CASA USER MANUAL

Rob Kremer
Department of Computer Science
University of Calgary
2500 University Dr.
Calgary, Alberta, Canada, T2N 1N4
Email: kremer@cpsc.ucalgary.ca

July 21, 2014

Abstract

CASA (Collaborative Agent Systems Architecture) is a framework for writing collaborative agents. CASA aims to support many different agent conversational paradigms, such as social commitment, BDI (Belief, Desire, Intention) and ad hoc models, and to support various standards such as the FIPA standard. But CASA also aims to avoid forcing the programmer to commit to any particular paradigm. This paper summarizes the structure of agents as well as the basic processing tasks of agents at runtime.

Contents

1 Introduction

CASA (Collaborative Agent Systems Architecture) is a framework for writing collaborative agents. CASA aims to support many different agent conversational paradigms, such as social commitment, BDI (Belief, Desire, Intention) and ad hoc models, and to support various standards such as the FIPA standard. But CASA also aims to avoid forcing the programmer to commit to any particular paradigm.

CASA is written in Java 6.0 and uses Armed Bear Common Lisp [Common-Lisp.net, 2011] for ontology, conversation and policy declarations, as well as for scripting and run-time control. CASA programmers may define agent behaviour in terms of either Java or Lisp code, and have that behaviour called upon by policies defined in either Java or Lisp code.

CASA agents interpret what to do with incoming and outgoing messages or other events in terms of *policies*, which call on either Java methods or Lisp functions to execute defined behaviour. Policies may be either *global* or embedded in *conversations* which define protocols for exchange of messages. Conversations, in turn, may be nested, and can contain state information. Policies typically identify events using identifiers in *event descriptors* which are usually interpreted flexibly using CASA's built-in hierarchical ontology (see §5).

When an agent isn't interpreting events (such as messages), it is free to perform other tasks. For example, an agent could monitor sensors, do computations, or perform a search while it has no events to interpret. As another example, when a *social commitment* agent interprets messages, it doesn't usually execute behaviour directly, but instantiates *social commitments*; so when a social commitment agent isn't interpreting events, it attempts to discharge it's social commitments by executing specific behaviours to eliminate them.

2 Basic agent structure

Figure 1 depicts the structure of a basic agent. An agent consists of two threads:

1. A thread whose primary function is to listen for messages on a port, to decode them into message objects (see §7), to wrap the message in a message event object, and then to queue them as events to the agent's event queue.

2. A main agent thread that does the things one normally thinks of as an agent doing, including dequeuing events (e.g. received message events) from the event queue, and processing them appropriately.

Agent classes are implemented in a layered fashion (see Figure 2) to allow users of the CASA framework to inherit the appropriate capabilities. All agents run in a thread, so the fundamental (abstract) agent type, `AbstractProcess` inherits from the java.lang package's `Thread` class. `TransientAgent` is the most basic concrete class and adds knowledge about messages, ontologies, etc. The `Agent` class adds persistence capabilities. The remainder of the classes in Figure 2 will be discussed in sections 3 and 14.

3 Basic types of agents

This section explains the different types of agents considered the basic agents. All of these are bundled with CASA. Typically, you would subclass either `Agent` (if you want persistence) or `TransientAgent`.

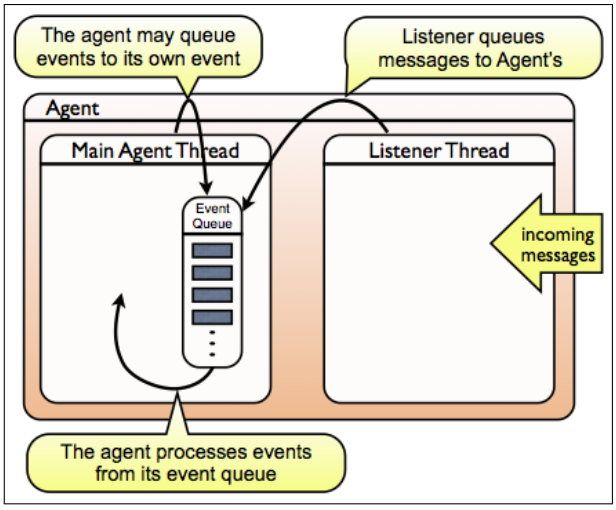


Figure 1: An agent, consisting of two threads.

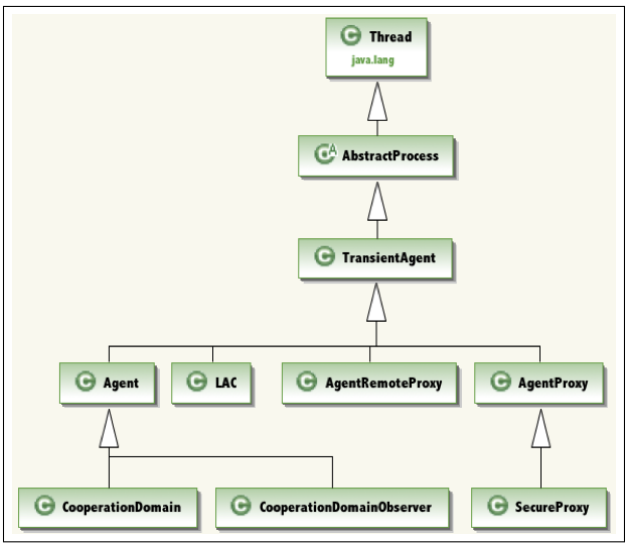


Figure 2: The basic agent class structure.

