SIXTH FRAMEWORK PROGRAMME

PRIORITY 2

Information Society Technologies

Project n° 033841

EMIL

Emergence In the Loop: simulating the two way dynamics of norm innovation

Deliverable 3.3

EMIL-S: THE SIMULATION PLATFORM

Due date of the deliverable: 31 Aug 2008 + 45 dd.

Actual submission date: 1 Oct 2008

Start date of the Project: **1 Sept 2006** Duration: **36 months**

Organization responsible for this Deliverable: UNI KO-LD

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)	

Contents

1 Preliminary remark

Deliverable D 3.3 is mainly the preliminary version of EMIL-S, the simulator which allows to run simulations in which EMIL-A agents interact with each other and from whose interactions might emerge. This document describes the implementation of the simulator and its interfaces formally. Here, Interfaces mean: A User interface for the description of the dynamical structure of the application (the EMIL-S in a narrower sense), a user interface for the initialisation and simulation control and, an interface to the experiment software.

The general design of the simulator was presented during the EMIL workshop in Budapest in June 2008 (as were the prototypes of two applications). Those parts of the implementation of EMIL-S that are responsible for defining and running concrete simulation applications were programmed mainly in July 2008. EMIL-S was tested with the two applications described in Deliverable 3.2. In August a short meeting between the Koblenz and Budapest team defined the interface between EMIL-S and the experiment software MEME [1].

Other applications (HUME and the different social contexts model) will be described for EMILS-S in the last three months of 2008 and are likely to lead to some extensions of EMIL-S. The definition of the other applications will be done together with other EMIL members who will take advantage of short training meetings during which they will be introduced to the modelling capabilities of EMIL-S.

2 EMIL-S Architecture

The simulation suite consists of three sub-systems (Figure 1):

- An available simulation tool, which provide a concrete application scenarios in which EMIL-A agents could be embedded (e.g. TRASS [4, 5], models developed in REPAST [8] or NETLOGO [7]).

- The EMIL-S component for describing software agents with norm formation abilities, acting in a scenario provided by the simulation tool.

- An experimentation environment, which allows the flexible and comfortable execution of simulation experiments (e.g. searching parameter spaces systematically for sensitivity analyses), realized by an interface to the available MEME software.

Dependencies between these sub-systems can be described briefly:

- Within the scenario generated by the simulation tool, the agents implement an interface (or wrapper) to EMIL-S. This wrapper converts EMIL-A messages to commands in the scenario and perceptive events in the scenario to EMIL-A messages

- EMIL-S implements agents on basis of the agent architecture defined in this paper. EMIL-S agents sends EMIL-A messages to the wrapper and receives EMIL-A messages from the wrapper, and, thus, are able to communicate with and about the agents and the environment on the simulation tool level

- The experimentation environment has access to the EMIL-S agents for configuring and analyzing simulation runs; it further has access (possibly indirect via another wrapper which then will be situated within the EMIL-S sub-system) to manage simulation runs

The remaining sections of this paper concentrate especially on the EMIL-S sub-system, with the components EMIL-S agent (see section 3) and the componet EMIL-S AgentDesigner (see section 4), and the interface to the Experimentation Environment MEME (see section 5).

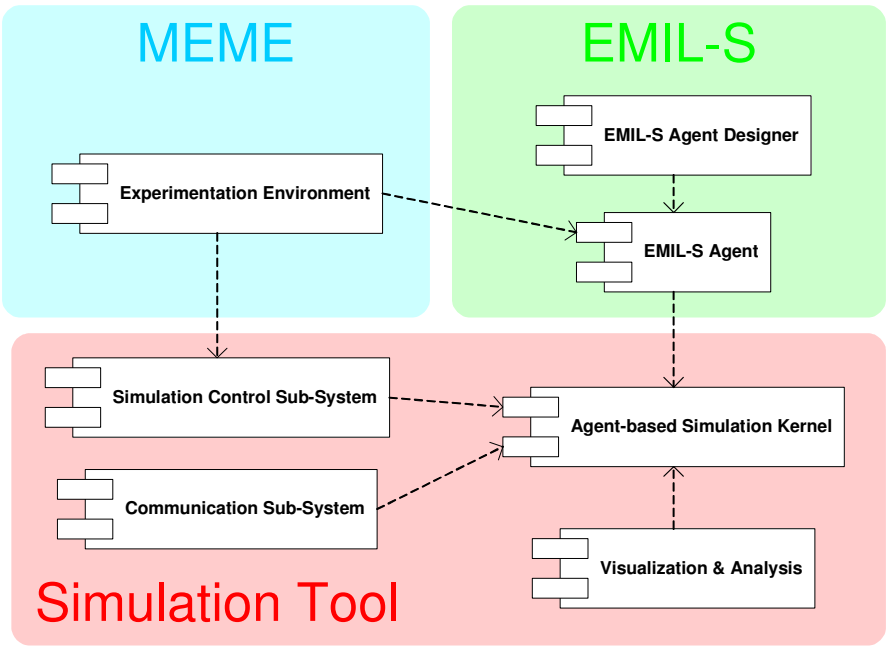

Figure 1 Component diagram of the entire simulation suite

3 Formal Description of EMIL-S

3.1 Event Board and Normative Frame

Generically, a scenario can be described by a finite set of event-action trees and an associated set of selection probability functions. Therefore, event-action trees are the basic elements to specify agent dynamics in EMIL-S. The following shows how these behavioural fragments can be put together within a simulation process to develop behaviour with increasing complexity, originating from regularities or normative beliefs and finally resulting in commonly accepted norms.

As one can see from the sample scenarios in Deliverable 3.2, it is possible to distinguish between two classes of events: First, events originating from the agent perception, including the EMIL-A modals A (assertion) and B (behavior) and second, events that occur when an agent receives a notification or valuation from another agent, including the EMIL-A message modals V (valuation), D (deontic) and S (sanction). In the following, the former are called **environmental events** and the latter **norm invocation events**.

In the same way actions are classified as environmental when influencing the environment directly, and as norm invocation when related with learning in the broadest sense. At the same time, the set of environmental actions forms the state space for all possible environmental agent states.

If one considers this type of description (i.e. event-action trees with selection probabilities) as a kind of meta language that enables agents to converse about an environment, this approach should be appropriate for describing simulation scenarios in general. Additionally, this implies that the modeller has to provide an environment by some means or other. Within this “real” environment, (environmental) events have to be generated by perceptive processes, while the (environmental) actions are transformed into actions influencing the particular environment.

On basis of this meta language, a general simulation process can be defined. This process requires at least two kinds of agent-internal memories: Firstly, an **event board** (or fact base), memorizing a history of incoming events including the conducted actions. And, secondly, a **normative frame** (or rule base), holds regularities and norms, derived from the event board as a result of the simulation.

