Bringing Autonomic Services to Live

Deliverable D2.2 Pervasive Supervision Component Set

Detailed Component Specification

Status and Version:	Final Version	
Date of issue:	05/07/2007	
Distribution:	Public	
Author(s):	Name	Partner
	Peter H. Deussen	FOKUS
	Matthias Baumgarten	UU
	Kieran Greer	UU
	Rosario Alfano	TI
	Giordano Tamburrelli	DEI
	Luciano Baresi	DEI
Checked by:	Peter H. Deussen	FOKUS
	Antonio Manzalini	TI
	Luciano Baresi	DEI
	Mathias Baumgarten	UU
	Kevin Curran	UU

Abstract

This document presents the detailed specification of the components that define the Cascadas supervision infrastructure. This document includes two appendixes:

– Appendix 1: a proof-of-concept implementation of the supervision infrastructure, together with a short document that describes how the proof of concept works.

– Appendix 2: a detailed component specification of the DriftAnalyser, as well as a proof of concept implementation thereof.

The purpose of the document is (a) to prepare the upcoming integration of the concepts developed so far in WP2 into the ACE infrastructure prototype delivered by WP1, (b) to identify open issues and to provide an implementation roadmap, and (c) to describe initial experiments and proof-of-concept implementations that will be re-used for the development of an ACE based version of the supervision system.

As the ACE infrastructure will only be available from the beginning of month 18 of the project (and thus essential aspects like the interaction of the supervision system with the system under supervision, i.e. monitoring and actuation, were not definable yet), we are not delivering a complete prototype, but concentrate on specification work and proof-of-concept implementations of single components.

Table of Contents

1 Introduction

1.1 Purpose and Scope

This document presents the detailed specification of the components that define the Cascadas supervision infrastructure. This document includes two appendixes:

– Appendix 1, a proof-of-concept implementation of the supervision infrastructure, together with a short document that describes how the proof of concept works.

– Appendix 2: a detailed component specification of the DriftAnalyser, as well as a proof of concept implementation thereof.

The purpose of the document is (a) to prepare the upcoming integration of the concepts developed so far in WP2 into the ACE infrastructure prototype delivered by WP1, (b) to identify open issues and to provide an implementation roadmap, and (c) to describe initial experiments and proof-of-concept implementations that will be re-used for the development of an ACE based version of the supervision system.

As the ACE infrastructure will only be available from the beginning of month 18 of the project (and thus essential aspects like the interaction of the supervision system with the system under supervision, i.e. monitoring and actuation, were not definable yet), we are not delivering a complete prototype, but concentrate on specification work and proof-of-concept implementations of single components.

1.2 Reference Material

1.2.1 Reference Documents

[1] IBM, "Policy Management for Autonomic Computing – Autonomic Computing Expression Language", http://www.alphaworks.ibm.com/tech/pmac

[2] D. C. Luckham. Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs. *Texts and Monographs in Computer Science*, Oct 1990.

[3] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *Department of Computer Science, Iowa State University, TR 98-06-rev27*, April, 2005.

[4] OMG, "Object Constraint Language", www.omg.org/docs/ptc/03-10-14.pdf

[5] L. Baresi and S. Guinea. Dynamo and Self-Healing BPEL Compositions. In 29th International Conference on Software Engineering (ICSE'07 Companion), pages 69–70. IEEE Computer Society, 2007.

[6] P. H. Deussen, "Towards a Mathematical Framework for Pervasive Supervision", CASCADAS Milestone Deliverable D2.1, accompanied document.

[7] P. H. Deussen, M. Baumgarten, R. Alfano, L. Baresi, M. Plebani, "Report on Pervasive Supervision - State of the Art, Basic Algorithms and Approaches, and Basic Supervision Architecture", CASCADAS Milestone Deliverable D2.1.

1.2.2 Acronyms

ACE	Autonomic Communication Element
DA	DriftAnalyzer
GN/GA	Goal Need/Goal Achievable Protocol
MAPE	Monitor—Analyze—Plan—Execute
MBS	Model Based Supervision
OCL	Object Constraint Language
SUS	System Under Supervision
SVS	Supervision System

1.3 Document History

Version	Date	Authors	Comment
---	2007/04/20	L. Baresi, R. Alfano, M. Baumgarten, P. H. Deussen, A. Mannalla	Initial Inputs
0.1	2007/06/20	L. Baresi, R. Alfano, M. Baumgarten, P. H. Deussen	1st consolidated version
0.2	2007/06/25	L. Baresi	Harmonisation of Inputs
1.0	2007/07/05	L. Baresi, P.H. Deussen	Final version based on comments by M. Baumgarten, P.H. Deussen, A. Manzalini
Final	2008/04/30	C. Moiso	Implementation of comments included in second ESR

1.4 Document Overview

This document comprises a specification of components of a basic supervision system. It provides a first set-up of a supervision architecture for self-management of ACE based service configuration, and it is based on ACEs by itself. We aim on an architecture that:

1. Integrates seamlessly in the architecture provided by WP1, utilizing common functions of ACEs for its particular purposes;
2. Is flexible enough to instantiate and to use only those functions which are really needed to serve a particular supervision task, thus supporting light-weight configurations as well complex analyses and control.
3. Is generic in the sense that supervision tasks are derived automatically – to a certain extend – by utilizing ACE self-models as functional specifications as well as service specific goals.

Let us discuss these objectives in more detail. To define the integration of a supervision architecture into an ACE based service configuration we have to answer two basic questions:

1. How to exploit ACE mechanisms (protocols, interfaces) to (a) define the possible interactions between the configuration under supervision, and the supervision system, i.e. what the supervision system is allowed to monitor, and which actions it is permitted to perform if corrective measures become necessary; (b) obtain run-time information (monitoring); and (c) interact if the interference of supervision functions becomes necessary (i.e. the configuration under supervision is in a (or approaches a) non-desirable state.
2. How to define the supervision system itself (a) either as an integrated part of the ACE architecture, or (b) as a component or configuration of components that is architecturally separated from the ACE configuration under supervision.

Interaction with supervised ACEs. To provide a certain service, an ACE basically executes a workflow generated from its *self-model*. This workflow is called a *plan*. It comprises of states and transitions leading from one state to another. A transition is performed by calling a specific function from the function repository of the ACE (which is assumed to be stateless), or by sending/receiving external messages. Additionally, a session object is maintained that stores data necessary for the service computation. This session object may be modified by specific functions.

The self-model thus defines what is supposed to be the correct or intended function of an ACE. It explains the structure of states, and which actions are available to change from one state into another and how to update state information, Furthermore, a special attribute is has been introduced that is used to assess the desirability of a certain state, e.g. whether it is an initial, intermediate, or end state, an error state, etc. Thus the self-model can be understood not only as a functional specification of ACEs, but also as an expression of computation goals and purposes. Supervision can now be described as the task to make sure that those goals/purposes are fulfilled, by means of the functions that are made available by an ACE.

To answer question 1(a), we envision using the GA/GN protocol developed in WP1 to commit a contract between the ACEs to be supervised and the supervision system. The contract comprises the self-models of these ACEs, together with monitoring and control permissions for actions.

To answer questions 1(b) and 1(c), let us now have a closer look into the architecture of a supervision system: Figure 1 provides a summary of the elements of the supervision system (orange) and their relationship to the constituents of an ACE under supervision (gray).

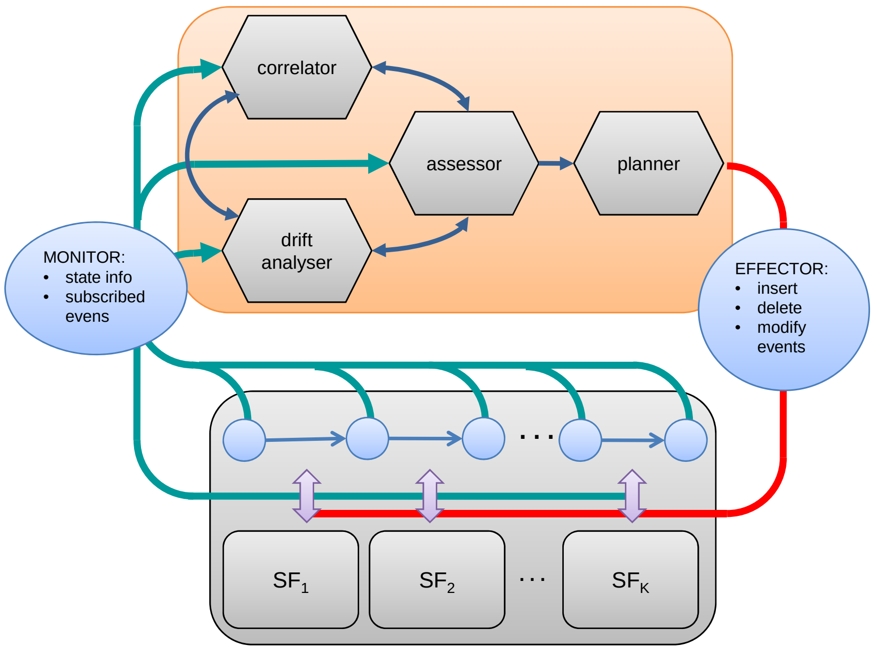


Figure 1. Architecture overview (singular ACE)

Figure 1 shows only one ACE under supervision, but it is possible to put several ACEs under the control of a single supervisor. For monitoring, ACEs provide interfaces (a) to access state and session object, (b) to subscribe on information on specific function invocation or incoming/outgoing messages. Technically, this is solved by hooking up on the ACE internal publish/subscribe communication infrastructure.

We are now going to elaborate briefly on set of components that is available to build supervision systems for ACEs or ACE configurations. All these components will be implemented as ACEs.

– **Monitor** components links the supervision system with the ACE under supervision by utilizing the access mechanisms described above.

– **Correlators** are responsible to aggregate monitored data from distributed sources and to correlate them, in order to extract meaningful indicators of the current health condition of the system under supervision

– **DriftAnalysers** try to anticipate future problem situations in the system under supervision. Additionally, information from the environment may be used to supplement the analytical process.

Thus if Correlators are concerned with the current state of the system, DriftAnalysers are concerned with the possible evolution of the system. Finally,

– **Assessors** make assumptions on the current (or future) system health on the basis of raw data or the output of Correlators and DriftAnalysers, and invoke a Planner if necessary.

Monitors, Correlators, DriftAnalysers, and Assessor thus form an analysis system. The reactive part is provided by the following components:

– On the basis of the assessments generated by the Assessor, the **Planner** tries to compute a course of actions that is intended to resolve the detected problem. Planning is based on the actions described in the self-model of the ACE (or ACEs) under supervision.

- **Effectors** are responsible to execute plans. This is again done by intercepting internal events of the ACE under supervision. Events can be deleted, modified, or inserted.

Integration of Supervision Functions. Let us now answer question 2. We use alternative 2.b for the following reasons.

- Separation of concerns. Integration of supervision functions into the ACE architecture itself would blow up the ACEs unnecessarily. In some cases, supervision functions may not be desired, possible, or necessary.
- Widening of scope. Some supervision functions require the coordinated interaction with a number of ACEs, which would be difficult to realize in the intrinsic scenario.
- Flexibility. In a number of cases, the described analysis/reaction cycle is obviously unnecessarily heavy-weighted. The external approach allows us to formulate the software architecture in a way that it is possible to flexibly use only the components that are needed, while others are not instantiated at all.

Thus we implement each of the components described in the previous paragraphs as a specialized ACE that runs asynchronously to the other components. This allows the flexible definition and setup of arbitrary complex control cycles comprising for instance chains of Correlators and DriftAnalysers, hierarchical Planners, and coordinated Effectors.

Now since these components are ACEs by itself, the GA/GN protocol can be used to discover available supervision components and the contracting mechanism developed in WP1 is available for the set-up of the configuration of the supervision system and the parameterization of its components (e.g. by rules, models, etc.). Therefore, supervision systems become an integral and pervasive part of service configurations, utilizing functions of the underlying platform for discovery and orchestration.

This document is organized as follows: Section 2 describes the high-level architecture and the principle interactions between its components in more detail, providing a global picture that is refined in the following Sections 3.1 - 3.6 in detail for each component. Section 4 exemplifies the architecture by means of a simple case study, illustrating the interaction pattern between the components of the supervision system. The final Section 5 concludes the document by addressing a number of open issues. It contains also a roadmap for the implementation work of WP2 towards self-organized, dynamic supervision pervasions.

2 High-level Architecture

Figure 2 presents the components (ACEs) that define a supervision system. In this case, and for the sake of simplicity, we assume a single instance of each type of components, but real systems might comprise different cooperating instances.

The meaning and the interfaces of each component are explained in detail in the next sections. Here, we only want to describe the high-level interactions among the different elements. These components are a refinement of those already presented in [7].