3.1 AbstractProcess

`AbstractProcess` is actually not an agent at all, since it is not aware of the semantic aspects of agent behaviour. but is a superclass to all CASA agents. It inherits directly from `Thread`, so code executing in the main agent tread can always get a reference to the agent using `(AbstractProcess) Thread.currentThread()`, however the method `AbstractProcess.getAgent()` is recommended, and will return the agent even if the code is executing in a "subthread" of the main agent thread (but one should be cautious, as it may return null if the agent isn't accessable).

`AbstractProcess`'s basic responsibilities are:

- initializing the event queue, and the listener thread (see §**??**).
- running the `eventLoopBody()` method continually until signalled to exit.
- all communications functions through its `sendMessage()` method and related methods.

3.2 TransientAgent

The `TransientAgent` class inherits directly from `AbstractProcess` and is the most primitive form of agent. It adds "semantics" to the base behaviour in interpreting events using policies and conversations. It is also responsible for reading in ontology, initialization files (containing policies and conversation definitions), and script files during agent initialization.

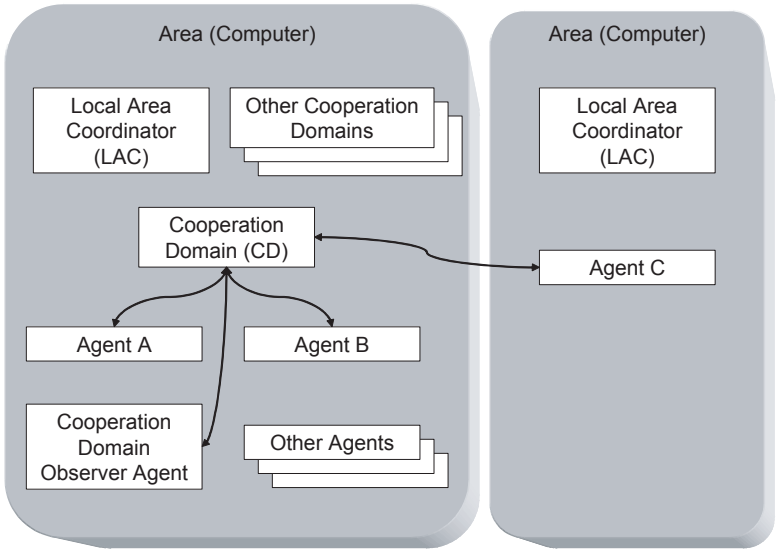


Figure 3: CASA LACs, CDs, and ordinary Agents

3.3 Agent

The `Agent` class inherits director from `TransientAgent` and adds persistence services to the base behaviour. Note that you must actually switch on the persistence for it to work. This can be done by setting the agent initialization Lisp command key parameter `:persistent T`.

3.4 LAC

The `LAC` class inherits directly from `TransientAgent`. An *area* is defined nominally as a single computer (but could be a cluster of several computers, or a partition on a single computer). There is exactly one *local area coordinator* (`LAC`) per area, which is responsible for local coordination and tasks such as:

- registering agents as part of their start-up process
- resolving agent URLs on behalf of other agents (see Section **??**)
- "waking up" a local agent on behalf of a remote agent.

All application agents reside in one or more areas.

3.5 CooperationDomain

As shown in Figure 3, *Cooperation Domains* (CDs) act as a central "hub" for multi-agent conversations. Participants may send messages directly to the Cooperation Domain for point-to-point, multi-cast, type-cast, and broadcast communication within the Cooperation Domain (a group of agents working together on some task). Agents within a cooperation domain may also use the cooperation domain to store persistent data that will permanently be associated with the conversation, giving the conversation a lifetime beyond the transient participation of the agents. Cooperation Domains may also store transaction histories for future playback of the chronological development of the conversation artifacts.

4 Agent activity

An agent has a lifetime, which of course, consists of a beginning, a middle, and an end. These are *initialization*, the *event loop*, and *exiting*. When the agent first starts, it must initialize before it can start normal agent activity (see §4.2). After it is initialized, the agent enters its event loop, where it processes *events* or processes any *commitments* it may have, or just sleeps if it doesn't have anything to do (see §4.3). When an agent is somehow signalled to exit, it may need to take certain actions (such as informing other agents it is exiting, or saving data), and then finish up processing the events in its event queue before actually exiting (see §4.4).