The event board is a chronologically sorted sequence (or list) whose elements contain the following data (Figure 2):

- the time stamp of the incoming event;
- the event message;
- the current (environmental) agent state (e.g. velocity and perception mode for a car driver in the traffic scenario);
- the associated action tree with the individual selection probability function;

At any point of time, the actual event board is completed by

- a classifier (i.e. a value reflecting the similarity of different sequences independent from the chronological order or length), which will be calculated from a subsequence of the event board (starting from the recent event entry) and allows comparisons of subsequences in the norm formation process later on.

Event board sequences describe consecutive fragments of agent behaviour, thus introducing a higher level of complexity. Presumably within this complexity level regularities and norms are residing.

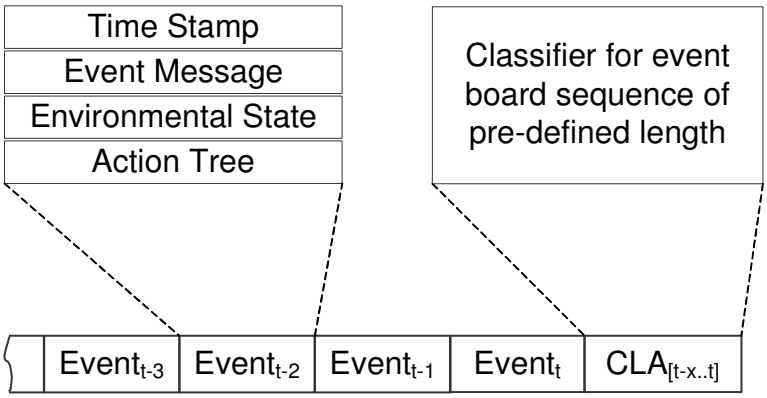

Figure 2 Event board

Thus, a subsequence of the event board is transformed into a rule for the normative frame when a norm invocation event (i.e. modal V, S, D) is detected and a similar entry (the level of similarity is defined by the classifier) is not present in the normative frame yet. In the case that such an element is already present, the event is added to the valuation history of the existing element and a learning process on this element is triggered. Consequently, an entry of the normative board contains the following elements (Figure 3):

- the associated classifier of the event board subsequence;
- the events from the corresponding event board subsequence
- a generated rule (a merged action tree);
- a rule state, specified by a category (regularity, normative belief, norm) and a type (permission, obligation, prohibition);
- a valuation history, holding a statistical report of the valuations received on the respective rule.

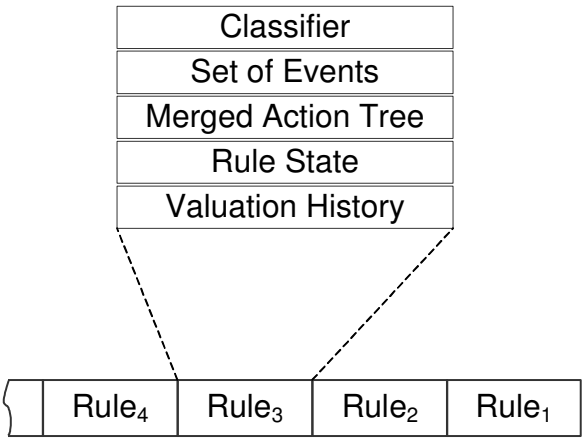

Figure 3 Normative frame

With the components of a normative frame introduced so far, the process of norm formation can be defined in detail.

3.2 Norm Formation in EMIL-S

As mentioned in Deliverable 3.2 the norm formation process is formed by four modules:

The norm recogniser: This component of an agent receives messages or makes observations of different types and can interpret them.

The norm adoption engine: This component of an agent is responsible for generating goals from the contents of the normative frame.

The decision maker: This component generates the normative intention from the goal or goals provided by the norm adoption engine.

The normative action planner: This component has to work out the concrete actions that have to be taken in order to fulfil the intention produced by the decision maker.

These modules represent the core functionality of the norm formation process and serve insofar as a basic frame for the agent architecture. Nevertheless, the dependencies between the modules are fairly complex and the boundaries are not always that clear on the operational level. Even more, there will be no explicit appearance of “goals” and “normative intentions” in the following explanation, but these concepts are “hidden” within the applied specification of agent dynamics as introduced in the previous section.

Figure 4 gives an overview of the complete norm formation process as realized. This course outline, as well as the following formal specification of the process details are presented as UML [2, 6] activity diagrams. According to the standards for proper specification of software systems, these diagrams are attached by textual descriptions (and UML class diagrams as well as pseudo-code fragments where indicated).

Despite the slightly different semantic compared to the EMIL-A process modules, names (and order) for the [main] activities are kept. In contrast, the two modules “Norm Adoption” and “Decision Maker” are integrated into one activity, and a pre-processing activity is added.

Furthermore, the diagram contains two state variables. The variable **Mode** is a flag that distinguishes between environmental and norm invocation events. The other state variable **Role** determines the current role of the agent. Each agent can behave as an actor on the one hand, and as an observer on the other hand. Between these two roles, there are two main differences regarding the event processing:

- An actor receives event messages with modal A from the perception as well as messages with modals D, V or S from other agents. An actor can send messages with modals B to the environment and messages with modals D, V or S to other agents.

- An observer receives the same messages as an actor and additionally messages regarding the actions of the observed agent (with modal B). In contrast, an observer is not able to execute environmental actions.

Each of the activities are detailed in the following subsections. Furthermore, the data structures referred to throughout the entire paper are formally specified.

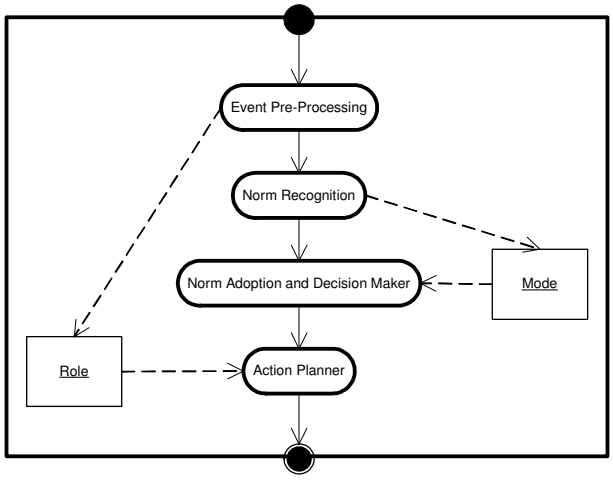

Figure 4 Architecture of the norm formation process.

3.2.1 Event Pre-processing

Starting the norm formation process, each incoming event first passes a **pre-processing module** where the addressee of the event is checked and the agent role (i.e. observer or actor) regarding this event is determined (Figure 5).

