FP6-004381-MACS

MACS

Multi-sensory Autonomous Cognitive Systems Interacting with Dynamic Environments for Perceiving and Using Affordances

Instrument: Specifically Targeted Research Project (STReP)

Thematic Priority: 2.3.2.4 Cognitive Systems

D4.4.1 A software prototype for an affordance monitoring module with empirical testing using various MACS robotics platforms – The Event and Execution Monitor Module (EEM)

Due date of deliverable: November 30, 2006
Actual submission date: October 19, 2006

Start date of project: September 1, 2004 Duration: 36 months

Linköpings Universitet (LiU-IDA)

Revision: Version 2

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

EU Project MACS

Deliverable D4.4.1

A software prototype for an affordance monitoring module with empirical testing using various MACS robotics platforms – The Event and Execution Monitor Module (EEM)

Fredrik Heintz, Patrick Doherty, Björn Wingman, Piotr Rudol, Mariusz Wzorek

Number: **MACS/4/4.1**
WP: *4.4*
Status: *draft, version 2*
Created at: *October 1, 2006*
Revised at:
Internal rev: *v2 – October 19, 2006*

FhG/AIS Fraunhofer Institut für Intelligente Analyse- und Informationssysteme, Sankt Augustin, D
JR_DIB Joanneum Research Graz, A
LiU-IDA Linköpings Universitet, Linköping, S
METU-KOVAN Middle East Technical University, Ankara, T
OFAI Österreichische Studiengesellschaft für Kybernetik, Vienna, A

This research was partly funded by the European Commission's 6th Framework Programme IST Project MACS under contract/grant number FP6-004381. The Commission's support is gratefully acknowledged.

© **LiU-IDA 2006**

Corresponding author's address:

Prof. Dr. Patrick Doherty
Linköpings Universitet
Dept. of Computer and Info. Science
Linköping 581 83, Sweden

Fraunhofer Institut für Intelligente
Analyse- und Informationssysteme
Schloss Birlinghoven
D-53754 Sankt Augustin
Germany

Tel.: +49 (0) 2241 14-2683
(Co-ordinator)

Contact:
Dr.-Ing. Erich Rome

Joanneum Research
Institute of Digital Image Processing
Computational Perception (CAPE)
Steyrergasse 9
A-8010 Graz
Austria

Tel.: +43 (0) 316 876-1769

Contact:
Dr. Lucas Paletta

Linköpings Universitet
Dept. of Computer and Info. Science
Linköping 581 83
Sweden

Tel.: +46 13 24 26 28

Contact:
Prof. Dr. Patrick Doherty

Middle East Technical University
Dept. of Computer Engineering
Inonu Bulvari
TR-06531 Ankara
Turkey

Tel.: +90 312 210 5539

Contact:
Asst. Prof. Dr. Erol Şahin

Österreichische Studiengesellschaft
für Kybernetik (ÖSGK)
Freyung 6
A-1010 Vienna
Austria

Tel.: +43 1 5336112 0

Contact:
Prof. Dr. Georg Dorffner

Contents

Part I

Background

Chapter 1

Introduction

This document describes the event and execution monitor (EEM) of the affordance-based robot control architecture developed in the MACS project as described in the "Development of an Affordance-based Control Architecture" [9]. The EEM is implemented using a knowledge processing middleware called DyKnow which has been developed in order to facilitate the creation and management of knowledge in a distributed real-time system.

The document is structured as follows. This first part describes the EEM, formula progression and chronicle recognition using DyKnow, and how the EEM is implemented by DyKnow. The second part is a step-by-step tutorial on how to use the EEM to implement a small application for finding and examining objects.

Chapter 2

The Event and Execution Monitor

2.1 Overview

The purpose of the event and execution monitor (EEM) module is to recognize the occurrence of events and to monitor the execution of actions and behaviors. Conceptually the EEM is part of the execution component in the architecture, but it may be used by any of the components to recognize event occurrences as required by other modules.

For example, for the constraints associated with a cue or outcome descriptor to be satisfied, a cue event or an outcome event must be recognized. This is the type of functionality provided by the EEM. To do this the EEM should receive an event occurrence request. Based on the request it should call the entity structure generation module (ESGM) with a policy to extract a state sequence of the appropriate set of attributes associated with the the cue or outcome to be used to verify the occurrence or non-occurrence of the event within a specified temporal duration. An event type is represented as a simple temporal network [8] where the nodes represent time-points of changes in attribute values. To recognize an event instance is to find a set of time-points where such value changes occur and where the set of time-points satisfy the temporal constraints of the network. Each such event occurrence recognized should be reported back to the original caller. According to the specification [9] it should also be possible to detect non-occurrence of events, but this is currently not supported by the EEM. The reason is that it is unclear what it would mean to recognize a non-occurrence of an event type over a possibly infinite state sequence. A possible interpretation that could be implemented is that no occurrences started in a particular interval and/or ended in a particular interval could be found.

The other task of the EEM is to monitor the execution of actions and behavior instances. This is done in order to determine whether a behavior or action instance is successful or not. To start this monitoring the EEM should receive an execution monitoring request. Based on the request it should call the ESGM with a policy to extract a state sequence of the appropriate set of attributes associated with the request, just like in the case with the event occurrence request. An execution monitoring request is expressed as one or more metric interval temporal logic (MITL) formulas [2]. When a formula is evaluated to `true` or `false` this should be reported back to the original caller.

Chapter 3

DyKnow

3.1 Introduction

The main purpose of DyKnow is to provide generic and well-structured software support for the processes involved in generating state, object, and event abstractions about the environments of complex systems. Generation of state, object, and event representations is done at many levels of abstraction beginning with low level quantitative sensor data and often resulting in qualitative data structures which are grounded in the world and can be interpreted as knowledge by the system. To produce these structures the system has to support operations on data and event streams at many different levels of abstraction. For the result to be useful, the processing must be done in a timely manner so that the robotic agent can react in time to changes in the environment. The resulting structures are used by various functionalities in a deliberative/reactive architecture for control, situation awareness and assessment, monitoring, and planning to achieve mission goals. DyKnow provides a declarative language for specifying these structures needed by the different components of the agent. Based on this specification it creates representations of the external world and the internal state of an agent based on observations and a priori knowledge, such as facts stored in databases, with the desired properties and quality of service guarantees. DyKnow also allows easy integration of existing sensors, databases, reasoning engines and other knowledge producing services.

The motivation for the work is to provide an agent with the necessary knowledge to make decisions and act. This is why we call it knowledge processing middleware instead of e.g. information processing middleware. Since the purpose is acting the processing time is an important part of the knowledge process. It is not sufficient to produce the best possible estimate of the position of the agent if it takes an hour to do it. Neither is it sufficient to produce a fast and dirty estimation of the position if it is not accurate enough to be useful for the agent. The two conditions are necessary, but not sufficient on their own. An important problem is to provide tools to help the agent designer describe and make the trade-offs between efficiency and accuracy.

3.2 Execution Monitoring

An agent acting in a dynamic and uncertain environment will sooner or later experience unanticipated or exceptional events leading to unintended failures. Therefore the agent needs to detect and handle these exceptional situations in order to perform robustly in a noisy environment. One approach to the problem is to make each action handle all possible circumstances, which leads to

very complex actions and duplication of functionalities. Therefore, execution monitoring has been suggested in order to separate the acting from the monitoring of exceptions and the recovery from these unintentional situations [5; 7; 10; 11; 13; 18].

One motivation for separating the monitoring from the execution is to increase the separation of concerns and the modularity. For example, assume we have a UAV which should fly a box to a destination using a trajectory following behavior. To make sure that the box is not dropped the UAV will monitor that it has not dropped the box. This means the action being executed is flying along a trajectory, but the condition that is being monitored is that a box attached to the UAV is not dropped. We could add checks in the trajectory following action to handle the situation where the box is dropped, but this has nothing to do with flying along a trajectory. Instead we separate the concerns and monitor the behavior of the UAV with another component, the execution monitor. This separation makes it much easier to monitor conditions that are not part of the action itself but should be invariant during the action, such as not dropping a box while flying. By making the monitor external to the action we can also monitor scenarios where the UAV is not only flying along a trajectory but is also doing other actions along the path to the destination of the box, such as taking photographs of buildings, without changing the monitoring condition. The separation is even necessary if we want to monitor the execution of the implementation of the action in the software system, for example to check that the action does not get stuck in an infinite loop or crashes.

One approach to execution monitoring is to use a model to derive the expected behavior of an agent and then compare this expected behavior with the output from the sensors [5; 13]. For example, Gat et al [13] takes the output from their path planner and simulates the expected sensor outputs using a simulator. From these expectations an interval within which the sensor readings are expected to be is derived for each time-point. While following the generated path the robot will check that the sensor readings are within the expected interval. If a discrepancy is detected the robot switches to a recovery mode which tries to handle the unintended situation.

Another approach is to focus on the unexpected situations instead of the expected behavior. The idea is to monitor as many exceptional cases as possible. This can for example be done using hierarchical monitors [10], where the top monitors are very general and the leaf monitors are very specific. The top monitors cover many cases, but usually detect the exception with a large delay and do not give very much information about the cause of the exception to aid the recovery. For example, a monitor to check that the action has taken longer than the expected maximum execution time is a top level monitor in Fernandez et al [10]. It will cover very many cases, but it will not trigger until the maximum execution time has passed, which can be a long time. The leaf monitors on the other hand are usually very specific, covering few cases but will detect these exceptions fast and provide detailed information about the possible causes to the recovery mechanism. An example of a leaf monitor, also from Fernandez et al [10], is the spinning in place monitor that checks that the robot is not spinning around indefinitely. If this monitor is triggered then the robot knows exactly what exception has occurred and what recovery action to perform.

A third approach is to use a logic of action and change to model the effects of actions and represent potential failures by abnormality predicates in the effects [11]. These abnormality predicates are false by default, but if the execution of an action fails then the controller can hypothesize that one or more of the abnormality predicates are true. Taking this new information into account the controller can derive a new plan to handle the exceptional situation.

The approach we have based our work on is by Lamine and Kabanza [18]. They use ideas from model checking where desired properties of the system are expressed in a logical formalism and

checked against models of the system [6]. But, instead of checking the properties against a model of the system they check the properties against the execution trace of the robot. Two categories of properties are usually discerned, those properties that are never allowed to be violated, called *safety properties*, and those properties that are required to be true indefinitely, called *liveness properties*. Since only the finite execution trace up to the current time point is checked it is not possible to check liveness properties, which are only defined on infinite traces, using this approach.

Like Lamine and Kabanza [18] we use a declarative formalism, a linear temporal logic, to express the properties to be monitored. Since the monitors are expressed in a declarative language we can reason about the monitors, as well as allow the agent itself to reason about its needs and derive the appropriate formulas. The development of new monitors is also simplified since no programming is needed, instead the user express the desired condition in the declarative language. Since there is no need to recompile anything the formulas can be added dynamically at run-time.

Compared to the other approaches we use a combination of ideas from the first two approaches. We do not do any explicit prediction, but at the same time the formulas express expected future behavior. Instead of making one prediction, we express a set of legal future states which are those states we predict are not going to make the execution fail. Therefore the negation of these formulas could be seen as expressing unexpected situations which we would like to detect in order to manage them when they occur.

To summarize, execution monitoring is about detecting and recovering from exceptional situations while performing actions in a dynamic and uncertain environment. By separating the actions, the monitoring conditions and the recovery actions and expressing the conditions using a declarative formalism the agent can reason about the conditions and possibly decide at run-time what monitors are needed for a specific situation as well as what recovery action is most appropriate. Another benefit is that the same action and the same sensors can be used for many different monitoring conditions, each specified by a declarative formula. A good monitoring condition is one that detects a class of exceptions as soon as possible after it has occurred and provides enough information about the exception in order for the agent to recover from the it.

3.2.1 Execution Monitoring With MITL

To describe the expected behavior we will use a linear temporal logical formalism called metric interval temporal logic (MITL) [2]. The choice suits the application very well since we only check the past, which is always linear. The fact that there exist efficient algorithms for evaluating MITL formulas online, as explained in Section 3.2.2, is another reason. A MITL formula can express that a condition F must hold either at all times, $\mathrm{always}\, F$, or at some time, $\mathrm{eventually}\, F$. It can also express that a condition F must hold at all times up to the time when a condition G holds, $F\,\mathrm{until}\, G$. The condition can either be an MITL formula or a first order predicate formula. It is also possible to define temporal operators over specific intervals, such as $\mathrm{always}_{[i,j]}\, F$ which says that F must hold over the interval $[i, j]$. Since a formula is evaluated in a state the interval is relative to the state, for example, if the timestamp of the current state is time-point 4 then the formula $\mathrm{always}_{[3,5]}\, f$ says that f should be true from time-point 7 to time-point 9. Similar logics are commonly used in model checking to verify safety and liveness properties of formal models of systems .