A typical agent will spend the vast majority of its lifetime in the middle, event loop stage, so the event loop is the most important stage to concentrate on. As the name implies, this stage typically focuses on processing *events*. But not all agents are purely server agents, so many kinds of specific agents will also spend much of their time generating events for other agents, or fulfilling other kinds of commitments (such as those on behalf of their users or owners). Even purely server agents may hold commitments to other agents if they do not, or cannot, immediately fulfill requests. So we say that an agent spends its time doing two things: processing events, and processing commitments. Events are explained in §4.1, but CASA is very flexible[1] as to how an agent may handle commitments so an explanation of commitments is deferred to §8.

A central assumption of CASA is that agents may choose to lay anywhere on a continuum between purely reactive and purely contemplative. However, a purely contemplative agent that doesn't react in some way to its environment hardly meets our expectations of an agent at all, so it seems any agent has to react to environmental events to some degree. In CASA, an event can be anything from the receiving, sending, or observation of a message, to a change in physical sensors, to an internal software event such as timer event, to a specific task deferred at an earlier time from an agent to itself to do in the future. See Figure ??.

4.1 Events and handling threads

Events are relatively easy to create. For example, to create a event that will send a message to another agent every minute, one can make use of the `RecurringTimerEvent` class:

```
new RecurringTimeEvent("keepTellingTime", this,
                       60000, 60000) {
  @Override
  public void fireEvent() {
    super.fireEvent();
    sendMessage(ML.INFORM,
      "keepTellingTime",
      url,
      ML.CONTENT,
      new Time(System.currentTimeMillis())
                .toString());
  }
}.start();
```

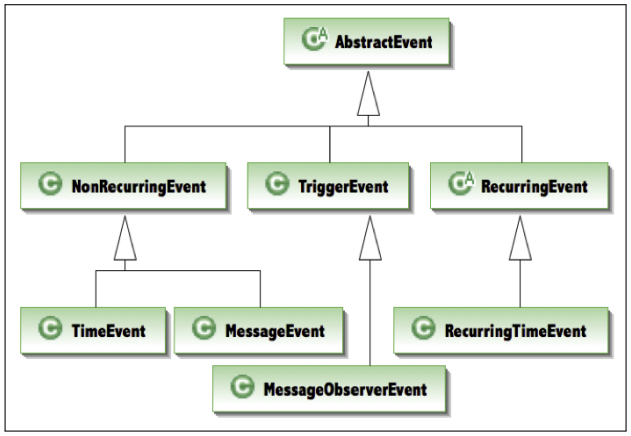


Figure 4: The CASA event classes.

which will result in queueing an event (named "keepTellingTime") to `this` agent's event queue in 60 seconds (to be repeated every 60 seconds) that will sent an *inform* message to the agent at address `url`.

Since a CASA agent's main thread is responsible for dequeueing events and processing commitments, it cannot be stopped to wait for an event to happen, since the waited-for event won't be dequeued (and therefore,

[1] CASA provides for a library of commitment-handling mechanisms

the event won't "happen"). For example, if we are executing in the agent thread (such as processing an event), and we need to get some information from another agent, we would synchronously send a message and wait for a reply:

```
StatusObject<MLMessage> stat = sendRequestAndWait(msg, timeout);
```

which will send the request *msg* and wait *timeout* milliseconds for a response. However, if we do that in the main agent thread, the tread will sleep, and it will never pick up the response message arriving[2]. So, to avoid this issue we need to create a new thread in which to execute the sleeping code, which we can do conveniently by calling the agent's `makeSubthread()` method[3]:

```
makeSubthread(new Runnable() {
  @Override
  public void run() {
    StatusObject<MLMessage> stat = sendRequestAndWait(msg, timeout);
    if (stat!=null && stat.getStatusValue()==0) {
      // process an successful reply
    }
    else {
      // process a failed reply
    }
  }
}).start();
```

Now, this new, independent thread is free to sleep while waiting on the response from another agent. However, if, for some reason, in processing these messages you need to execute some of the code in the agent's main thread, you need to create an evert and place it on the agent's event queue. You can do this easily:

```
agent.defer(new Runnable() {
  public void run() {
    //code to run in the agent's thread.
  }
}
```

which will create an event and immediately queue it to the agent's event queue.

If you need to run this code in the agent's thread and return some result to the current thread, you'll need to specialize the `Runnable` class and carefully handle timing it out and waiting on the event without being fooled

[2] Actually, `sendRequestAndWait()` will detect the problem and throw an exception.

[3] The `makeSubthread()` method is preferable to simply creating a new `Thread` because it will do additional tasks such as making the agent object easily accessible from the tread, and naming the thread regularly to simplify debugging.

by spurious interrupts that may occur during your thread's sleep. All this is actually fairly easy to do:

```
private class MyRunnable implements Runnable {
  public Status result = null;
  public boolean completed = false;
  public MyRunnable() {}
  @Override
  public void run() {
    result = // some result to return
    completed = true;
  }};
MyRunnable runnable = new myRunnable();
defer(runnable);
long timeout = System.currentTimeMillis()+3000; // timeout is 3 seconds
while (System.currentTimeMillis()<timeout) {
  try { // in this case we, want WANT the interrupt exception to occur, if it didn't the event timed out.
    sleep(timeout-System.currentTimeMillis());
  } catch (InterruptedException e) { // other, spurious, exceptions may happen
    if (runnable.completed)
      break;
  }
}
if (runnable.completed) {
  Status stat = runnable.result; // succes !
}
else {
  // timed out
}
```

This works efficiently because the `defer()` method automatically arranges to have an interrupt called on the calling thread when the `Runnable` completes.