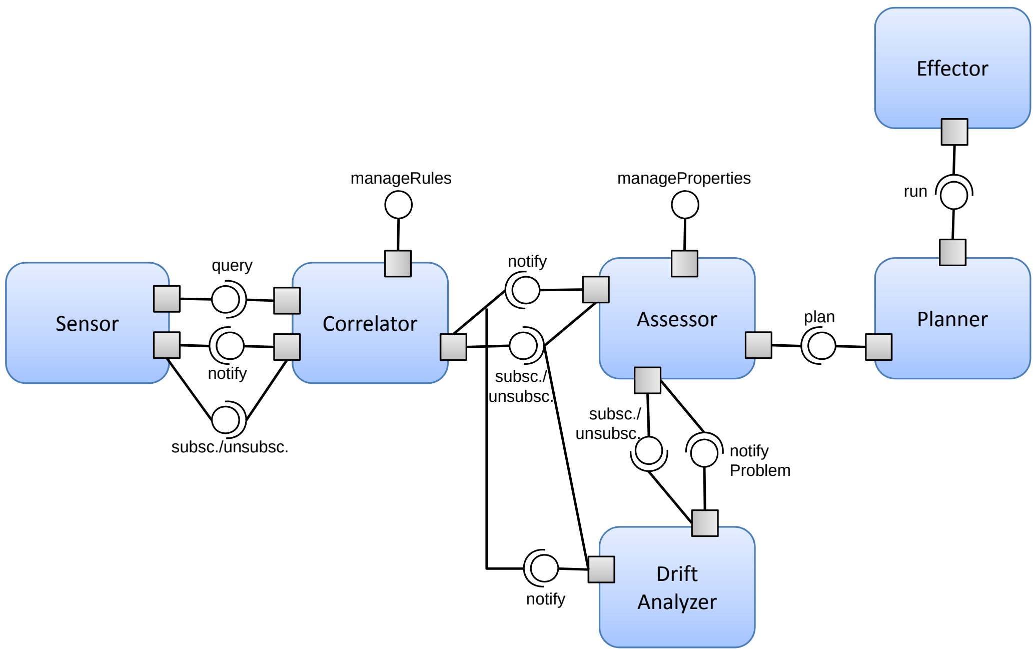


Figure 2. Components of the supervision system.

The supervision process starts with a Sensor that collects some data from the field. This information is passed - either synchronously or asynchronously - to the Correlator, which is in charge of analyzing, filtering, and correlating received data. After this step, the Correlator notifies both the Assessor and the Drift Analyzer. The former is in charge of detecting any possible anomaly as soon as it arises, while the latter adopts a wider spectrum and considers the drifts in the behaviour of the supervised system. The latter informs the former if there is a problem.

In case of an anomaly, the Assessor is in charge of contacting a Planner to ask for a new plan, that is, a recovery action to try to keep the system on track. The Planner produces it and asks the Effectors to execute it, that is, to "modify" the behaviour of supervised elements.

After presenting the main elements, we need to clearly state that:

– All the different components will be implemented as ACEs. They might be independent ACEs, part of the system just for the supervision, or they might be part of application ACEs, and supplement them with supervision features.

– The component interaction mechanisms supports both push and pull approaches. The interfaces presented in Figure 2 allow us to both query the elements to retrieve significant data, and let them notify the others as soon as they have significant data (based on a publish and subscribe approach).

– Besides the interfaces shown in Figure 2, all these components will exploit the GN/GA functionality to discover the other elements they have to interact with. More specifically:

  - A component can send a GA (in broadcast) containing the goal it is able to achieve.
  - A component can send a GN (in broadcast) containing the goal it needs to complete its task.
  - When a component receives a GA, it can decide to accept it or not. In case it accepts it, the component establishes a new contract with the component that sent the GA and it invokes the service described in the GA.
  - When a component receives a GN, it can decide to provide its capability or not. In case it provides it, it sends the GA to the requestor node.

- A component can send a GA (in broadcast) containing the goal it is able to achieve and a GN (in broadcast) containing the goal it needs to complete its task.
- This means that Figure 2 actually presents a snapshot of an already created supervision subsystem, but before this, we need a suitable discovery phase to properly identify who interacts with whom.
- For future development of the interaction mechanism, it might be not expected that all the components will share the same domain-specific knowledge, and thus that will be able to decode any message that will be sent to them. It might be useful overcoming this situation by constraining the components interaction to a limited number of conversation types, each conversation being highly structured and associated with the appropriate constraints on message content.

3 Components

Interfaces are described as sets of “methods” or “procedure calls”; we however do not impose any synchronous communication mechanisms. Instead, we assume that all functions of the components described below are implemented by some asynchronous message exchanges (e.g. by some request / replay scheme). We use a “free” formalism that resembles an object oriented class description language.

3.1 Sensor

Needless to say, a Sensor is in charge of sensing data from the field, in this case from the different ACEs. Currently, we present Sensor as a separate component, but we are already envisioning it as a dedicated feature, which can be switched on/off, provided by the different application ACEs.

We also envision two different kinds of Sensors:

- Sensors are *active* if they can provide the other components relevant information on their own. They distribute their data as soon as they become available. Components can subscribe and then they are informed “immediately”.
- Sensors are *passive* if they do not communicate spontaneously, but they need to be triggered (polled) to provide the information they have collected. This way, it is the component interested in get data from these sensors that must query them.

Interfaces

An active *Sensor* provides the following interface:

`bool subscribe(DataType)`

This operation is used to let the other components subscribe to the different types of data produced by the Sensor.

`bool unsubscribe(DataType)`

This operation is the dual of the previous one and is used to let a component unsubscribe for a given data type. If the component is not subscribed to that type, the operation has no effect.

A passive *Sensor* provides the following interface:

`DataType query()`

This operation can be used by those components that want to interact with passive Sensors. Needless to say the returned value is the datum (set of data) acquired by the Sensor.

3.2 Correlator

This component is in charge of correlating the events retrieved from the sensors. The Correlator must be able to interact with both passive and active Sensors, as described above). This means that a Correlator must be able to be notified when we data are retrieved from a Sensor, but it must also be able to poll the Sensor for new data.

The behaviour of this component is defined by means of user-defined rules to specify how retrieved information has to be correlated. Correlation rules are defined on data types. The default is that if no rules exist for a given type, the information is forwarded unchanged. These rules use standard correlation operators (e.g. [1]) to correlate retrieved data.

Interfaces

The *Correlator* provides the following interface:

`bool setRule(CorrelationRule)`

This operation is used to add a new correlation rule to the component. The execution of each rule is triggered by the reception of a new datum of the type(s) to which the rule applies.

`bool unsetRule(RuleId)`

This operation is the dual of the previous one and is used to get rid of a previously added correlation rule.

`bool subscribe(DataType)`

This operation is used to let the Assessor(s) subscribe to the different types of data produced by the Correlator.

`bool unsubscribe(DataType)`

This operation is the dual of the previous one and is used to let a component unsubscribe for a given data type. If the component is not subscribed to that type, the operation has no effect.

`bool notify(Datum)`

This operation is used by the other components (the Sensor, in this case) to inform the component of the availability of new data, passed as parameter. Informally, the structure of the Datum comprises both the information itself and also the data class described above.

3.3 Drift Analyzer

This component is in charge of analyzing drift behaviour of a SUS. While its internal structure is presented in Appendix 2, this section concentrates on the external interfaces of the DA with respect to the overall supervision system. Simplified, this component has an external behaviour that is similar to that provided by the Correlator. They both receive data from the Sensor(s) and can feed the Assessor with additional or more meaningful elaborations of the raw data retrieved from the SUS. The difference to the Correlator is that the DA is capable of storing and processing historical data thus being able to observe a system over time as well as providing trends and, ideally, information about the "direction" a SUS is drifting towards.

Interfaces

As seen from the overall architecture, the Drift Analyzer will be able to communicate with the Sensor, the Correlator and the Assessor, thus provides the following interfaces:

`bool notify(Datum)`

This operation may be used by other components (in this case, the Sensor, the Correlator and the DA itself) to inform the registered component of the availability of new data, which is passed as parameter. Informally, the structure of the Datum comprises both the information itself and also the data class described above.

`bool subscribe(Problem)`

This operation is used to let the Drift Analyzer know who wants to be notified when a problem or any other interesting phenomenon is detected. We assume that notifications are governed by problem types. Possible components to be notified include the Correlator, Assessor and the DA itself.

`bool unsubscribe(Problem)`

This operation is the dual of the previous one and is used to let a component unsubscribe for a given problem. If the component is not subscribed to that problem, the operation has no effect.

3.4 Assessor

The Assessor is used to understand if and how retrieved data are consistent with respect to the hypotheses set by the application. This component only works newly retrieved data to understand whether they meet supervision constraints. If a problem is detected, the Assessor triggers the Planner, which is required to provide a new plan to keep the system on track.

Supervision properties work on classes of sensed data, which might be specific of the different Sensor types, or might be produced by the Correlator. These properties will be specified using a language that resembles conventional assertion languages (like Anna [2], JML [3], OCL [4], or WSCoL [5]), along with meta data to define the importance of the check, its validity, and additional constraints on the classes of data on which they must be applied.

The Assessor can work in two different ways: synchronously and asynchronously. In the first case, the Correlator or the Drift Analyzer may ask the Assessor to check a particular property on a given datum, while in the second case, the Correlator and Drift Analyzer only inform the Assessor about the availability of new data, which triggers the evaluation of set supervision properties (given their class).

Interfaces

As for the synchronous behaviour, the *Assessor* provides the following interface:

`bool check Property(Datum, Property)`

This operation is called to check a particular property on a specific datum. Both the property and datum are passed as parameters. The returned bool(ean) value communicates the result of the evaluation.

As for the asynchronous behaviour, the *Assessor* provides the following interface:

`bool notify(Datum)`

This operation is used by the other components (the Correlator, in this case) to inform the component of the availability of new data, passed as parameter. Informally, the structure of the Datum comprises both the information itself and also the data class described above.

`bool setProperty(Property)`

This operation is used to add (set) a new supervision Property to the Assessor, which starts evaluating it as soon as the setup is complete. The returned bool(ean) value is used to check the actual completion of the operation. The evaluation of a property may result in calling the Planner for a new plan.

`bool unsetProperty(Property)`

This operation is used to delete (unset) an existing supervision Property identified by means of its id. When a property is deleted, the Assessor stops evaluating it. The returned bool(ean) value is used to check the actual completion of the operation.

`bool notifyProblem(Problem)`

This operation is used by the Drift Analyzer to communicate a possible problem and thus trigger the reaction flow, thus is to ask the Assessor to provide a new plan and then ask the Effector to execute it.

3.5 Planner

The purpose of the Planner is to compute potential executions (plans) that are supposed to lead the system under supervision (SUS) out of a problem situation into a more desirable system state. We assume the following:

Planning bases on the notion of a supervision model. Models comprise a set of states S and a set of transitions that lead from one state into another one. Transitions are labelled by actions from an action alphabet A. Intuitively, actions are the functions that the SUS provides, i.e. that can be called or invoked, but also the observable effects of the execution of those functions that can be observed. In the ACE based set-up that is currently developed by WP1, states and actions are given by the self-model of an ACE. In this document, we model actions and their (expected) effects by a partial mapping $\delta: S \times A \to S$, i.e. for some state $s \in S$ and some action $a \in A$, $\delta(s, a)$ denotes the state that is reached after the execution of a at the state s. If $\delta(s, a)$ is undefined, than the action a cannot be executed at the state s. We further denote by

$$\textbf{enabled}(s) = \{ a \in A \mid \delta(s, a) \text{ is defined} \}$$

the set of actions that can be executed at a state $s \in S$.

Some actions can be controlled by the SVN, some cannot (for instance because there are controlled by the environment of the SUS, the SVN has no permission to access the implementing functions, etc.). Furthermore, the execution of some actions can be observed, while other actions are executed in a "hidden" way, and can monitored only indirectly by observing their effects (state changes). Thus for each action $a \in A$ we assume two attributes a.cnt and a.obs of type Boolean indicating whether the action is controllable and observable, respectively. We further need a notion of "desirability" of system states. This is done by a predicate **acceptable**(s) that assigns a Boolean value to each system model state. This model is actually a simplified form of the notion that we gave in the document [6] that does not incorporate aspects concerning hierarchical supervision.

A plan is now essentially a sub-model of the underlying supervision model, where a model M_1 is a sub-model of a model M_2 if

1. the state set of M_1 is a subset of the state set of M_2;
2. the action set of M_1 is a subset of the actions set of M_2;
3. an action of M_1 is enabled at some state s of M_1 if it is enabled at s in M_2.

We say that a sub-model M_1 of a model M_2 is a plan the following properties do hold:

1. if it is deterministic on controllable actions, i.e. whenever s is a state of M_1, and a_1, a_2 are actions of M_1 such that $a_1, a_2 \in$ **enabled**(s), then a_1.cnt and a_2.cnt implies $a_1 = a_2$.
2. It is acyclic (this property seems to be too strong, but for the moment we can live with it), and has a defined start state **start**(M_1), and furthermore a set of end states **end**((M_1) – with the usual definitions of these terms
3. We have $\neg$**acceptable**(**start**(M_1)) and **acceptable**(s) for all $s \in$ **end**(M_1).

To turn this abstract notion into a computation object, we furthermore have to face the following problem: If an observable action is expected to be observed, how long do we wait until we conclude that this observation will never take place? When do we conclude that the SUS (or its environment) does not behave as expected? The usual solution for this concrete manifestation of the "halting problem" is of course to define timeouts. Thus we extend the notion of an action – for those which are used in plans as observable actions – by an expiration time. Considered plans, actions are thus equipped with an attribute *a*.expirationTime;

Remark: The notion of a plan give above differs from the notions presented in [6]. The difference is however motivated by one of the remarks made at the end of that document. In [6] we considered "unconditional" plans in the sense that they assumes a predefined unique reaction on the SUS to their execution, which is – considering the fact that the SUS may behave non-deterministically – a pretty strong restriction. We therefore planned as future work the definition of "conditional plans" which are supposed to be able to react more flexible on different reactions of the SUS. The automata model presented above is our solution to this problem. Details on the construction and interpretation of those conditional plans will be given in a later deliverable.

Interfaces

The *Planner* provides the following interface:

`bool init(Model)`

This operation initializes the Planner with the Model for which we want to create a plan. The returned bool(ean) value is used to check the actual completion of the operation. Informally, a Model is finite state machine, which mimics the evolution of the supervision system.

`Plan createPlan(State)`

This operation tries to create a plan given the initial state of the model we are interested it. If the operation takes too long or it does not terminate at all, we think of raising an exception to stop the planning activity.

Informally, a State is a state of the finite state machine behind the model. A Plan is another automaton whose states identify the actions needed to move the system under supervision in a consistent state.

`bool stop();`

This operation can be used to stop the Planner. The returned bool(ean) value is used to check the actual completion of the operation.

