp) (status start)) at 14:32
(safing-status (system ro) (status nominal)) at 14:48
(startup-status (system ro) (status start)) at 14:48
(safing-status (system aes) (status safe)) at 14:48
(startup-status (system aes) (status stop)) at 14:48
(safing-status (system pps) (status nominal)) at 14:49
(startup-status (system pps) (status start)) at 14:50
(safing-status (system aes) (status nominal)) at 14:52
(startup-status (system aes) (status start)) at 14:52
(lorc-rorc-situation) from 03:56 to 14:52
(safing-status (system aes) (status safe)) at 15:44
(startup-status (system aes) (status stop)) at 15:44
(safing-status (system aes) (status nominal)) at 15:49
(startup-status (system aes) (status start)) at 15:49

Figure 5: Transcript of CERA Events Showing a LORC-RORC Situation (Transcript edited for space) Event

After the LORC event, the RO and AES subsystems both enter safe mode at 04:04 and the PBBWP system follows at 04:05 with the PPS subsystem not far behind at 04:09. With all four subsystems entering safe mode after the LORC event, a LORC situation is recognized and signalled at 04:09. Notice that this event begins at 03:56 when the loss of communications actually began.

Finally, the subsystems begin coming back on line and at 14:52 the AES restarts completing the LORC-RORC situation pattern and a LORC-RORC situation event is signalled, running from 03:56 to 14:52. Notice that the AES subsystem does restart earlier, at 14:32 but then it stops again at 14:48, before the PPS subsystem restarts. The semantics of the `all` pattern combined with the fact that AES stopping contravenes the AES starting given our representation, means that CERA will wait for all four subsystems to be started simultaneously before signalling the LORC-RORC event. Alternative behaviors can be obtained by using different patterns in the recognizer.

CERA Pattern and Recognizer Types

CERA supports a variety of pattern and pattern recognizer types. What follows are brief descriptions of these patterns and pattern recognizers.

One patterns are the simplest of the pattern types. As their name implies, they consist of exactly one element. The recognizer that a **one** pattern creates is complete as soon as it recognize that element. **Repeat-match** bind with signals (such as the telemetry events seen above) and provide one way to allow maintenance of state while monitoring.

A **one-of** pattern is an unordered collection of elements. A **one-of** recognizer is complete when any one of the elements specified by the pattern that created it occurs.

An **in-order** pattern is a collection of elements in a specified order. It follows then that an **in-order** recognizer will be complete if each of the elements of the **in-order** pattern occurs in the correct order. This definition is somewhat lenient in that it is acceptable for other elements to occur in between two of the elements in the pattern, and even for the elements of the pattern to occur out of order, since events that do not directly affect the next element in the list are ignored by the recognizer.

An **all** pattern is a collection of elements with no regard given to the order in which they occur. Once all of the elements of a given **all** pattern have occurred, an **all** recognizer created by that pattern would be complete.

Recognizers for **in-order** and **all** patterns can also make use of the the idea of *event contravention*⁴. The basic idea is this: in a complex multichannel environment, many events will occur that are not relevant to the completion or futility of a recognizer. For example, while watching for all subsystems to go into safe mode, many events are irrelevant, but some events *contravene* the recognition of this pattern—for example, having once noticed that a subsystem has gone into safe mode, that subsystem's restarting (that is to say, going into “non-safe” mode) contravenes the overall pattern

⁴Previously we called this *event contradiction* (Fitzgerald, Firby, & Hannemann 2003).

recognition. To support this idea, software classes that represent base signals can be assigned a method that returns true or false when compared to other base signals. This predicate can be as simple or as complex as necessary for a given application.

A **within** pattern consists of a single element and a time duration. A **within** recognizer can only be complete if the element that it is looking for occurs with start and finish times such that the difference between the start and finish times is less than the duration specified by the pattern. A **without** pattern consists of a single item and a time interval. The recognizer created by a **without** pattern can only be completed if its subrecognizer returns complete with a start time **after** the finish time of the specified interval or a finish time **before** start time of the specified interval. Otherwise, the recognizer is said to be FUTILE. CERA also provides patterns based on the possible relationships between the intervals over which event occur (Allen 1983).

