



## CASCADAS

---

Bringing Autonomic Services to Life

# Autonomic Communication Elements and the ACE Toolkit

Borbala Katalin Benko (bbenko@hit.bme.hu)  
Budapest University of Technology and Economics

November 25, 2008, Milan



## About the topic

- Autonomic communication is a hot topic
- 2005: 4 sister projects in EU-FP6
  - ANA, Bionets, CASCADAS, Huggle
- CASCADAS:
  - 14 partners (Italy, Germany, France, UK, Ireland, Belgium, Greece, Hungary)
  - Achievements:
    - Designed a unified model of autonomic services
    - Developed and released a platform for autonomic services
      - Sourceforge: acetoolkit
- Goal today at this tutorial:
  - Get known with the ACE model
  - Get known with the ACE Autonomic Toolkit
    - Theory and practice

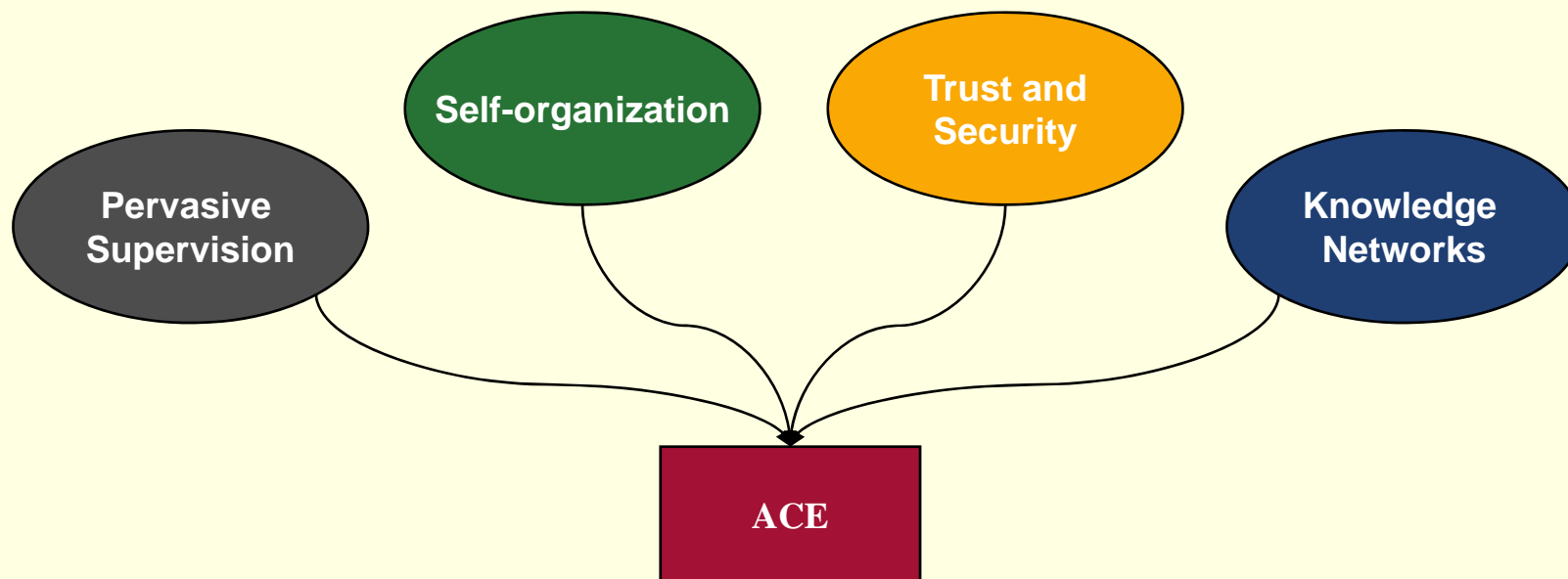


# IBM autonomic computing vision

- Autonomic Systems:
  - Self-management of systems
  - Human operator designs the self-management process
- Main functional areas:
  - Self-Configuration
  - Self-Healing
  - Self-Optimization
  - Self-Protection
- Autonomic communication proceeds on this path

## The CASCADAS vision

- Future is: situation-aware, self-healing, autonomic services
- Project goal: explore problems, design and evaluate solutions
- Common abstraction: Autonomic Communication Element





# Agenda

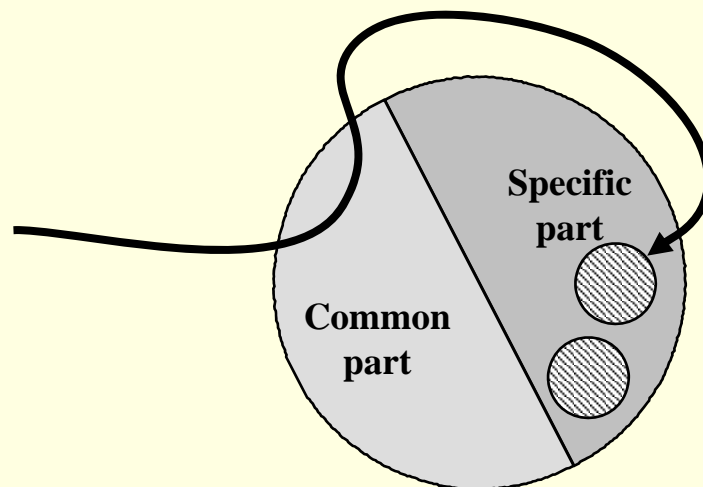
- Get known with the ACE model
  - Motivation
  - Conceptual model
  - Architecture and details
- How the four research areas were plugged in
- Get known with the ACE Toolkit
  - Capabilities
  - How to use
  - Settings, examples



# The ACE model

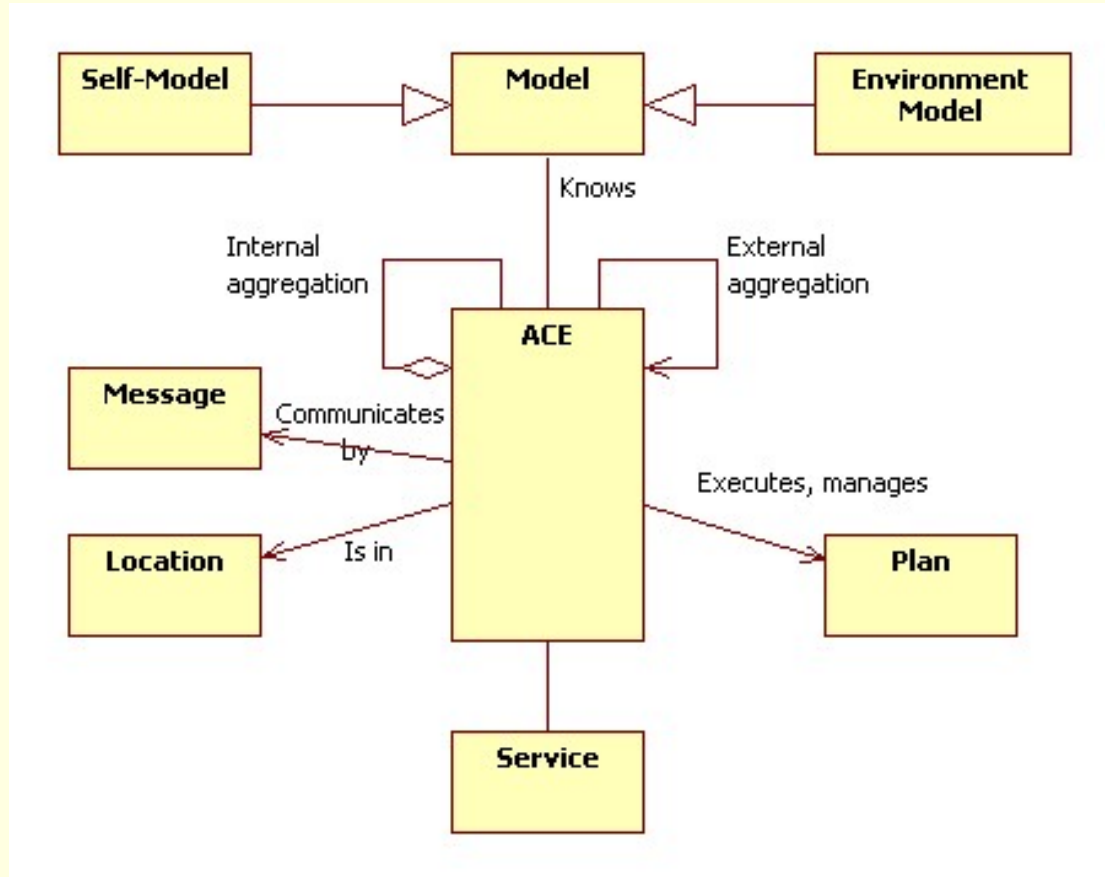
## Specific part and common part

- The ACE consists of two parts:
  - Common part: available in all instances
  - Specific part: varies from ACE to ACE
    - Can be discovered through the common part