Planning Algorithm

This section details a first planning algorithm based on the refinement of the concepts introduced above.

```
Class Model {
      Class State  { } // states are not specified
      Class Action {
            Attribute cnt, obs: Boolean;
      }
      Attribute StateSet: Set of State;
      Attribute ActionSet: Set of Action;
      Attribute delta(State, Action): State
      Attribute enabled(State): Set of Action;
      Attribute acceptable(State): Boolean;
```

```
        // A generic definition of the project operation depends of course not on a particular
        // component but on its type. Thus we assume that there is an object
        // ComponentType associated with each component – this might be already the
        // self-model of the component
        Attribute start(): State
        Attribute end(): Set of State:
}

Class Plan extends Model {
        Class TimedAction extends Action {
                // meaningful only if obs = true;
                Attribute expirationTime;
        }
}
```

A first elementary planning algorithm is described below. It makes use of a number of variables and constrants:

- P is the plan to be constructed; a local procedure plan() is responsible for that.
- *max_depth* is a constant that defined the maximum recursion depth of the planning algorithm.
- *default_time* is the default expiration time form actions – in a later version of this algorithm we might go to make this a tunable parameter.

```
Algorithm createPlan(State s): Plan raises NoSuccessException {
        Procedure plan(State s, Plan P, depth d): Plan
        raises NoSuccessException {
                if acceptable(s) then return P;
                if d > max_depth then raise NoSuccessException;
                select a ∈ SupervisionModel.enabled(s)
                such that a.cnt;
                a.expirationTime = 0;
                s' = copy(SupervisionModel.delta(s, a)); // define a new state
                P.states + s'; P.actions += a; P.delta(s, a) = s';
                plan(delta(s, a), P, d + 1);
                forall s  ∈ SupervisionModel.enabled(s)
                such that a.obs {
                        a.expirationTime = default_time;
                        // define a new state
                s' = copy(SupervisionModel.delta(s, a));

                P.states + s'; P.actions += a; P.delta(s, a) = s';
                plan(delta(s, a), P, d + 1);
            }
        }
        Plan P = new Plan(); P.States = { s }; P.Actions = { };
        return plan(s, P, default_depth)
}
```

The algorithm makes furthermore use of a function select() that heuristically selects a controllable action to be executed at a given state. This basically is the place where the planning intelligence is implemented. In a more advanced version of this algorithms, select() might use adjustable transition weights that indicate success rates and costs for particular state/action pairs.

The algorithm presented above defines a deterministic plan: The supervision system does never have a choice what to do next. Non-determinism is only external, as the SUS or its environment may generate events (observable actions) at random. As described above, projection of a plan to the set of local components that is under the responsibility of a local SVS yields plans for these components. Note that projection might change the status of actions from observable and controllable to un-observable and non-controllable.

In a more advanced version of this algorithm we will take the fact into account that actions might be executed concurrently. Plans will then be "interpretations of distributed alphabets" as described in D2.1 – Mathematical Framework.

3.6 Effector

The Effector is responsible for executing these plans (recovery actions). An Effector executes a plan. Effectors are assumed to act from within the SUS, as part of internal control structures - in CASCADAS terms, as aggregated functions. Discussions within WP1 indicate that a control structure that acts on the internal message bus of an ACE is an appropriate implementation of an Effector. Since all internal control messages of an ACE (all actions) are distributed over this bus, the Effector can easily control the invocation of specific functions, and monitor internal and external events (observable actions in the terminology introduced above). This however requires a more detailed description of the ACE-internal communication, which is not available at the time we are writing this document.

Interfaces

The *Effector* provides the following interface:

```
bool run(Plan)
```

This operation is called to ask the Effort to execute the newly produced plan. The returned bool(ean) value is used to check the actual completion of the operation.

4 Case Study

In this section we are going to illustrate the interaction between the components of the supervision system by means of a simple case study. We start with a brief explanation of the system under supervision, which is a rather simplified version of the Behavioural Pervasive Advertisement scenario.

Consider a set of display screens located in some larger room or area (a shopping mall, a bazaar, etc.). Each screen receives user profiles from the mobiles of people in its proximity, and displays contents that meet the majority of the received profiles.

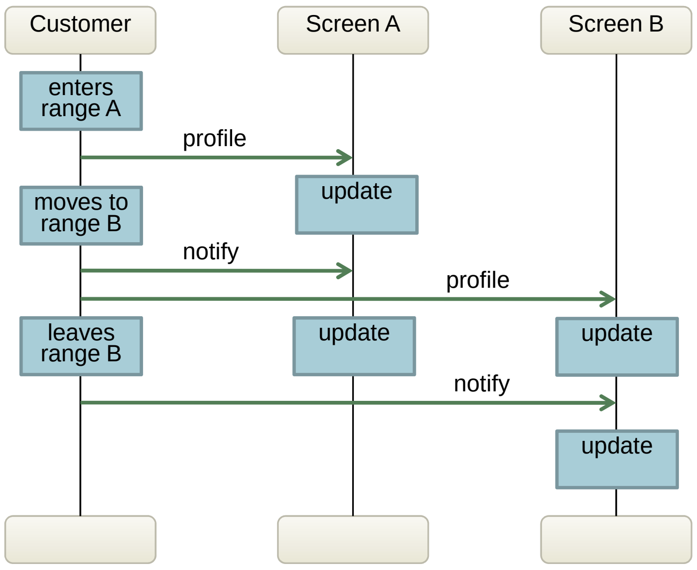


Figure 3. Basic Interaction

Figure 3 illustrates the interactions performed between customer mobile and screens in two neighboured screens A and B. The message exchange starts when the customer enters the range of screen A and sends its profile to screen A. Screen A then updates its contents according to the new profile distribution. When the customer moves out of the range of screen A into that one of screen B, it sends a notification to screen A, and its profile to screen B, causing updates of the contents of both screens. Finally, if the customer leaves the area, she notifies screen B which updates its display again.

Let us now have a look on supervision aspects. Clearly, the example is extremely simplified, thus it is not a surprise that our supervision task is simply to ensure the "business logic" of the screens. Let us concentrate on a particular failure, namely a failed update of one of the screens. Detection of this failure is – in our simplified example – done by simply by comparing the actually displayed contents with the expected ones.

What are possible the possible causes of this failure?

1. A profile message or a notification has not been processed correctly, was dropped, etc. (we leave open the question how the supervision system can be more knowledgeable about the current distribution of profiles than the screen itself).
2. The screen is not function correctly

In the first case, overwriting the currently display majority profile by the correct one is a valid countermeasure. In the second case, a reset of the screen might help. If this fails also, performing a shutdown of the screen might be our last option.

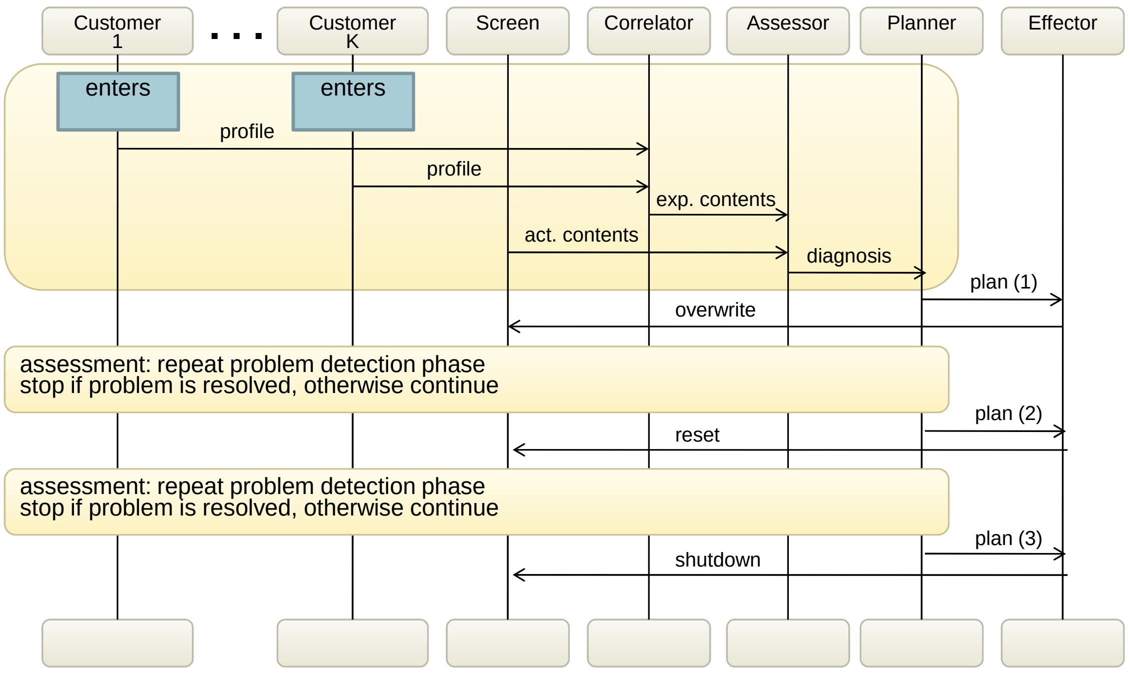

Customer 1
. . .
Customer K
Screen
Correlator
Assessor
Planner
Effector
enters
enters
profile
profile
exp. contents
act. contents
diagnosis
plan (1)
overwrite
assessment: repeat problem detection phase
stop if problem is resolved, otherwise continue
plan (2)
reset
assessment: repeat problem detection phase
stop if problem is resolved, otherwise continue
plan (3)
shutdown

Figure 4. Supervision procedure

Figure 4 illustrates the interaction of the supervision system with the customer mobiles and the screen. The Correlator collects the user profiles of all mobiles in range and correlates the expected contents to be displayed. The Assessor compares these data with the actually displayed profile and invokes the planner with corresponding diagnostics if the expected and the observed profile do not match. The planner then develops a contingency plan that comprises the three actions discussed above. When an action is performed, it is validated by another monitoring/correlation/assessment phase. This in particular implies that the current state of the supervision procedure is have to be stored in the planner, which has to decide which of the action has to be issued next.

Maintaining state information is known to increase the complexity of a system. Figure 5 shows an alternative supervision procedure that utilizes a message that is sent by the screen after a reset command to notify the supervision system about the success or failure of the reset action.

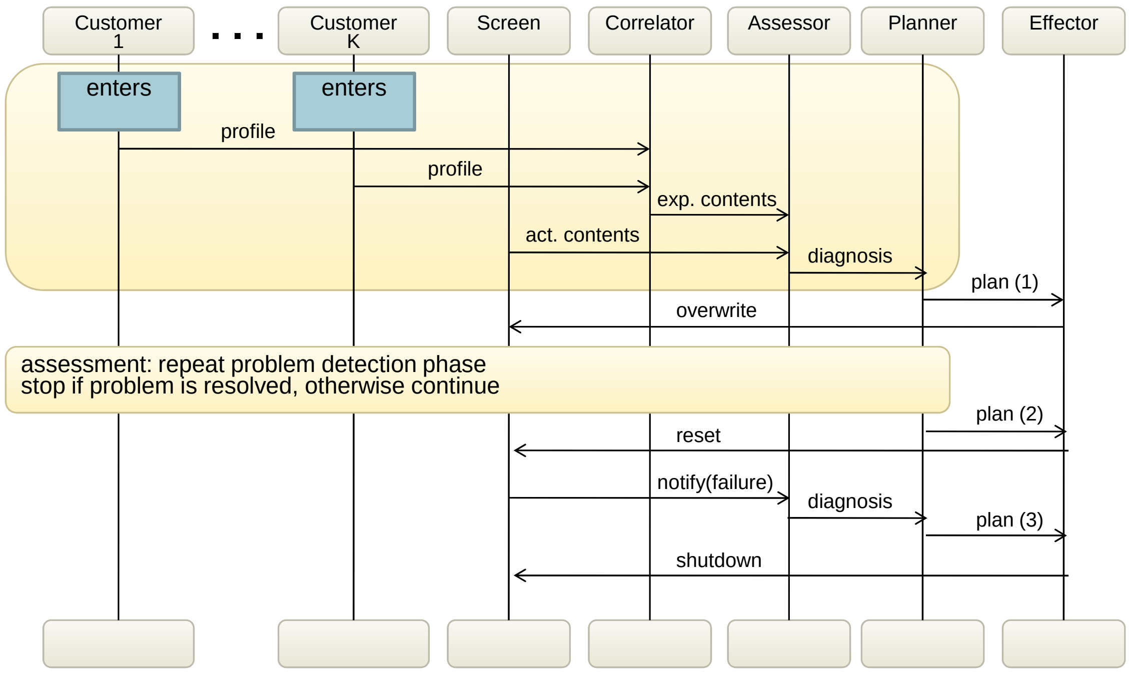


Figure 5. Supervision procedure (reduced state)

This example illustrates that supervision procedures can be simplified if state information is stored directly (and reliable) in the system under supervision. Since ACE based systems maintain a plan that controls the execution of the provided services, and hence is able to provide state information on request, reduction of the state information to be stored in the Planner is an attractive opportunity.

Let us look now into a second failure which illustrates a reduced set-up of the supervision system that does not depend on complex correlation and planning algorithms. Consider two neighboured screens with areas that do overlap. If a mobile located in the overlapping area sends its profile it is received by both screens. According to the business policy however each profile should be considered by at most one screen.