Not only the role but also information on the addressee has a major impact on the following process stages: The processing of events directed to the agent (in the “actor” role) on the one hand, and the noticed events to or from each of the observed agents on the other hand must be treated as different [“mental”] sub-processes. This implies that for each message addressee a distinct view on the event board (in combination with different classifiers) must be provided.

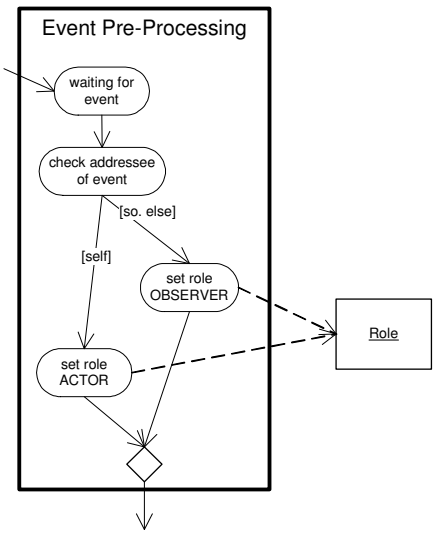

Figure 5 UML activity diagram for event pre-processing

3.2.2 Norm Recognition

Generally, the task of the **norm recognition module** (Figure 6) is to find out whether there exist rules in the normative frame which describe similar environmental situations as sequences of events in the recent perception history. Depending on the type of the received event (i.e. environmental or norm-invocation) there are two different operation modes which influence the conducted activities during further processing.

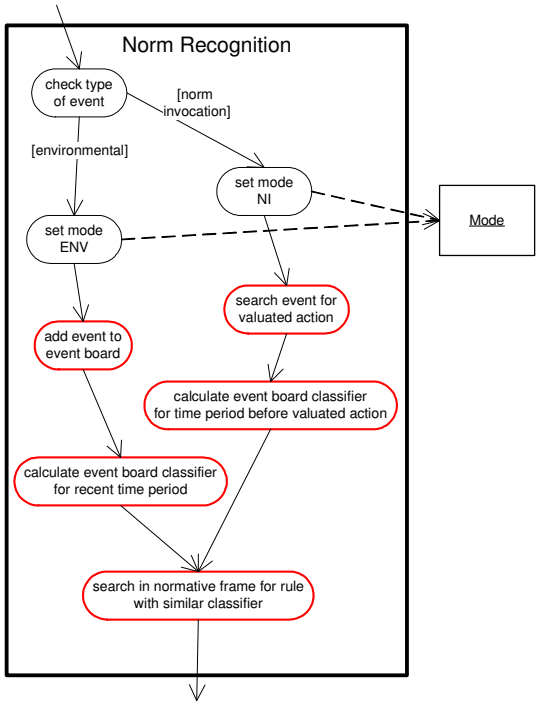

Figure 6 UML activity diagram for the norm recognition process.
Activities with red marking are detailed in chapter 3.3.

In case of an environmental (ENV) mode, a new entry for the event is generated and added to the event board (for details see section 3.3.1). Otherwise, the event board is searched for an entry related to an action that is valuated by the currently received norm-invocation (NI) event (see section 3.3.2). If no entry is found (e.g. because the valuated action is outdated and thus already removed from the event board), the process terminates.

In both cases the classifier (CLA) is then calculated (see section 3.3.3) for a time period (of an arbitrary length that has to be determined for each particular scenario) before the corresponding event board entry. This classifier is then used to search in the normative frame for entries with similar classifiers, i.e. rules that were generated in similar environmental situations (see section 3.3.4). The aim of finding similar entries is

- in case of an environmental mode, to recognise typical and already known situations within the environment at an early stage and execute adequate actions (e.g. to avoid undesired incidents), and
- in case of a norm-invocation mode, to allow conditioning of rules for more specific environmental situations.

This information about the found normative frame entries is passed to the norm adoption component.

3.2.3 Norm Adoption and Decision Making

Subject to the result of the search for entries with similar classifier in the normative board, the **norm adoption** process starts with different actions (Figure 7). If no matching or similar rule is found and the norm invocation mode is active, a rule is generated from the event board sequence that is covered (regarding the time period) by the just calculated classifier, stored into the normative frame and added to the set of found rules. The rule generation is an important sub-process that involves the merging of the action trees for all events occurring in the sequence (see section 3.3.5). In case of an environmental mode, the corresponding initial event-action tree is retrieved and passed to the action planner as result of the norm adoption and decision making process. On the other hand, if similar rules are found in the normative frame, one of them is chosen and passed to a decision making stage. In case of not obeying the rule the process is terminated after all possible rules were declined. Otherwise, the invocation history for the obeyed rule is modified and an evaluation of the norm invocation history is started with the goal to adjust the state of the rule and to trigger normative learning (see section 3.3.6). There are two components constituting the rule state:

- the category of the rule: regularity, normative belief, norm;
- the type of the rule: permission, obligation, prohibition.

Finally the action tree is extracted from the rule and passed to the action planner.

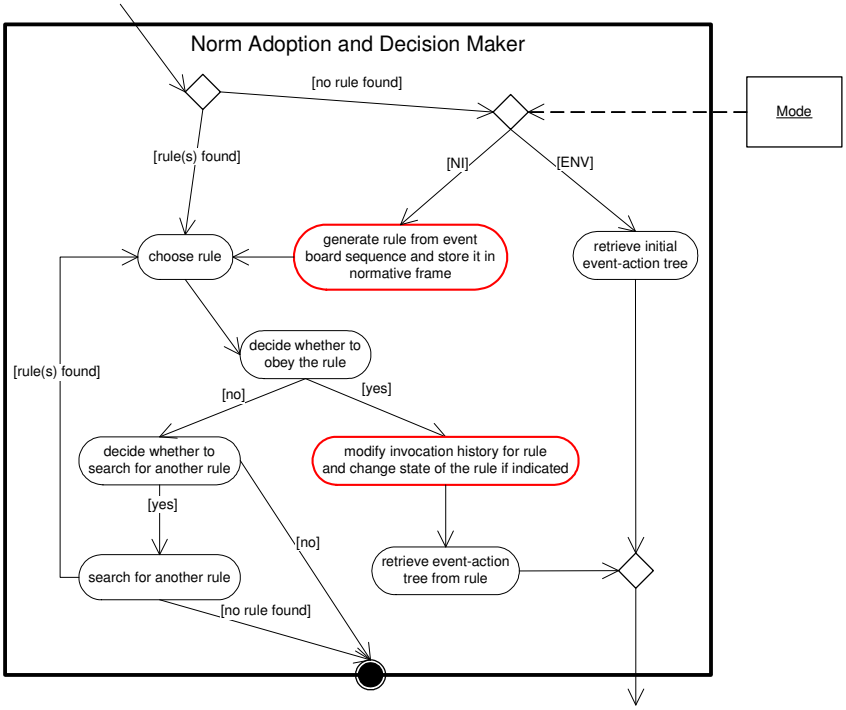

Figure 7 UML activity diagram for the norm adoption process. Activities with red marking are detailed in chapter 3.3.