Specific functionality

## (Early) conceptual model





# ACE in a nutshell

- The primary goal of ACEs is to use/provide services (service eco-system)
- Situation-aware, context-aware:
  - monitors the internal and external environment
- Adaptive and self-optimizing
  - Adapts itself to the changing conditions
- Self-aware
  - Knows its capabilities and how to adapt
- Self-organizing
- Self-healing
  - Through adaptation
  - Through supervision
- May use knowledge networks
- May use security
  
- Let's see how it's realized



## Competitive, on-demand, semantically aided service discovery

- How do ACEs find each other?
    - No central registry
    - Peer-to-peer overlay (just for this purpose)
    - Message based: GN (goal needed) – GA (goal available)
  
  - On demand
  - Semantically aided
  - Competitive
- 
- Why is this good?
    - Robust (peer-to-peer)
    - Flexible (ACEs may change their services any time)
    - Open



# Contracts and role based group communication

- For cooperation, ACEs can “contract” each other (form a group)
  - This creates a direct communication channel between partners\
    - Bilateral
    - Multilateral
  - Roles
  - Role based addressing
  - Communication is till message based
- Why is this good?
  - A complex service can be implemented by an interactive collective of ACEs (instead of a single one)
  - The contract adds guarantees
  - Multi-lateral, so not just the user-provider (client-server) model is possible



## So, communication is

- Service discovery: peer-to-peer
  - Flexible, open
- Service provisioning: contract based, direct communication
  - Reliable, efficient
- Why messages?
  - An alternative would be synchronous communication (e.g. remote method call)
  - Asynchronousness
    - Helps decoupling the components
    - Helps dealing with error resulting from the highly dynamic environment
    - Is more transparent (e.g. an ACE may move in an unnoticed way, during interaction)



## Internal working logic made explicit

- Internal working logic: application logic, which is usually implemented as program code
- We made this explicit
  - XML format for describing the current application logic
  - XML format to describe the autonomic part (how to adapt, etc.)
- So: a new layer between the black box layer (input-output of the ACE) and the program code
  
- Why is this good?
  - It is observable, supervisable
  - Abstracts the program code → more self-knowledge → more place for reasoning, optimization



# Separation of the current behaviour and the autonomy

- Plan: current behavior
  - When and what to do
  - How to react to an incoming message
  - Implementation: EFSM
- Self-Model: the core autonomy
  - How to generate plans
  - How to detect situation changes
  - How to modify/replace/remove plans
  - Implementation: a lot of rules
- Why is this good?
  - Separates the current (short-time) and the long-term behavior
  - Adaptation, situation-awareness etc. comes naturally
  - Incomplete planning is also possible



# Open autonomy

- The autonomic part is defined by the ACE developer!
  - Plan generation rules
  - Plan modification rules
  - What to listen to, what triggers the reasoning
- The rules can define any kind of autonomic heuristics
  - Adaptation
  - Self-healing
  - Optimization
  - Or other, application-specific ones



## “What” and “How” got separated

- Abstract ACE layer (Self-model and Plan): defines the “what to do”
  - Symbolic functionality invocation
- Actual program code of a functionality: takes care of “how to to it”
  - Plugged into the system transparently, though XML descriptors
- Why is this good?
  - Functionalities can be easily added to an ACE
  - High-level processes (e.g. reasoning) is not disturbed by the details



## Intrinsic ability for simulation

- Functionalities are running in a sandbox
  - They reach the outside world through references
  - If we pass “fake”, simulated references, the functionality call will automatically have its effects on the simulated environment instead of the real one
- Why is this good?
  - Planning: select the best way through simulation
  - Supervision, error detection: check if the program code part works properly



## Migration and cloning

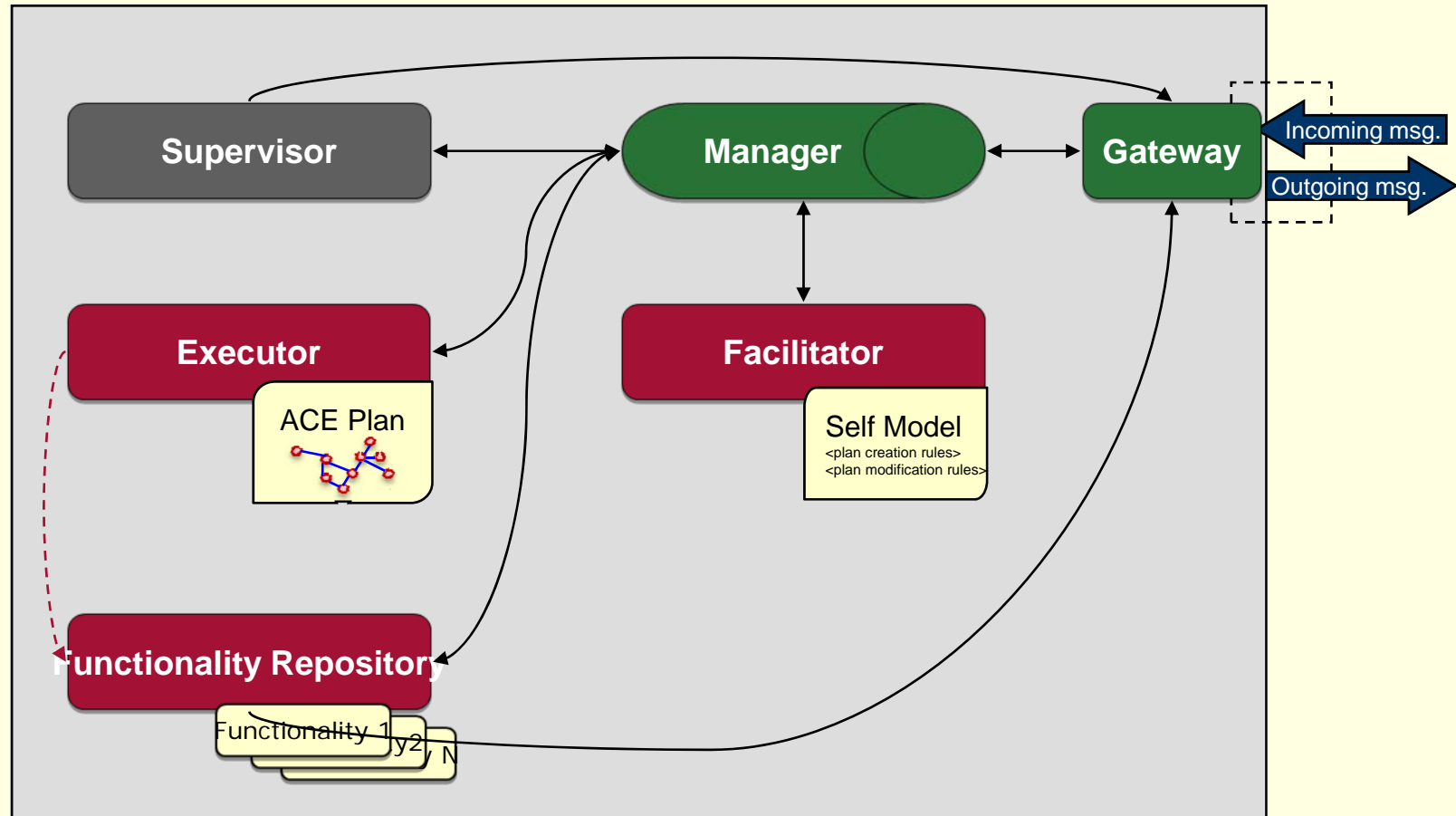
- ACEs can freely migrate between environments
  - Autonomic decision
  - Restriction: physically bound functionalities (e.g. sensor handler)
- ACEs can clone themselves
  - Autonomic decision
  - Restriction: dedicated, non-duplicable functionalities (e.g. robot hand)
- Why is this good?
  - Load balancing
    - Move nearer to users
    - Duplication when load is high
  - Optimization
    - Move into a more optimal environment to provide a better service