Software Availability

The CERA system was originally developed in Common Lisp, and we took advantage of the ease with which Common Lisp allowed us to write patterns declaratively with procedural attachment (as in the examples given). We have also written procedural versions of the CERA software in both Python and C++. The Python and C++ versions of CERA are available on the CERA software project site at <http://sourceforge.net/projects/cera> and are released under an open-source license which allows them to be easily incorporated into new projects. The proprietary Lisp version of CERA (Fitzgerald, Firby, & Hannemann 2003) is available from I/NET Inc.

Conclusions

The immediate current state of autonomous systems is often reflected in the telemetric stream of sensor and actuator signals. To monitor the state of a system in semantically meaningful terms often requires watching for complex patterns of signals and events. As we have demonstrated in the WRS prototype, CERA is a useful architecture for this kind of complex system monitoring. Furthermore, monitoring the completion or non-completion of CERA event pattern recognizers provides further insight to system history and possible future states. Using a system such as CERA allows system developers to recognize hierarchical patterns of events which leads to deeper insight into the system state. This is especially useful with systems which operate for long periods and do so autonomously most of the time.

CERA has also been used to provide the direct human-machine interface for a mobile robot that acts as a judge in a multi-modal memory game (Fitzgerald, Kairys, & Phillips 2004), and can be useful for building the immediate multi-modal interfaces to monitoring systems (as well as monitoring the systems themselves).

By providing open-source versions of the CERA software, we hope to develop a community of users that will lead to new event pattern descriptions as well as innovations in

multi-modal user interface development. Readers are invited to try the CERA software in their own projects.

Acknowledgements

CERA was originally developed under NASA SBIR Projects 99-1-08.04-1856 and 99-2-08.04-1856. The Python and C++ code was developed a 2003 Summer NSF REU Program at Hope College. We gratefully acknowledge useful comments by Michael Hannemann on an earlier version of this paper.

References

- Allen, J. 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM* 26:832–843.
- Bonasso, P.; Kortenkamp, D.; and Thronesbery, C. 2003. Intelligent control of a water-recovery system: Three years in the trenches. *AI Magazine* 24(1):19–44.
- Firby, R. J. 1989. Adaptive execution in complex dynamic worlds. Technical Report YALEU/CSD/RR #672, Computer Science Department, Yale University.
- Fitzgerald, W.; Firby, R. J.; and Hannemann, M. 2003. Multimodal event parsing for intelligent user interfaces. In *Proceedings of the 2004 International Conference on Intelligent User Interfaces*, 253–261. ACM Press.
- Fitzgerald, W.; Kairys, J.; and Phillips, A. 2004. Complex event pattern recognition software for multi-modal user interfaces. Available on-line at <http://cera.sourceforge.net>.
- Krasner, G. E., and Pope, S. T. 1988. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* 1(3):26–49.
- Martin, C.; Schreckenghost, D.; Bonasso, P.; Kortenkamp, D.; Milam, T.; and Thronesbery, C. 2004. Aiding collaboration among humans and complex software agents. In *Proceedings of AAAI 2004 Spring Symposium on Interaction Between Humans and Autonomous Systems over Extended Operation*.
- Schreckenghost, D.; Bonasso, R. P.; Kortenkamp, D.; and Ryan, D. 1998a. Three tier architecture for controlling space life support systems. In *IEEE Symposium on Intelligence in Automation and Robotics*.
- Schreckenghost, D.; Ryan, D.; Thronesbery, C.; Bonasso, R. P.; and Poirot, D. 1998b. Intelligent control of life support systems for space habitats. *AAAI Innovative Applications of AI*.
- Thronesbery, C., and Schreckenghost, D. 2003. Situation views: Getting started handling anomalies. In *Proceedings of the Systems, Man, and Cybernetics Conference. Washington, D.C.: IEEE*.