The safety constraint mentioned in the introduction, that if a box is attached and the UAV goes into the trajectory following mode then the box must stay attached until the UAV is no longer in the trajectory following mode, can be expressed in MITL as $\mathrm{always}(\mathsf{attached} \wedge \mathsf{traj\_following} \rightarrow \mathsf{attached}\,\mathrm{until}\,\neg\mathsf{traj\_following})$. The constraint will be violated if the UAV drops the box while

following a trajectory. Even though it is a quite simple condition to monitor, it is not something that we can do only with a sensor since the condition only applies in certain modes of operation. It is not enough that the UAV reacts every time the attachment sensor registers that a box has been detached, it is only those detachments that occur while flying a trajectory that are interesting. It is possible to achieve the same behavior by only using the sensor if it is only turned on when the UAV is in the trajectory following mode or if the control mode is checked each time the sensor goes from on to off. The drawback is that the UAV needs a controller for turning the sensor on and off which could otherwise be passively turned on all the time. It will also restrict the use of the sensor since it may only be used for this particular task. By using an execution monitor the sensor can be used for many different monitoring tasks, each specified by a declarative formula.

Given that we have a MITL formula containing a number of variables the question is how to create a state sequence representation of those variables using DyKnow so that the evaluation of the MITL formula on the representation gives the correct result with respect to the world. When we speak of "the world", we are in fact referring to the world as it is perceived by the agent through its sensors, i.e. we assume that the sensors produce a correct representation of the world. This means that DyKnow does not reason about the relation between the data produced by the sensors and the actual state of the world. However, DyKnow can support such reasoning carried out by other systems at the agent's disposal, by providing those systems with streams of data meeting specified requirements. Instead, we are interested in describing the properties of the output from the sensors, the desired properties of a representation and how to actually derive a representational structure with those desired properties. If it is not possible to do this then it should be detected and reported so the agent can handle the situation. The purpose of DyKnow is to create such representational structures from the output of the sensors. Before we describe how this is done we will describe the semantics of MITL and one method to evaluate MITL formulas.

The semantics of an MITL formula is defined on an infinite sequence of states where each state is associated with a time interval and assigns a value to every variable in the formula. A formula on the form $\mathrm{always}\, f$ is true over a state sequence $s_1, \ldots$ iff f is true in each state s_i in $s_1, \ldots$. Over the same state sequence a formula $\mathrm{eventually}\, f$ is true iff f is true in some state s_i. The formula $f\ \mathrm{until}\ g$ is true iff there exists a state s_i in $s_1, \ldots$ such that f is true in all states before, but not including, s_i and g is true in s_i. A first order formula f is true in a state s iff the interpretation of the variable in the state s entails f. If intervals are added to the temporal operators then the intervals of the states must be considered as well. For example, a formula $\mathrm{always}[s, e] f$ is true over a state sequence $s_1, \ldots$ iff f is true in each state s_i in $s_1, \ldots$ where the time interval associated with s_i overlaps the interval $[b, e]$. The formula $\mathrm{eventually}[s, e] f$ is true over the same state sequence it there exist at least one state s_i where f is true and where the interval associated with s_i overlaps $[s, e]$.

3.2.2 Evaluating MITL Formulas Using Progression

Since we want to detect violations of safety constraints as soon as possible we need a suitable method to evaluate the MITL formulas. One such method is a technique called progression, which evaluates a formula over a state sequence one state at the time. The technique is perfectly suited for an execution monitoring application since we can progress the formula each time a new state is available until the scenario is finished or the formula is progressed to either `true` or `false`.

Progression takes an MITL formula and the first state in a state sequence and returns a new MITL formula, such that the result of evaluating the new formula on the rest of the state sequence is the same as evaluating the original formula on the whole state sequence. A picture of the pro-

gression of an MITL formula F is shown in Figure 3.1. The progression can be applied recursively to evaluate a formula over a state sequence by evaluating a single state at a time. If the formula is progressed to either `true` or `false` no more progression is required since they are progressed to themselves for all states. The progression algorithm we have used is taken from [17], which is similar to the progression algorithm described in [3].

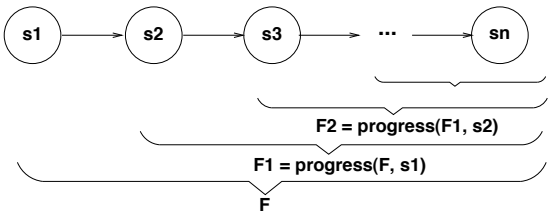

Figure 3.1: The evaluation of an MITL formula F over the state sequence $s_1, \ldots, s_n$ by progression. The result of progressing F over the first state s_1, F_1, is evaluated over the state sequence $s_2, \ldots, s_n$, by successive progression.

A drawback of using progression is that we only evaluate formulas on finite state sequences, while the semantics of MITL is defined on infinite sequences. This means that we can not infer that a system has typical liveness properties such as F holds infinitely many times during the execution, expressed by $\mathrm{always}\,\mathrm{eventually}\,F$. But, by introducing intervals on the temporal operators it is possible to verify limited liveness properties, such as at any time during the ten hour execution of the system the condition F holds within 10 minutes, expressed by $\mathrm{always}_{[0,10\mathrm{hours}]}\,\mathrm{eventually}_{[0,10\mathrm{minutes}]}\,F$.

In order to use MITL and progression to do execution monitoring we have to generate a state sequence correctly representing the behavior of the UAV. To do this we have to process the information available from the sensors and produce a state sequence representation of the behavior. There are two issues to consider. First, what properties are required of the state sequence in order for the evaluation of the MITL formula to be correct with respect to the output from the sensors. Second, how can we from the output of the sensors generate a state sequence which satisfies these properties.

The first question is specific to progression of MITL formulas while the second question is more general. The second question is really about how to generate a representation structure with certain properties. In this case it is about taking the output from the sensors and deriving a state sequence with certain properties. One purpose of DyKnow is to provide a language where such representational structures can be specified declaratively. Such a specification is called a *policy*. These policies are used by DyKnow to actually derive the representation, with the specified properties. If the policy is violated this can be detected and reported in order for the agent to react to the situation and handle it appropriately.

The general idea is to use a declarative policy to specify the properties of a representational structure and to show that these properties are sufficient for the purpose of the representation. The same specification also gives the implementation of DyKnow the possibility to detect deviations from this specification. The actual instance of the representational structure is usually dependent on some external source, such as a sensor. If the external source does not conform with its policy then a policy violation will be detected and the agent can react to the event in whatever way appropriate.

By using formulas that are triggered by many different state sequences, such as limiting the

allowed execution time of an action, it might be difficult to recover from the failure since not enough information is available. For example, if the execution time monitor is triggered then the only thing we know is that the execution took too long time, not what caused the stalling. If we have historic information available about the behavior of the UAV, then a human operator or the UAV itself could query the history by using MITL to express conditions on the behavior of the UAV. Based on the answers a more detailed explanation to the condition of the UAV could be obtained. In fact, this can be done using DyKnow since it is possible to specify the amount of history to be cached for each fluent stream. As long as the content of the cache is enough, we can easily create and evaluate MITL formulas ad-hoc on the history of the system to classify the error. This is also another benefit of using a declarative language to specify the properties, since it is easy to add new properties at run-time, without any need for recompilation.

3.3 Chronicle Recognition

In many applications it is of interest to describe and recognize complex events. One popular formalism for defining complex events is temporal constraint networks (TCN) [8]. A TCN is an acyclic directed graph where the nodes represent primitive events and the edges sets of disjunctive temporal constraints on the occurrence of the primitive events. A TCN corresponds to a temporal constraint satisfaction problem (TCSP) where a set of variables, one for each node, should be assigned a time-point in such a way that the binary constraints defined by the edges are satisfied. If it possible to find such an assignment then the TCN is consistent. To determine if a TCSP is consistent, i.e. has at least one solution, is NP-complete [8]. But, if we restrict the problem to only allow a single temporal constraint on each of the edges in the TCN then we get a simple temporal constraint network whose corresponding simple TCSP can be solved in polynomial time [8].

One formalism which uses TCN as the basis for recognizing complex events is the IxTeT chronicle recognition system. It is developed at LAAS in Toulouse, France, by Ghallab et al. [14]. Since it is based on TCN it is capable of representing constraints as before, equal, after and their disjunctions as well as numerical constraints between time points, but not their disjunctions since then the problem would become NP-complete. The numerical constraints are intervals $[I^-, I^+]$ corresponding to lower and upper bounds on the temporal distance between the time-points. A benefit of the chronicle formalism is that given a stream of time-stamped primitive events it will recognize all possible instances of a chronicle, i.e. TCN, over the event sequence. To do this IxTeT keeps track of all possible developments (Ghallab calls this prediction) in an efficient manner. A recognized chronicle can also be seen as an event in itself which can be used to create recursive chronicles.

One application area for chronicle recognition is in the surveillance of dynamic systems. It has for example been used with success to monitor gas turbines [1] and telecommunication networks [4].

Other similar approaches to IxTeT have been proposed. One example is Fontaine and Ramaux [12] who tries to match temporal constraint networks, instead of satisfying them, by constructing one graph for the chronicle being recognized and one for the actually observed events, called the session, and trying to match them. An approach close to text pattern recognition techniques using finite-state automaton whose transitions correspond to the observed events is presented by Lévy [19]. His approach is very efficient in recognizing sequential chronicles while other structures impair the performance.

A related research area is plan recognition where you infer what plan an agent is executing by

observing the agents' interactions with the world and by maintaining a model of the mental states of the agents, which is a much more general concept than a chronicle or scenario [16]. According to Pynadath and Wellman [20] the difference between general pattern recognition (to which you can count chronicle recognition) and plan recognition is that plans are made by a rational agent with a mental state which could be used to improve the matching. In chronicle recognition you only take the actual events into account not why the agent causing the events did them. Another difference is that plan recognition systems usually do not deal explicitly with time, instead they focus on the plans and what actions or sub plans they are made of.

Chapter 4

Implementing the EEM Using DyKnow

The event and execution monitor is actually a wrapper around two modules which are part of DyKnow, the *formula progressor* and the *chronicle recognition engine* (CRE). Depending on the type of the request one of them will be called in order to perform the actual monitoring. The monitoring of executions are done by the formula progressor while the event monitoring is done by the chronicle recognition engine.

4.1 Implementing Execution monitoring

In order to use the MITL progression algorithm to implement an execution monitoring application an appropriate state sequence must be generated for each formula. In this section we explain how this can be done using DyKnow. First, we describe the properties required from the state sequence to use it for progressing MITL formulas, then how such a state sequence can be extracted from the data streams produced by the sensors using DyKnow.

To guarantee that the state sequence has certain properties, which are required for the MITL progression, we have to assume that the output from the sensors has certain properties. To specify these properties of the sensor streams and the desired properties of the state sequences we introduce a declarative language. In the language these properties are specified by *policies*. Based on a policy DyKnow can check that the actual output from the sensors satisfies the properties. If a policy violation is detected it can be reported to the agent. The agent can react to this event and take the appropriate actions. This is one benefit of using a declarative language. Another benefit is that the user only has to specify what she wants, not how to compute it. We can reason about the data streams produced on an abstract level and prove properties of it. The same specification can be used by the implementation to make sure that the actual data streams satisfies the properties. If they are satisfied then we know that the output has the proved properties, otherwise we are able to detect the violation and can try to handle the situation anyway.

The policy, with the constraints, restricts the set of legal fluent streams. If a fluent stream is produced by the sensors that does not satisfy these constraints then a policy violation will be detected and reported. The agent will have to react to this event and take the appropriate actions. To derive a state sequence from these fluent streams we first derive fluent generator representational structures, i.e. total functions from time to value instead of partial functions as the fluent streams are, from the fluent streams and then sample these representational structures with a regular interval. Each sampling of the fluent generator representational structures will give a state in the state sequence.

To derive a fluent generator representational structure from the fluent stream we have to specify how to approximate the value at those time-points where no sample exists in the fluent stream. The reason is that a fluent stream only has samples at certain time-points while a fluent is a total function from time to value. For example, if a fluent stream has values at time-points 0, 100, and 200, then what are the values at time-points 50, 113, and 222? The simplest way to approximate the value is to assume that all changes in the value will have explicit samples and therefore a value is valid until a sample with a higher valid time is added. This is called the *most recent value approximation* and is part of the fluent generator policy.

To evaluate several formulas over the same state sequence they are collected into a *context*. Each context consists of a set of formulas and a state sequence described by a policy. There are two ways to define the variables that will be part of the state sequence of a context. Either they can be explicitly specified when the context is created or they are extracted from the first formula added to the context. In either case the variables in the state will never change during the lifetime of the context. This means that if the implicit method for adding variables to the state is used then the first formula must contain all the variables to be used for any formula in this context. Since it's easy to write a formula which is evaluated once and then removed this is not very restrictive, but somewhat clumsy. When a context is created formulas can be added and removed. If someone tries to add a formula which contains variables not in the state then an exception will be generated to inform the user about the condition. From an optimization point of view it is important to have as few contexts as possible, since it is much cheaper to evaluate formulas than to generate state sequences.