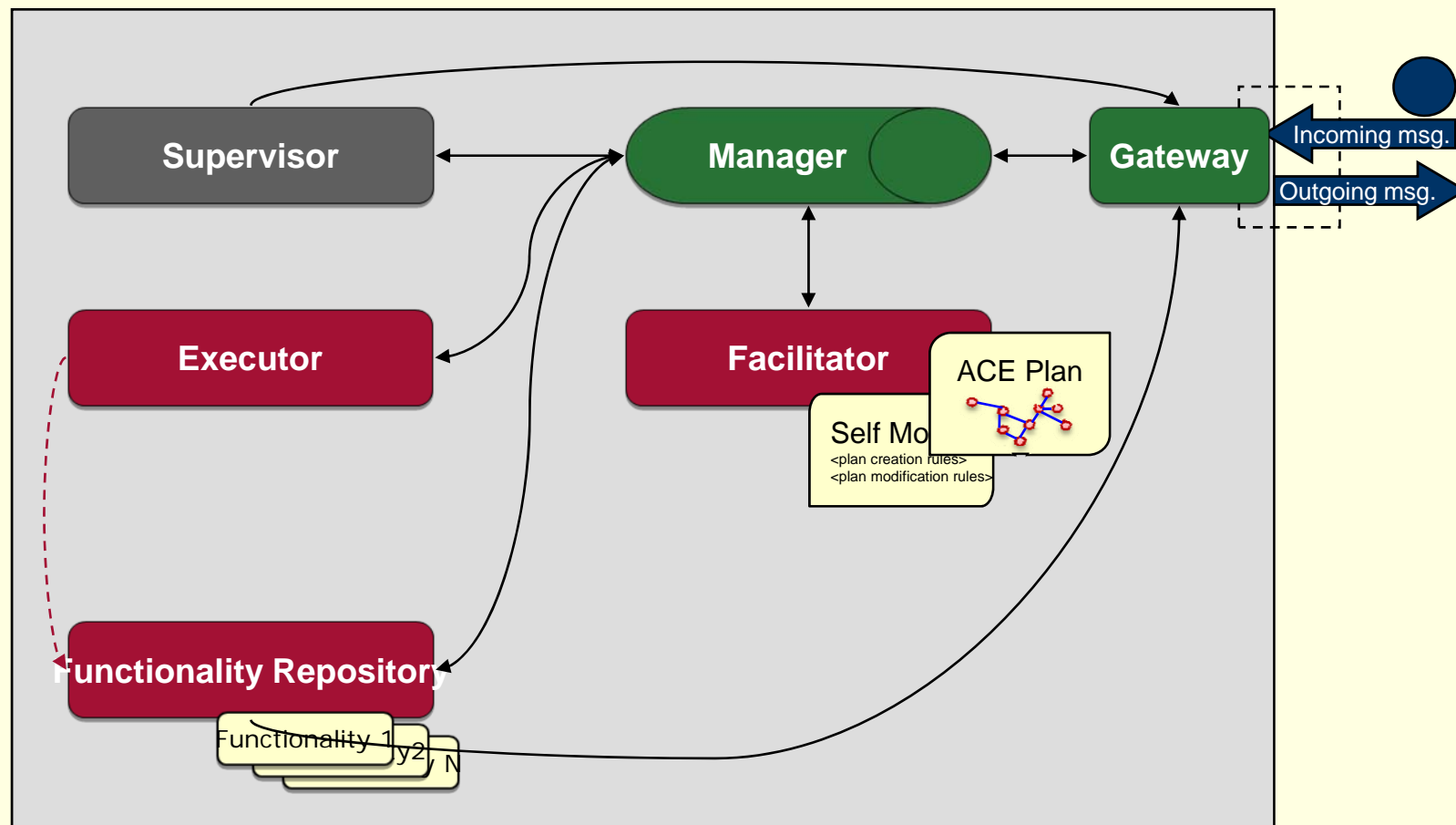
# Organ model

- Functional decoupling of an ACE: organs
  - Name is biologically motivated ☺
  - Organs are orthogonal to each other
  - Together they form an autonomic element
- ACE organs:
  - Facilitator: takes care of the Self-Model, creates/modified/removes Plans
  - Executor: executes the Plans
  - Functionality Repository: hosts the functionality codes, sandbox
  - Gateway: takes care of communication facilities (peer-to-peer overlay and direct channels)
  - Manager: local inter-organ communication, life-cycle management
  - Supervision organ: monitoring and healing

# ACE organs



# ACE organs





## So, this was the ACE component model

- Three concepts:
  - Self-Model
  - Plan
  - Functionalities
  
- Five organs:
  - Facilitator – the intelligent core
  - Executor – the one who executes the plan
  - Functionality Repository – the sandbox for program code
  - Manager – internal matters
  - Gateway – external communication
  - Supervision organ – observation and interference point



## One more word about functionalities

- Common functionalities: available in all ACE instances. E.g.
  - Send a simple GN or GA
  - Send a Service Usage Event to another ACE over a contracted connection
  - Update the local context
  - ...
- Specific functionalities: varies from ACE to ACE
  - Developed by the designer of the specific ACE
  - Application dependent logic is there



How to plug in the four scientific concepts?



## Place of Supervision

- Supervision was implemented as an ensemble of ACEs (six roles)
- The Supervisor Pervasion monitors the ACE under supervision through the Supervision organ
  - Sees all internal and external messages
  - Sees all Plans
  - Gets notification about each state change and action in the plan
- When needed, interferes through the Supervision organ
  - Places messages into the ACE under supervision
  - Or restarts the faulted organ



## Place of Self-organization

- One place: Self-Model
  - Contracting
  - Migration and Cloning
- Other place: clustering
  - Active or passive clustering is possible
  - (the Gateway and a Common functionality are involved)



## Place of Security and Trust

- Security is implemented as a service
- Must be contracted in order to use.
  