3.2.4 Action Planning

Finally, the **action planning module** (Figure 8) selects the related actions according to the agent role and the probabilities attached to the submitted action tree. For an actor, corresponding environmental and norm-invocation actions are executed as well as the selection probabilities for the applied rule are modified. If the agent is an observer, no environmental actions are allowed.

The “choose actions” activity conceals a sub-process that is crucial for the entire agent operation. Details can be found in section 3.3.7.

The actions that are chosen for execution can belong to one of three different types:

- environmental actions, influencing directly the environmental state of the agent;
- communication acts (as one subset of the norm-invocation actions), valuating an action (or state) of another agent;
- (reinforcement) learning actions (as another subset of norm-innovation actions), adjusting the selection probabilities of the agent itself (these can insofar also called as “internal actions”). The learning process is designed in such a way that an action retrieves a higher probability when either often executed or positively valuated by other agents. Rarely executed actions or negative valuations have, of course, the contrary effect on the selection probabilities.

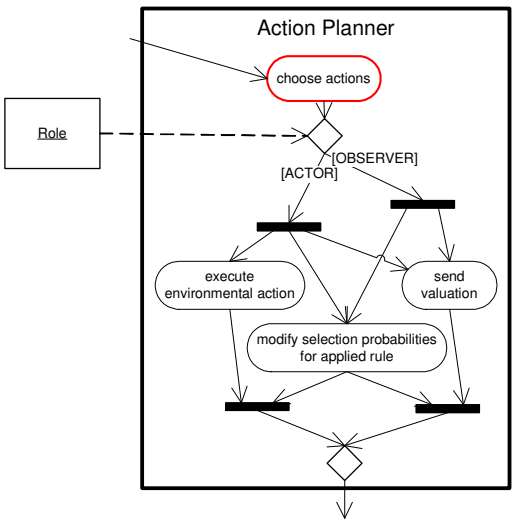

Figure 8 UML activity diagram for the decision making process.
Activities with red marking are detailed in chapter 3.3.

3.3 Activities of the Norm Formation Process and Associated Data Structures

The following subsections describe some of the activities introduced in section 3.2 in detail. The activities are represented by methods; pseudo-code is used for specification of methods and algorithms. Since for development and treatment of any kind of algorithm it is essential to have knowledge about type and structure of the data to be processed, a number of class diagrams constitute the introduction of this section.

Figure 9 introduces three enumerations specifying the relevant message modals and categories as well as types of rules.

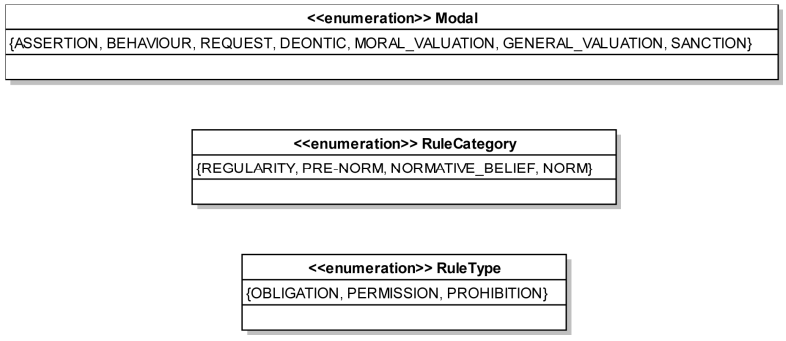

Figure 9 Important enumerations

Figure 10 shows the realization of the abstract EMIL-A message within the EMIL-S system. For the message content a separate class is defined, introducing a time stamp to a message. This content part of a message plays an important role within the implementation: both events and (external) actions are represented by `Content` objects.

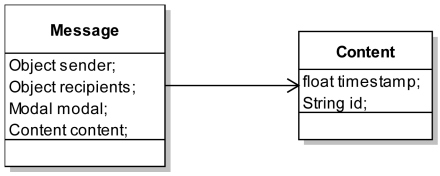

Figure 10 EMIL-A message and content classes

Accordingly, these classes provide the basis for the specification of the process-related data structures (Figure 11). The class diagram shows the four memories each agent is able to access:

- The `EventBoard` is one of the two agent-individual memories. It has associations to the following classes:
  - `EventEntry`, an entry of the event board.
  - `Classifier`, allows for comparison of two different sequences of event entries.
  - `EventMarking`, is used by `Classifier` to collect the considered events in combination with a set of all environmental states (represented by the class `StateField`) the event occurred together with.
  - `State`, the environmental agent state.
  - `StateField`, is used by `EventMarking` and is parent to `State`.
  - `ActionTree`, the decision tree composed of `ActionGroups`.
  - `ActionGroup`, represents a set of mutually exclusive actions.
  - `AbstractAction`, parent of `InternalAction` and `ExternalAction`.
  - `InternalAction`, specifies an action affecting the agent on EMIL-S level and is used mainly for agent learning.
  - `ExternalAction`, specifies an environmental action.

- The `NormativeFrame` is the second agent-individual memory and is associated with:
  - `NormativeFrameEntry`, an entry of the `NormativeFrame`.
  - `Rule`, a complex behavioural element on base of several merged action trees and the associated events.
  - `RuleState`, the state reflecting the “importance” of a rule.
  - `ValuationInfo`, an entry of the valuation history.

- The `NormativeBoard` is the “global version” of the normative frame and serves as a kind of “statute book” in which all norms are stored.

- For each agent type the `InitialRuleBase` holds the appropriate set of initial `EventActionTrees`.