The interface to the execution monitoring part of the EEM consists of three methods:

```
long monitor_execution(in string formula,
                       in ExecutionMonitorCallback callback);

LongSeq monitor_execution_group(in StringSeq formulas,
                                in boolean dependent,
                                in ExecutionMonitorCallback callback);

void stop_monitor_execution(in long formula_id);
```

The first two are used to create contexts and start monitoring formulas. The difference between them is that the the second one takes a set of formulas which are all evaluated in the same context while the first method creates a context with a single formula in it. The last method is used to remove either a formula or a context (if it is the last formula in the context then the context is removed). The `monitor_execution_group` method is currently not implemented. The policy used to create the state sequence is currently based on sampling the fluent generator of each of the variables in the state every 100 milliseconds. With the current interface it is not possible to change the sampling rate or to specify the start and end of the monitoring, but a part from testing it is easy to add.

As soon as a formula is progressed to either `true` or `false` a call is made to the callback in order to notify the creator of the execution monitoring request. After the callback has been made the formula is removed from the context since it will not be progressed any further. This means that if a formula should be monitored at all times it has to be added again if it is evaluated to `true` or `false`.

4.2 Implementing Event monitoring

The IxTeT implementation that we use is called C.R.S. and is made by France Telecom. In order to use chronicle recognition to recognize event occurrences the event must be expressed in the chronicle formalism and a suitable stream of primitive events must be generated. Since the chronicle formalism does not require the same type of states as the formula progression this is a much simpler problem. In fact, the EEM can subscribe to each entity trajectory structure that is part of a chronicle independently of the others. The only requirement is that the samples arrive ordered by valid time. The reason is that all the information for a specific time-point has to be available before the temporal network can be updated with this new information. This means that whenever a new sample arrives with the valid time t the network is propagated up to the time-point $t-1$ and then the new information is added. If a sample arrives out of order it will be ignored. To add new information the EEM checks whether the value of the attribute of the entity structure has changed compared to the last sample, if it has changed then an event is generated. For example, if a chronicle uses the entity structure `obj` which has the attributes `pos` and `speed` then the EEM would subscribe to the entity trajectory structure for `obj`. Each time a new sample arrives it would check if the `pos` or `speed` attributes have changed. If the `speed` attribute of an entity structure with the name "robot" changes from `high` to `low` at time-point t then an event "the speed attribute of the entity type obj of the instance robot has changed from high to low" is generated at time-point t. This is the type of events which are used to define chronicles and used as input to the chronicle recognition engine.

The interface to the event monitoring part of the EEM consists of two methods:

```
void monitor_event(in string chronicle,
                   in string chronicle_file,
                   in EventCallback callback);

void stop_monitor_event(in string event);
```

The first method is used to starting monitoring an event as defined by the chronicle `chronicle` found in the compiled chronicle file `chronicle_file`. As soon as an instance of the chronicle is detected a call is made to the event callback. The event callback is a function that takes four arguments, the name of the chronicle, the arguments of the chronicle and the interval over which the chronicle instance was recognized. To stop monitoring an event a call to the `stop_monitor_event` method has to be made. Unlike the execution monitoring formulas the event is not removed when an instance has been detected but rather it is continuously monitored until explicitly turned off.

Part II

Tutorial

Chapter 5

Introduction

This tutorial gives a practical introduction to using the event and execution monitor (EEM) centered around a concrete ground robot example. The purpose is to show how to monitor events as well as the execution of a robot in an application. The tutorial is self-contained and should be enough to get started using the EEM.

Consider the following scenario, which is a continuation of the scenario presented in the ESGM report [15]. A wheeled ground robot is moving around in an in-door environment looking for objects to inspect. When it has found an interesting object it should drive there and examine it. In the ESGM report we described how entity trajectory structures with the current object of interest could be generated using computational units and sensor interfaces. In this scenario we will focus on the implementation of an object following behavior which needs to be monitored and which uses the recognition of events to guide its execution. To simplify the scenario we will replace the computed trajectory of objects with a primitive one. The ComputeDirection function will remain to compute the desired driving direction based on the current position of the object of interest and the position of the robot. The entity trajectory generated by this computational unit will be feed together with the current driving direction of the robot into two different computational units in order to calculate the quantitative and qualitative deviation from the desired driving direction. These functions are called DirectionDeviation and QualitativeDeviation. Their output will be used to do the event and execution monitoring. An overview of the data flow in the application is shown in Figure 5.1.

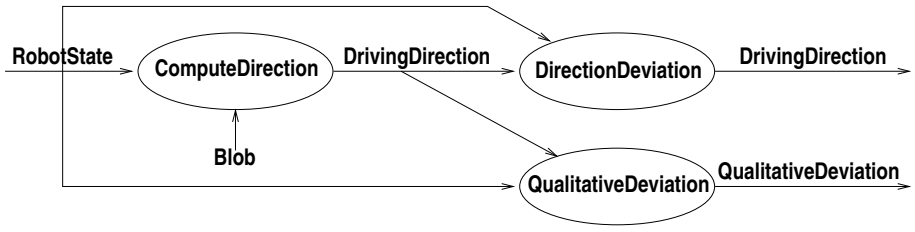

Figure 5.1: An overview of the data flow in the tutorial application. The ellipses are computational units and the lines are entity trajectory structures labeled with the name of its entity structure type.

All the code in the tutorial is written in C++ and is available in the supplementary file called `eem_tutorial.cc`.

Chapter 6

Getting Started

This chapter describes how to install and run the EEM.

6.1 Installing the EEM

- Check out the MACS code from Gibson.
- Follow the instructions in the README file in the LiU/dyknow directory on how to download the DyKnow binaries (which includes the required C.R.S. binaries and libraries) and compile the ESGM and EEM.
- Get an activation key for the C.R.S. software used to do implement the chronicle recognition by following the instructions on the webpage `http://crs.elibel.tm.fr/register/index.mhtml`. Install the activation key by assigning it to the environment variable `CRSKEY`. If no valid activation key is present then the EEM will not be able to recognize any chronicle instances. Currently no checks are made if the key is present so no errors will be caused.

6.2 Running the EEM

To run the EEM you only have to start the program `eem`. It is configured to start all the servers which are required by the EEM by default. When the text `(EEM) ORB running...` is printed in the terminal the EEM is up and running. The EEM uses the ESGM which will also be started unless the option `--no-fork-esgm` is given at the command line.

6.3 Retrieving the EEM

Before the EEM can be used in an application a reference to the actual EEM CORBA server has to be retrieved. This can be done using the helper function `get_object_from_iorserver` which takes a pointer to an `IorServerProxy`, the name of the server, and a variable to store the reference in. The following code retrieves the default IorServer and retrieves a reference to the EEM stored under the name `EEM_NAME`.

```
Witas::Util::IorServerProxy ior_server;
```

```
Eem_var eem;
get_object_from_iorserver(&ior_server, EEM_NAME, eem);
```

If the function is successful the variable *eem* can now be used to call the methods in the EEM interface.

Chapter 7

Setting up the Scenario

To set up a scenario like this four things needs to be done. First the entity structure types must be defined. They describe the entities the application can handle. Then sensors and functions must be made available to the ESGM. This is done by implementing sensor interfaces and computational units. The last step is to create entity trajectory structures which contain the actual data produced by sensors and computed by the functions. This tutorial will not go into the details of how this is done, since this is already described in the ESGM tutorial [15]. Instead we will only show some of the code that is used.

7.1 Creating the Entity Structure Types

This scenario uses four entity structure types: Blob, RobotState, DrivingDirection, and QualitativeDeviation. Of these only the last one is new. The QualitativeDeviation entity structure type represents the qualitative deviation of a robot. It has the following attributes:

- left : bool; true iff the robot is deviating to the left relative to the driving direction of the robot,
- right : bool; true iff the robot is deviating to the right,
- straight : bool; true iff the robot is not deviating but driving in the desired direction.

Like the other entity types their is a wrapper, `QualitativeDeviationWrapper`, to make it easy to get and set the attributes of the entity structures.

The actual code to create the entity structure types are:

```
string blob_type_name("Blob");
string robot_state_type_name("RobotState");
string driving_direction_type_name("DrivingDirection");
string qualitative_deviation_type_name("QualitativeDeviation");
esgm->create_entity_structure_type(blob_type_name.c_str(),
                                   BlobWrapper::type());
esgm->create_entity_structure_type(robot_state_type_name.c_str(),
                                   RobotStateWrapper::type());
esgm->create_entity_structure_type(driving_direction_type_name.c_str(),
                                   DrivingDirectionWrapper::type());
esgm->create_entity_structure_type(qualitative_deviation_type_name.c_str(),
                                   QualitativeDeviationWrapper::type());
```

7.2 Creating the Computational Units

The scenario uses three computational units: ComputeDirection, DirectionDeviation, and QualitativeDeviation. The computational unit to compute the driving direction of the robot is the same as in the ESGM scenario. The other two are new and are used to compute the current deviation from the desired driving direction. The first does a quantitative calculation while the other provides a qualitative classification of the deviations.

7.2.1 Computing the Deviation From the Desired Driving Direction

To compute the deviation from the desired direction we create a function which takes the current robot state and the current desired driving direction as input and compute the absolute difference between them. Since we reuse the DrivingDirection entity structure which also has a speed attribute not used we always set it to 0. The actual implementation is available in the files `DrivingDirectionDeviationCU.h` and `DrivingDirectionDeviationCU.cc` which are included as Appendix III.

7.2.2 Computing the Qualitative Deviation From the Desired Driving Direction

The chronicle recognition engine used to do define events needs input in the form of qualitative primitive events. Therefore we need to transform the numeric deviation into a set of qualitative categories. This classification is done by the QualitativeDeviation computational unit. The classification is quite simple and only looks at the difference between the actual driving direction and the desired direction. If they are almost the same then the robot is driving straight and not deviating at all. If the actual driving direction is less than the desired direction then the robot is deviating to the left, otherwise it is deviating to the right. The actual implementation is available in the files `QualitativeDeviationCU.h` and `QualitativeDeviationCU.cc` which are included as Appendix III.

7.3 Creating the Entity Trajectory Structures

Since we are not using any sensors in this scenario the last step is to create the actual entity trajectory structures. Since this is covered in detail in the ESGM report we will only describe what trajectories are created and what they contain.

First two primitive trajectories with the input to the scenario are created. One with the current robot state and one with the current blob to be followed. They are called `robot_state` and `current_blob`. Then a trajectory called `driving_direction` with the desired driving direction computed from the function DrivingDirection is created. Finally two more computed trajectories are constructed from the functions DirectionDeviation and QualitativeDeviation. They are called `driving_direction_deviation` and `qualitative_deviation`. The actual code is available in Appendix III.

Chapter 8

Monitoring the Object Following Behavior

Now that we have created the data flow in the application we are ready to describe the object following behavior and the monitoring of it.

The goal of the behavior is to minimize the deviation from the desired driving direction to the currently followed object. This means that it will turn the robot towards the object if it detects that the deviation is too large. If the deviation is within limits then drive towards the object until close enough, then stop.

With respect to this behavior we would like to monitor two conditions. First of all, at any time if the deviation is too large then within a few seconds the robot should be turned towards the object. This should be true whether the object following behavior is executing or not. This can be seen as a global monitor which makes sure that the object following behavior is invoked and manages to guide the robot in the right direction within a few seconds. We call this a global monitor since it is not tied to a particular behavior but monitored at all times. The second condition is that at all times the object following behavior is running the robot should never be deviating from the desired direction for more than 500 milliseconds. This means that it should not take more than 500 milliseconds for the behavior to reach an acceptable deviation. If this condition is violated, then one reason for the failure could be that the robot is driving too fast. Therefore the recovery action will be to stop the robot in order to be able to safely turn in the desired direction. We call this a local monitor since it is only active when the object following behavior is active.

8.1 Implementing the Object Following Behavior

The main loop controlling the robot basically checks the current state and decides which behavior to run. In our simple scenario the choice is only whether to run the object following behavior or not to do anything. The object following behavior itself checks the deviation from the desired driving direction, if it is too large (in our case more than 2 degrees) then it turns towards the object. If the deviation is smaller then it checks the distance to the object. If the distance is larger than 1 meter then it will stop, otherwise drive towards the object with a constant speed of 1 m/s.

To keep track of the current state a class called `EemTutorialState` is created. It contains all the variables we are interested in and will also be used in order to implement the callbacks for reporting monitoring violations and recognized events. The code for the state can be found in Appendix III.

The main loop, which is run as its own thread, also simulates the movement of the objects and the failures introduced in order to demonstrate the monitoring capability. The function implementing the main loop is called `thread_main` and can be found in Appendix III.

8.2 Adding the Global Monitor

The purpose of the global monitor, which should always be checked, is to make sure that the robot will start moving towards the current object of interest within a few seconds. This monitor could for example be violated if the robot does not switch to the object following behavior quick enough or if the object following behavior fails for some reason. The formula used is:
$\text{always}(driving\_direction\_deviation.dir > 5$
$\rightarrow \text{eventually}[0, 3000]driving\_direction\_deviation.dir < 5)$. It says that if the deviation from the desired driving direction, represented by the attribute `dir` on the entity trajectory structure `driving_direction_deviation`, is more than 5 degrees then it should be below 5 degrees within 3 seconds (the timeunit in the interval is milliseconds). To violate this formula the deviation has to be more than 5 degrees continuously for more than 3 seconds.