- Available now:
  - Encryption
  - Decryption
  
- Is possible to implement later:
  - Reputation based trust



## Place of Knowledge Networks

- KN is implemented as a service
  - Knowledge Atom ACEs: contain knowledge
  - Knowledge Container ACEs organize the knowledge
- The KN can be queried



## Organs in details



## Place of Knowledge Networks

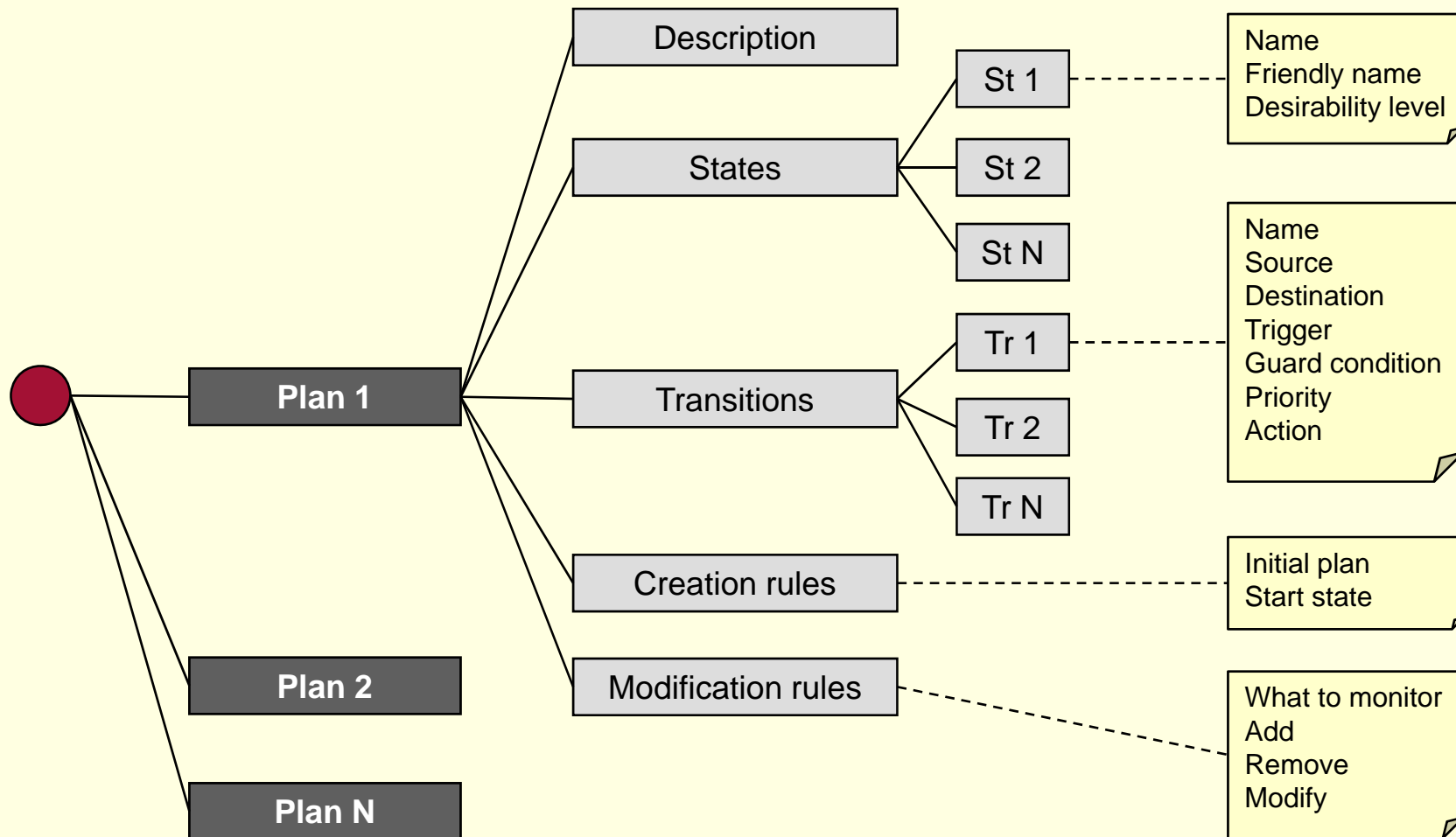
- KN is implemented as a service
  - Knowledge Atom ACEs: contain knowledge
  - Knowledge Container ACEs organize the knowledge
- The KN can be queried



# Facilitator and the Self-Model

- Overall operation:
  - On bootstrap: loads the self-model
  - During life-cycle:
    - Monitors the external and internal environment
    - Deducts, applies rules (e.g. to create/modify/remove Plans)
- Self-model:
  - Set of possible states and transitions
  - Rules to combine states and transitions
- Rules can be:
  - Non-conditional (e.g. default Plan)
  - Dependent on events
  - Dependent on the context

# Sample Self-Model



```

<plan id="Plan1" default="true">
  <description></description>
  <states>
    <state id="state0">
      <friendly_name>set_timer_for_gn_dresscode</friendly_name>
      <desirability_level>1</desirability_level>
    </state>
    <state id="stater1">
      <friendly_name>ready</friendly_name>
      <desirability_level>1</desirability_level>
    </state>
    ...
  </states>
  <transitions>
    <transition id="tr0">
      <source>state0</source>
      <destination>stater1</destination>
      <priority>1</priority>
      <trigger>@auto</trigger>
      <guard_condition></guard_condition>
      <action>add_timer_service(timerId=gnDresscode, millis=1000)</action>
    </transition>
    <transition id="tr1">
      <source>stater1</source>
      <destination>state2</destination>
      <priority>1</priority>
      <trigger>@auto</trigger>
      <guard_condition></guard_condition>
      <action>gn_sender_service(goalName=dresscode_simulation,
        myAddress=?globalSession://aceAddress)</action>
    </transition>
    ...
  </transitions>

```



```

<creationRuleML>
  <Assert>
    <And>
      <Atom closure="universal">
        <Rel>createState</Rel>
        <Ind>state0</Ind>
        <Ind>state1</Ind>
        ...
      </Atom>
      <Atom closure="universal">
        <Rel>initState</Rel>
        <Ind>state0</Ind>
      </Atom>
      <Atom closure="universal">
        <Rel>createTransition</Rel>
        <Ind>tr0</Ind>
        <Ind>tr1</Ind>
        ...
      </Atom>
    </And>
  </Assert>
</creationRuleML>

<modificationRuleML>
  <Assert>
    <And>
      <Implies closure="universal">
        <Atom closure="universal">
          <Rel>weather</Rel>
          <Ind>rainy</Ind>
        </Atom>
        <Atom>
          <Rel>deleteTransition</Rel>
          <Ind>tr3</Ind>
        </Atom>
      </Implies>
      ...
    </And>
  </Assert>
</modificationRuleML>
</plan>

```

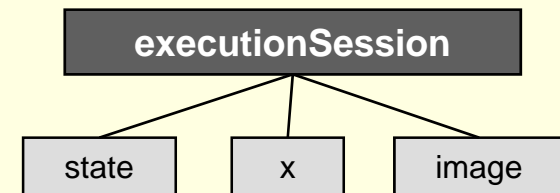
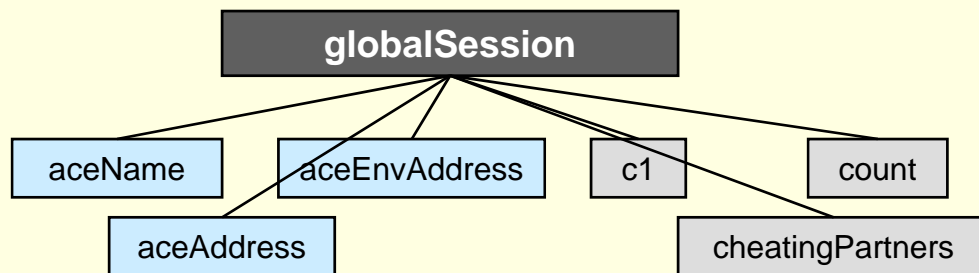


# Executor and the Plan

- Overall operation
  - At bootstrap: initializes the Ace resources (the “context”)
  - During lifecycle:
    - Receives Plans
    - Executes the Plans
- Plan:
  - Active state
  - Outgoing transitions
    - Trigger
    - Guard condition
    - Priority
    - Action
- ACE resources: globalSession, executionSession

## Session objects

- GlobalSession: for the life time of the ACE
  - ACE name
  - ACE address
  - ACE environment address
  - Any application-specific key-value pair
- ExecutionSession: for a single plan only
  - Any application-specific key-value pair





## Transitions in the plan (1)

```
<transition id="tr0">
  <source>state0</source>
  <destination>state1</destination>
  <priority>1</priority>
  <trigger>@auto</trigger>
  <guard_condition></guard_condition>
  <action>add_timer_service(timerId=gnDresscode, millis=1000)</action>
</transition>
```