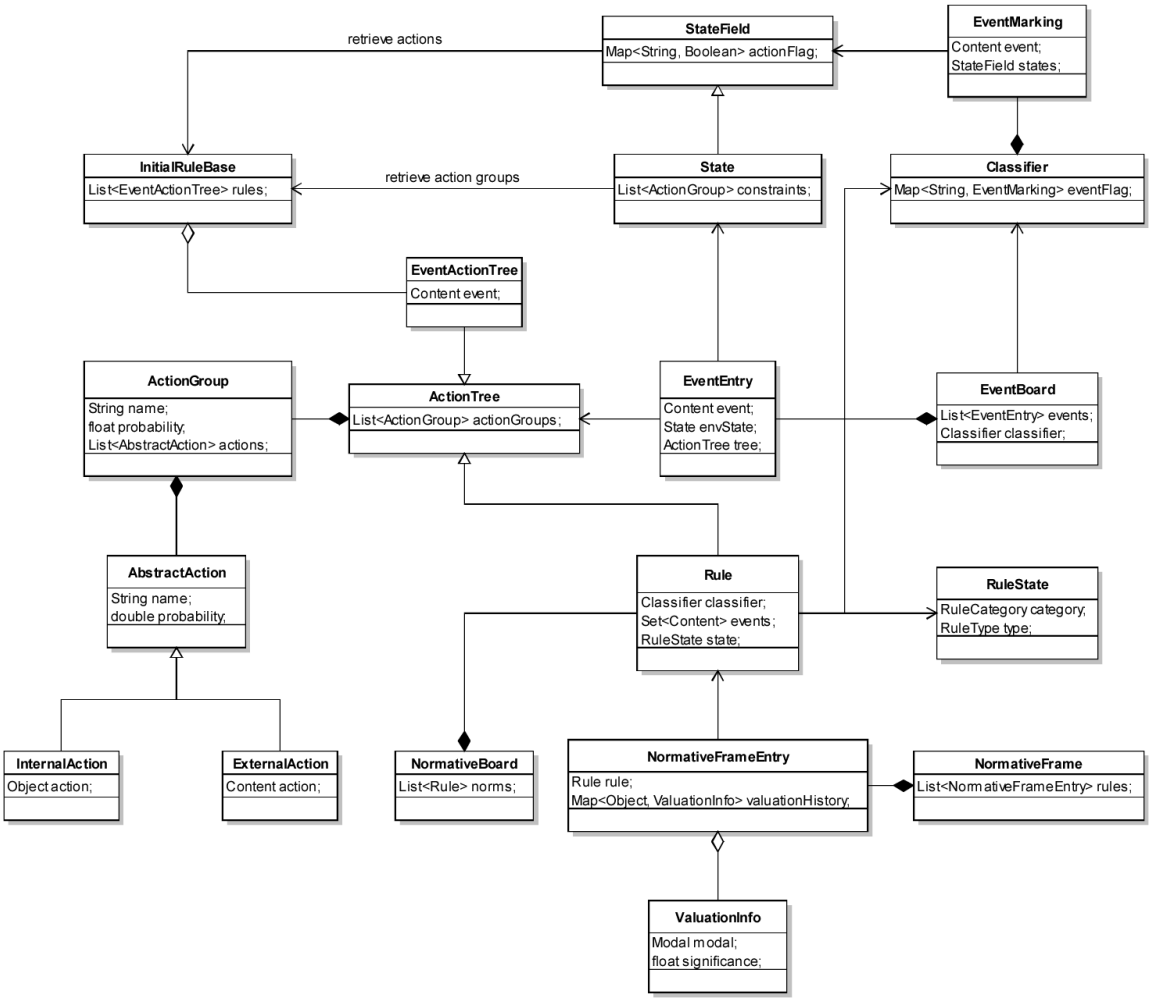

Figure 11 Data oriented class diagram of EMIL-S

3.3.1 "Add event to event board"

The activity "add event to event board" is detailed in the pseudo-code of the method `addEventToEventBoard`. It simply generates an event board entry from the information transmitted by the `message` parameter.

```
void addEventToEventBoard(Message message) {

    create new EventEntry{
        time = current simulation time;
        event = message.content;
        state = current environmental state;
        tree = empty; // will be set later after action planning
    }
    add created entry to event board;

}
```

3.3.2 “Search event for valuated action”

The activity “search event for valuated action” is detailed in the pseudo-code of the method `searchEventForValuatedAction`. It uses the information from the `message` parameter to determine the search criterion: either the time of the valuated action or the type of the action. The method returns an entry from the event board.

```
EventEntry searchEventForValuatedAction(Message message) {

    case Message.content is

        when contains time when valuated action happened =>
            return first event board entry older than time;

        when contains information about action =>
            repeat
                get next event board entry
                (beginning from the newest not inspected entry);
            until entry is found where valuated action was conducted;

            return found event board entry;

}
```

3.3.3 “Calculate event board classifier”

The activities “calculate event board classifier for recent time period” and “calculate event board classifier for time period before valuated action” both use a procedure that is detailed in the pseudo-code of the method `calculateEventBoardClassifier`. Only the event board entry which determines the newest event of the classifier scope differs within the two activities. Thus, the event board entry is method parameter; the result is a classifier object.

```
Classifier calculateEventBoardClassifier(EventEntry startEntry) {

    create new Classifier{
        eventFlag = empty;
    }

    while considered eventBoardEntry is temporally within
            classifier scope
        loop
            get next (older) eventBoardEntry (beginning
            with startEntry);

            if not exist flag with the key event.id in classifier
                then
                    create new EventMarking{
                        event = eventBoardEntry.event;
                        states = empty;
                    }
                    add marking to classifier with key event.id;
                else
                    get marking with key event.id from classifier;
            end if;

            get state from eventBoardEntry and make disjunction with
            field marking.states;
            store the disjunction as new value of marking.states;

        end loop;

    return classifier;

}
```

3.3.4 "Search in normative frame for rule with similar classifier"

The activity "search in normative frame for rule with similar classifier" is detailed in the pseudo-code of the method `searchRuleInNormativeFrame`. It uses the parameter `classifier` as search criterion and returns a list of matching normative frame entries (which contain the requested rules). The method consists of two parts:

- It firstly checks whether there exist a matching norm in the (global) normative board. If there exists an appropriate norm, then the norm is copied into the (individual) normative frame of the agent.

- Afterwards the normative frame is scanned for matching entries. These entries are collected in a list and returned as method result.

The threshold for decision on the similarity can be a global as well as an individual model parameter.

```
List<NormativeFrameEntry> searchRuleInNormativeFrame
    (Classifier classifier) {

    if normative board shall be considered
        then
            repeat
                get next rule from normative board;
                compare(classifier, rule.classifier);
                if percentage of congruence is above threshold
                    then
                        store norm as normative belief in normative frame;
                end if;
            until no more rules exist in normative board;
    end if;

    repeat
        get next entry from normative frame;
        compare(classifier, entry.rule.classifier);
        if percentage of congruence is above threshold
            then
                add entry to resultList;
        end if;
    until no more rules exist in normative frame;

    return resultList;

}
```

Within the pseudo-code another method `compare` is applied which calculates the similarity of two classifiers. An exemplary implementation of such a `compare` algorithm is shown in the following pseudo-code fragment, but other (more efficient) algorithms are also applicable (in particular hashing functions).

```
float compare(Classifier classifier1, Classifier classifier2) {

    numberOfEvents = 0.0;
    numberOfSharedEvents = 0.0;

    foreach in InitialRuleBase specified event id
        loop
            if marking with id exist in classifier1 and/or classifier2
                then
                    numberOfEvents = numberOfEvents + 1;
                    if marking with id exist in both classifiers
                        then
                            compare states from both markings with a
                            similar (to this compare method) procedure
                            with a result value of range [0..1];
                            numberOfSharedEvents =
                                numberOfSharedEvents + result;
                    end if;
            end if;
        end loop;

    if numberOfEvents > 0
        then
            return numberOfSharedEvents / numberOfEvents;
        else
            return 0;
    end if;

}
```

3.3.5 "Generate rule from event board sequence and store it in normative frame"

The activity "generate rule from event board sequence and store it in normative frame" is detailed in the pseudo-code of the method `generateRule`. The originating event board sequence is specified by the two parameters

- `seqClassifier` which is the classifier for the considered event board sequence;
- `startEntry` which is the newest entry of the considered event board sequence.