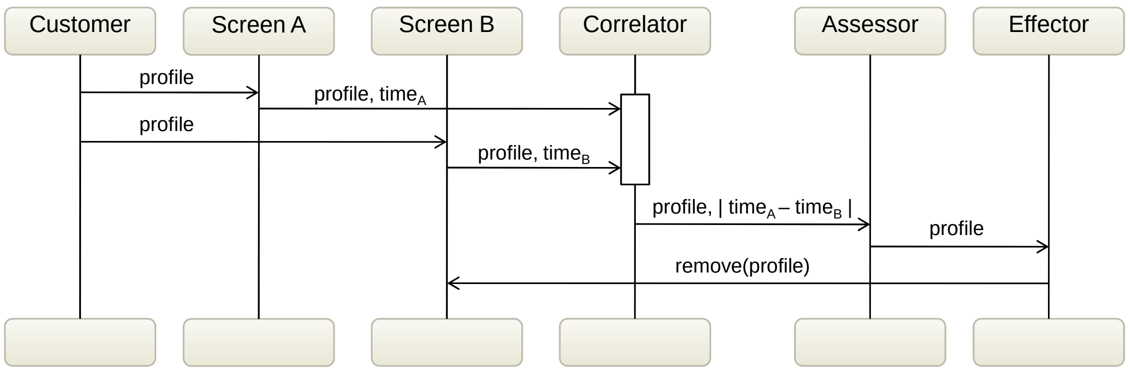


Figure 6. Overlapping ranges

Figure 6 illustrates a supervision procedure to alleviate this problem. Assume that there is a time synchronization mechanism available (such as NTP) that synchronizes the local clock of the screen. Thus update messages can be imagined sent by the screens to the Correlator about receiving a new profile information from a customer's that are equipped with a timing information. The Correlator then computes the absolute value of the difference between the point in time a profile is received by screen A and the point in time at which the same profile is received at screen B. If the time difference is sufficiently small (range of milliseconds), then the Assessor concludes that the same profile is received twice because the mobile is in the overlapping area and triggers an effector to send a request to remove one of the profiles to one of the screens (B in our example).

This example shows how interactions in a reduced supervision system architecture works that is concerned with supervision task that do not require complex planning or analysis activities, maintenance of states, etc. Those supervision tasks are supposed to be accomplishable by a (probably fine grained and cascaded) set of simple local reaction rules (like the illustrated one).

5 Outlook

Workpackage 2 currently investigates two basic types of supervision approaches, which can be referred to as model based supervision (MBS) and decentralized intrinsic supervision (DIS). MBS based on the idea to have use a distributed set of model objects that define, on a given level of abstraction, supervision objectives as well as supervision procedures. This is done in terms of a so-called supervision model. In the milestone deliverable D2.1 [7] and the accompanied document [6] a generic notion of supervision models have been introduced as (not necessarily finite) state-transition systems together with a value structure. Values are associated with system states. A state with a value that is below a certain threshold is considered as a "bad" or "unwanted" system state that requires some corrective measure by the supervision system. Once a bad state have been considered, the supervision system constructs a plan that is intended to lead the system to a "good" state, and executes this plan. Planning is done on the basis of the supervision model that defines the functions available for management and control purposes of the system under supervision (SUS). After the execution of the plan, it is validated whether the system state reached is really in the set of desired ones (i.e. has a value that is above the given threshold).

It has been noted that the procedure (comprising the steps: Monitor → Analyse → Plan → Execute → Validate) is somewhat heavy-weighted. In many cases, a simplified procedure that bases on local "involuntary" rules is more preferable. This approach bases on the continuous monitoring of the SUS and the immediate execution of some action (or set/sequence of actions) if a certain condition is met. The set of information items that is considered to trigger the execution of those rules is local in the sense that only a limited number of distributed components is queried (usually the neighbourhood of the component that performs the execution of the rule). Triggers might be trivial. In the easiest case, the supervision rule is executed repeatedly with a certain frequency. The intended behaviour of the system (the non-reachability of undesired system states) is then supposed to "emerge" from the execution of these local rules.

Note that as already outlined in Section 1.4, from the architectural point of view there is no difference between the MBS and DIS approach – the supervision system for the DIS approach assumes a simplified architecture comprising only Monitors, Correlators, Assessors, and Effectors (the contingency plan that is generated by the Planner in the MBS approach is now either "hard-coded" or statically configured in the Effector).

Let us discuss both approaches on the example of the simplified pervasive behavioural advertisement example (compare Section 4). Recall that each screen located in a certain area is supposed to display contents that matches a value computed by some functions f() with takes as input the user profiles of all mobiles currently located in that area. Thus if profiles as well as contents are modelled by colours, f() returns the colour that matches the majority of profiles.

Hence, a set of distributed supervision rule can be given as follows:

1. If C $\neq$ f(P) then C $\leftarrow$ f(P), where P is the current distribution of profiles
2. if C $\neq$ f(P) then reset()
3. if C $\neq$ f(P) then shutdown()

What is not described here is a control mechanism to guide the execution order of these rules (rule (3) is to be executed only if rules (1) and (2) have failed to resolve the problem, rule (2) is to be executed only if rule (1) has failed). There are a number of ways to establish such a control mechanism (maintaining internal states, using priorities, etc.), the details are not important here.

But of course it is possible to translate each set of rules of the above form into a state-transition system (with transitions triggered by input events). To see this, let us fix some syntax for rules:

<event>: **if** <cond> **then** <action>

Here, a <event> denotes a specification of events that triggers the execution of the rule, <cond> is a predicate that defines additional conditions, and <action> is an action to be performed if the rule is executed. Then the transition associated with a rule

E: **if** *c* **then** *a*

is enabled at all system states *s* that satisfy the predicate *c*, and accepts input symbols from the set {*x*: *E*(*x*)}. Its effect of the state at which the transition is applied to is described by the action *a*. Details of the construction are again not of importance here. On the other hand, if we consider a "plan" generated by the MBS approach as action part of a rule, and add the event/condition that lead to the generation of the plan as a trigger, than each generated plan can be used as a rule. Of course, nothing is said about the suitability of this generated rule sets. Hence, we state as a conjecture:

MBS and DIS are equivalent in the sense that (ignoring performance considerations) the interaction of supervision systems following these approaches with the SUS cannot be distinguished.

Provided the conjecture turns out to be true, we therefore can imagine a procedure that evaluates and assesses the suitability of a generated plan for a given situation or class of situations utilizing the validation step of the MBS approach. A plan that have been proven to improve the reliability of the SUS in a sufficient number of problem situations will then turned into a rule, and executed in an "involuntary" way without performing complex analysis, planning, and validation activities. The detailed elaboration of this procedure will be a major research topic for Workpackage 2.

5.1 Towards ACE based supervision pervasions

This Section describes elements of an approach to integrate the software architecture (and its refinements) into the ACE framework that is developed in WP1. We start with a brief summary of features related to supervision that are supported by the current implementation of the framework. We then provide some details of utilizing the GA/GN mechanisms to this purpose. We continue with an elaboration on the envisioned procedure for supervision of aggregated ACEs. The Section is concluded with a tentative roadmap defining an order of implementation of the discussed features.

ACE Supervision Support

The ACE framework prototype that is delivered as a part of the WP1 milestone for month 18 provides a number of mechanisms to support supervision of ACE based systems:

1. Access to the self-model of an ACE, and to its currently executed plan.
2. Access to the actual state of the plan the ACE assumes, including the session object that is used to store information necessary for the currently executed computation

3. Access to all messages that are internally exchanges, including send and receive events that corresponds to outgoing and incoming messages, as well as invocations of specific functions. This feature is implemented by means of a subscribe mechanisms that hooks up to the message bus which is used for all ACE internal communications.
4. Moreover, a "controlled mode" is supported. In this mode, a subscribed message is intercepted by the supervision system and not delivered until the supervision system permits its. The supervision system is able to delete a message from the bus, to insert a new one, and to modify an intercepted message. Therefore, this feature allows comprehensive supervision of an ACE at all levels (including common functions like the GA/GN protocol).

By means of these features, the supervision system is able to be aware and to control all aspects of the current execution of an ACE, on the level of the self-model of an ACE. The supervision system is by default not able to access and modify data related to internal processes in specific functions that are invoked from the plan execution engine.

Initialization of supervision

The current set-up of the supervision system architecture assumes that the system under supervision is a single ACE of a configuration of interacting ACEs, but does not take into account the structure of such a configuration defined by the "contracts" between its elements, such as the super/sub-service relations, processing chains, control structures, etc. We nevertheless use this section to describe initial ideas on how to achieve supervision for aggregated ACEs. The envisioned binding mechanism to couple an (singular or aggregated) ACE with a supervision system bases on the understanding of supervision as a service, thus a contract has to be committed between the ACE under supervision and the supervision system. This assumes that there is an actual entity on both sides that is able to perform the activities necessary to establish contract and binding. Whether this is – on the side of the supervision system – done by a dedicated component or by any of the components described in this document is not decided yet[1]. By assuming that supervision is basically a service that can be employed by other ACEs, we utilize the GA/GN protocol to couple a supervision system and an ACE (or ACE aggregate) to be supervised. The supervision system advertises (GA) a generic supervision service. Once a client for this service is identified (GN), a contract is committed these peers that includes the delivery of the self-model and current plan to the supervision system, and the subscription of observable and controllable messages as described above. If the ACE to be supervised an aggregated one (i.e. the access point of a composite service), the supervision system commits also contracts with the elements of the aggregate and configures itself according to the structure of the aggregate.

Hierarchical Supervision

Here, we assume a hierarchical structure of the aggregate in the sense that elements that dominate a set of other ACEs do not necessarily control each aspect of their interworking, but are at least aware of the service that is commonly provided by these ACEs, and provide service access points. Figure 7 illustrates this basic assumption. The ACE A_{sap} provides access to the service provided by the ACEs A_1 to A_4 (forming a sort of processing chain). A_2 is furthermore access point for a subservice provided by B_1 and B_2 (which are invoked in parallel).

[1] In principle, any of these components can be equipped with means to set-up initial contracts and to bootstrap the complete supervision system configuration.

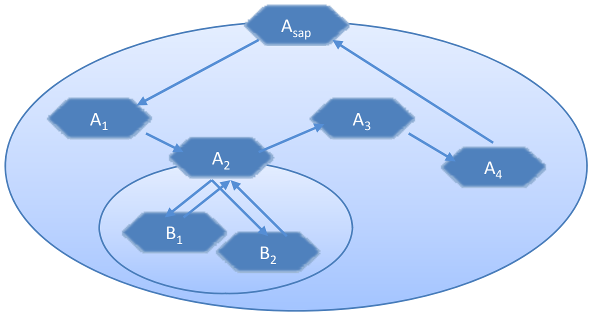


Figure 7 - Aggregated ACEs

The envisioned procedure works as follows: Assume that all the ACEs shown in the example have an associated supervisor. Assume there is a problem state detected at A_3. Then in a first step the supervisor SV(A_3) of A_3 tries to solve the problem locally, i.e. by only interacting with A_3. This of course can fail for various reasons:

1. There is no local solution at all. For instance imagine A_3 to be crashed, and a re-start might not possible. Then there might be an alternative sequence of actions that provides a similar (e.g. degraded) service but does not involve actions of A_3.
2. There is a local solution, but it requires that A_2 and A_4 perform certain actions in a specific order.

In either case, a solution of the problem (if it exists) involves certain actions to be performed by the other ACEs in the configuration. The problem is reported up one level, i.e. to the supervisor SV(A_{sap}) of the ACE A_{sap}. SV(A_{sap}) then constructs a global plan that involves actions of the ACEs A_1 to A_4, and sends the parts of the plan relevant to A_i to the supervisor SV(A_i) of that ACE. The hierarchical planning algorithm that we will employ bases on the use of model trees and the associated notion of "zooms", i.e. local refinement of models according to the reverse application of an abstraction map of a component (a local ACE self-model, in this case) that is embedded in the model we which zoom into (see [6] for details). In case of A_2 (which is also an aggregated ACE), SV(A_2) might refine this the local plan further to involve SV(B_1) and SV(B_2) in the supervised execution. Thus to perform planning for hierarchical system it is not necessary to even construct a global system model if no problem is detected or a problem can be solved locally – hierarchical supervision can be characterized as an "on-demand" approach.

Note that it is necessary in some cases to set up further means to coordinate the distributed execution of the global plan, e.g. by means of the exchange of synchronization messages between the local supervisors. An alternative approach that avoids synchronization problems is the centralized execution of the global plan by SV(A_{sap}). It will be part of the further work of WP2 to evaluate both approaches.

5.2 Implementation Roadmap

We conclude this outlook on further work by elaborating on a road map to implement the described concepts. This road map, leading from of monolithic supervision systems associated with each ACE to a pervasion of interacting supervisors is divided in the following phases (Figure 8):

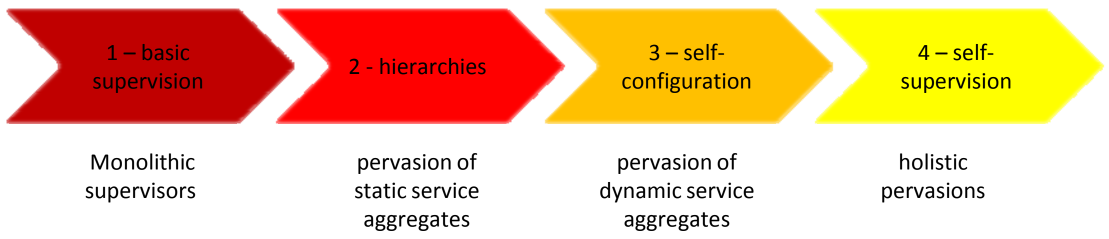


Figure 8 - WP2 Roadmap

- Phase 1 – “Basic supervision”: This includes the implementation of the components described in this document, under the assumption that the system under supervision is a singular ACE or an unstructured set of ACEs. It comprises the basis monitoring and effection mechanisms explained above. Self-models of ACEs will be employed as supervision models for planning, i.e. we consider also the establishment of contracts between supervision system and system under supervision.
- Phase 2 – “Hierarchies”: In this phase, hierarchical supervision is considered. In this phase we will stipulate the assumption that service hierarchies are a-priory given, and do not change during service execution. This simplification will allow concentrating of hierarchical planning and distributed execution of hierarchical plans.
- Phase 3 – “Self-configuration”: The assumption that hierarchies are fixed will be dropped in this phase. We now consider ACE configuration with dynamically adapt their structure, and what this dynamism implies to the structure and function of the associated supervision pervasion.
- Phase 4 – “Self-supervision”: A final phase concludes addresses the question how a supervision pervasion itself can be made robust against problems.