4.2 Agent initialization

There are 3 main phases to an agent's creation and startup, all of which have initialization methods associated with them that a programmer can override in agent subclasses to add specific behaviour for the new type of agent. As depicted in Table 1, these three phases are:

Construction where `initializeConstructor()` is called from the agent's constructor and executes in the **caller's thread**. This is where the base structure of the two-threaded agent is created. In general, subclasses should not be dong much in this method other than perhaps initializing data structures.

Thread initialization where `initializeThread()` is called from the agent's thread's run() method and executes in the **agent's independent main thread**. This is where ontologies and policies are loaded from *.ont.lisp and *.init.lisp files respectively. After thread initialization is complete, `AbstractProcess.isInitialized()` returns true. Subclasses may begin activity in an override of this method if they are not dependent on the agent being registered, but deferral of activity to `initializeAfterRegistration (boolean)` is recommended.

Agent registered where `initializeAfterRegistration (boolean)` is called after the agent has successfully (the parameter is true) or unsuccessfully (the parameter is false) negotiated registration with the Local Area Coordinator (LAC) (see Section 3.4). In either case, `initializeAfterRegistration (boolean)` is called exactly once. At this point the agent is fully initialized and communicating with other agents. Any

Phase	Init Method	Tasks
construction	initialize-Constructor()	***AbstractProcess***: calls makeOptions() to initialize the options variable (allowing subclasses to provide extended versions of the `Options` class ***AbstractProcess***: Initializes the agent's URL, including correcting the port number if necessary ***AbstractProcess***: starts up the agent listener (SocketServer)
thread initialization	initializeThread()	***AbstractProcess***: sets the trace file name (from the URL) ***AbstractProcess***: sets runtime options from the command line (`resetRuntimeOptionsFromCommandLine()` ***AbstractProcess***: sets the security package if necessary (`from resetSecurtiyPackage(value of SECURITY command parameter)`) ***TransientAgent***: initializes policies and the commitment processor through `initializePolicies()` ***TransientAgent***: reads in the ontologies from *agentname*.ont.lisp, *classname*.ont.lisp, or *superclassname*.ont.lisp ... where these are searched according to §A (all these names are fully qualified with delimiter ".") ***TransientAgent***: executes the Lisp initialization files from *most-general-superclass-name*.init.lisp, ... *most-specific-superclass-name*.init.lisp, *agentname*.init.lisp, where these are searched according to §A (all these names are fully qualified with delimiter ".") ***TransientAgent***: initialize proxy agents if specified by :PROXIES on the cmd line ***TransientAgent***: register this agent in the `AgentLookUpTable` ***TransientAgent***: initialize the Jade semantic extensions (belief base) ***TransientAgent***: sets up a default banner string to be used by UI interfaces (`getBanner()`) ***TransientAgent***: send the request/registerAgent to the LAC if the specified LAC port > 0 (by the command line :LACPORT or the LACdefaultport preference (see §B, default 9000)) or otherwise call `intiializeAfterRegistered(false)`
agent registered	initializeAfter-Registration (boolean has_registered)	***TransientAgent***: reset the banner with the after-registration data ***TransientAgent***: executes any Lisp script files from *most-general-superclass-name*.lisp, ... *most-specific-superclass-name*.lisp, *agentname*.lisp, where these are searched according to §A (all these names are fully qualified with delimiter ".") ***Agent***: if :PERSISTENT is true (or omitted): ***Agent***: initialized the persistent file and load it, being careful to restore command line options ***Agent***: read in the _*ontology node from the file if persistentOntology is true ***Agent***: reset the port number in the properties ***Agent***: set the CreateDate attribute in the properties if this is its first creation

Table 1: Initialization phases, initialization methods, and their function.

agent script files (*.lisp) are called here. It is recommended that agents override this method to begin their activity, however, be attentive to the boolean parameter, as it may be that case the agent actually *isn't* registered.

When overriding any of the initialization methods, care must be taken to call the *super* version of the method in the overriding method, preferably at the beginning of the method body.

4.3 The event loop

Once the main agent thread (see Figure 1) is initialized (see Section 4.2), it has two tasks: processing events from the event queue, and processing any internal commitments that might arise from processing the events[4].