@auto: no trigger  
 @wait: any message  
 p1.MyEvent: specific message

```
<transition id="tr1">
  <source>state1</source>
  <destination>state2</destination>
  <priority>1</priority>
  <trigger>@auto</trigger>
  <guard_condition></guard_condition>
  <action>
    gn_sender_service(
      goalName=dresscode_simulation,
      myAddress=?globalSession://aceAddress)
  </action>
</transition>
```

simple String

?globalSession://key  
 ?executionSession://key

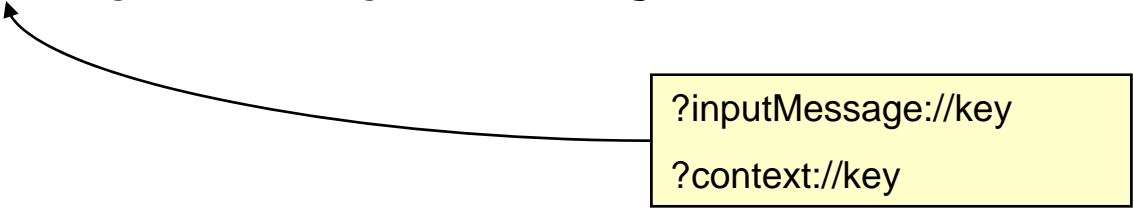
## Transitions in the Plan (2)

```

<<transition id="tr2">
  <source>state2</source>
  <destination>state0</destination>
  <priority>2</priority>
  <trigger>cascadas.ace.event.TimerExpiredEvent</trigger>
  <guard_condition>EQUALS(?inputMessage://timerId,gnDresscode)</guard_condition>
  <action></action>
</transition>

<transition id="tr3">
  <source>state2</source>
  <destination>state3</destination>
  <priority>1</priority>
  <trigger>cascadas.ace.event.GoalAchievableEvent</trigger>
  <guard_condition>EQUALS(?inputMessage://goalName,dresscode_simulation)</guard_condition>
  <action>generic_add_to_execution_session_service(
    dresscode_simulation_serviceName=?inputMessage://serviceName,
    dresscode_simulation_providerAdress=?inputMessage://providerAddress)
  </action>
</transition>

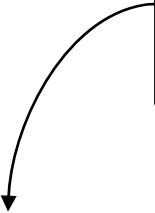
```



## Transitions in the Plan (3)

```
<transition id="tr4">
  <source>state3</source>
  <destination>state4</destination>
  <priority>1</priority>
  <trigger>@auto</trigger>
  <guard_condition></guard_condition>
  <action>
    contract_n_establishment_service(
      user=?globalSession://aceAddress, provider=?inputMessage://providerAddress,
      contractId=dresscode_simulation_Contract)
  </action>
</transition>
```

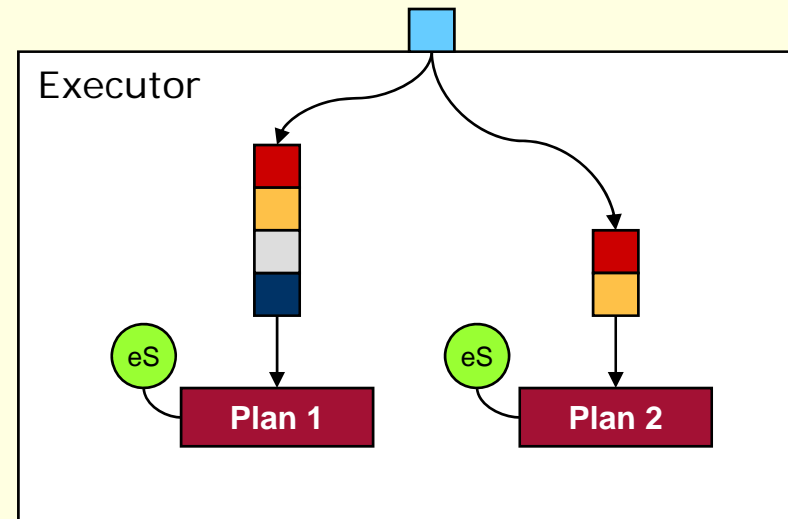
?inputMessage refers to the last inputMessage (so it's also valid in trigger-less transitions)



```
<transition id="tr6a">
  <source>state6a</source>
  <destination>state6</destination>
  <priority>1</priority>
  <trigger>@auto</trigger>
  <guard_condition></guard_condition>
  <action>service_caller_service(
    contract=?executionSession://dresscode_simulation_Contract,
    serviceName=?executionSession://dresscode_simulation_serviceName,
    targetRole=provider)
  </action>
</transition>
```

## Plans: normal, child, daemon

- Parallel plans are possible
- Each plan has:
  - Input queue
  - Execution session
- User plans:
  - Normal plan
    - New execution session, New (empty) input queue
  - Child plan
    - Shared execution session, Copy of parent's input queue
- Built-in daemon plan
  - Context acquisition (works up the incoming context info)