The method consists of two parts:

- Firstly, a rule is generated and initialized with the event set and the merged action tree.
- Secondly, a normative frame entry is generated which attaches a valuation history to the rule. This entry is also returned by the method.

```
NormativeFrameEntry generateRule
    (Classifier seqClassifier,  EventEntry startEntry) {

    create new Rule{
        classifier = seqClassifier;
        state = RuleCategory.REGULARITY;
        events = empty;
        actionGroups = empty;
    }

    // initialize events and actionGroups of created rule
    while considered eventBoardEntry is temporally within
            classifier scope
        loop
            get next (older) eventBoardEntry (beginning
            with startEntry);

            add eventBoardEntry.event to rule.events;

            foreach actionGroup in eventBoardEntry.tree
                loop
                    add actionGroup to rule.actionGroups
                end loop;
        end loop;

    create new NormativeFrameEntry{
        rule = rule just created;
        valuationHistory = empty;
    }
    add created entry to normative frame;
    return entry;

}
```

3.3.6 "Modify invocation history for rule and change state of the rule if indicated"

The activity "modify invocation history for rule and change state of the rule if indicated" is detailed in the pseudo-code of the method `revaluateRuleState`. The two method parameters are:

- `entry`, the normative frame entry which is to be updated;
- `message`, the norm invocation message which was responsible for triggering the current activity. From the message the information about modal and sender are used.

The method has a tripartite structure:

- Extending the valuation history by adding a new entry for the current norm invocation message.
- Updating the rule state on basis of the valuation history.

- Normative learning: when the rule state changes to “normative belief”, the rule is published in the normative board where all other agents can access the norm and transfer it to the individual normative frames. By spreading such rules, another type of learning is introduced (beyond reinforcement strategies).

```
void revaluateRuleState(NormativeFrameEntry entry, Message message) {

    create new ValuationInfo{
        modal = message.modal;
        significance = retrieve significance from authority board
            (in future versions);
    }
    add created info to entry.valuationHistory with key message.sender;

    calculateRuleState(entry.valuationHistory);
    store new state to entry.rule.state;

    if state.category equals RuleCategory.NORMATIVE_BELIEF
        then
            add entry.rule to normative board;
    end if;

}
```

An important part of the procedure is the calculation of the rule state, hidden within the method `calculateRuleState`. An exemplary implementation of such an algorithm is shown in the following pseudo-code fragment.

```
RuleState calculateRuleState(Map<Object, ValuationInfo> valuationHistory){

    create new RuleState{
        category = RuleCategory.REGULARITY;
        type = none;
    }

    if valuationHistory contains entries from more than e.g. 20% of agent
    population or with a significance above a threshold
        then
            state.category = RuleCategory.PRE-NORM;
        else
            if valuationHistory contains entries from more than e.g. 90%
                of agent population or with maximum significance
                then
                    state.category = RuleCategory.NORMATIVE_BELIEF;
            end if;
    end if;

    set state.type according to modals of valuation infos in
    valuationHistory using a similar procedure as for determining the
    rule category;

    return state;

}
```

3.3.7 "Choose Actions"

The activity "choose actions" is detailed in the pseudo-code of the method `chooseActions`. It uses an action `tree` as method parameter and calculates for each action group a "score" for all of the attached actions. This score represents the combined selection probabilities for actions occurring more than one time within the tree. Due to the merging of action trees during the rule generation it might happen that contradictory actions (i.e. actions from one and the same action group) get an equal probability for firing. Thus, it must be granted that for every processing step maximal one action from each action groups is executed.

The method returns a list of all actions that are allowed for executing. Which actions eventually are considered is decided afterwards, determined by the current agent role.

```
List<AbstractAction> chooseActions(ActionTree tree){

    create actionScore, a new Map<AbstractAction, Float>;

    foreach actionGroup in tree
        loop
            evaluate selection probability for actionGroup;
            if actionGroup shall be considered
                then
                    foreach action in actionGroup
                        loop
                            add action to actionScore (if not already
                            present) and modify score for action by
                            combining the action's selection probability
                            with the current score;
                        end loop;
            end if;
        end loop;

    foreach in InitialRuleBase specified actionGroup
        loop
            get actions from actionGroup;
            if actionScore contains retrieved actions
                then
                    choose action with highest score and add it to the
                    resultList;
            end if;
        end loop;

    return resultList;

}
```

4 User interface for describing EMIL-S agents

As pointed out in deliverable 3.2., the proposed simulator should allow the definition of different scenarios in which norm formation can be observed and the definition should be

done in a way, that a modeler can endow agents with all the necessary features they need in a certain scenario. Therefore the agent model of EMIL-S must be completed by a user interface (wizard) for the specification of agents (see also the Agent Model Designer in section 2). Basically, the functionality of this wizard follows the event-action tree concept (rules), which was introduced in the section before to describe agent dynamics in norm formation processes.

The main steps of an agent type definition can be distinguished conceptually as follows:

- Definition of **actions**
  Definition of single actions and categorize them to action groups (to specify mutual exclusive actions)
  - **Environmental actions:**The content of each action is sending a message of mode B (Behavior)
    (e.g. Traffic scenario: Driver action group accelerate/slow down/stop)
  - **Norm invocation action:** The content of each action is sending a message of mode V (Valuation), D (Deontic) or S (Sanction)
    (e.g. Traffic scenario: (Send) positive/negative notification)

- Definition of **events**
  - **environmental events**
    The content of each event is the receiving a message of mode (Assertion)
    (e.g. Traffic scenario : Crossroad ahead, pedestrian on road)
  - **norm invocation events**
    The content of each event is the receiving a message of mode V (Valuation), D (Deontic) or S (Sanction)
    (e.g. Traffic scenario : (Receive) positive/negative notification)

- Definition of **dependencies between events and actions**
  - Initial selection probabilities between events and action groups
  - Initial selection probabilities between action groups and corresponding actions (summed up to 100%)

  Input could be done via tables or graphical representation of the trees and should be supported by an import function (combo/selection box, auto completion, …)

- Previously defined event-action trees can be assigned to two different **agent roles**:
  - **Actor**
    Selection of environmental event-action trees
    (e.g. Traffic scenario: Pedestrian on road → accelerate/slow down/stop)
    Selection of norm invocation event-action trees
    (e.g. Traffic scenario: Receiving of sanction → Adjust probability distribution)
  - **Observer**
    Selection of norm invocation event-action trees
    (e.g. Traffic scenario: Receiving of sanction → Adjust probability distribution)

- Definition of **"global" model parameters**

(e.g. “norm adoption willingness”, see in the activity “check where to obey the norm” in the norm adoption activity diagram)

Regarding the internal representation of the user input, XML (Extensible Markup Language)-based data structures seems to be the most appropriate solution [3]:

- XML as an representation language for hierarchically structured data seems to be the most intuitive choice for the event-action tree approach in EMIL-S
- The description of XML-documents takes place by scheme languages (e.g. DTD, XML-Schemes)
- XML-Parser-API’s are already available (e.g. DOM, StAX, XML Beans)
- XML-schemes are widely-used as common data exchange format

Therefore, the user input (e.g. event-action trees, parameters, agent roles) can be done mainly based on a XML-based editor at first. But applying the already available XML-tools, it is also possible to develop a more sophisticated user interface even in a very early state of the system development as well.