To monitor the formula two things are needed. First, a callback has to be created which is used by the EEM to notify the application that a formula was evaluated either to `true` or `false`. Since this is an always formula, which is `true` if and only if all states satisfy the condition, this means that it can never be evaluated to `true` since we never have the complete state sequence. Therefore it would be a bug in the formula progressor if it was ever evaluated to `true`. Second, the formula and the callback has to be registered with the EEM in order for it to be evaluated.

To create a callback you have to implement the `ExecutionMonitorCallback` interface defined in the IDL-file `eem_callbacks.idl` (see Appendix III). This interface defines two methods, `formula_violated` and `formula_satisfied`, which has to be implemented:

```
interface ExecutionMonitorCallback
{
  void formula_violated(in long formula_id,
                        in Time begin_time,
                        in Time end_time);

  void formula_satisfied(in long formula_id,
                         in Time begin_time,
                         in Time end_time);
};
```

To simplify the creation of these callbacks a general implementation exists, called `EemEmCallback` which takes two functions of two arguments (which is the start and end time of the interval over which the formula was evaluated) as input. The first function will be called when the formula is evaluated to `false` and the second when it is evaluated to `true`. These functions could for example be methods on an object, which is what is used in the tutorial. There it is two methods, `violate_global_monitor` and `satisfy_global_monitor`, on an `EemTutorialState` object. The code for this is:

```
FormulaFun
  gviolated(bind(&EemTutorialState::violate_global_monitor,
            state, _1, _2));
FormulaFun
  gsatisfied(bind(&EemTutorialState::satisfy_global_monitor,
```

```
            state, _1, _2));
EemEmCallback global_callback_servant(gviolated, gsatisfied,
                                      state->verbose);
ExecutionMonitorCallback_var
  global_monitor_callback = global_callback_servant._this();
```

Since the monitor is removed when the formula is evaluated to `true` or `false` it is important that the monitor identifier is removed in the callbacks. For example, this is the actual callback used in the tutorial when the global monitor is violated:

```
void
EemTutorialState::violate_global_monitor(Time begin, Time end)
{
  global_monitor = -1;
  speed = 0;
}
```

The first line resets the global monitor identifier to -1. The second line tells the robot to stop moving by setting the speed to 0.

The next step is to register the formula and the callback. This is done through calling the `monitor_execution` method in the EEM interface. The method takes the formula as a string and a reference to the callback CORBA object as inputs and returns a formula identifier. If the identifier is -1 then the formula was not added for some reason. Currently there does not exist any good feedback but instead you have to look what the EEM prints out. The most common error is that the formula is not syntactically correct. The second most common error is that the variables used in the formula are not defined in the ESGM. The formula id is used when you want to stop the monitoring, so remember to store it. In our case the code to start the monitoring is:

```
state->global_monitor =
  state->eem->monitor_execution(formula.c_str(),
                                global_monitor_callback.in());
```

Where `formula` is the string "always ((driving_direction_deviation.dir $>$ 5) -> (eventually [0, 3000] (driving_direction_deviation.dir $<$ 5)))". The `state` variable is a pointer to the current `EemTutorialState` which contains the formula identifier of the global monitor and a reference to the EEM server. If the variable `global_monitor` is not -1 then the formula is being monitored. It will be monitored until either it is explicitly stopped or evaluated to `true` (which should never happen) or `false`. This means that the formula has to be registered again if it has been violated and it should continue to be monitored after the violation has been resolved. This is done in the main loop where in each iteration it is checked if the global monitor is -1 then it is added again. This assures that the formula will always be monitored.

To test the monitoring the tutorial application will not invoke the object following behavior the first time a new object is recognized. After three seconds the formula will be evaluated to `false` and the monitor is violated. The violation callback is called and the application sets a flag which invokes the correct behavior. The monitor should not be violated after that.

8.3 Adding and Removing the Local Monitor

The purpose of the local monitor, which is only checked when the object following behavior is running, is to make sure the robot never deviates from its desired course for more than a short period of time. This monitor will be violated if the robot isn't able to turn in the right

direction fast enough. One reason for this could be that it is driving too fast or the object it is following is moving too fast. Since the robot can't control the object it will assume that it is driving too fast and stop, and then try to steer in the right direction. The formula used is: $\text{always}(\text{eventually}[0, 500]driving\_direction\_deviation.dir < 5)$. It says that it always the case that within 500 milliseconds the robot will have a deviation of less than 5 degrees. To violate this formula the robot must have a deviation of more than 5 degrees continuously for more than 500 milliseconds. This formula could be made more restrictive by removing the interval. If we had the formula $\text{always}\, driving\_direction\_deviation.dir < 5$ then the robot would not be allowed to deviate from the course at all. This has two downsides. First, the formula would have to be added only after the robot has turned and reached the desired direction. Second, the condition is very likely to be violated even if the execution is going smoothly possibly due to errors in the measurements or sporadic delays in the values which would make them unsynchronized for a short period. Therefore it is better to have a somewhat forgiving condition which doesn't react to the slightest changes.

To monitor the formula we have to create a callback and tell the EEM to start monitoring the formula, just like we did with the global monitor. This is done with the following code:

```
FormulaFun
  lviolated(bind(&EemTutorialState::violate_local_monitor,
            state, _1, _2));
FormulaFun
  lsatisfied(bind(&EemTutorialState::satisfy_local_monitor,
            state, _1, _2));
EemEmCallback local_callback_servant(lviolated, lsatisfied,
                                     state->verbose);
ExecutionMonitorCallback_var
  local_monitor_callback = local_callback_servant._this();
```

The main difference is that now the formula should be added whenever the object following behavior is started and it should be removed whenever the behavior is stopped. To start monitoring the formula the following code is used:

```
if ( state->follow_object ) {
  if ( not state->following_object ) {
    state->following_object = true;

    if ( state->local_monitor == -1 ) {
      state->local_monitor
       = state->eem->monitor_execution(formula.c_str(),
                                       local_monitor_callback.in());
    }
  }
}
```

Where `formula` is the string "always (eventually [0,500] (driving_direction_deviation.dir < 5))". The code says that if we should follow the object and we are not currently following the object, then start monitoring the execution unless it is already started. To monitor the execution the method `monitor_execution` in the EEM interface should be called.

To stop the monitoring call the `stop_monitor_execution` method in the EEM interface. This is done when the behavior is no longer active. The actual code used in the tutorial is:

```
if ( state->local_monitor != -1 ) {
```

```
  state->eem->stop_monitor_execution(state->local_monitor);
  state->local_monitor = -1;
}
```

The code first checks that the local monitor is actually being monitored, then it removes it.

To test the monitoring of the local formula the tutorial application will not turn in the desired direction the first time the object following behavior is activated. After 500 milliseconds the formula will be evaluated to false and the monitor is violated. The violation callback is called and the application sets a flag which prevents the behavior from not turning (this means that there is no real recovery action associated with this monitor, only a flag that causes it to be triggered). The monitor should not be violated after that.

Chapter 9

Detecting a Zig-Zag Behavior

One problem which could occur is that the robot is not driving straight towards the object but is swaying back and forth in a zig-zag like manner. Given that the robot is always driving straight towards the object often enough (i.e. within 500 milliseconds) during the zig-zag movement it will not be detected by the monitoring formulas described above since the deviation is small enough often enough. Since this is an unwanted behavior we would like to detect it. One description of the situation is where the robot is first deviating to the left, then deviating to the right, and then deviating to the left again. This means that we would like to recognize the sequence of three events, deviating left, deviating right and then deviating left. One way of doing it would be to write a formula that captures the situation. This would be a quite complex formula with several nested temporal operators since an operator would be needed to capture each state change. The formula would look something like $\mathrm{eventually}(left \wedge \mathrm{eventually}(right \wedge \mathrm{eventually}\, left))$ and this would then be satisfied if such a behavior was recognized. To convert it to a monitoring formula which should be violated if the behavior is detected the formula must be negated. If we add the extra condition that it should be at most 500 milliseconds between the first and the second left deviation, then there is no simple formula that would capture this situation since we can not refer to explicit time-points in the formulas. But, the same information is easily expressed by a simple temporal network (STN) as shown in Figure 9.1.

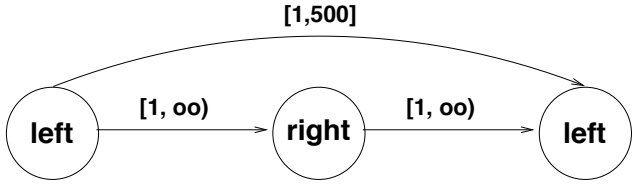

Figure 9.1: A simple temporal network describing a zig-zag behavior. The nodes represent time-points when events happen and the arrows represent temporal constraints on the time-points. The constraint [1,oo) means that it should be at least one timeunit between the events (oo is the same as infinity).

This type of simple temporal networks can also be expressed in the chronicle formalism. A chronicle representing the zigzag behavior consists of three parts. The domain definitions, the attribute definitions and the definition of the chronicle itself. The first part is the domain definition:

```
domain Boolean = { unknown, true, false }
domain Object = ~{}
```

This defines two domains `Boolean` which contains three possible values and `Object` which can contain any value ($\sim\{\}$) means everything that is not in the empty set). The second part is the attribute definitions:

```
attribute qualitative_deviation.left[?obj]
  { ?obj in Object ; ?value in Boolean }
attribute qualitative_deviation.straight[?obj]
  { ?obj in Object ; ?value in Boolean }
attribute qualitative_deviation.right[?obj]
  { ?obj in Object ; ?value in Boolean }
```

This defines three attributes representing the three possible qualitative deviations as represented by the entity trajectory structure `qualitative_deviation` and its three attributes `left`, `straight`, and `right`. When the EEM finds the attribute `a_b_c.d` then it knows that it should look for the entity trajectory called `a_b_c` and its attribute called `d`. If such an entity trajectory does not exist then the EEM will fail to register the chronicle. Each attribute has one argument, which will be the name of the entity structure. For example, if the `qualitative_deviation` entity trajectory structure contains an entity structure with the name "robot" and the three attribute value pairs $\{\langle left, \texttt{false}\rangle, \langle straight, \texttt{false}\rangle, \langle right, \texttt{true}\rangle\}$ this would correspond to the attribute `qualitative_deviation.left[robot]` having the value `false` at the valid time of the entity structure.

The third part is the actual chronicle definition itself:

```
chronicle zigzag[?obj]
{
  event(qualitative_deviation.left[?obj]:(false, true), t0)
  event(qualitative_deviation.right[?obj]:(false, true), t1)
  event(qualitative_deviation.left[?obj]:(false, true), t2)

  t1-t0 > 0
  t2-t1 > 0
  t2-t0 < 500
}
```

This defines the chronicle `zigzag` which takes one argument which is the object that is found to have the zigzag behavior. The chronicle consists of three events, one for each of the nodes in the simple temporal network. Each event represents the change of values in one attribute. The reason we introduced a three valued boolean domain was to represent the first event, namely the change of values from `unknown` to `true` or `false`. If we didn't have this third value then we would have to wait until the second value change before we could generate the first event (or always regard the first assignment of an attribute as a change). The first event statement should be read as: the `left` attribute of the entity trajectory structure `qualitative_deviation` changes value from `false` to `true` at time-point t_0. Then we have three temporal constraints encoding the arrows in the STN. The timeunit is milliseconds. The complete chronicle can be found in Append III.

9.1 Compiling the Chronicle

To be able to recognize instances of a chronicle the specification must currently be compiled. To do this run the chronicle compiler `ccrs` on the chronicle specification file:

```
> ccrs zigzag.crs
```

This will generate a file called `zigzag.xrs` which can be used by the EEM to register the chronicle.

9.2 Start Recognizing the Chronicle

To start recognizing instances of a chronicle it must be registered with the EEM using the `monitor_event` method. It takes three arguments, the name of the chronicle, the name of the file with the compiled chronicle and a reference to a callback object. The callback is used by the EEM to notify the application that an instance of the chronicle has been recognized. To create a callback the `EventCallback` interface must be implemented. The interface consists of a single method `event_detected` and is defined in the `eem_callbacks.idl` file:

```
interface EventCallback
{
  void event_detected(in string name,
                      in StringSeq args,
                      in Time begin_time,
                      in Time end_time);
};
```

The `name` argument is the name of the chronicle, `args` is the sequence of arguments to the chronicle, and `begin_time` and `end_time` defines the interval over which the chronicle instance took place. To simplify the creation of a callback a wrapper exists, similar to the one for the execution monitor callback, called `EemEventCallback`. It creates a callback from a function which takes four arguments, the same as the `event_detected` method takes. The code for creating a callback which calls the method `EemTutorialState::event` on the object `state` is:

```
EventFun event(bind(&EemTutorialState::event, &state, _1, _2, _3, _4));
EemEventCallback event_callback_servant(event, state.verbose);
EventCallback_var event_callback = event_callback_servant._this();
```

To register the chronicle and start detecting instances call the `monitor_event` method in the EEM interface:

```
string zigzag_name("zigzag");
string zigzag_file("/home/frehe/fhwitas/dyknow2/doc/zigzag.xrs");
eem->monitor_event(zigzag_name.c_str(),
                   zigzag_file.c_str(),
                   event_callback.in());
```

Where the first argument is the name of the chronicle, in our case “zigzag”, the second argument is the filename of the chronicle, and the third argument is a reference to a callback object.