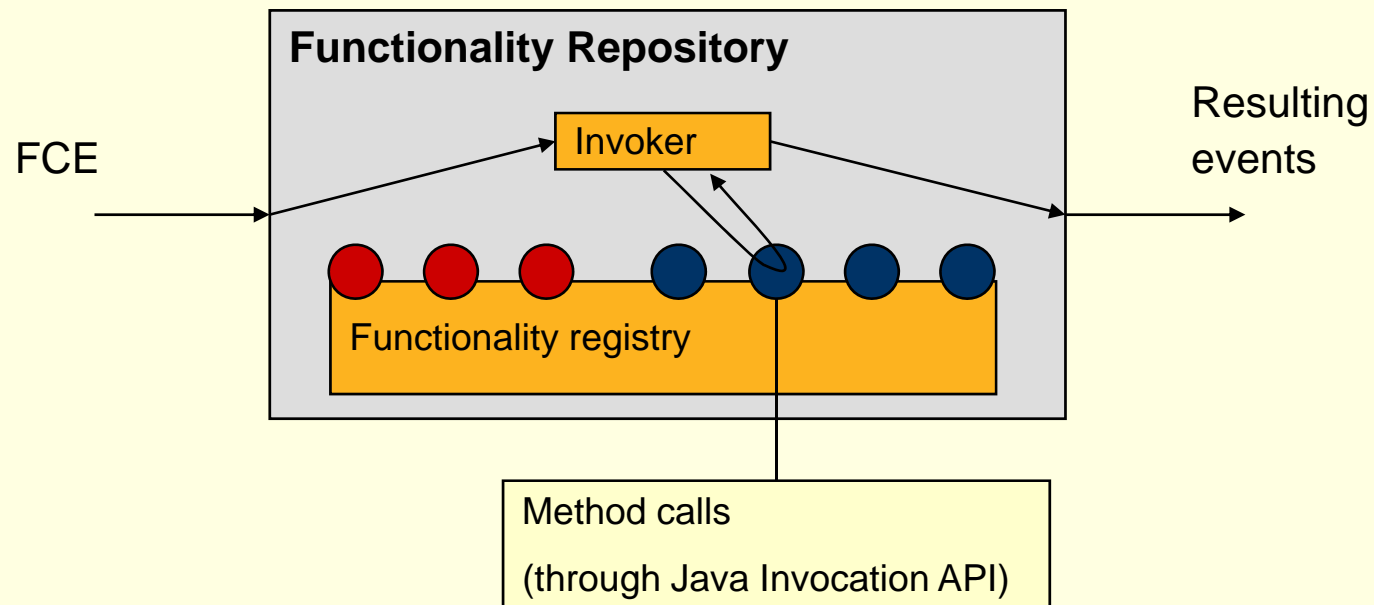




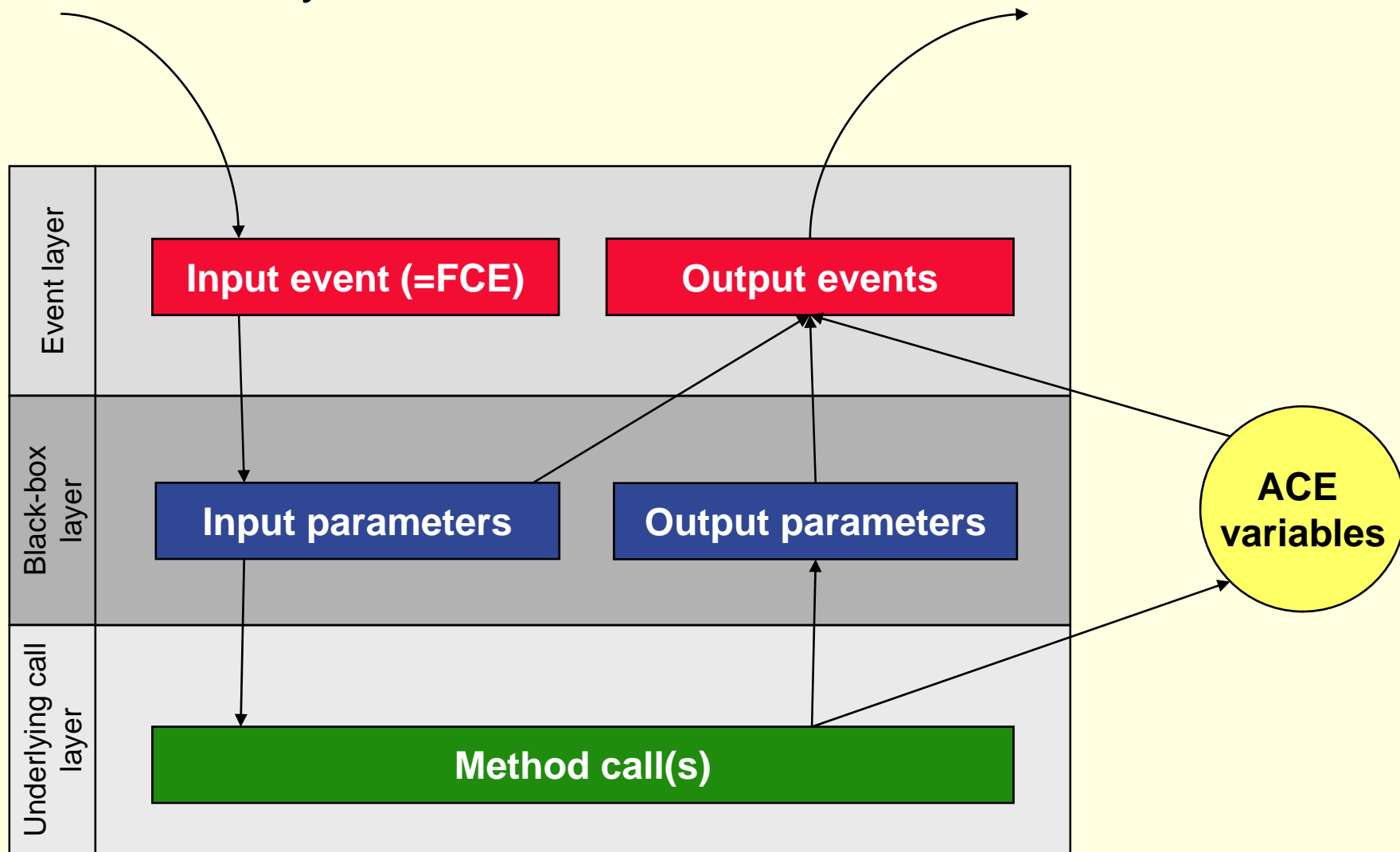
# Functionality Repository

- Overall operation:
  - At startup: loads the functionalities (based on their XML descriptors)
  - During ACE life time:
    - Receives call requests (from Plan actions)
    - Performs calls, sends the output events
- Call types:
  - Synchronous
  - Asynchronous
- Functionality model
  - Given as XML descriptor (functionality descriptor)
  - Functionality name (unique)
  - Details (input, output, classes/methods to invoke)

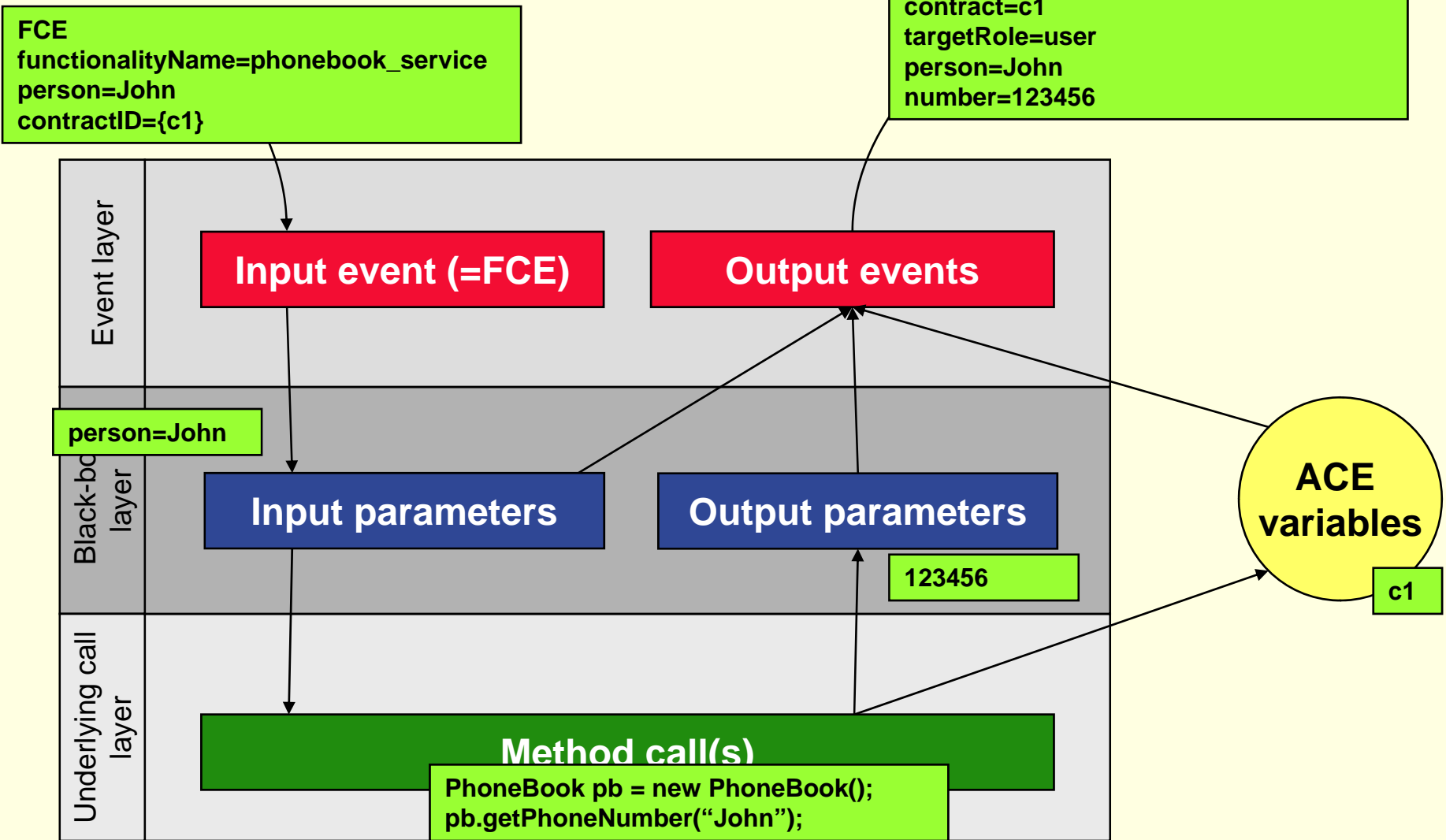
# Call flow



# Functionality Model



# Functionality Model





# A real Functionality Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE functionality SYSTEM "../..../dtd/functionality.dtd">

<functionality id="simple-phonebook-lookup-service">

  <black-box-description>
    <input>
      <param name="person" type="java.lang.String"/>
    </input>
    <output name="phone number" type="java.lang.String"/>
  </black-box-description>