[4]or may have arisen from the some other task the specific agent type might undertake – CASA leaves this up to the specific agent implementation

```
new Event event = eventQueue.getItem();
if (event == null) {
  boolean didSomething = processCommitments();
  if (!didSomething && eventQueue.isEmpty()) {
    try {
      synchronized (this) {
        wait(exit?500:0);
      }
    }
    catch (InterruptedException e) {}
  }
}
else {
  processEvent(event);
}
```

Figure 5: The body of the CASA event loop

So the agent behaviour is: If any work items are to be done (processing events and/or commitments), they are all done in sequence (priority to events) as fast as possible. When everything is complete, the agent sleeps until it is interrupted. The sleeping agent is typically interrupted by the listener thread (see Figure 1) queueing a new event.[5]

Thus, once initialized, an agent loops throughout its lifetime in an *event loop*, the body of which is implemented by the private final method `AbstractProcess.messageBufferLoopBody()`. As seen in Figure 5, the body of the event loop checks to see if there is anything in the event queue, and process one of them if there are any (`processEvent()`). If there is nothing in the event queue, the agent is given a chance to process any active commitments (`processCommitments()`). If there is nothing to do, the thread sleeps until it is interrupted (unless the agent is exiting, in which case it sleeps only 500ms at most).

Processing events

In `AbstractProcess`, every event taken from the event queue is processed by calling it's `private void processEvent(Event)` method. After some processing and checking, `processEvent(Event)` will normally call its abstract `handleEvent(Event)` method. This method is overridden by `TransientAgent` to do the actual semantic processing on the event. All this behavior is summarized in Table 2.

`AbstractProcess`'s `processEvent (Event)` method does the following:

1. if the event is not a EVENT_MESSAGE_RECEIVED or a EVENT_MESSAGE_OBSERVED then go on to the last step.

2. If there is a security filter, `processEvent()` calls the security filter's `processEvent()` method to decrypt the message if necessary (see Section 14).

3. If the message is not actually addressed to the agent and the agent is not observing all messages the method exits.

[5] Other events that may wake a sleeping agent are timer events, or some other thread specifically calling the agent thread's `interrupt()` method.

step	class	method
If this is an incoming message event, preprocess message event by security decoding and notifying observers of an EVENT_MESSAGE_RECEIVED; or by dropping the event if the security declines, it isn't for the agent, or it's a exit message. If the event hasn't been dropped, call `handleEvent()`.	AbstractProcess	private void processEvent(Event event)
Collect all applicable policies from either conversations or global policies; add in the always-apply policies; if we found none, use the last-resort policies. If we have policies, call the `processPolicies(...)` method, otherwise call the event's `fireEvent()` method.	TransientAgent	protected Status handleEvent (Event event)
Apply policies. If policies fail and this is a message event, return a "not understood" message.	TransientAgent	private void processPolicies(...)

Table 2: Event processing

4. If the message is not authorized (see Section 14), the method prints a warning to the log file and exits.
5. If this is an exit message (see Section 4.4), the method exits.
6. Notifies all observers (see Section 12) by calling `notifyObservers (ML. EVENT_MESSAGE_RECEIVED, msg)`
7. Calls the agent's `handleEvent(Event)` method.

`handleEvent(Event)` is not implemented by `AbstractProcess`; subclasses should override this method if they wish to handle incoming messages in their own way. However, `TransientAgent`[6] provides a default implementation:

1. If the event has an associated conversation ID (even some non-message events may have an associated conversation ID), then find all current conversations labeled with this conversation ID that have applicable policies. If more than one such conversation is found, log it as an error and choose the first one found. We now have zero or one selected conversations, and zero or more applicable policies.
2. If we have not found any applicable policies, then search the *agent-global* policies[7] for applicable policies.
3. If we have still not found any applicable policies, then search the *last-resort* policies[8] for applicable policies.
4. Add in any applicable *always-apply* policies[9] to the set of applicable policies.

[6] `TransientAgent.handEvent()` should only be called in the agent's main thread.
[7] *Agent-global* policies are policies that are only searched if there are no policies found in applicable conversations. For example, these may be policies that create conversations, or policies to handle simple inform messages.
[8] *Last-resort* policies are that are only searched if no other policies apply. For example, "Send a not-understood message in reply to a message where no applicable policy can be found."
[9] *Always-apply* policies are policies that are always searched, no matter what. These are often basic social norms, for example, "Always reply to a request."

5. If the option `threadedEvents` is set true, then call `processPolicies(...)` in a newly-created thread; otherwise simply call `processPolicies(...)` from this thread.

`TransientAgent.processPolicies(...)` does the following:

1. Calls the static method `PolicyContainer.applyPolicies(...)` (see Section 9.2) with the list of applicable policies.

2. If the above application returns null or a `Status` with a value of 138 or it returns a `StatusObject` containing a null object or a `List` object of size zero, then the application is deemed to have not handled the event.

3. For non-*executable* events:

   (a) If there were no applicable policies (except *ghost* policies[10]) then log it as a warning.

   (b) Otherwise, if the event was unhandled by the applicable policies, then log it as a error or warning (error if the event hand any observers, warning otherwise).

   (c) If the event was unhandled (whether there were or weren't any applicable policies) and it is a message event, then call the `unhandledMessage(MLMessage)` method, which by default, will just log the unhanded message as a *warning*.