In our scenario the chronicle is registered before the scenario is started in order to detect all zigzag instances. To test the monitoring of the event the tutorial application will deliberately make a zigzag movement if the flag `zig_zag` is set.

9.3 Stop Recognizing the Chronicle

When a chronicle has been registered all recognized instances of the chronicle will generate a call to the `event_detected` method on the callback object. To stop the recognition a call to the `stop_monitor_event` method of the EEM must be made. The method takes a single argument which is the name of the chronicle that should no longer be monitored:

```
eem->stop_monitor_event(zigzag_name.c_str());
```

In our scenario the chronicle is monitored the whole time and is not removed until the robot is stopped by `ctrl-c`.

To run the tutorial compile it, start the EEM, and then execute the tutorial binary. The print outs in the terminal should describe what is expected and what actually happens. This ends this tutorial.

Part III

Appendices

```
/**
 * @file eem.idl
 * Definition of the EEM interface.
 */


#ifndef EEM_IDL
#define EEM_IDL


#include "logable.idl"
#include "pingable.idl"
#include "statable.idl"
#include "resetable.idl"
#include "memory_statable.idl"
#include "eem_callbacks.idl"

#pragma prefix "ida.liu.se/witas"


module Witas {
  module Idl {
    module Dyknow {

      interface Eem
        : Pingable, Resetable, MemoryStatable, Statable, Logable
      {
        /**
         * Monitor the execution of the formula @a formula. If the
         * formula is evaluated to true or false a call to the
         * appropriate method in the callback will be made. The id of
         * the formula is returned. If it is -1 then the formula was
         * not added properly.
         */
        long monitor_execution(in string formula,
                               in ExecutionMonitorCallback callback);

        /**
         * Monitor the execution of all the formulas @a formulas. If
         * dependent is true then all formulas are stopped if one
         * formula is evaluated to false. If a formula is evaluated to
         * true or false a call to the appropriate method in the
         * callback will be made. The list of ids of all the formulas
         * is returned. If an id is -1 then the formula was not added
         * properly.
         */
        LongSeq monitor_execution_group(in StringSeq formulas,
                                        in boolean dependent,
                                        in ExecutionMonitorCallback callback);


        /** Stop monitoring the formula @a formula_id. */
        void stop_monitor_execution(in long formula_id);
```

```
        /**
         * Try to find instances of an event in the form of a
         * chronicle @a chronicle. The chronicle is read from the file
         * with the name @a chronicle_file. If an instance of the
         * event is detected the @a callback will be used to notify
         * the user about it.
         */
        void monitor_event(in string chronicle,
                           in string chronicle_file,
                           in EventCallback callback);

        /** Stop monitoring the event @a event. */
        void stop_monitor_event(in string event);
      };

    };
  };
};


#endif
```

```
/**
 * @file eem_callbacks.idl
 * Definition of the EEM callbacks.
 */


#ifndef EEM_CALLBACKS_IDL
#define EEM_CALLBACKSI_DL


#include "common_types.idl"

#pragma prefix "ida.liu.se/witas"


module Witas {
  module Idl {
    module Dyknow {

      interface ExecutionMonitorCallback
      {
        void formula_violated(in long formula_id,
                              in Time begin_time,
                              in Time end_time);

        void formula_satisfied(in long formula_id,
                               in Time begin_time,
                               in Time end_time);
      };


      interface EventCallback
      {
        void event_detected(in string name,
                            in StringSeq args,
                            in Time begin_time,
                            in Time end_time);
      };

    };
  };
};


#endif
```

```
/* -*- Mode: C++ -*- */

/**
 * @file eem_tutorial.cc
 *
 * The code from the tutorial on the event and execution monitor.
 *
 * Created by: Fredrik Heintz 2006-09-24
 */


#include "BlobWrapper.h"
#include "RobotStateWrapper.h"
#include "DrivingDirectionWrapper.h"
#include "QualitativeDeviationWrapper.h"
#include "DrivingDirectionCU.h"
#include "DrivingDirectionDeviationCU.h"
#include "QualitativeDeviationCU.h"
#include "SubCallback.h"
#include "EemTutorialState.h"
#include "eem_tutorial_opt.h"

#include "dyknowutil/EemEmCallback.h"
#include "dyknowutil/EemEventCallback.h"
#include "dyknowutil/SubscriptionProxy.h"
#include "dyknowutil/idl_dyknow_types_extensions.h"

#include "util/OrbProxy.h"
#include "util/witasgendefs.h"
#include "util/IorServerProxy.h"
#include "util/witasiornamesdefs.h"
#include "util/get_object_from_iorserver.h"
#include "util/idl_common_types_extensions.h"

#include "dyknow2/eemC.h"
#include "dyknow2/esgmC.h"
#include "ace/Signal.h"
#include "ace/Thread_Manager.h"

#include <boost/bind.hpp>
#include <iostream>


using std::cout;
using std::cerr;
using std::endl;
using std::flush;
using std::boolalpha;
using std::string;

using boost::bind;

using Witas::Idl::Time;
```

```
using Witas::Idl::mStringSeq;
using namespace Witas::Util;
using namespace Witas::Dyknow;
using namespace Witas::Idl::Dyknow;


static sig_atomic_t finished = 0;
extern "C" void sig_handler(int sig)
{
  if ( finished == 1 ) {
    cerr << "Terminating tutorial with signal " << sig << endl;
    exit(1);
  } else {
    if ( finished == 1 ) {
      cerr << "2nd try: Terminating tutorial with signal " << sig << endl;
      exit(1);
    }

    cerr << "Shutting down tutorial with signal " << sig << endl;
    finished = 1;
  }
}


void*
thread_main(void* input)
{
  cout << "started robot thread" << endl;
  EemTutorialState* state = static_cast<EemTutorialState*>(input);
  assert ( state != 0 );


  // Create a callback servant for the global monitor
  FormulaFun gviolated(bind(&EemTutorialState::violate_global_monitor, state, _1, _2));
  FormulaFun gsatisfied(bind(&EemTutorialState::satisfy_global_monitor, state, _1, _2));
  EemEmCallback global_callback_servant(gviolated, gsatisfied, state->verbose);
  ExecutionMonitorCallback_var
    global_monitor_callback = global_callback_servant._this();

  // Create a callback servant for the local monitor
  FormulaFun lviolated(bind(&EemTutorialState::violate_local_monitor, state, _1, _2));
  FormulaFun lsatisfied(bind(&EemTutorialState::satisfy_local_monitor, state, _1, _2));
  EemEmCallback local_callback_servant(lviolated, lsatisfied, state->verbose);
  ExecutionMonitorCallback_var
    local_monitor_callback = local_callback_servant._this();

  while ( not finished ) {
    try {
      ++state->cycles_since_move;
      if ( state->cycles_since_move > 50 ) {
        state->cycles_since_move = 0;
        state->move_object = true;
```

```
  ++state->iteration;
  switch ( state->iteration ) {
  case 1:
    // Global monitor fail
    state->zig_zag = false;
    state->follow_object = false;
    state->fail_to_follow_object = false;
    cout << "NEW ITERATION: follow object should violate the global monitor" << endl;
    break;
  case 3:
    // Local monitor fail
    state->zig_zag = false;
    state->follow_object = true;
    state->fail_to_follow_object = true;
    cout << "NEW ITERATION: follow object should violate the local monitor" << endl;
    break;
  case 5:
    // Zig zag
    state->zig_zag = true;
    state->follow_object = true;
    state->fail_to_follow_object = false;
    cout << "NEW ITERATION: follow object should trigger the zig zag event" << endl;
    break;
  default:
    // Normal
    state->zig_zag = false;
    state->follow_object = true;
    state->fail_to_follow_object = false;
    cout << "NEW ITERATION: no violations or events expected" << endl;
  }
}

// Add the global monitor, if it is not added and we have started the scenario
if ( state->global_monitor == -1 and state->iteration > 0 ) {
  cout << "add global monitor" << endl;
  //string formula = "always ( (prev(driving_direction.dir) != driving_direction.dir) -> eventually [0, 3000] (driving_d
  string formula = "always ( (driving_direction_deviation.dir > 5) -> (eventually [0, 3000]
  state->global_monitor =
    state->eem->monitor_execution(formula.c_str(),
                                  global_monitor_callback.in());
}

if ( state->move_object ) {
  cout << "move object" << endl;
  state->following_object = false;
  state->speed = 0;

  if ( state->local_monitor != -1 ) {
    cout << "remove local monitor" << endl;
    state->eem->stop_monitor_execution(state->local_monitor);
    state->local_monitor = -1;
  }
} else {
```

```
  if ( state->follow_object ) {
    if ( not state->following_object ) {
      cout << "start following the object" << endl;
      state->following_object = true;

      // Add the local monitor for this behavior
      if ( state->local_monitor == -1 ) {
        cout << "add local monitor" << endl;
        string formula = "always (eventually [0,500] (driving_direction_deviation.dir < 5))";
        //string formula = "always ( driving_direction_deviation.dir < 5 )";
        state->local_monitor
          = state->eem->monitor_execution(formula.c_str(),
                                          local_monitor_callback.in());
      }
    }
  }
}

if ( state->following_object ) {
  if ( fabs(state->robot.dir() - state->desired_dir) > 2.0 ) {
    if ( not state->fail_to_follow_object ) {
      // If the desired direction is different from the current
      // driving direction then change the current driving direction
      // to the desired direction.
      state->robot.dir(state->desired_dir);
      cout << "New robot direction is " << state->desired_dir << endl;
    }
  } else {
    if ( state->zig_zag ) {
      static bool zig = true;
      double new_dir;
      if ( zig ) {
        new_dir = state->desired_dir-60*drand48();
        cout << "Zig: New robot direction is " << new_dir << endl;
      } else {
        new_dir = state->desired_dir+60*drand48();
        cout << "Zag: New robot direction is " << new_dir << endl;
      }
      zig = not zig;
      state->robot.dir(new_dir);
    }
    if ( state->distance_to_object() > 1.0 ) {
      if ( state->speed < 0.1 ) {
        cout << "start moving" << endl;
        state->speed = 0.5;
      }
    } else if ( state->speed > 0 ) {
      cout << "close to object, stop" << endl;
      state->speed = 0;
    } else {
      //cout << "stopped close to object" << endl;
    }
  }
```

```
      }

      // Update robot state
      state->update_robot_state();

      // Update blob
      state->move_blob();
    } catch (const CORBA::Exception& e) {
      cout << "caught CORBA::Exception " << e << endl;
    } catch (const std::exception& e) {
      cout << "caught std::exception " << e.what() << endl;
    } catch (...) {
      cout << "caught unknown exception" << endl;
    }

    usleep(100*1000);
  }

  if ( state->global_monitor != -1 ) {
    cout << "remove global monitor" << endl;
    state->eem->stop_monitor_execution(state->global_monitor);
    state->global_monitor = -1;
  }

  state->done = true;

  cout << "killed the robot thread" << endl;
  state->grp_id = -1;
  return 0;
}


int main(int argc, char* argv[])
{
  // Initialize the ORB
  OrbProxy::init(argc, argv);
  OrbProxy* orb = OrbProxy::instance();


  // Parse the rest of the arguments
  optionProcess(&eem_tutorialOptions, argc, argv);


  try {
    // Get parameters from the command line
    int verbose = OPT_VALUE_VERBOSE;
    string ior_server_address(OPT_ARG(IORSERVER));
    Time sample_period = OPT_VALUE_SAMPLE_PERIOD*TIMEUNITS_PER_MILLI_SECOND;

    string esgm_name(OPT_ARG(ESGM_NAME));
    if ( esgm_name == "" ) {
      esgm_name = ESGM_NAME;
    }
```

```
string eem_name(OPT_ARG(EEM_NAME));
if ( eem_name == "" ) {
  eem_name = EEM_NAME;
}

bool reset = false;
if ( ENABLED_OPT(RESET) ) {
  reset = true;
}

if ( verbose > 0 ) {
  cout << " verbose = " << verbose
       << "\n ior_server_address = " << ior_server_address
       << "\n esgm_name = " << esgm_name
       << "\n eem_name = " << eem_name
       << "\n sample_period = " << sample_period
       << "\n reset = " << boolalpha << reset
       << endl;
}


// Register handler of ctrl-c and other signals
ACE_Sig_Set ss;
ss.sig_add(SIGINT);
ss.sig_add(SIGTERM);
ss.sig_add(SIGQUIT);
ss.sig_add(SIGHUP);
ACE_Sig_Set sm;
ACE_Sig_Action sa(ss, static_cast<ACE_SignalHandler>(sig_handler), sm);


// Retrieve the iorserver
cout << "retrieving the ior server " << ior_server_address << "... " << flush;
IorServerProxy ior_server(ior_server_address);
cout << "done" << endl;

// Retrieve the ESGM from the iorserver
cout << "retrieving the ESGM " << esgm_name << "... " << flush;
Esgm_var esgm;
get_object_from_iorserver(&ior_server, esgm_name, esgm);
cout << "done" << endl;

if ( reset ) {
  esgm->reset();
}

// Retrieve the EEM from the iorserver
cout << "retrieving the EEM " << eem_name << "... " << flush;
Eem_var eem;
get_object_from_iorserver(&ior_server, eem_name, eem);
cout << "done" << endl;
```

```
if ( reset ) {
  eem->reset();
}

// Create the entity structure types
string blob_type_name("Blob");
string robot_state_type_name("RobotState");
string driving_direction_type_name("DrivingDirection");
string qualitative_deviation_type_name("QualitativeDeviation");

esgm->create_entity_structure_type(blob_type_name.c_str(),
                                   BlobWrapper::type());
esgm->create_entity_structure_type(robot_state_type_name.c_str(),
                                   RobotStateWrapper::type());
esgm->create_entity_structure_type(driving_direction_type_name.c_str(),
                                   DrivingDirectionWrapper::type());
esgm->create_entity_structure_type(qualitative_deviation_type_name.c_str(),
                                   QualitativeDeviationWrapper::type());


// Create a primitive entity trajectory structure for the robot
// sensor
string robot_state_name("robot_state");
cout << "create " << robot_state_name << " entity trajectory structure... " << flush;
EntityTrajectoryStructure_var robot_state
  = esgm->create_primitive_entity_trajectory_structure(robot_state_name.c_str(),
                                                       robot_state_type_name.c_str());
cout << "done" << endl;

// Create a primitive entity trajectory structure called
// current_blob which represents the blob currently being followed
string current_blob_name("current_blob");
cout << "create " << current_blob_name << " entity trajectory structure... " << flush;
EntityTrajectoryStructure_var current_blob
  = esgm->create_primitive_entity_trajectory_structure(current_blob_name.c_str(),
                                                       blob_type_name.c_str());
cout << "done" << endl;


// Create an instance of the driving direction comp unit
string driving_dir_cu_name("DrivingDirection");
cout << "create " << driving_dir_cu_name << " comp unit servant... " << flush;
DrivingDirectionCU driving_direction_cu_servant(verbose);
cout << "done" << endl;

cout << "create " << driving_dir_cu_name << " comp unit corba object... " << flush;
CompUnit_var driving_direction_cu = driving_direction_cu_servant._this();
cout << "done" << endl;

// Register the driving direction comp unit
cout << "add the " << driving_dir_cu_name << " comp unit... " << flush;
esgm->add_comp_unit(driving_dir_cu_name.c_str(),