5 Interface to experiment software

5.1 MEME Overview

Basically, MEME is a universal experimentation environment with powerful parameter sweep functionality. The design of MEME is strongly oriented on the REPAST simulation framework; the programming language used for implementation is JAVA.

MEME is composed of three main components:

- Module for analyzing, presenting and graphically visualizing data from simulation run log files (presented in REPAST CSV format);
- Parameter sweep wizard, dedicated for analyzing the simulation model and supporting the user to specify an adequate parameter space for multiple simulation runs;
- Batch controller for (distributed) simulation runs, allows sweeping automatically through the specified parameter space.

One of the most important advantages of MEME is the functionality of the parameter sweep wizard to conduct multiple simulation runs parallel on PC clusters or grids without the need to modify the simulation tool or even the simulation model. This is achieved by providing a distributed batch controller, relying on a ProActive-based client-server architecture.

A possible drawback (at least from the EMIL-S point of view) results in the REPAST-oriented design of MEME. This becomes problematic mainly because of the following issues:

- Parameter and recordable retrieval: MEME scans directly REPAST classes for members or getter/setter methods for possible model parameters and variables ("recordables").

- Usage of the REPAST-internal batch controller for simulation execution on single computers.

In the near future AITIA plans to modify MEME in order to make it independent from REPAST and to provide its functionality for other simulation tools like NETLOGO and (in particular) TRASS.

It is planned to use MEME as experimentation tool for EMIL-S. While model verification will be done within a native EMIL-S runtime environment, the role of MEME mainly lies in model validation and parameter sensitivity analysis.

In a meeting of EMIL project partners UNI-KO and AITIA, held in Budapest during August 2008, a generic interface and a set of related data types were defined. This can be seen as a first step to couple the EMIL-S simulator to the MEME experimentation environment. The following sections summarise the results of the meeting: Firstly, the functionality of the MEME parameter sweep wizard is described. Based on this, the requirements for the interface are deduced afterwards.

5.2 MEME Parameter Sweep Wizard

The process of using the MEME parameter sweep wizard includes the following steps:

1. After loading a simulation model and entering various general settings the wizard retrieves a list of parameters from the simulation model and displays all recognized parameters to the user.
   For each parameter the user must enter information about the parameter space. Three types of variables are supported: constant, list of values, range. Afterwards the user can define dependencies between the single parameters. This means that the user can arrange the parameters hierarchically in a tree, resulting in convoluted iterations of simulation runs. For example (Table 1), if parameters x and y are each associated with a list of values [10, 20, 30], and y is "child" of x, than nine simulation runs with the parameter value distributions shown in Table 1 are executed. For multiple parameters on the same tree level, the minimum number of elements in the parameter lists or of steps in parameter ranges determines the overall number of parameter combinations applied for simulation runs.

2. In the following step the wizard retrieves a list of "recordables" from the simulation model. A recordable is a model attribute that represents a part of the simulation state. It is represented either by a class member/attribute, a getter function or a script. Scripts are provided by the user and accessing only other recordables for calculations. The user can specify which recordables are

included for model exploration.
The user has to define and parameterize one or more recorders, to which one or more recordables can be attached. A recorder is a small code fragment/module/program, situated within the agent code/simulation tool which collects the values of the attached recordables at a specified frequency and writes the collected data to a CSV file at predefined points of time.
Furthermore, stop conditions (e.g. number of rounds, fixed termination of simulation, recordable threshold) must be provided by a user.

3. After all necessary information is complete, the wizard starts with simulation executions until all defined parameter combinations were processed. This is done by cycling through the following steps:

    a. Instantiating the simulation platform/tool (if applicable).

    b. Initializing the simulation model with a new configuration from the defined parameter space.

    c. Activating and initializing the defined recorders within the simulation models.

    d. Running the simulation.

    e. Collecting the simulation results from CSV files and writing the data in a (MEME-internal) database.

Run	Value of parameter x	Value of parameter y
1	10	10
2	10	20
3	10	30
4	20	10
5	20	20
6	20	30
7	30	10
8	30	20
9	30	30

Table 1 Parameter distribution

5.3 Interface Design

As indicated in the previous section, the interface has two different functions:

- It provides information about the simulation model to MEME by supplying lists of all simulation parameters and of all relevant recordables.

- It allows MEME to configure, initialize and supervise simulation runs.

For this, two JAVA interfaces were defined, which must be implemented by the simulation tool according to the processes sketched above. MEME communicates with the simulation tool exclusively by invoking the interface methods.

Figure 12 shows `IModelInformation`, the JAVA interface for information retrieval. It contains only the two methods `getParameters()` and `getRecordables()`. The first method returns a list of parameter descriptions, the second a list of recordable descriptions.

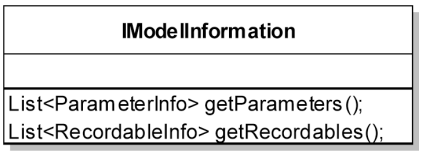

Figure 12 Interface IModelInformation

A single parameter is specified by the JAVA class `ParameterInfo` and the subclasses for the three available parameter types (Figure 13). The most important attributes are:

- `name` of the parameter.
- `type` of the parameter; the value can be one of the classes `Boolean`, `Number` or `String`.
- An informative `description` of the parameter.
- A `defaultValue` for the parameter.
- A definitionType; the enumeration embraces the elements `CONSTANT`, `LIST` and `RANGE` and contains also the respective parameter values. Depending on the `definitionType` the value or domain of the parameter must be defined:
    - a single `value` for a constant parameter
    - a collection of `values` for parameter list;
    - `startValue`, `endValue` and an `incrementValue` for a parameter range.