4. Fire the event (call the event's `fireEvent()` method).

5. Call the event's `delete()` method. Note that the event may or may not actually delete itself – for example, it may not delete itself if it is a repeating event that has not been cancelled.

6. Interrupt the agent's thread if this method is not running in the agent's thread.

`PolicyContainer.applyPolicies()` takes the list of applicable policies, filters them, and sorts them (see Section 9.2) and then calls the `apply()` method of each of these policies. In the case of CASA's Lisp-defined policies (see Section 11), this involves executing the lisp code for consequent part of the policy.

Processing commitments

`AbstractProcess` does not define its method `processCommitments()`, and subclasses are free to override it. But `TransientAgent` provides a default: it calls its `commitmentProcessor`'s (and instance of `CommitmentProcessor`) `processCommitments()` method. `CommitmentProcessor` is an abstract class, and by default the concrete subclass `casa.policies.sc2.ConcreteCommitmentProcessor` is used. Subclassing agents may replace the `CommitmentProcess` by using `setCommitmentProcessor`.

The default commitment processor (`ConcreateCommitmentProcessor`)'s `processCommitments()` method will choose a single active commitment from its `CommitmentStore` and execute it. `ProcessCommitments()` returns true if it processed a commitment (or attempted) and false otherwise to signal the event loop see §5) that it can sleep without calling `processCommitments()` again immediately.

[10] *Ghost* policies are policies that we do not want to count as being applied. For example, the policy to fulfill a commitment to send or receive a message by that message being sent or received.

4.4 Exiting

An agent is signalled to exit by calling the `AbstractProcess.exit()` method. But the agent does not exit immediately; it does the following first:

1. It sets the `exit` flag so that the event loop (Section 4.3) will terminate after all pending events are cleared.
2. It calls its abstract `pendingFinishRun()` method. Subclasses should override `pendingFinishRun()` to take appropriate actions before exiting.
3. It queues a dummy exit event to keep the event queue open for at least another 200ms.

Agent classes `TransientAgent` and `Agent` both override the `pendingFinishRun()` method to add their own behaviors:

`TransientAgent.pendingFinishRun()` withdraws the agent from an CDs (see Section 3.5) it has joined and unregisters the agent from the LAC (see Section 3.4). Both of these tasks are accomplished by carrying out conversations with the relevant agents, and this is the need for the event queue to remain operational: the event queue is used to process the messages in these conversations.

`Agent.pendingFinishRun()` writes out persistent data to the datafile allocated by the LAC upon registration.

5 Ontology

Since CASA deals with performatives and acts in messages as full types and not just as tokens, it must be capable of supporting a type system. CASA supports two ontology systems, and may be extended to support others. OWL2 is fairly "standard" with W3C and has several good tools supporting it. Therefore, OWL2 is the recommended ontology system to use.

Both ontology systems support common Java and Lisp interfaces.

The Java interface is defined by `casa.ontology.Ontology`:

```
add(String)
addIndividual(String, String)
addIndividual(String, String...)
addType(String, String)
addType(String, String...)
declMaplet(String, String, String)
declRelation(String, String, Set<Property>, Constraint, Constraint, Object...)
describe(String)
describeIndividual(String)
describeRelation(String)
describeType(String)
extendWith(String)
getName()
instanceOf(String, String)
isa(String, String)
isCompatable(Ontology)
```

```
isCompatableThrow(Ontology)
isIndividual(String)
isObject(String)
isRelation(String)
isType(String)
relatedTo(String, String)
relatedTo(String, String, String)
toString()
```

The Lisp interface is:

```
ONT.ASSERT,DECLMAPLET set a type-to-type relationship in the specified relation.
ONT.GET,GET-ONTOLOGY Retrieves ontology int either from the shared memory or from a file of
the same name ([name].
ONT.GET-RESIDENT,GET-RESIDENT-ONTOLOGIES Retrieves a list of the names of the ontologies
in shared memory.
ONT.INDIVIDUAL,DECLINDIVIDUAL Declare an individual in the A-box.
ONT.IS-INDIVIDUAL,DECLINDIVIDUAL Return true iff the parameter is an individual in the
ontology.
ONT.IS-OBJECT,DECLINDIVIDUAL Return true iff the parameter is a type or an individual in
the ontology.
ONT.IS-TYPE,DECLINDIVIDUAL Return true iff the parameter is a type (not an individual) in
the ontology.
ONT.RELATED-TO If RANGE is specified, return T iff the DOMAIN is related to the RANGE by the
specified RELATION, otherwise return the set of elements in range of DOMAIN by RELATION.
ONT.RELATION,DECLRELATION Define a relation in the agent's casa ontology.
ONT.SET-DEFAULT,SET-DEFAULT-ONTOLOGY Sets the agent's default ontology either from the
shared memory or from a file of the same name ([name].
ONT.TYPE,DECLTYPE Declare a type in the T-box.
ONTOLOGY,DECLONTOLOGY,WITH-ONTOLOGY Declare a new Ontology or extends an existing Ontology.
```

You can get more detailed information on the Lisp commands in CASA by using the Lisp command `(describe '<commandName>)`.