Appendix 1:

Supervision System Prototype Description

A1.1 Introduction

This appendix describes the WP2 proof of concept prototypes developed as part of deliverable M18. The main goal is to demonstrate the architecture of the pervasive supervision system under development by WP2, and to assess its feasibility on top of DIET.

To demonstrate how the supervision system works, we developed a simple distributed application (on top of DIET), described in Section 2. This also allowed us to provide the context for the supervision infrastructure. Section 3 briefly the architecture of the supervision system and Section 4 explains how the architecture is currently implemented in this first prototype.

A1.2 Smart advertisements

In this appendix, we assume the supervised application is placed in a big shopping mall. Customers are identified by means of their mobile devices, while the mall provides "smart" advertisement panels to help customers understand what they need. Customers move throughout different areas, which are equipped with different panels, and their portable electronic devices (e.g. their Bluetooth enabled mobile phones) contain ad-hoc software agents that communicate customer preferences to the smart advertisement system. The different panels can then display different commercials according to the preferences of the customers that are in proximity of the different displays. Everything is implemented on top of the DIET agent platform[1]: each customer is an agent, as well as each display, and both the advertisement system and the supervision infrastructure are a set of cooperating agents.

The advertisement system comprises different rooms, each containing three advertisement screens. The panels "sense" the presence of nearby customers (through the power of their mobile device's signal) and change their commercial advertisements to fit user preferences. To simplify the problem, user preferences are rendered through colours. Rooms host DIET agents that detect users and their preferences and display advertisements (colours) on the screens following a particular policy (Figure 1): the first screen displays the colour required by the majority of the clients in the room, the second displays the second colour in the list, while the third display renders the third colour.

[1] http://diet-agents.sourceforge.net/

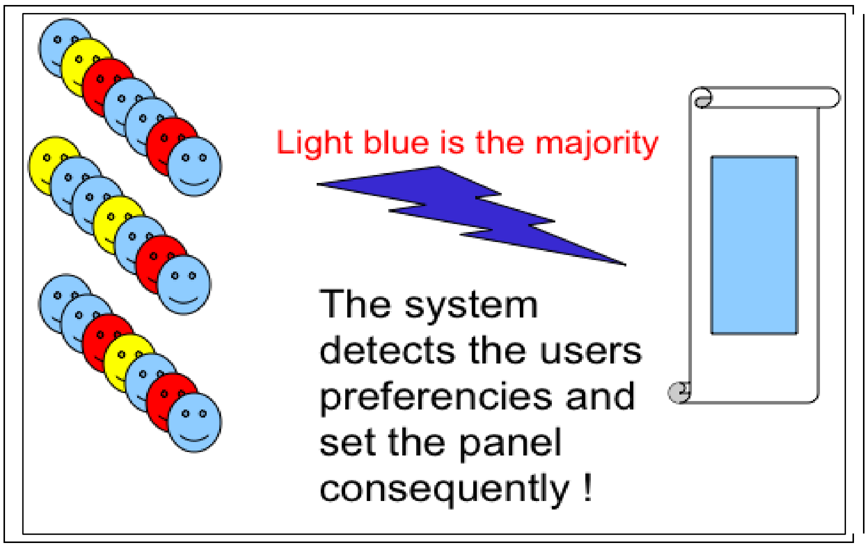

Light blue is the majority
The system detects the users preferencies and set the panel consequently !

Figure 1: A simplified scenario of the system for smart advertisements.

The supervised application comprises collaborating agents that implement some basic features (user detection, preference ranking, etc.). The following agents are defined for each room (DIET agents are called ACEs to pave the ground to the upcoming ACE-based implementation):

- The UserACE agent is the component installed in the user devices.
- The UserManagerACE agent manages the users that are detected in a particular room.
- The PolicyManagerACE agent provides the policy used to select the contents to be displayed.
- The ScreenManagerACE agent manages the screens available in a particular room.
- The SystemManagerACE agent exploits the services offered by the others agents to decide the colour to be shown on the screens.

Basically, when a user enters a room, and his device contains the right software, agent UserACE sends a message to the UserManagerACE, which is in charge of maintaining an up to date list of the users currently in the room. The message contains the user's identity and his/her preferences (i.e. a colour). At fixed intervals, the UserManagerACE forwards the complete lists of users to the SystemManagerACE, in charge of deciding the colours that must be shown on the advertisement panels. In order to decide correctly, the SystemManagerACE retrieves the appropriate policy from the PolicyManagerACE.

As soon as the SystemManagerACE receives the information needed to decide the colours to be shown, it sends a ranked lists of colours to the ScreenManagerACE which updates the screens available in that room accordingly. Figure 2 illustrates the main components.

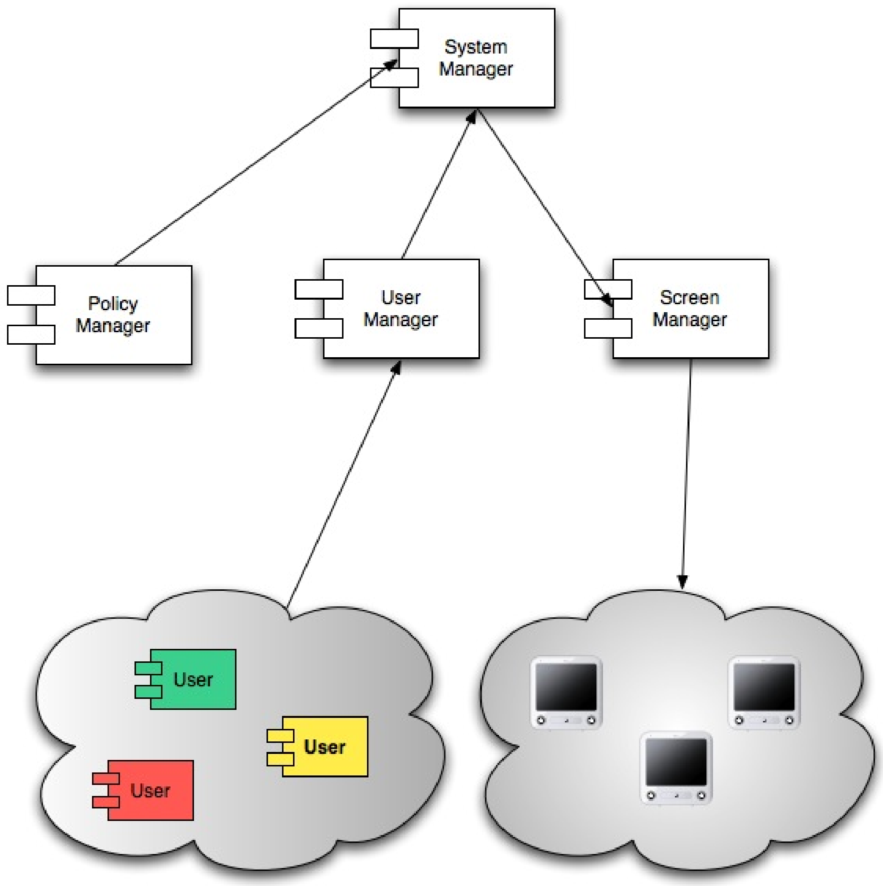


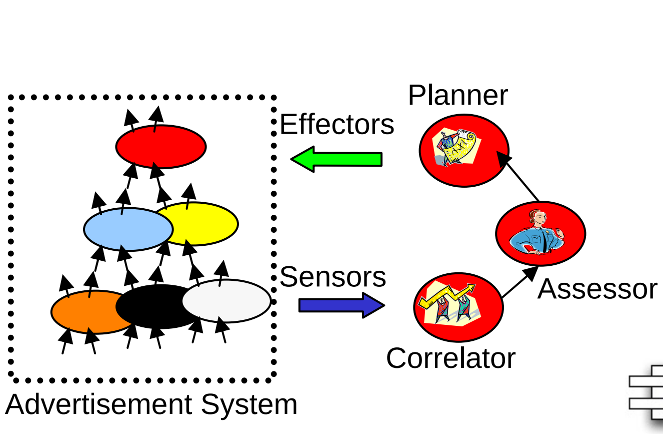
Figure 2: Main agents of the supervised application.

A1.3 Supervision system

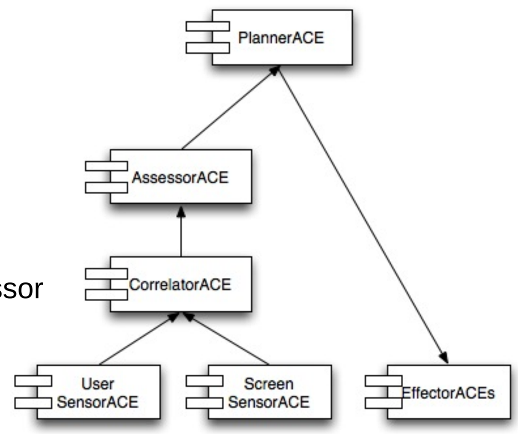
The supervision system is composed of different agents. Their cooperation permits both fault detection and system management. The supervision system is composed of the following agents, which mimic the components already identified in [7].

- UserSensorACE, PolicySensorACE, ScreenSensorACE
- CorrelatorACE
- AssessorACE
- PlannerACE
- EffectorACE

Figure 3(a) sketches the simplified supervision process: Sensors are distributed in the room and they are in charge of detecting all the events and messages useful for supervision purposes.



(a) High-level process



(b) Main components

Figure 3: Simplified supervision process.

This information is then forwarded to the correlator which groups the information received from the different sensors: both the client devices and the displays must send their data. The accessor understands if there is a problem and generates the faulty event for the planner, which creates a new plan to recover from the problem (i.e., it resets the displays), and informs the effectors to execute it.

Figure 3(b) presents the main components: a CorrelatorACE groups the data received from the different sensors (UserSensorACE and ScreenSensorACE) and sends aggregated data to the AssessorACE. This last agent detects possible faulty conditions and informs the PlannerACE, by means of a particular message. It elaborates a plan and uses the EffectorACEs to apply it.

A1.4 Our prototype

When started, the prototype generates three rooms (environments according to the DIET jargon). Each environment contains an advertisement sub-system composed of the agents described above. Each room contains three screens.

Figure 4 shows a room that detects two users, the small rectangles, one with a yellow preference and the another one with a blue preference. It is important to notice how the advertisement system updates the two screens to satisfy the preferences of the two users. The third screen remains unused since there are only two users.

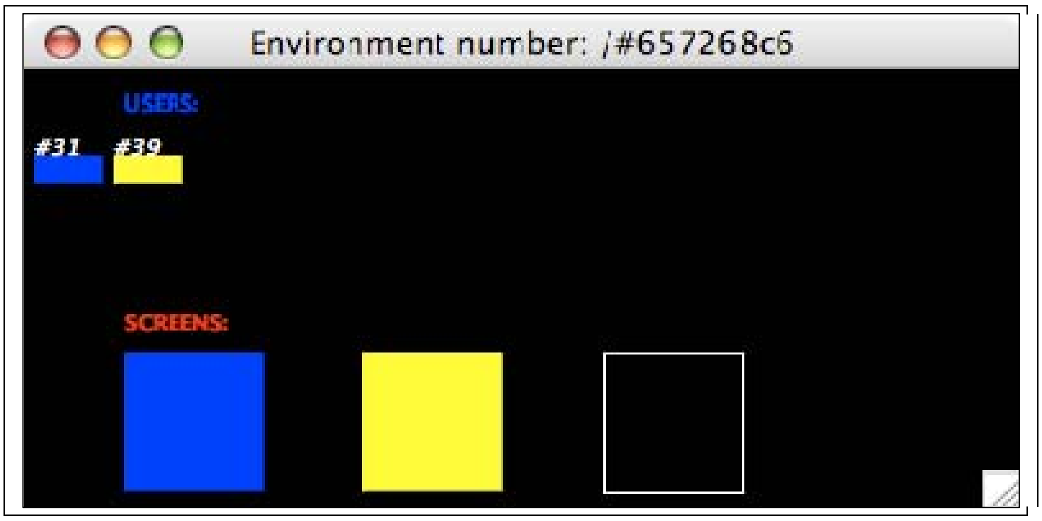


Figure 4: A room with two customers and three screens.

The users can enter and leave the system and they can also move around and migrate autonomously from one room to another. The effect, in terms of the supervised system, is that the displays update their colours, that is their advertisements.

To demonstrate how the supervision works, the prototype has a failure generator capable of injecting faulty behaviours into the systems (this means that the screens display the wrong colours). Figure 5 shows the GUI we use to expose the SystemManagerACE of a particular room to a faulty behaviour. The failure is caused by a BadGuy agent that sends a CORRUPTED message to the SystemManagerACE.

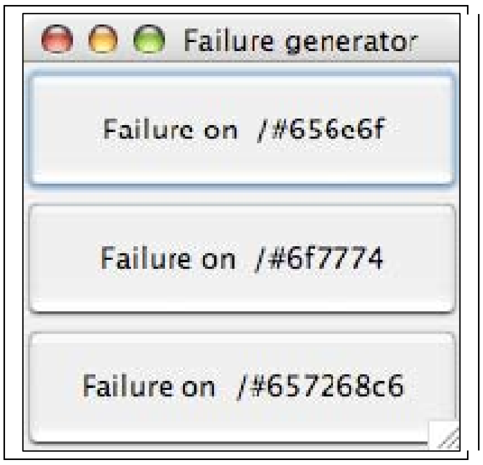


Figure 5: Failure generator.