  <simple-call-details
    class-name="example.testfunctionality.MyPhoneBook"
    method-name="lookupPhoneNumber"/>

  <output-event-mappings>
    <mapping
      event="cascadas.ace.event.ServiceResponseEvent"
      target-role="user">
      <value ref="person"/>
      <value ref="phone number"/>
    </mapping>
  </output-event-mappings>

</functionality>
```

```
<functionality id="advanced-phonbook-lookup">
  <black-box-description>
    <input>
      <param name="surname" type="java.lang.String"/>
      <param name="first name" type="java.lang.String"/>
      <param name="phone book file" type="java.lang.String"/>
    </input>
    <output name="phone number" type="java.lang.String"/>
  </black-box-description>
  <complex-call-details>
    <call class-name="example.testfunctionality.MyPhoneBook" method-name="loadPhoneBook" >
      <arg ref="phone book file"/>
    </call>
    <call class-name="example.testfunctionality.MyPhoneBook" method-name="lookupPhoneNumber">
      <arg ref="surname"/>
      <arg ref="first name"/>
      <return ref="phone number"/>
    </call>
  </complex-call-details>
  <output-event-mappings>
    <mapping event="cascadas.ace.event.ServiceResponseEvent" target-role="user">
      <value ref="phone book file"/>
      <value ref="surname"/>
      <value ref="first name"/>
      <value ref="phone number"/>
    </mapping>
  </output-event-mappings>
</functionality>
```



# Output event generation

- Mapping: XML
- Mapper: own class
  - implements the `OutputEventManager` interface
- Mappers and Mappings can be mixed

```
<output-event-mappings>
  <mapping event="cascadas.ace.event.ServiceResponseEvent" target-role="user">
    <value ref="x"/> // input-output
    <value ref="?globalSession://y"/> // global session
    <value ref="?executionSession://z"/> // execution session
    <value ref="?contextData://w"/> // context
    <value ref="?callWideContext://v"/> // call wide context
  </mapping>

  <mapper mapper-class="example.MyOutputEventManager"/>
</output-event-mappings>
```



# Sample Mapper

```
public class MyOutputEventMapper implements OutputEventMapper {

    /** Sends the same message to all listed roles one by one */
    public Set<Event> createOutputEvents(
        CallParameters params,
        Session executableBackground,
        Session globalBackground,
        CallWideContext callWideContext) {

        Set<Event> ret = new HashSet<Event>();

        Set<String> names = (Set) callWideContext.get("names");
        for (String name : names) {
            ServiceResponseEvent evt =
                new ServiceResponseEvent((Contract) params.get("contract"));

            evt.setDestinationRole(name);
            evt.addParam("message", params.get("notification message"));
            ret.add(evt);
        }
        return ret;
    }
}
```



## Advanced features in functionalities

- Instances:
  - No state information is preserved between calls!
    - Field values are lost!!!
  - Use the sessions (global or execution) to preserve data
- Access to ACE resources: implement the concerning interface
  - SessionAware
  - CallWideContextAware
  - ThreadPoolAware
  - LogAware
  - OutputDeclaring

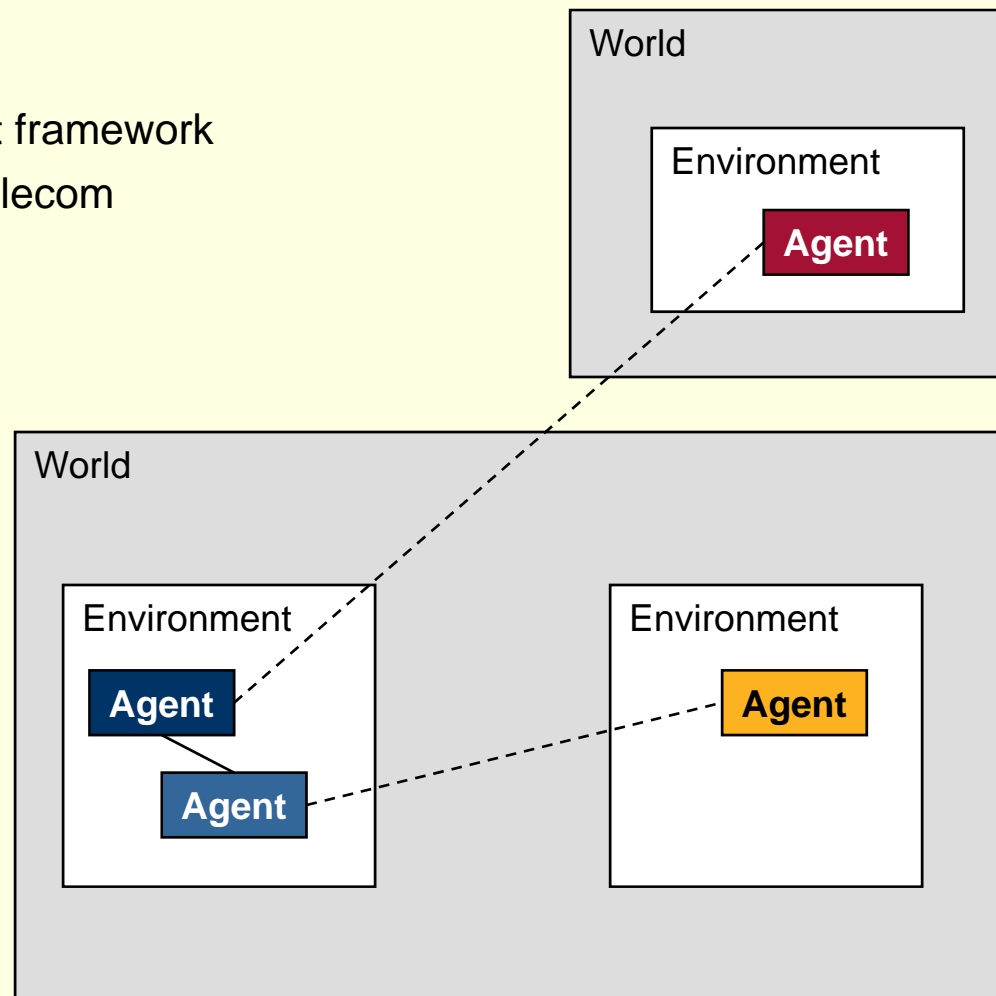


# Gateway

- In charge of communicating with the external world
- Overall operation:
  - At startup: sets up facilities
  - During ACE life time:
    - Sends/Receives p2p overlay messages
    - Sets up direct communication channels, sends/receives messages over them
- P2P overlay: REDS
- Direct communication: DIET Agents

## Intermezzo: DIET Agents

- Robust, scalable, reliable agent framework
- Development lead by British Telecom
- Sourceforge, GPL licence
- World
- Environment
- Agent
- Mirror agents
- ACE
  - 1 ACE = 1 agent
  - Only agent in its env.





# Manager

- Consists of:
  - Life-cycle manager: startup, migration, cloning, shutdown
  - Bus: internal, subscription based message dispatcher



## Supervisor organ

- Plug point for external supervisors
- Provides tools to observe the internal matters (to make sure everyone is playing along)
  - Internal messages (Bus traffic)
  - External messages (Gateway traffic)
  - Self-Model
  - Plans
  - State changes in plans, actions invoked
- Provides tools to
  - suppress,
  - insert, or
  - modify messages.



# ACELandic

- A language to make developer's life easier
  - No manual creation of plans
  - No manual creation of functionality descriptors
- A procedural language
  - (Translates into self-model XML and into functionality descriptor XML)



## Sample ACELandic code