5.1 OWL2 Ontology

OWL2 [Motik et al., 2011] is a standard language that can be written in several different forms, and has several tools for ontology editing and analysis. One of these tools is Protege [Stanford Center for Biomedical Informatics Research, 2013] which offers ontology editing and and graphical analysis.

OWL2 is CASA's default ontology. For documentation on OWL2, see Table 3.

For information on loading and using CASA ontologies, see §5.3.

5.2 CASA Native Ontology [Deprecated]

CASA maintains a lattice-based type system written in java, but which can be more conveniently accessed in lisp (see §11). In CASA's ontology, *relations*, *types* and *individuals* are defined. Relations are defined as either *primitive*:

Type	Document
For Users	Document Overview. A quick overview of the OWL 2 specification that includes a description of its relationship to OWL 1. This it the starting point and primary reference point for OWL 2. http://www.w3.org/TR/owl2-overview/#
Core Specification	Structural Specification and Functional-Style Syntax defines the constructs of OWL 2 ontologies in terms of both their structure and a functional-style syntax, and defines OWL 2 DL ontologies in terms of global restrictions on OWL 2 ontologies. http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/
Core Specification	Mapping to RDF Graphs defines a mapping of the OWL 2 constructs into RDF graphs, and thus defines the primary means of exchanging OWL 2 ontologies in the Semantic Web. http://www.w3.org/TR/2012/REC-owl2-mapping-to-rdf-20121211/
Core Specification	Direct Semantics defines the meaning of OWL 2 ontologies in terms of a model-theoretic semantics. http://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/
Core Specification	RDF-Based Semantics defines the meaning of OWL 2 ontologies via an extension of the RDF Semantics. http://www.w3.org/TR/2012/REC-owl2-rdf-based-semantics-20121211/
Core Specification	Conformance provides requirements for OWL 2 tools and a set of test cases to help determine conformance. http://www.w3.org/TR/2012/REC-owl2-conformance-20121211/
Specification	Profiles defines three sub-languages of OWL 2 that offer important advantages in particular applications scenarios. http://www.w3.org/TR/2012/REC-owl2-profiles-20121211/
For Users	OWL 2 Primer provides an approachable introduction to OWL 2, including orientation for those coming from other disciplines. http://www.w3.org/TR/2012/REC-owl2-primer-20121211/
For Users	OWL 2 New Features and Rationale provides an overview of the main new features of OWL 2 and motivates their inclusion in the language. http://www.w3.org/TR/2012/REC-owl2-new-features-20121211/
For Users	OWL 2 Quick Reference Guide provides a brief guide to the constructs of OWL 2, noting the changes from OWL 1. http://www.w3.org/TR/2012/REC-owl2-quick-reference-20121211/
Specification	XML Serialization defines an XML syntax for exchanging OWL 2 ontologies, suitable for use with XML tools like schema-based editors and XQuery/XPath. http://www.w3.org/TR/2012/REC-owl2-xml-serialization-20121211/
Specification	Manchester Syntax (WG Note) defines an easy-to-read, but less formal, syntax for OWL 2 that is used in some OWL 2 user interface tools and is also used in the Primer. http://www.w3.org/TR/2012/NOTE-owl2-manchester-syntax-20121211/
Specification	Data Range Extension: Linear Equations (WG Note) specifies an optional extension to OWL 2 which supports advanced constraints on the values of properties. http://www.w3.org/TR/2012/NOTE-owl2-dr-linear-20121211/

Table 3: OWL2 Documentation Overview [Motik et al., 2012]

```
(ont.relation "isa-parent")
```

or in terms of another relation with further *properties*:

```
(ont.relation "isa" :base isa-parent :transitive :reflexive)
```

Here, the relation `isa` is defined in terms of the early-defined relation `isa-parent`, but it is further defined as being both reflexive ($(isa\, x\, x)$ is always true), and transitive ($((isa\, a\, b) \wedge (isa\, b\, c) \rightarrow (isa\, a\, c))$. One can define actual elements of a relation:

```
(ont.assert isa-parent "action" TOP)
(ont.assert isa-parent "performative" action)
```

meaning that `action` is a related to `TOP` and `performative` is a related to `action` in the `isa-parent` relation. By the definition of `isa` above, the system infers `(isa action action)`, `(isa performative performative)` and `(isa performative TOP)`.

CASA's ontology system is implemented in Java, and has interfaces in both Java and Lisp. The ontology system is hierarchical; that is, an ontology may extend a "parent" ontology (called the *superontology*). Ontologies all have names, and public ontologies are all registered globally (per process) and accessible via the Java `CASAOntology.getOntology(String name)` method (or the Lisp `(get-ontology <name>)` function). An ontology contains up to three types of objects: *types*, *individuals* and *relations*. *Types* and *individuals*, of course represent types and individuals respectively. *Relations* are somewhat more complex.