A faulty SystemManagerACE communicates erroneous data to the ScreenManagerACE that consequently displays colours that do not correspond to the user preferences as shown in Figure 6.

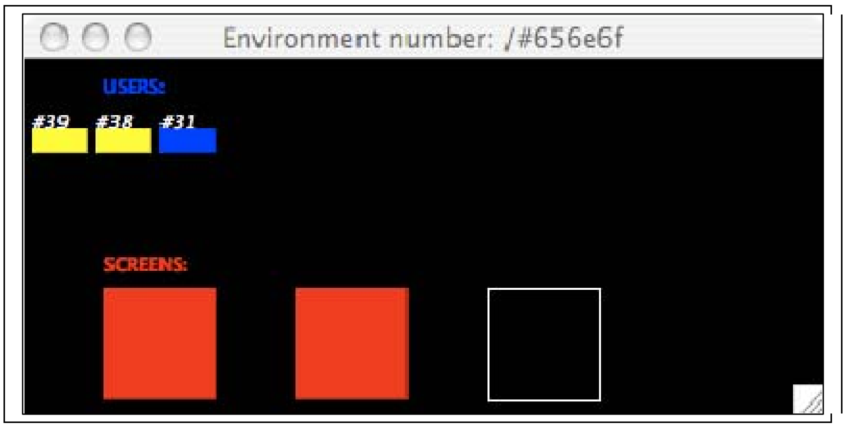


Figure 6: A faulty behaviour.

After a few seconds, the minimum amount necessary to obtain a consolidated set of data, the supervision system produces a new plan, that is, it decides to re-set the displays, and re-start the screens by applying the explained algorithm. Figure 7 shows the transition.

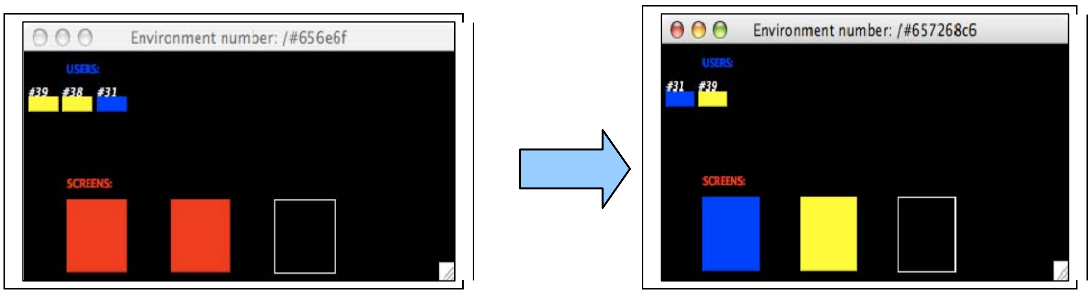


Figure 7: Reaction to a faulty condition.

A1.5 How to run it

The prototype is a plug & play Eclipse project. To run this application, create a new project and include the lib folder to the build path. The main class is AdvApp.java

Appendix 2:

Concept Drift for Pervasive Supervision

A2.1 Introduction

There is strong motivation for new perspectives on generic supervision methodologies in order to provide more resilience in the face of ever more complex systems. In particular, future autonomic systems that ideally operate with no or only a limited user input require such advanced supervision to control and adapt different variables of existing systems but more importantly to supervise the dynamic aggregation of individual autonomous working components as found in e.g. upcoming service oriented architectures or SOA. This research seeks to explore the requirements for a supervision mechanism that is capable of observing and analysing complex and dynamically constructed models that reflect a real world service or computational system. Furthermore, the method proposed will be able to operate at different levels of granularity with respect to the system supervised and as such supports the methodological framework for pervasive supervision. Subsequent sections explore the requirements for different observation methodologies for distributed and network like knowledge structures, in particular exploring how such knowledge can be gathered, represented and what type of mechanism can be used to detect so called drift behaviour within the observed data. A secondary objective is formed by the problem of how such drift behaviour can be used to (a) adapt individual components of a supervised system and (b) achieve a stable state of more global oriented systems, which could then freely evolve within pre-defined boundaries that describe the functional correctness of the system under supervision. In particular, the use of a lower and upper bound as well as the so called ideal state of individual variables will be explored.

A2.2 Generic Architecture

In general, state of the art supervision methodologies and systems mainly implement a closed control loop approach which implements the following three concepts.

- **Monitoring**: Gathering of information from the system that is under supervision. Additional tasks may include correlation and translation activities in order to pre-process incoming information to improve the quality of the monitored data and to reduce information overhead.
- **Analytics**: Dedicated methods testing for certain conditions, violations etc. that are of interest to the supervision process. Current analytical methods often implement a static rule- or policy-based methodology where individual rules or policies are "hard coded" for each system and as such are not dynamic and often difficult to adapt to changing conditions. While such methods are sufficient for traditional applications working in non-distributed environments, future autonomic systems will require more dynamic, highly intelligent and fully automated services that are able to operate in distributed context aware environments and as such are able to not only adapt the system but, more importantly, the supervising system itself
- **Reaction**: the reactive part of a supervision system closes the loop to actually achieve supervision. That is guiding a system within the boundaries it is allowed to operate in. The challenge for this part is not to realise and control the so called actuators which realise individual corrective measures on a supervised system. On the contrary the correlation of a given problem, that has been detected, with the correct countermeasures at different levels of granularity can be seen as the biggest challenge for the reactive part of a pervasive supervision framework.

The same three concepts are also relevant for a more long-term oriented and evolutionary-based supervision principle as envisioned here. That is with one important extension. In order to allow a system to evolve over time but at the same time assure the correctness of the underlying logic, advanced forecasting and prediction methods are required which allow the system to:

- forecast the "direction" of a supervised system;
- predict individual attributes based on past behaviour or on other attributes;
- and finally, detect critical states before they actually occur.

For that to be realised, it is necessary to build up a history of all monitored attributes of the system that is under supervision. Dedicated forecasting and prediction methods could then be used to predict future states and events based on the past behaviour of the model that is under supervision. Specific reaction mechanism may then be linked to the monitored model in order to register dedicated corrective measures to specific parts of the system under supervision. Figure 1 depicts schematically the general supervision architecture and highlights individual aspects for each part which will be discussed throughout this section. For convenience, individual monitor or actuator units as relevant for a complete supervision system are not shown in Figure 1.

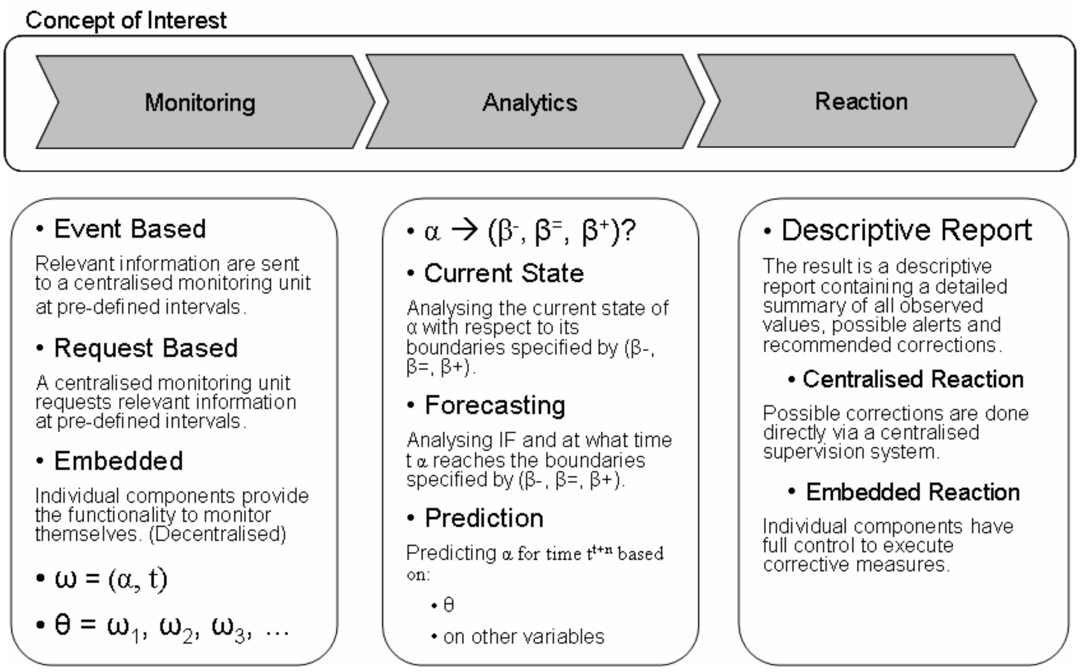

Concept of Interest
Monitoring
Analytics
Reaction
• Event Based
Relevant information are sent to a centralised monitoring unit at pre-defined intervals.
• Request Based
A centralised monitoring unit requests relevant information at pre-defined intervals.
• Embedded
Individual components provide the functionality to monitor themselves. (Decentralised)
• $\omega = (\alpha, t)$
• $\theta = \omega_1, \omega_2, \omega_3, \ldots$
• $\alpha \rightarrow (\beta^-, \beta^=, \beta^+)$?
• Current State
Analysing the current state of α with respect to its boundaries specified by (β-, β=, β+).
• Forecasting
Analysing IF and at what time t α reaches the boundaries specified by (β-, β=, β+).
• Prediction
Predicting α for time t^{t+n} based on:
• θ
• on other variables
• Descriptive Report
The result is a descriptive report containing a detailed summary of all observed values, possible alerts and recommended corrections.
• Centralised Reaction
Possible corrections are done directly via a centralised supervision system.
• Embedded Reaction
Individual components have full control to execute corrective measures.

Figure 1: Schematic Supervision Architecture, Concept Drift

A2.2.1 Monitoring

Independent of the technique used to monitor an individual source, the goal can be summarised as to collect (a) a tuple ω = (α, t), where α is the observed value and t a timestamp referring to the time the value / event has been observed. A dedicated history service could then used to build up a context history of the observed source such as θ = ω1, ω2, ω3, … . With respect to an overall monitoring mechanism three distinct monitoring mechanisms have been identified to be relevant. These are:

- Event Based Monitoring: The observed source posts relevant information at pre-defined intervals or at certain events (e.g. the value of the observed attribute has changed) to a centralised monitoring unit.
- Request Based Monitoring: A centralised monitoring unit requests at pre-defined intervals or at certain events (e.g. an outside alert) requests relevant values from observed sources.
- Embedded Monitoring: Individual components provide the functionality to monitor themselves. That is a specialised component is linked to the component under supervision which facilitates relevant monitoring tasks.

	Event Based	Request Based	Embedded
Decentralised Monitoring	no	no	yes
Lightweight Components	yes	yes	no
Controllability & Configurability	medium	good	difficult
Time to Answer	slow	slow	fast
Monitor Delay	medium	high	none
Complexity	medium	medium	high
Attribute Correlation	possible	difficult	
Cascade able	no	no	Build-in
Overall	++	+	+++

Table 1: Monitoring Techniques

As shown in the table above, embedded monitoring mechanisms are best suited due to a fast response time and the possibility of supporting an underlying cascadeable framework.

A2.2.2 Analytics – Concept Drift

As mentioned earlier, a concept of interest is a phenomenon that describes a real world model and is defined by underlying contextual information or raw data. By nature, it is likely to change over time which is referred to as concept drift. Drift may occur in the underlying concept of interest if it:

- is not static e.g. dynamic models
- can not be described in its entirety e.g. incomplete models
- if its values are subject to change in any way e.g. changing context

Three distinct analytical methods have been identified for detecting drift. These are:

- Current State (Sudden drift behaviour): Analysing the current state of α with respect to its own value / state and / or with respect to pre-defined boundaries as specified by (β-, β=, β+)[1].

[1] See Section 3.3

- Forecasting (Visible, continues drift behaviour): Analysing if and at what time t α reaches e.g. a critical state as specified by pre-defined boundaries.
- Prediction (hidden, continues drift behaviour): Predicting α for time tt+n based on: θ or more interestingly based on other variables that are correlated to a given α. At this stage it is important to stress that prediction and forecasting are seen as two different concepts of detecting drift behaviour. While the former relates to standardised statistical functions that e.g. establish trends based on the history of the system under supervision, predictive mechanisms may utilise dedicated AI mechanism that include other related or unrelated aspects.

A2.2.3 Reaction

Based on the closed loop approach, the reaction part of a supervision process is concerned with the identification, configuration and execution of relevant measures to counteract incorrect behaviour or to invoke a specific recovery mechanism. With respect to the discussed monitoring and analytical mechanisms identified so far, two possible reaction mechanisms are deemed relevant.

- Direct Reaction: Corrective measures are invoked whenever an illegal state or violation is detected. This mechanism is particularly relevant for autonomous micro-supervision systems that are fully aware of what to supervise, how to supervise it and finally how to react if something goes wrong.
- Descriptive Reporting: If a system is not able to react on an illegal state or violation or if a system is forced to invoke countermeasures on a more global aspect of a system, then individual components may choose to report their current 'health' to conceptually higher oriented supervision components. Obviously, such a reporting mechanism should be as complete as possible containing information about the sender, the fault, possible reasons (if the fault already has been analysed locally) and if known, relevant corrective measures.

Both mechanisms may be realised in a centralised way where possible corrective measures are identified and executed via a centralised system or, alternatively, in a decentralised system that is embedded, where individual components have full control in executing corrective measures. The latter obviously requires that each component is aware of the reactive measures it can invoke. Furthermore, a dedicated component may be implemented that assesses individual reactive measures on a higher level of granularity.

A2.3 Drift Analyser – DA

The following sections describe in more detail the technical specifications for implementing (a) the overall component that will handle the problem of drift behaviour and (b) individual components thereof envisioned to facilitate dedicated tasks. It is envisioned that the DA as well as its sub components are realised as independent services to allow for maximum flexibility with respect to the dynamic orchestration of specific supervision mechanisms.