                    driving_direction_type_name.c_str(),
```

```
                    mValueSortSeq(ENTITY_FRAME_TYPE),
                    driving_direction_cu.in());
cout << "done" << endl;

// Create a computed entity trajectory structure called
// driving_direction which computes the driving direction to the blob
string driving_dir_name("driving_direction");
cout << "create " << driving_dir_name.c_str() << " entity trajectory structure... " << flush;
EntityTrajectoryStructure_var driving_direction
  = esgm->create_computed_entity_trajectory_structure(driving_dir_name.c_str(),
                                                      driving_dir_cu_name.c_str(),
                                                      mStringSeq(robot_state_name,
                                                                 current_blob_name));
cout << "done" << endl;


// Create an instance of the driving direction deviation comp unit
string driving_dir_dev_cu_name("DrivingDirectionDeviation");
cout << "create " << driving_dir_dev_cu_name << " comp unit servant... " << flush;
DrivingDirectionDeviationCU driving_direction_deviation_cu_servant(verbose);
cout << "done" << endl;

cout << "create " << driving_dir_dev_cu_name << " comp unit corba object... " << flush;
CompUnit_var driving_direction_deviation_cu = driving_direction_deviation_cu_servant._this();
cout << "done" << endl;

// Register the driving direction comp unit
cout << "add the " << driving_dir_dev_cu_name << " comp unit... " << flush;
esgm->add_comp_unit(driving_dir_dev_cu_name.c_str(),
                    driving_direction_type_name.c_str(),
                    mValueSortSeq(ENTITY_FRAME_TYPE),
                    driving_direction_deviation_cu.in());
cout << "done" << endl;

// Create a computed entity trajectory structure called
// driving_direction_deviation which computes the driving
// direction deviation to the blob
string driving_dir_dev_name("driving_direction_deviation");
cout << "create " << driving_dir_dev_name.c_str() << " entity trajectory structure... " << flush;
EntityTrajectoryStructure_var driving_direction_deviation
  = esgm->create_computed_entity_trajectory_structure(driving_dir_dev_name.c_str(),
                                                      driving_dir_dev_cu_name.c_str(),
                                                      mStringSeq(robot_state_name,
                                                                 driving_dir_name));
cout << "done" << endl;


// Create an instance of the qualitative deviation comp unit
string qualitative_dev_cu_name("QualitativeDeviation");
cout << "create " << qualitative_dev_cu_name << " comp unit servant... " << flush;
QualitativeDeviationCU qualitative_deviation_cu_servant(verbose);
cout << "done" << endl;
```

```
cout << "create " << qualitative_dev_cu_name << " comp unit corba object... " << flush;
CompUnit_var qualitative_deviation_cu = qualitative_deviation_cu_servant._this();
cout << "done" << endl;

// Register the driving direction comp unit
cout << "add the " << qualitative_dev_cu_name << " comp unit... " << flush;
esgm->add_comp_unit(qualitative_dev_cu_name.c_str(),
                    qualitative_deviation_type_name.c_str(),
                    mValueSortSeq(ENTITY_FRAME_TYPE),
                    qualitative_deviation_cu.in());
cout << "done" << endl;

// Create a computed entity trajectory structure called
// qualitativeection_deviation which computes the driving
// direction deviation to the blob
string qualitative_dev_name("qualitative_deviation");
cout << "create " << qualitative_dev_name.c_str() << " entity trajectory structure... " << flush;
EntityTrajectoryStructure_var qualitative_deviation
  = esgm->create_computed_entity_trajectory_structure(qualitative_dev_name.c_str(),
                                                      qualitative_dev_cu_name.c_str(),
                                                      mStringSeq(robot_state_name,
                                                                 driving_dir_name));
cout << "done" << endl;


{
  // Create an object to represent the current state
  EemTutorialState state(eem.in(),
                         robot_state.in(),
                         current_blob.in(),
                         verbose);


  cout << "subscribe to the desired driving direction... " << flush;
  EFFun dd_fun(bind(&EemTutorialState::set_desired_dir, &state, _1));
  SubCallback desired_dir_callback(dd_fun, verbose);
  SubscriptionProxy dd_sub_proxy(driving_direction->subscribe(),
                                 mSymbol(driving_dir_name),
                                 &desired_dir_callback);
  cout << "done" << endl;


  cout << "subscribe to the driving direction deviation... " << flush;
  EFFun dev_fun(bind(&EemTutorialState::set_deviation_dir, &state, _1));
  SubCallback deviation_dir_callback(dev_fun, verbose);
  SubscriptionProxy devd_sub_proxy(driving_direction_deviation->subscribe(),
                                   mSymbol(driving_dir_dev_name),
                                   &deviation_dir_callback);
  cout << "done" << endl;


  cout << "subscribe to the qualitative deviation... " << flush;
  EFFun qdev_fun(bind(&EemTutorialState::set_qualitative_deviation, &state, _1));
```

```
  SubCallback qualitative_dev_callback(qdev_fun, verbose);
  SubscriptionProxy qdev_sub_proxy(qualitative_deviation->subscribe(),
                                   mSymbol(qualitative_dev_name),
                                   &qualitative_dev_callback);
  cout << "done" << endl;


  // Create a callback servant for the zigzag event
  EventFun event(bind(&EemTutorialState::event, &state, _1, _2, _3, _4));
  EemEventCallback event_callback_servant(event, state.verbose);
  EventCallback_var event_callback = event_callback_servant._this();

  // Start detecting zigzag events
  string zigzag_name("zigzag");
  string zigzag_file("/home/frehe/fhwitas/dyknow2/doc/zigzag.xrs");
  eem->monitor_event(zigzag_name.c_str(), zigzag_file.c_str(),
                     event_callback.in());


  cout << "create robot thread... " << flush;
  ACE_Thread_Manager* thr_mgr = ACE_Thread_Manager::instance();
  state.grp_id = thr_mgr->spawn(&thread_main, &state);
  cout << "done" << endl;


  cout << "scenario started" << endl;
  while ( not state.done ) {
    ACE_Time_Value tv(1, 100000);
    orb->run(tv);
  }
  cout << "scenario finished" << endl;

  cout << "wait for robot thread to finish... " << flush;
  while ( state.grp_id != -1 ) {
    usleep(200*1000);
  }
  cout << "done";


  // Stop detecting the zigzag events
  eem->stop_monitor_event(zigzag_name.c_str());
}


// Destroy the entity trajectory structures created
cout << "destroy " << robot_state_name << " trajectory... " << flush;
esgm->destroy_entity_trajectory_structure(robot_state_name.c_str());
cout << "done" << endl;

cout << "destroy " << current_blob_name << " trajectory... " << flush;
esgm->destroy_entity_trajectory_structure(current_blob_name.c_str());
cout << "done" << endl;
```

```
  cout << "destroy " << driving_dir_name << " trajectory... " << flush;
  esgm->destroy_entity_trajectory_structure(driving_dir_name.c_str());
  cout << "done" << endl;

  cout << "destroy " << driving_dir_dev_name << " trajectory... " << flush;
  esgm->destroy_entity_trajectory_structure(driving_dir_dev_name.c_str());
  cout << "done" << endl;

  cout << "destroy " << qualitative_dev_name << " trajectory... " << flush;
  esgm->destroy_entity_trajectory_structure(qualitative_dev_name.c_str());
  cout << "done" << endl;


  // Remove the computational units
  cout << "destroy " << driving_dir_cu_name << " CU... " << flush;
  esgm->remove_comp_unit(driving_dir_cu_name.c_str());
  cout << "done" << endl;

  cout << "destroy " << driving_dir_dev_cu_name << " CU... " << flush;
  esgm->remove_comp_unit(driving_dir_dev_cu_name.c_str());
  cout << "done" << endl;

  cout << "destroy " << qualitative_dev_cu_name << " CU... " << flush;
  esgm->remove_comp_unit(qualitative_dev_cu_name.c_str());
  cout << "done" << endl;


  // Destroy the entity structure types created
  cout << "destroy " << blob_type_name << " type... " << flush;
  esgm->destroy_entity_structure_type(blob_type_name.c_str());
  cout << "done" << endl;

  cout << "destroy " << robot_state_type_name << " type... " << flush;
  esgm->destroy_entity_structure_type(robot_state_type_name.c_str());
  cout << "done" << endl;

  cout << "destroy " << driving_direction_type_name << " type... " << flush;
  esgm->destroy_entity_structure_type(driving_direction_type_name.c_str());
  cout << "done" << endl;

  cout << "destroy " << qualitative_deviation_type_name << " type... " << flush;
  esgm->destroy_entity_structure_type(qualitative_deviation_type_name.c_str());
  cout << "done" << endl;
} catch (const CORBA::Exception& e) {
  cout << "caught CORBA::Exception " << e << endl;
} catch (const std::exception& e) {
  cout << "caught std::exception " << e.what() << endl;
} catch (...) {
  cout << "caught unknown exception" << endl;
}

cout << "exit" << endl;
return 0;
```

```
}
```

```
/* -*- Mode: C++ -*- */

/**
 * @file EemTutorialState.h
 * Include file for the class EemTutorialState.
 *
 * Created by: Fredrik Heintz 2006-10-03
 */


#ifndef EEM_TUTORIAL_STATE_H
#define EEM_TUTORIAL_STATE_H


#include "BlobWrapper.h"
#include "RobotStateWrapper.h"

#include "dyknow2/eemC.h"
#include "dyknow2/entity_trajectory_structureC.h"


struct EemTutorialState
{
  EemTutorialState(Witas::Idl::Dyknow::Eem_ptr eem_,
                   Witas::Idl::Dyknow::EntityTrajectoryStructure_ptr robot_state_,
                   Witas::Idl::Dyknow::EntityTrajectoryStructure_ptr current_blob_,
                   int verbose_ = 0);

  Witas::Idl::Dyknow::Eem_var eem;
  Witas::Idl::Dyknow::EntityTrajectoryStructure_var robot_state;
  Witas::Idl::Dyknow::EntityTrajectoryStructure_var current_blob;


  int grp_id;
  bool done;
  int iteration;
  int verbose;

  bool move_object;
  bool follow_object;
  bool fail_to_follow_object;
  bool following_object;
  bool zig_zag;
  int cycles_since_move;

  Witas::Dyknow::BlobWrapper blob;
  Witas::Dyknow::RobotStateWrapper robot;
  double desired_dir;
  double speed;

  int global_monitor;
  int local_monitor;
```

```
  double distance_to_object() const;

  void update_robot_state();
  void move_blob();

  void set_robot_state(const Witas::Idl::Dyknow::EntityFrame& ef);
  void set_desired_dir(const Witas::Idl::Dyknow::EntityFrame& ef);
  void set_deviation_dir(const Witas::Idl::Dyknow::EntityFrame& ef);
  void set_qualitative_deviation(const Witas::Idl::Dyknow::EntityFrame& ef);

  void violate_global_monitor(Witas::Idl::Time begin, Witas::Idl::Time end);
  void satisfy_global_monitor(Witas::Idl::Time begin, Witas::Idl::Time end);

  void violate_local_monitor(Witas::Idl::Time begin, Witas::Idl::Time end);
  void satisfy_local_monitor(Witas::Idl::Time begin, Witas::Idl::Time end);

  void event(const std::string& name, const Witas::Idl::StringSeq& args,
             Witas::Idl::Time begin, Witas::Idl::Time end);
};


#endif
```

```
/* -*- Mode: C++ -*- */

/**
 * @file EemTutorialState.cc
 * Implementation file for the class EemTutorialState.
 *
 * Created by: Fredrik Heintz 2006-10-03
 */


#include "EemTutorialState.h"
#include "DrivingDirectionWrapper.h"
#include "QualitativeDeviationWrapper.h"
#include "dyknowutil/idl_dyknow_types_extensions.h"
#include "util/idl_common_types_extensions.h"
#include <iostream>


using std::cout;
using std::endl;
using std::string;
using Witas::Idl::Time;
using Witas::Idl::StringSeq;
using namespace Witas::Dyknow;
using namespace Witas::Idl::Dyknow;


static const double PI = 3.14159265358979323846;
static const double DEG2RAD = PI / 180.0;
static const double RAD2DEG = 180.0 / PI;


EemTutorialState::EemTutorialState(Eem_ptr eem_,
                                   EntityTrajectoryStructure_ptr robot_state_,
                                   EntityTrajectoryStructure_ptr current_blob_,
                                   int verbose_)
  : eem(Eem::_duplicate(eem_)),
    robot_state(EntityTrajectoryStructure::_duplicate(robot_state_)),
    current_blob(EntityTrajectoryStructure::_duplicate(current_blob_)),
    grp_id(-1),
    done(false),
    iteration(0),
    verbose(verbose_),
    move_object(false),
    follow_object(false),
    fail_to_follow_object(false),
    following_object(false),
    zig_zag(false),
    cycles_since_move(0),
    blob("blob", 10, 10, 10),
    robot("robot", 0, 0, 0),
    desired_dir(0),
    speed(0),
```

```
    global_monitor(-1),
    local_monitor(-1)
{
  robot_state->create_at_now(robot.name, robot.attributes);
  current_blob->create_at_now(blob.name, blob.attributes);
}


double
EemTutorialState::distance_to_object() const
{
  double dx = blob.x() - robot.x();
  double dy = blob.y() - robot.y();
  return sqrt(dx*dx + dy*dy);
}


void
EemTutorialState::update_robot_state()
{
  robot.x(robot.x() + speed*cos(robot.dir()*DEG2RAD));
  robot.y(robot.y() + speed*sin(robot.dir()*DEG2RAD));
  robot_state->create_at_now(robot.name, robot.attributes);
  //cout << "New state for " << robot << endl;
}


void
EemTutorialState::move_blob()
{
  if ( move_object ) {
    if ( robot.x() > blob.x() )
      blob.x(blob.x()+int(10*drand48()));
    else
      blob.x(blob.x()-int(10*drand48()));
    if ( robot.y() > blob.y() )
      blob.y(blob.y()+int(10*drand48()));
    else
      blob.y(blob.y()-int(10*drand48()));
    current_blob->create_at_now(blob.name, blob.attributes);
    cout << "Changed current blob to " << blob << endl;
    move_object = false;
  }
}


void
EemTutorialState::set_robot_state(const EntityFrame& ef)
{
  robot = RobotStateWrapper(ef);
}
```

```
void
EemTutorialState::set_desired_dir(const EntityFrame& ef)
{
  desired_dir = DrivingDirectionWrapper(ef).dir();
  //cout << "new desired direction is " << desired_dir << endl;
}


void
EemTutorialState::set_deviation_dir(const Witas::Idl::Dyknow::EntityFrame& ef)
{
  if ( verbose > 0 ) {
    double deviation_dir = DrivingDirectionWrapper(ef).dir();
    cout << "deviation dir = " << deviation_dir << endl;
  }
}


void
EemTutorialState::set_qualitative_deviation(const Witas::Idl::Dyknow::EntityFrame& ef)
{
  if ( verbose > 0 ) {
    QualitativeDeviationWrapper qd(ef);
    cout << "qdev: left = " << std::boolalpha << qd.left()
         << "; straight = " << std::boolalpha << qd.straight()
         << "; right = " << std::boolalpha << qd.right() << endl;
  }
}


void
EemTutorialState::violate_global_monitor(Time begin, Time end)
{
  cout << "violated global monitor over interval [" << begin
       << ", " << end << "]" << endl;
  global_monitor = -1;
  speed = 0;
  follow_object = true;
}


void
EemTutorialState::satisfy_global_monitor(Time begin, Time end)
{
  cout << "Error: satisfied global monitor over interval [" << begin
       << ", " << end << "]" << endl;
  global_monitor = -1;
}


void
EemTutorialState::violate_local_monitor(Time begin, Time end)
{
```

```
  cout << "violated local monitor over interval [" << begin
       << ", " << end << "]" << endl;
  local_monitor = -1;
  speed = 0;
  fail_to_follow_object = false;
}


void
EemTutorialState::satisfy_local_monitor(Time begin, Time end)
{
  cout << "Error: satisfied local monitor over interval [" << begin
       << ", " << end << "]" << endl;
  local_monitor = -1;
}


void
EemTutorialState::event(const string& name, const StringSeq& args,
                        Time begin, Time end)
{
  cout << "detected event " << name << "[" << args
       << "] over the interval [" << begin
       << ", " << end << "]" << endl;
}
```

```
/* -*- Mode: C++ -*- */

/**
 * @file QualitativeDeviationWrapper.h