```
repository {
  emitting myFunctionality [
    display-contract : cascadas.ace.session.Contract,
    destination : java.lang.String] |-> package1.MyClass : myMethod;
} // repository

selfmodel provider {
  initial plan contracting {
    forever {
      reveal data;
      accept -> operational.contract;
      forever {
        call myFunctionality[
          display-contract <- global/operational.contract,
          destination <- display];
        on cancellation operational.contract { exit; }
        wait 2000;
      } // forever
    } // forever
  } // plan
} // selfmodel
```

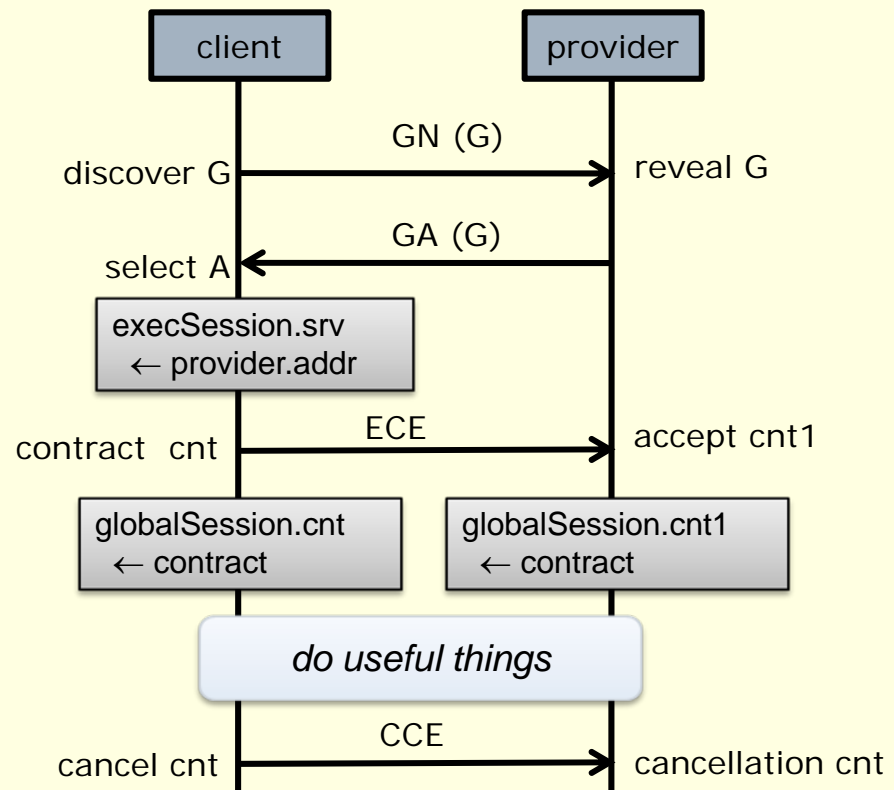
# Service discovery and Contracting in ACELandic

## client

```
discover G {
  select srv { . . . }
}
contract cnt [
  role1 <- self/address,
  role2 <- local/srv];
// do useful things
cancel cnt;
```

## provider

```
reveal G;
accept -> cnt1;
// do useful things
cancellation { . . . }
```





## What (else) can you do in ACELandic?

- Repository declarations
  - Aliases (less typos)
  
  - Initial plan
  - Other plans
  - GN/GA
  - Message sending and receiving
  - Value assignments (globalSession, executionSession)
  - Loops
  - Conditions
  - Functionality calls
  - Timer handling
  - Waiting
  - Handling broken contracts
- Not possible (yet):
- Plan modification rules (replacement is possible)



## The core ACE Toolkit



# The ACE Toolkit

- Core ACE Toolkit
  - Everything that is needed to create and run ACEs
- Extension libraries:
  - Supervision
  - Self-organization (clustering)
  - Security
  - KN
- Helper tools:
  - ACELandic compiler
- Demo Applications
  - Simple demos
  - Complex demo: runs on 100+ PCs (ICL testbed)



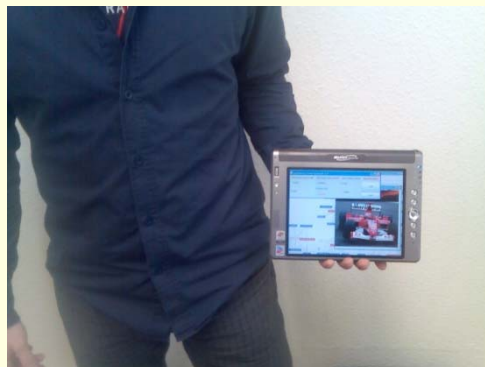
# Hardware and software requirements

- Java 1.5 (recommended: 1.6) and networking
- Full version runs on (tested on):
  - PC
    - Windows
    - Mac (Java 1.5!)
    - Unix
  - Mobile devices
    - Minicomputers, palmtops, ...
    - Nokia N800 mobile phone
    - Google Android (emulator)
- TinyACE (limited functionality) runs on: JavaME enabled mobile phones

# ACE Toolkit on some mobile devices



IBM X41 Tablet PC



Motion Computing LS800 Tablet PC



Sony Vaio VGN-U71P



Pannel PC  
Wincomm WLP-6820



OQO 1+



Google Android  
(emulator)



## Toolkit directory structure

- Conf
  - settings.properties
  - commonfunctionalities
  - aces
    - ace\_type\_1
    - ace\_type\_2
  - commonbehaviours
- Misc
  - Chainsaw config, javadoc and jar config
- Lib
- Src
- build.xml



## Directory structure of an ACE

- conf/aces/type\_1
  - ace\_type.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <comment></comment>
  <entry key="model">./conf/aces/type_1/model/selfmodel.xml</entry>
  <entry key="functionalDescription">./conf/aces/type_1/repo/</entry>
</properties>
```

- repo folder
  - functionality1.xml
  - functionality2.xml
  - ...
- model folder
  - selfmodel.xml
- Specific functionalities:
  - Anywhere in the classpath



## aces.xml

- Defines
  - Which ACEs to instantiate
  - What parameters to pass to them

```
<?xml version="1.0" encoding="UTF-8"?>
<aces>
  <ace name="SystemMonitor" type="conf/aces/system_monitor/ace-type.xml"/>

  <ace name="automigrator_cpu@95%" type="conf/aces/automigrator/ace-type.xml">
    <param id="max-cpu-usage" type="int">95</param>
  </ace>
  <ace name="automigrator_cpu@85%" type="conf/aces/automigrator/ace-type.xml">
    <param id="max-cpu-usage" type="int">85</param>
  </ace>

  <ace name="BruceWillis" type="conf/aces/filmstar/ace-type.xml">
    <param id="accent" type="java.lang.String">american</param>
    <param id="breathePerMinute" type="int">10</param>
  </ace>
</aces>
```



# settings.properties

- General settings
  - Name of aces.xml
  - Location of common functionalities
  - OS-dependent network settings (should be customized on Mac)
- Logging
  - Level should be customized
- DIET settings
  - Important if more than one environments or more than one machines
- REDS settings
  - Important if more than one machines are involved



Let's see some demos



# Practice



## How to create an ACE?

1. Design how it will work.
2. Write the specific functionalities. Include the compiled version in the classpath.
3. Create the home folder of your ACE type under conf/aces.
  - Create a repo folder in the home.
  - Create a model folder in the home.
  - Create ace\_type.xml.
    - Don't just copy, pay attention to the referred path.
4. Create XML descriptor for specific functionalities, and place them to the repo folder.
5. Create self-model. Place it to the model folder.
6. Edit aces.xml to create an instance (or more) from your ACE type.
7. Start the application.
  - Either directly (cascadas.ace.AceFactory)
  - Or via Elvis (com.btexact.diet.elvis.Elvis)



# Practice

- Create two ACEs
- Quote collector
  - Looks for a “quote” service (GN)
  - When finds one
    - Sends a ServiceCallEvent to it (no parameter)
    - Receives the ServiceResponseEvent
      - Extracts the “quoteText” from it
      - Prints it to the screen
- Quote provider
  - Answers “quote” GNs with GA
  - When a SCE arrives, sends back a quote
    - It’s a SRE, containing a parameter “quoteText”



**Thanks!**  
**Questions?**

More info:

<https://sourceforge.net/projects/acetoolkit/>

<http://cascadas-project.org/>

Borbala Katalin Benko, [bbenko@hit.bme.hu](mailto:bbenko@hit.bme.hu)