A2.3.1 Technical Architecture

The main DA service will allow for the orchestration of sub-services related to the general analytics of concept drift. Depending on specific bootstrap or runtime configurations individual services may be loaded in order to perform individual supervision tasks. However, if no service is loaded then its functionality may be ignored. For example, if there is no forecasting service, then forecasting will not be performed even if an actuator registers for it.

The high level architecture of the DA as depicted in Figure 2 is based on the closed control loop and as such the DA will connect to a monitoring mechanism (the Sensor Interface) as well as a reaction mechanism shown (the Assessor Interface). Both of these interfaces reflect the standardised interfaces and form the basic I/O mechanism of the DA with respect to the overall supervision architecture as outlined earlier. The DA architecture itself comprises the following three concepts:

- A generic service based architecture where individual modules can dynamically be (un-)loaded. Already identified services include a dedicated forecasting mechanism, a mechanism to detect boundary violations as well as the monitoring of the ideal "state" of the system under supervision.
- A communication bus enabling event based communication among individual services
  - An actuator registry where applications, services (such as the Assessor) and other DA's may register themselves to be notified by selected services of the DA there are register with.

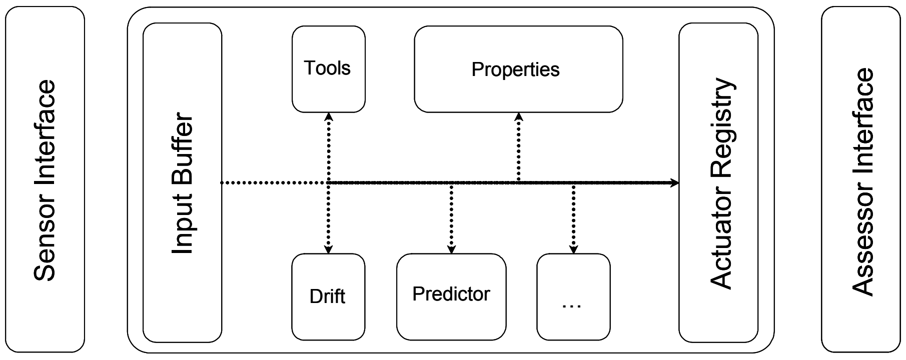


Figure 2: DA – Architecture

The proposed architecture supports both the general ACE concept as well as the DIET methodology to an extent that both are to be orchestrated via sub-components (here services) which are capable of communicating among each other and with the outside world via a dedicated communication mechanism. Nevertheless, specific design adaptations may be necessary once a final framework of ACE's is submitted.

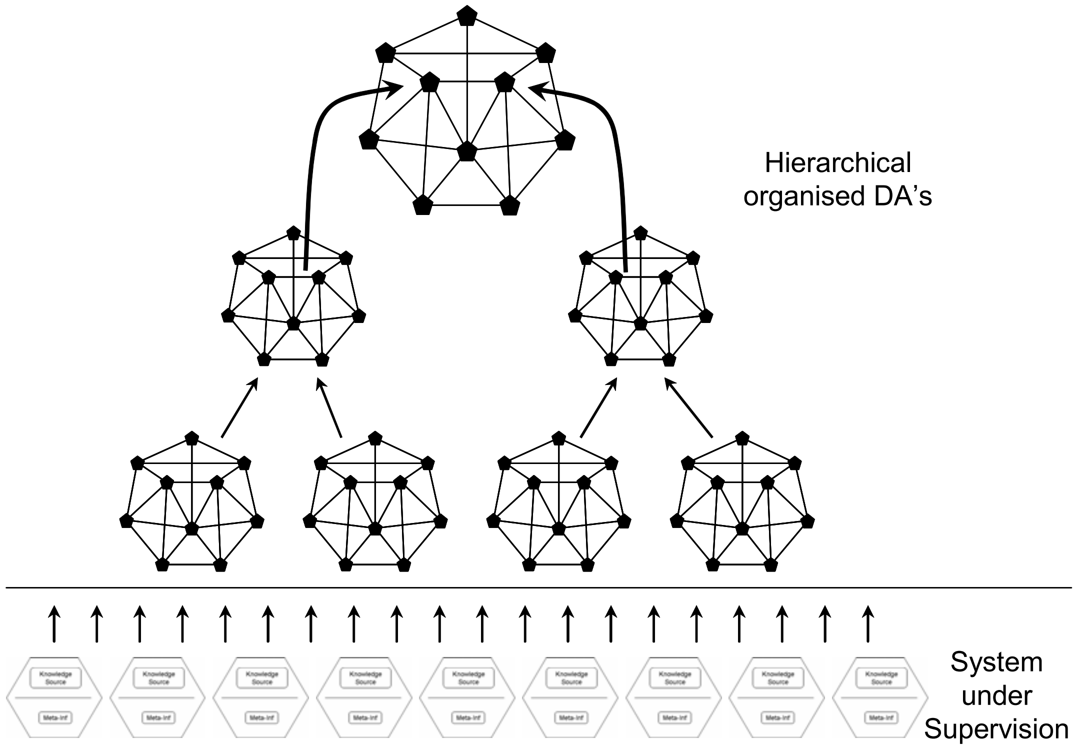


Figure 3: Hierarchical DA Organisation

So far, the outlined architecture provides only limited supervision functionality in a sense that each DA is only capable of supervising a single concept of interest. Although it is envisioned, that individual concept of interests are not limited to just primitive types, but instead could also reflect a more complex model or structure, the supervision within a single hierarchical level will, in most cases, not be sufficient. To achieve a more powerful and eventually fully pervasive supervision potential, relevant mechanisms need to be available at different levels of granularity on (a) the system under supervision and (b) the supervising system (independent if this system is localized distributed). In order to provide for the hierarchical, or in fact network, like orchestration of DA's the selected notification mechanism, currently the actuator interface, will also serve as a potential input source for the monitoring unit.

As depicted in Figure 3 individual DA's may register with lower or even higher oriented DA's to be notified of certain events. If connected to dedicated reasoning mechanisms, incoming messages stemming from other DA's or from other monitored concepts may be properly correlated and analysed. While this not only provides the modelling of more meaningful concepts of interests it also allows for supervision at different levels of granularity yet maintaining the self-similarity of individual supervision components that is required to serve the underlying ACE model as it may provide the structure of the system to be supervised.

A2.3.2 Numeric vs. Symbolic Monitoring

The sources may provide information that is either numerical or symbolic in nature. Numerical information can be evaluated with statistical processes that are similar to time series evaluation. It may not be as simple to detect drift in symbolic values. One possibility would be to build up a statistical record of the symbols recently used and then detect when different sets of symbols are being returned. For example, one could split up the history buffer of symbolic values into two parts and compare the values in the second part with the values in the first part. If they are different then maybe drift has occurred. In general, symbolic values can be processed when at least one of the following conditions is provided for.

- The set of symbolic values is finite and known beforehand.
- A sort of ordering can be induced over the symbolic values to be monitored.
- A dedicated logic may be used to evaluate the incoming stream of symbolic values.

A2.3.3 Boundaries

Identifying the general 'path' of a system and as such drift behaviour is a powerful method to identify if a system suddenly or slowly (but continually) moves into a specific direction or towards an unwanted state. Normally, for a system it is indifferent to whether such a state represents only an annoyance or a more seriously maybe critical situation of the system under supervision. Thus boundaries may be utilised to (a) identify if a system is in an illegal state; (b) predict the time it takes to reach an illegal state; or (c) to self–organise it around an ideal state, which is either pre-defined or the mean of its boundaries. Therefore, the boundaries of a given concept of interest can be utilised to specify the states a system can evolve in without violating its general purpose and as such are key to the overall analytical process.

A2.3.3.1 Lower and Upper Bound

As depicted in Figure 4, the lower and upper bound (β^-, β^+) define the borders a system can evolve (operate) in. Overlaid trends would then allow predicting long term directions so that out of bound violations could be identified at an early state. If a system violates the boundaries a possible alarm or other action may be triggered or corrective measures may be induced. Note that individual lower and upper bounds do not need to be static. Depending on a changing context individual boundaries may change as well. However, dynamically adjusting or even identifying initial boundaries is error prone as falsified values may be interpreted as new bounds.

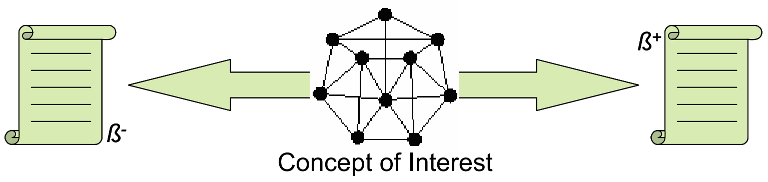


Figure 4: Boundaries for Concept Drift

A2.3.3.2 Ideal State Situation

More interesting in the context of autonomic computing is the organisation of a system around a so called ideal state, $\beta^=$, which is the situation a system should attempt to achieve. For instance, an autonomic system regulating the temperature of a building has to react to a multitude of factors, e.g. outside temperature, number of people in the building, open windows, etc. Nevertheless, based on its configuration its ultimate goal could be as simple as "keeping the temperature at 27 degree Celsius". In this case the ideal state of is reflected directly by its goal. Never mind the fact that, in this case, it would be pretty hot in that building.

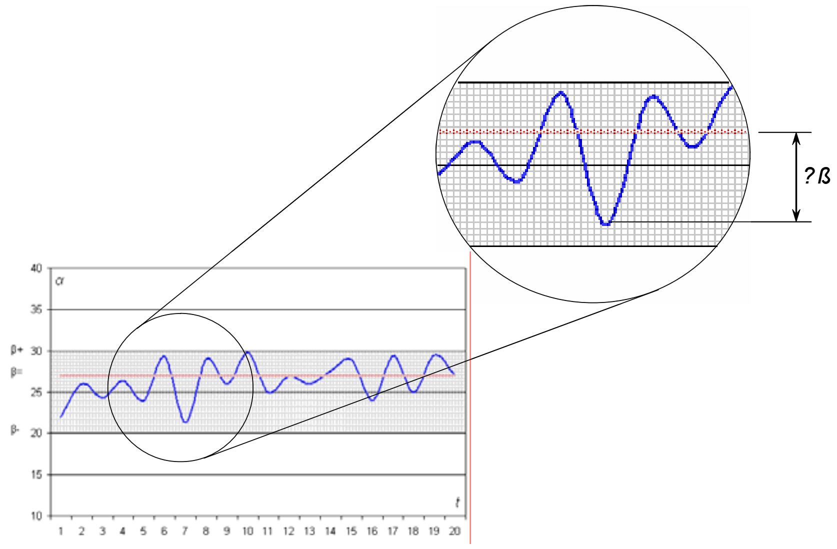


Figure 5: Ideal State

This concept is visualized in Figure 5 where the difference between the actual state and the ideal state is constantly monitored.

A2.4 Functional Specifications – Plug-Ins

Plug-ins actually do provide the functionality of the DA in a sense that specific supervision tasks are supplied via individual modules and then orchestrated within the DA which acts as a container component. Plug-ins may come in the form of services that implement the required functionality to monitor the concept drift or aspects that are deemed relevant for supervision purposes. The general idea is that a specific DA instance can be configured by dynamically loading dedicated plug-ins in order to perform specialised supervision rather than generic observation. The components described in the sequel represent concepts that have been identified as highly relevant for the problem of analysing aspects related to drift behaviour but the list of components discussed here is by no means complete and may be extended as required.

A2.4.1.1 Peak Buffer

As visualised in Figure 6, this service aims to de-sensitise the system under supervision by averaging the stream of input values. For instance, it could read a series of values over a specified time interval calculates the average and then sends this value to the rest of the system as the actual value. Note that in this case there is an issue with out of bound oscillations, where values can oscillate from too high to too low but still produce an acceptable average. This however is a minor issue which can be accommodated for in different ways. The peak buffer is envisioned to be a standard service which is always loaded into the system but may be bypassed if desired. The rationale for this is that it also functions as a safety mechanism for incoming false values or exceptions. Finally, the method employed to de-sensitise the input stream may also be loaded dynamically to allow for different types of stream manipulations as well as the handling of symbolic values.

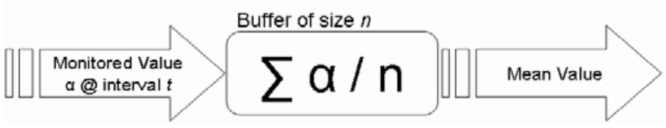


Figure 6: Peak Buffer

Peak Buffer
-Buffer Size : long -Polling Interval : long -Evaluation Method
+Output() : <unspecified> «signal»+Monitor Interface()

Figure 7: Class Description of the Peak Buffer Service

A2.4.1.2 History Buffer

As the name suggests a history buffer will store a sequence of events (values) over a specified time period to be used e.g. to perform calculations such as trend analysis. The buffer stores the values along with a timestamp of when the value was read (see Figure 8).

History Buffer
-Buffer Size : long -Polling Interval : long
+Output() : Event History «signal» +Monitor Interface()

Figure 8: Class Description of the History Buffer Service

A2.4.1.3 Property Management

The rational of this component is to constantly observe the underlying system in order to identify possible boundaries based on the systems history. Ideally this component will serve as an input to other components such as current state or ideal state, providing a more dynamic boundary mechanism that can operate without the need of pre-setting the boundaries in which a system can operate in or around, in the case of an ideal state monitoring (See Figure 9).

Property Management
+Output1() : Suggested Ideal Value +Output2() : Suggested Bounds «signal» +Monitor Interface()

Figure 9: Class Description of the Property Management Service

A2.4.1.4 Current State

As visualised in Figure 10, the current state is constantly read and validated against the boundaries. If outside of the boundaries a dedicated notification message is sent to the actuator registry. A class description of the current State Service is illustrated in Figure 11.