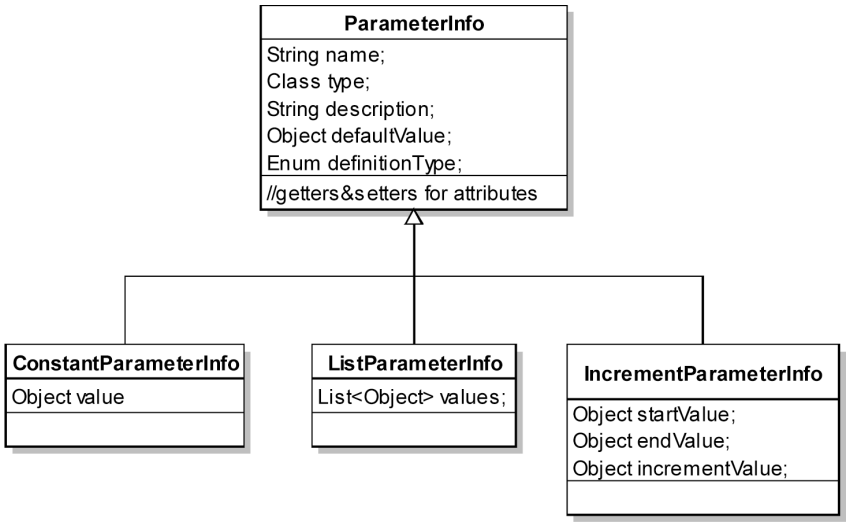

Figure 13 Class ParameterInfo

A single recordable is specified - similar to the parameter specification - by the JAVA class `RecordableInfo` (Figure 14). The most important attributes are:

- `name` and `type` are equal to the corresponding attributes of `ParameterInfo`.
- `accessibleName` is an expression under which the recordable can be accessed in a script.

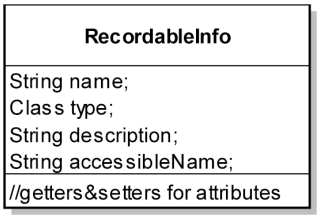

Figure 14 Class RecordableInfo

The second JAVA interface `IBatchController` for initializing and controlling simulation runs shows Figure 15. The methods have the following functions:

- `setParameters()`, `setRecorders()`: Initializing the batch controller.
- `setStoppingCondition()`, `startBatch()`, `stopBatch()`, `stopCurrentrun()`: Administration of simulation runs.
- `getMaxRun()` retrieves the cardinality of the set of all possible parameter distributions defined by the parameter tree. This number equals the maximum number of simulation runs and has to be provided to MEME for monitoring purposes.
- `addBatchListener()`, `removeBatchListener()`: Adding and removing listener objects for monitoring of simulation runs.

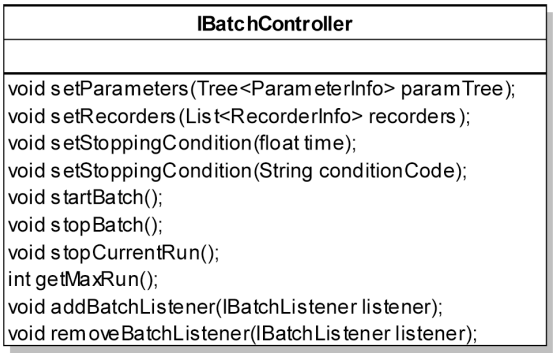

Figure 15 Interface IBatchController

While the elements of `paramTree` are objects of the above defined class `ParameterInfo` (and subclasses, respectively), another JAVA class `RecorderInfo` for the specification of recorders is necessary (Figure 16). The attributes are:

- `name` of the recorder.
- `outputFile` specifies the CSV file in which the recordable data is written during simulation runs.
- `delimiter` defines the character which is used to separate the columns in the CSV file.
- `recorderType` determines when (how often) the values of the recordables are captured by a specified Boolean condition (e.g. when the value of a recordable exceeds a threshold). The possible modes are defined in the `RecordOptions` enumeration:
    - at every (timer) event;
    - at the end of a simulation run;
    - at a specified condition (e.g. when the value of a recordable exceeds a threshold).
- `writeType` determines when (how often) the recorded values are written to the CSV file. The possible modes are defined in the `RecordOptions`.
- List of `recordables`.

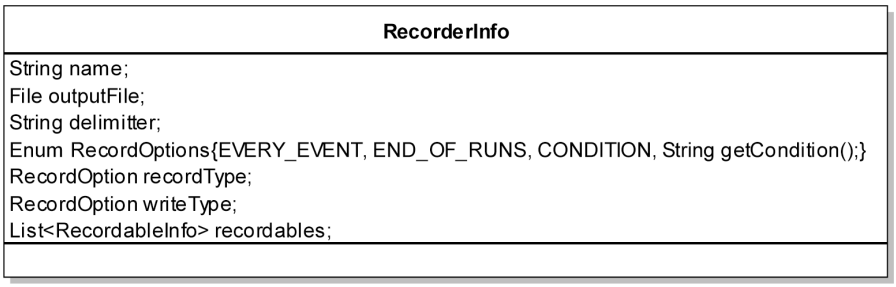

Figure 16 Class RecorderInfo

The simulation system can send notifications to MEME using a call-back functionality. For this purpose an interface `IBatchListener` defines methods that can be implemented by MEME and are triggered by the batch controller of the simulation tool (Figure 17).

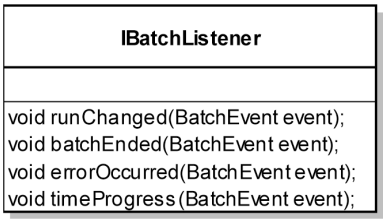

Figure 17 Interface IBatchListener

6 Conclusion

This document formally describes EMIL-S and its environment. A prototypical version of EMIL-S can now be run, but some changes might still appear necessary when the first applications have been tested.

References

1. AITIA: Model Exploration Module (MEME), http://www.aitia.hu/meme

2. Booch, J., Jacobson, I, Rumbaugh, J.: The Unified Modelling Language User Guide. Boston: Addison-Wesley (1999)

3. Harold, R.E., Means, W.S.: XML in a Nutshell. Sebastopol,CA: O'Reilly (2004)

4. Lotzmann, U.: TRASS - A Multi-Purpose Agent-based Simulation Framework for Complex Traffic Simulation Applications. Accepted for: Ana L. C. Bazzan and Franziska Klügl (eds.): Multi-Agent Systems for Traffic and Transportation. IGI Global (2008)

5. Lotzmann U., Möhring, M.: A TRASS-based agent model for traffic simulation. In: Loucas S. Louca, Yiorgos Chrysanthou, Zuzana Oplatková, Khalid Al-Begain (eds.):

Proceedings of the 22nd European Conference on Modelling and Simulation ECMS 2008. Nicosia, Cyprus (2008) 97-103

6. Miles, R., Hamilton, K.: Learning UML 2.0. Sebastopol, CA: O'Reilly (2006)

7. Netlogo, http://ccl.northwestern.edu/netlogo/

8. Recursive Porous Agent Simulation Toolkit (Repast), http://repast.sourceforge.net