 * Include file for the class QualitativeDeviationWrapper.
 *
 * Created by: Fredrik Heintz 2006-10-03
 */


#ifndef QUALITATIVE_DEVIATION_WRAPPER_H
#define QUALITATIVE_DEVIATION_WRAPPER_H

#include "dyknow2/entity_structureC.h"
#include "dyknowutil/EntityStructureWrapper.h"

namespace Witas {
  namespace Dyknow {

    class QualitativeDeviationWrapper
      : public EntityStructureWrapper
    {
    public:
      QualitativeDeviationWrapper(const std::string& name);
      QualitativeDeviationWrapper(const std::string& name,
                                  bool left, bool straight, bool right);
      QualitativeDeviationWrapper(const Idl::Dyknow::EntityFrame& ef);

      ~QualitativeDeviationWrapper();

      void check_attributes();

      void left(bool val);
      bool left() const;

      void straight(bool val);
      bool straight() const;

      void right(bool val);
      bool right() const;

      static Idl::Dyknow::EntityStructureType type();
    };

  }
}


#endif
```

```
/* -*- Mode: C++ -*- */

/**
 * @file QualitativeDeviationWrapper.cc
 * Implementation file for the class QualitativeDeviationWrapper.
 *
 * Created by: Fredrik Heintz 2006-10-03
 */


#include "QualitativeDeviationWrapper.h"


namespace Witas {
  namespace Dyknow {

    QualitativeDeviationWrapper::QualitativeDeviationWrapper(const std::string& name)
      : EntityStructureWrapper(name)
    {
      set_bool("left"); set_bool("straight"); set_bool("right");
    }


    QualitativeDeviationWrapper::QualitativeDeviationWrapper(const std::string& name,
                                                             bool left,
                                                             bool straight,
                                                             bool right)
      : EntityStructureWrapper(name)
    {
      set_bool("left", left); set_bool("straight", straight); set_bool("right", right);
    }


    QualitativeDeviationWrapper::QualitativeDeviationWrapper(const Idl::Dyknow::EntityFrame& ef)
      : EntityStructureWrapper(ef)
    {
      check_attributes();
    }


    QualitativeDeviationWrapper::~QualitativeDeviationWrapper() {}


    void
    QualitativeDeviationWrapper::check_attributes()
    {
      if ( not exists_attribute("left", Idl::Dyknow::BOOL_TYPE) ) set_bool("left");
      if ( not exists_attribute("straight", Idl::Dyknow::BOOL_TYPE) ) set_bool("straight");
      if ( not exists_attribute("right", Idl::Dyknow::BOOL_TYPE) ) set_bool("right");
    }


    void QualitativeDeviationWrapper::left(bool val) { set_bool("left", val); }
```

```
    bool QualitativeDeviationWrapper::left() const { return get_bool("left"); }


    void QualitativeDeviationWrapper::straight(bool val) { set_bool("straight", val); }
    bool QualitativeDeviationWrapper::straight() const { return get_bool("straight"); }


    void QualitativeDeviationWrapper::right(bool val) { set_bool("right", val); }
    bool QualitativeDeviationWrapper::right() const { return get_bool("right"); }


    Idl::Dyknow::EntityStructureType
    QualitativeDeviationWrapper::type()
    {
      static Idl::Dyknow::EntityStructureType qd_type;
      static bool first = true;
      if ( first ) {
        qd_type.name = CORBA::string_dup("QualitativeDeviationType");
        qd_type.attributes.length(3);
        qd_type.attributes[0].name = CORBA::string_dup("left");
        qd_type.attributes[0].type = Idl::Dyknow::BOOL_TYPE;
        qd_type.attributes[1].name = CORBA::string_dup("straight");
        qd_type.attributes[1].type = Idl::Dyknow::BOOL_TYPE;
        qd_type.attributes[2].name = CORBA::string_dup("right");
        qd_type.attributes[2].type = Idl::Dyknow::BOOL_TYPE;

        first = false;
      }
      return qd_type;
    }

  }
}
```

```
/* -*- Mode: C++ -*- */

/**
 * @file DrivingDirectionDeviationCU.h
 *
 * The header file for the driving direction deviation computational
 * unit used in the MACS tutorial.
 *
 * Created by: Fredrik Heintz 2006-10-02
 */


#ifndef DRIVING_DIRECTION_DEVIATION_CU_H
#define DRIVING_DIRECTION_DEVIATION_CU_H


#include "wcorba/dyknow2/comp_unitS.h"


/** The implementation of the driving direction computational unit. */
class DrivingDirectionDeviationCU
  : public virtual POA_Witas::Idl::Dyknow::CompUnit
{
public:
  DrivingDirectionDeviationCU(int verbose);

  virtual Witas::Idl::Dyknow::Sample*
  compute(const Witas::Idl::Dyknow::SampleSeq& inputs, Witas::Idl::Time qtime)
    ACE_THROW_SPEC ((CORBA::SystemException,
                     Witas::Idl::Dyknow::ValueNotAvailable));


private:
  int verbose_;
};


#endif
```

```
/* -*- Mode: C++ -*- */

/**
 * @file DrivingDirectionDeviationCU.cc
 *
 * The implementation file for the driving direction deviation
 * computational unit used in the MACS tutorial.
 *
 * Created by: Fredrik Heintz 2006-10-02
 */


#include "DrivingDirectionDeviationCU.h"
#include "DrivingDirectionWrapper.h"
#include "RobotStateWrapper.h"

#include "dyknowutil/dyknow_type_conversion.h"
#include "dyknowutil/idl_dyknow_types_extensions.h"

#include <iostream>

using std::cout;
using std::endl;
using Witas::Idl::Time;
using namespace Witas::Dyknow;
using namespace Witas::Idl::Dyknow;


DrivingDirectionDeviationCU::DrivingDirectionDeviationCU(int verbose)
  : verbose_(verbose)
{
  if ( verbose_ > 1 ) cout << "DrivingDirectionDeviationCU(" << verbose_ << ")" << endl;
}


Sample*
DrivingDirectionDeviationCU::compute(const SampleSeq& inputs, Time qtime)
  ACE_THROW_SPEC ((CORBA::SystemException,
                   ValueNotAvailable))
{
  if ( verbose_ > 1 )
    cout << "DrivingDirectionDeviationCU::compute(" << inputs << ", " << qtime << ")" << endl;
  try {
    // input[0] is the robot state and input[1] the driving direction
    if ( inputs.length() == 2
         and inputs[0].vtime >= inputs[1].vtime ) {
      RobotStateWrapper robot_state(get_EntityFrame(inputs[0].val));
      DrivingDirectionWrapper driving_dir(get_EntityFrame(inputs[1].val));
      Time now = qtime;
      // Compute the difference in driving directions, i.e. the
      // difference between the desired direction and the actual
      // driving direction.
      double dir_diff = driving_dir.dir() - robot_state.dir();
```

```
      while (dir_diff < -180) dir_diff+=360;
      while (dir_diff > 180) dir_diff-=360;
      double dir = fabs(dir_diff);
      double speed = 0.0;
      DrivingDirectionWrapper driving_dir_dev(std::string(robot_state.name), dir, speed);
      if ( 0 < verbose_ and verbose_ <= 1 )
        cout << " DrivingDirectionDeviationCU(" << robot_state << "@" << inputs[0].vtime
             << ", " << driving_dir << "@" << inputs[1].vtime << ")";
      if ( verbose_ > 0 ) cout << " = " << driving_dir_dev << endl;
      if ( verbose_ > 0 )
        cout << " vtime diff = " << inputs[0].vtime - inputs[1].vtime << endl;
      return new mSample(mValue(driving_dir_dev), inputs[0].vtime, qtime, now);
    }
  } catch ( const CORBA::Exception& e ) {
    if ( not (verbose_ > 1) )
      cout << "DrivingDirectionCU::compute(" << inputs << ", " << qtime << ")";
    cout << "DrivingDirectionDeviationCU::compute threw " << e << endl;
  }
  throw ValueNotAvailable();
}
```

```
/* -*- Mode: C++ -*- */

/**
 * @file QualitativeDeviationCU.h
 *
 * The header file for the qualitative deviation computational unit
 * used in the MACS tutorial.
 *
 * Created by: Fredrik Heintz 2006-10-03
 */


#ifndef QUALITATIVE_DEVIATION_CU_H
#define QUALITATIVE_DEVIATION_CU_H


#include "wcorba/dyknow2/comp_unitS.h"


/** The implementation of the qualitative deviation computational unit. */
class QualitativeDeviationCU
  : public virtual POA_Witas::Idl::Dyknow::CompUnit
{
public:
  QualitativeDeviationCU(int verbose);

  virtual Witas::Idl::Dyknow::Sample*
  compute(const Witas::Idl::Dyknow::SampleSeq& inputs, Witas::Idl::Time qtime)
    ACE_THROW_SPEC ((CORBA::SystemException,
                     Witas::Idl::Dyknow::ValueNotAvailable));

private:
  int verbose_;
};


#endif
```

```
/* -*- Mode: C++ -*- */

/**
 * @file QualitativeDeviationCU.cc
 *
 * The implementation file for the qualitative deviation computational
 * unit used in the MACS tutorial.
 *
 * Created by: Fredrik Heintz 2006-10-03
 */


#include "QualitativeDeviationCU.h"
#include "DrivingDirectionWrapper.h"
#include "QualitativeDeviationWrapper.h"
#include "RobotStateWrapper.h"

#include "dyknowutil/dyknow_type_conversion.h"
#include "dyknowutil/idl_dyknow_types_extensions.h"

#include <iostream>

using std::cout;
using std::endl;
using std::flush;
using Witas::Idl::Time;
using namespace Witas::Dyknow;
using namespace Witas::Idl::Dyknow;


QualitativeDeviationCU::QualitativeDeviationCU(int verbose)
  : verbose_(verbose)
{
  if ( verbose_ > 1 ) cout << "QualitativeDeviationCU(" << verbose_ << ")" << endl;
}


Sample*
QualitativeDeviationCU::compute(const SampleSeq& inputs, Time qtime)
  ACE_THROW_SPEC ((CORBA::SystemException,
                   ValueNotAvailable))
{
  if ( verbose_ > 1 )
    cout << "QualitativeDeviationCU::compute(" << inputs << ", " << qtime << ")" << flush;
  try {
    // input[0] is the robot state and input[1] the driving direction
    if ( inputs.length() == 2
         and inputs[0].vtime >= inputs[1].vtime ) {
      RobotStateWrapper robot_state(get_EntityFrame(inputs[0].val));
      DrivingDirectionWrapper driving_dir(get_EntityFrame(inputs[1].val));
      Time now = qtime;

      // Compute the deviation
```

```
      double dir_diff = driving_dir.dir() - robot_state.dir();
      while (dir_diff < -180) dir_diff+=360;
      while (dir_diff > 180) dir_diff-=360;

      // Compute the qualitative deviation
      bool left = dir_diff >= 0.1;
      bool right = dir_diff <= -0.1;
      bool straight = not (left or right);

      QualitativeDeviationWrapper qualitative_dir(std::string(robot_state.name),
                                                  left, straight, right);
      if ( 0 < verbose_ and verbose_ <= 1 )
        cout << " QualitativeDeviationCU(" << robot_state << "@" << inputs[0].vtime
             << ", " << driving_dir << "@" << inputs[1].vtime << ")";
      if ( verbose_ > 0 ) cout << " = " << qualitative_dir << endl;
      if ( verbose_ > 0 )
        cout << " vtime diff = " << inputs[0].vtime - inputs[1].vtime << endl;
      return new mSample(mValue(qualitative_dir), inputs[0].vtime, qtime, now);
    }
  } catch ( const CORBA::Exception& e ) {
    if ( not (verbose_ > 1) )
      cout << "QualitativeDeviationCU::compute(" << inputs << ", " << qtime << ")";
    cout << " threw " << e << endl;
  }
  throw ValueNotAvailable();
}
```

```
/* -*- Mode: C++ -*- */

/**
 * @file EemEmCallback.h
 * Include file for the class EemEmCallback.
 *
 * Created by: Fredrik Heintz 2006-10-03
 */


#ifndef EEM_EM_CALLBACK_H
#define EEM_EM_CALLBACK_H


#include "dyknow2/eem_callbacksS.h"
#include <boost/function.hpp>


typedef boost::function<void (Witas::Idl::Time begin, Witas::Idl::Time end)> FormulaFun;

class EemEmCallback
  : public virtual POA_Witas::Idl::Dyknow::ExecutionMonitorCallback {
public:
  EemEmCallback(const FormulaFun& violated_fun,
                const FormulaFun& satisfied_fun,
                int verbose = 0);

  virtual void formula_violated(CORBA::Long formula_id,
                                Witas::Idl::Time begin_time,
                                Witas::Idl::Time end_time)
    ACE_THROW_SPEC ((CORBA::SystemException));

  virtual void formula_satisfied(CORBA::Long formula_id,
                                 Witas::Idl::Time begin_time,
                                 Witas::Idl::Time end_time)
    ACE_THROW_SPEC ((CORBA::SystemException));


private:
  FormulaFun violated_fun_;
  FormulaFun satisfied_fun_;
  int verbose_;
};


#endif
```

```
/* -*- Mode: C++ -*- */

/**
 * @file EemEmCallback.cc
 * Implementation file for the class EemEmCallback.
 *
 * Created by: Fredrik Heintz 2006-10-03
 */

#include "EemEmCallback.h"
#include <iostream>

using std::cout;
using std::endl;
using Witas::Idl::Time;


EemEmCallback::EemEmCallback(const FormulaFun& violated_fun,
                             const FormulaFun& satisfied_fun,
                             int verbose)
  : violated_fun_(violated_fun),
    satisfied_fun_(satisfied_fun),
    verbose_(verbose)
{
}


void
EemEmCallback::formula_violated(CORBA::Long formula_id,
                                Time begin_time,
                                Time end_time)
  ACE_THROW_SPEC ((CORBA::SystemException))
{
  if ( verbose_ > 0 ) cout << "formula " << formula_id << " false over [" << begin_time
                           << ", " << end_time << "]" << endl;
  violated_fun_(begin_time, end_time);
}


void
EemEmCallback::formula_satisfied(CORBA::Long formula_id,
                                 Time begin_time,
                                 Time end_time)
  ACE_THROW_SPEC ((CORBA::SystemException))
{
  if ( verbose_ > 0 ) cout << "formula " << formula_id << " true over [" << begin_time
                           << ", " << end_time << "]" << endl;
  satisfied_fun_(begin_time, end_time);
}
```

```
/* -*- Mode: C++ -*- */

/**
 * @file EemEventCallback.h
 * Include file for the class EemEventCallback.
 *
 * Created by: Fredrik Heintz 2006-10-04
 */


#ifndef EEM_EVENT_CALLBACK_H
#define EEM_EVENT_CALLBACK_H


#include "dyknow2/eem_callbacksS.h"
#include <boost/function.hpp>


typedef boost::function<void (const std::string& name,
                              const Witas::Idl::StringSeq& args,
                              Witas::Idl::Time start,
                              Witas::Idl::Time end)> EventFun;

class EemEventCallback
  : public virtual POA_Witas::Idl::Dyknow::EventCallback {
public:
  EemEventCallback(const EventFun& fun, int verbose = 0);

  virtual void event_detected(const char* name,
                              const Witas::Idl::StringSeq& args,
                              Witas::Idl::Time begin_time,
                              Witas::Idl::Time end_time)
    ACE_THROW_SPEC ((CORBA::SystemException));


private:
  EventFun fun_;
  int verbose_;
};


#endif
```

```
/* -*- Mode: C++ -*- */

/**
 * @file EemEventCallback.cc
 * Implementation file for the class EemEventCallback.
 *
 * Created by: Fredrik Heintz 2006-10-04
 */


#include "EemEventCallback.h"
#include "dyknowutil/idl_dyknow_types_extensions.h"
#include "util/idl_common_types_extensions.h"
#include <iostream>


using std::cout;
using std::endl;
using Witas::Idl::Time;


EemEventCallback::EemEventCallback(const EventFun& fun, int verbose)
  : fun_(fun),
    verbose_(verbose)
{
}


void
EemEventCallback::event_detected(const char* name,
                                 const Witas::Idl::StringSeq& args,
                                 Time begin_time,
                                 Time end_time)
  ACE_THROW_SPEC ((CORBA::SystemException))
{
  if ( verbose_ > 0 ) cout << "event " << name << "[" << args
                           << "] detected over [" << begin_time
                           << ", " << end_time << "]" << endl;
  fun_(name, args, begin_time, end_time);
}
```

```
domain Boolean = { unknown, true, false }
domain Object = ~{}

attribute qualitative_deviation.left[?obj] { ?obj in Object ; ?value in Boolean }
attribute qualitative_deviation.straight[?obj] { ?obj in Object ; ?value in Boolean }
attribute qualitative_deviation.right[?obj] { ?obj in Object ; ?value in Boolean }


chronicle zigzag[?obj]
{
        event(qualitative_deviation.left[?obj]:(false, true), t0)
        event(qualitative_deviation.right[?obj]:(false, true), t1)
        event(qualitative_deviation.left[?obj]:(false, true), t2)

        t1-t0 > 0
        t2-t1 > 0
        t2-t0 < 500
}
```