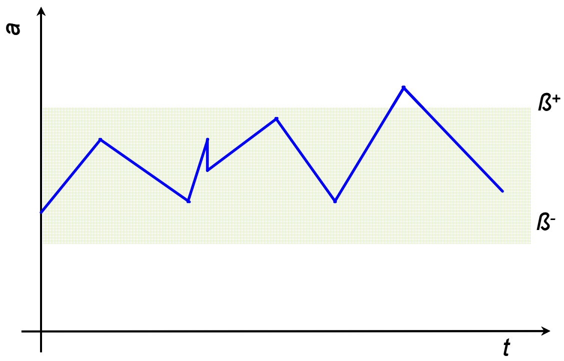


Figure 10: Current State

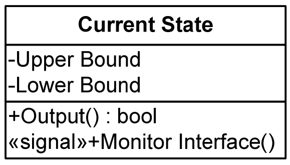


Figure 11: Class Description of the Current State Service

A2.4.1.5 Ideal State

The ideal state of a system is seen as its optimum health. As such a supervision mechanism should attempt to organise a system as close to this state as possible. As depicted in Figure 5, this service detects the difference between the current state and the desired state and notifies interested parties of the difference so that distinct countermeasures may be executed. The ideal state should be the first concept that is evaluated. The supervision class needs to be able to load in any evaluation class, so it could be based on the generic component that can load in services. The supervision class is then also a service that can be loaded into the basic autonomic component of the network (See Figure 12).

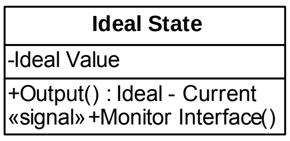


Figure 12: Class Description of the Ideal State Service

A2.4.1.6 Forecasting

Forecasting is used to measure the trend in the current data. It can be used to predict when a system may enter a critical sate or vice versa which state a system might have in the future. Figure 13 visualises the forecasting service as the trend line based on the values retrieved from the history buffer. Figure 14 shows the class description for the forecasting service.

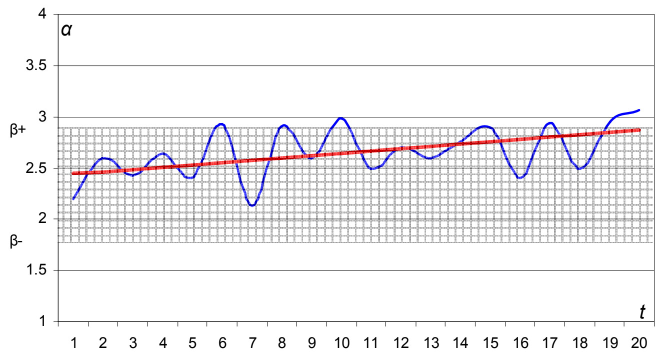


Figure 13: Forecasting

Forecasting
-History Buffer : History Buffer -Bounds : Bounds
+Output() : bool

Figure 14: Class Description of the Forecasting Service

A2.4.1.7 Prediction

This module provides a predictive component which is capable of analysing other related attributes or concepts of interests in order to predict a future condition of the concept of interest that is under supervision. However, predictive features are not envisioned within the first release of the DA. The main reason for this is the current lack of specificity with respect to generic supervision tasks. While it is theoretical possible to implement predictive features in a generic fashion, In reality this often leads to assumptions that are only of statistical value which in turn are not very useful because of (a) the relevance or accuracy is not good enough or (b) the result space is to large to be successfully used. A secondary incentive is the requirement to link related supervision components together to successfully reason over multiple concepts of interests. How and to what extend this will be possible is not yet clear yet and will mainly depend on the overall framework.

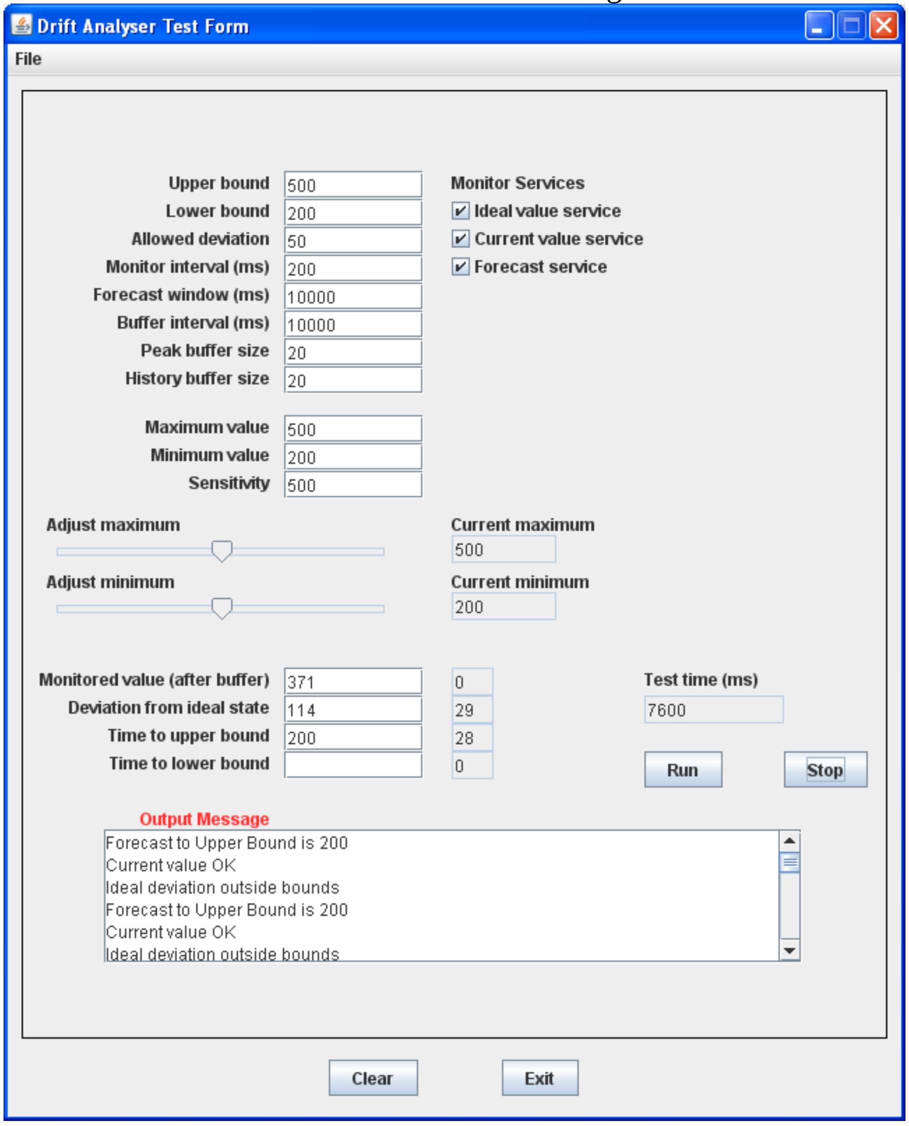
A2.5 Prototype

Although not fully integrated into the overall supervision framework the functionality of the DA may be evaluated utilising the test form shown below. Currently, the ideal state service, the current value service & the forecasting service have been realised and can be (de-)activated by (un-)checking the relevant check boxes. Relevant configuration parameters can be set or loaded from file and can be summarised as:

- Upper / Lower Bound: Boundaries that the system under supervision should remain within.
- Allowed Deviation: Maximum allowed amount of deviation from the ideal value before the deviation is considered a violation.
- History size: The maximum history buffer size.
- Peak size: the maximum peak buffer size.
- Monitor interval: The interval between checking concepts for concept drift.
- Forecast interval: The forecasting window in which the system tests for violations.
- Buffer interval: the maximum time interval for storing related concepts.

Just for generating test values, but not actually as part of the DA component, there are three values that define the random source values that can be generated. There are:

- Maximum value and minimum value: These values specify the bounds that a random value should be generated in.
- Sensitivity: This indicates by how much the initial maximum and minimum values can be changed, by using the slide bars, so that the trend of the current values can be altered. This is not to be confused with the volatility of the system but only reflects the minimum and maximum boundaries a value is generated in.



Drift Analyser Test Form

File

Upper bound	500
Lower bound	200
Allowed deviation	50
Monitor interval (ms)	200
Forecast window (ms)	10000
Buffer interval (ms)	10000
Peak buffer size	20
History buffer size	20
Maximum value	500
Minimum value	200
Sensitivity	500

Monitor Services
- ✓ Ideal value service
- ✓ Current value service
- ✓ Forecast service

Adjust maximum — Current maximum: 500

Adjust minimum — Current minimum: 200

Monitored value (after buffer)	371	0
Deviation from ideal state	114	29
Time to upper bound	200	28
Time to lower bound		0

Test time (ms): 7600

Output Message

```
Forecast to Upper Bound is 200
Current value OK
Ideal deviation outside bounds
Forecast to Upper Bound is 200
Current value OK
Ideal deviation outside bounds
```

Figure 15: DriftAnalyser – Test Form

As well as these, three other parameters can be configured to integrate the component into the overall framework as well as for security purposes. There are:

- Password: A password to access the service to be supervised.
- Service key: A unique key for loading/removal of sub-components.

- Parent: The parent component to which the supervision component is added to.

The parameters of the randomly generated monitored values can be adjusted using the two slide bars whereas the current minima and maxima are constantly shown to the right of the slide bars as illustrated in Figure 15. Below this is an output indicating the monitored states of the system. These are shown as follows:

- The monitored value shows the supervised concept (after the peak buffer). A count of the number of times that the current value falls outside of the allowed upper and lower bounds is indicated to the right of the text box.
- The deviation from the ideal state is the deviation of the current source value from the last ideal value returned from the history buffer. A value is output only if the deviation is greater than the allowed deviation. A count of the number of times that the allowed state is violated is indicated to the right of the text box.
- The time to the upper bound is the amount of time that would be required, as indicated by the current trend, for the ideal value to move outside of the upper bound. A value is output only if the time is inside of the forecast window time. A count of the number of times that the allowed state is violated is indicated to the right of the text box.
- The time to the lower bound is the amount of time that would be required, as indicated by the current trend, for the ideal value to move outside of the lower bound. A value is output only if the time is inside of the forecast window time. A count of the number of times that the allowed state is violated is indicated to the right of the text box.
- The total amount of time that the test has been running for is indicated in the Test Time box.
- There are buttons that can be used to start or stop the test. If you stop the test the values will be reset before it is started again.
- Finally, a text box is used to visualise messages stemming from individual services. Each service will output a message indicating the result of its last evaluation.

The test form is available as an executable jar file and a test configuration file is provided within the same package.

A2.6 Overall Research Directions

Future research directions come from a multitude of different areas and include individual supervision components as well as the overall architectural framework. For instance, what are the technological pre-requisites to build, configure and to interact with supervision pervasions, i.e. which detailed capabilities are required from components and component orchestrations to support (a) monitoring and interaction, but also (b) to model and to implement a supervision instance in an autonomic fashion?

Self-organization is seen as one of the key properties of autonomic systems. Within the context of supervision this translates into the dynamic configurations and maintenance of supervision configurations and, following the pervasive supervision paradigm, supervision subsystems are thus (a pervading) part of such configurations. Thus they have to follow basically the same organizational rules as the supervised sub-systems. On the other hand, interaction and modification of service configurations by exploiting self-organization rules may be used as a mean to interact with those configurations.

From the perspective of security, supervision is of course very critical. Thus dedicated mechanisms need to be realized to perform supervision tasks in the presence of security demands. Moreover, security modules and components are also error prone and as such might be non-functional; add unacceptable performance bottlenecks, etc. Thus security is by itself an application area for supervision techniques. Resolving this mutual relationship is another foreseen research direction.

Finally, situation-awareness is not yet properly reflected within current supervision approaches. Connecting the overall supervision architecture to relevant contextual sources could provide a new generation of supervision pervasiveness that is not yet possible. For instance, for a supervision component to monitor the temperature of a room it needs to be connected to a temperature sensor. However in order to supervise it, it also needs to be connected to an actuator unit that allows changing the room temperature in some way or the other. Now, for a system to know that e.g. the window has been opened in order to freshen up the air allows additional reasoning about if, how, when and what countermeasures should be employed. How such situation-awareness can be achieved and how it can be successfully exploited for supervision is yet another open issue to be investigated in more detail.

A2.7 Conclusions

This work deals with various approaches and results related to pervasive supervision and provides the foundation for a supervision mechanism that is based on drift behavior. In the first part, the closed control loop architectural has been discussed, which is the basic architectural paradigm employed by all supervision systems. While the analyses of concept drift is only one possible mechanism to allow for long term supervision, the concept of interest and the real world problem they reflect are seen as key principles to enable autonomic services to self-evolve in context aware environments.

One of the main problems in this area is that a system under supervision can, at different levels of granularity, contain any type of information. It is therefore not possible to create a generic evaluation function that is capable of evaluating any type of information or concept of interest respectively. This is based on the fact that an evaluation value is likely to be meaningless if it is derived in a generic fashion rather than based on the specific context of the system under supervision. Thus, it may be better to allow a system to try and self organize itself in a way that micro versions of the whole supervision system exist at different levels. If such micro supervision systems maintain a stable state, the overall system should be stable too. On the other hand, if this state changes in any way, a system may recognize this as odd behaviour and may react on this. The main advantage of this approach is that, individual concepts of interests are, on a micro level, more likely to be of primitive types rather than complex structures. If this is the case standardized evaluation criteria's may be employed to (a) assess them and (b) to configure the supervision system. Realizing a current state analysis combined with more advanced forecasting and prediction mechanisms will allow the detection of sudden as well as gradual drift behavior. If embedded in a virtual realization of a system under supervision, such drift behavior could be detected at early stages and effective countermeasures or a fail back mechanism could be invoked on specific components of a system rather than safeguarding the overall system.