Bibliography

[1] J. Aguilar, K. Bousson, C. Dousson, M. Ghallab, A. Guasch, R. Milne, C. Nicol, J. Quevedo, and L. Travé-Massuyès. TIGER: real-time situation assessment of dynamic systems. *Intelligent Systems Engineering*, pages 103–124, 1994.

[2] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

[3] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelliegnce*, 22:5–27, 1998.

[4] S. Bibas, M.-O. Cordier, P. Dague, C. Dousson, F. Lévy, and L. Rozé. Alarm driven supervision for telecommunication networks: I - off-line scenario generation *and* II - on-line chronicle recognition. *Annals of Telecommuncations*, pages 493–508, 1996.

[5] Marcus Bjäreland. *Model-based Execution Monitoring*. PhD thesis, Linköping Studies in Science and Technology, Dissertation No 688, 2001.

[6] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[7] Giuseppe De Giacomo, Raymond Reiter, and Mikhail Soutchanski. Execution monitoring of high-level robot programs. In *Principles of Knowledge Representation and Reasoning*, pages 453–465, 1998.

[8] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[9] Erich Rome et. al. Development of an affordance-based control architecture. Technical report, MACS/2/2.2, jun 2006.

[10] Joaquin L. Fernández and Reid G. Simmons. Robust execution monitoring for navigation plans. In *Proceedings of the Conference on Intelligent Robots and Systems (IROS)*, 1998.

[11] Matthias Fichtner, Axel Grossmann, and Michael Thielscher. Intelligent execution monitoring in dynamic environments. *Fundam. Inf.*, 57(2-4):371–392, 2003.

[12] D. Fontaine and N. Ramaux. An approach by graph for the recognition of temporal scenarios. *IEEE transactions on System, Man and Cybernetics*, 1997.

[13] E. Gat, M. G. Slack, D. P. Miller, and R. J. Firby. Path planning and execution monitoring for a planetary rover. In *Robotics and Automation, 1990. Proceedings., 1990 IEEE International Conference on*, pages 20–25 vol.1, 1990.

[14] Malik Ghallab. On chronicles: Representation, on-line recognition and learning. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pages 597–607, San Francisco, November 5–8 1996. Morgan Kaufmann.

[15] Fredrik Heintz, Patrick Doherty, Björn Wingman, Piotr Rudol, and Mariusz Wzorek. The entity structure generation module. Technical report, MACS/4/3.2, sep 2006.

[16] Henry A. Kautz and James F. Allen. Generalized plan recognition. In Tom Kehler and Stan Rosenschein, editors, *Proceedings of the Fifth National Conference on Artificial Intelligence*, Los Altos, California, 1986. American Association for Artificial Intelligence, Morgan Kaufmann.

[17] Jonas Kvarnström. *TALplanner and Other Extensions to Temporal Action Logic*. PhD thesis, Linköping Studies in Science and Technology, Dissertation No 937, 2005.

[18] K. Ben. Lamine and F. Kabanza. Reasoning about robot actions: A model checking approach. In *Advances in Plan-Based Control of Robotic Agents*, LNAI, pages 123–139, 2002.

[19] F. Lévy. Recognising scenarios: a study. In *Proceedings of the Fifth International Workshop of Diagnosis*, pages 174–178, 1994.

[20] David V. Pynadath and Michael P. Wellman. Accounting for context in plan recognition, with application to traffic monitoring. In Besnard, Philippe and Steve Hanks, editors, *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI'95)*, pages 472–481, San Francisco, CA, USA, August 1995. Morgan Kaufmann Publishers.