Types and Individuals

Types are symbols representing abstract types, whereas *individuals* are symbols representing concrete objects. Types and individuals are declared in Lisp like this:

```
(ont.type "person")
(ont.individual "fred")
```

This only works, of course, within the context of a specific ontology, but one may specify the ontology in the declaration like this:

```
(ont.type "person" :ontology <an-ontology-name>)
```

In Java, the same thing is accomplished like this:

```
CASAOntology myOntology = new CASAOntology("myOntology");
Type person = myOntology.declType("person");
Individual fred = myOntology.declIndividual("Fred");
```

When either types or individuals are declared, they are instantiated as symbols in the Lisp environment with their value being the string equivalent. This means that although you must declare types and individuals in Lisp with double quotes around them, once they are declared, you no longer need to use the double quotes.

Types are also implicitly declared when they are used in the domain position in a `(ont.assert ...)` Lisp command or a `Ontology.ont.assert(...)` Java method (see §5.2).

Relations

Relations represent binary relations among types and individuals. A *primitive relation* is, conceptually, nothing more than a set of ordered pairs of types or individuals. However, relations can have several properties such as symmetry, reflexivity, and transitivity.

Relation properties

Properties are implemented as *decorators* (à la the decorator pattern [Gamma et al., 1994]) on top of a primitive relation. Thus, we can define the relation `isa-parent` as a primitive relation, then further define `isa-ancestor` as transitive decorator on top of `isa-parent`. Furthermore, we can define the relation `isa` as a reflexive decorator on top of `isa-ancestor`. Thus, we have the `isa` relation defined as two decorators on top of a primitive relation:

→ `isa`: *reflexive* → `isa-ancestor`: *transitive* → `isa-parent`: *primitive*

`isa` may be defined in the Lisp in the following way:

```
(ont.relation "isa-parent")
(ont.relation "isa-ancestor" :base isa-parent :transitive)
(ont.relation "isa" :base isa-ancestor :reflexive)
```

Or, in Java, using the `declRelation(final String name, ConcreteRelation basedOn, Set<Relation.Property> properties, Constraint domConstraint, Constraint ranConstraint, Object... otherParams)` method:

```
ConcreteRelation isaParent, isaAncestor, isa;
isaParent = primitiveOntology.declRelation("isa-parent", null, null, null, null);
isaAncestor = primitiveOntology.declRelation("isa-ancestor", isaParent,
    new Relation.Property[]{Relation.Property.TRANSITIVE}, null, null);
isa = primitiveOntology.declRelation("isa", isaAncestor,
    new Relation.Property[]{Relation.Property.REFLEXIVE}, null, null);
```

As with types and individuals, relations are also instantiated in the Lisp environment when they are declared. But relations are instantiated as functions. Relational functions take either one or two arguments. The one argument version takes a domain object (type or individual) and returns a list of objects in the range for the domain (if any). The two argument version takes a domain object and a range object and returns T if the relation is satisfied for this pair, and NIL otherwise. For example, if we imagine the appropriate declarations:

```
(isa horse mammal) → T
(isa horse) → (mammal animal TOP)
```

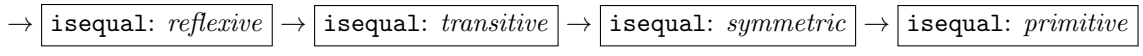
While all three of the above relations are useful, sometimes the intermediate relations are not of interest. For example, *equal* is reflexive, transitive, and symmetric, but we generally have not interest in the intermediate relations. So, we can simply define our relation `isequal` in Lisp as:

```
(ont.relation "isequal" :symmetric :transitive :reflexive)
```

or, in Java:

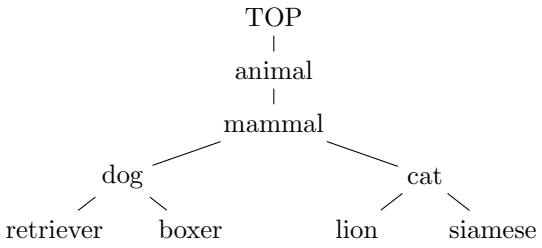
```
ConcreteRelation isequal;
isequal = primitiveOntology.declRelation("isequal", null, new Relation.Property[]{
      Relation.Property.TRANSITIVE,
      Relation.Property.REFLEXIVE,
      Relation.Property.SYMMETRIC}, null, null);
```

This forms the following decorator pattern:



In this case, all but the top decorator in the chain are hidden from normal use, so the fact that they all have the same name, "isequal", doesn't matter much.

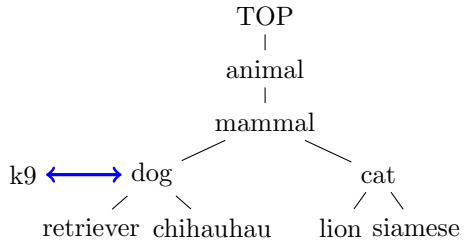
Now that we have defined the `isa` relation we can define a simple example ontology of mammals:




```
(ont.assert isa-parent "animal" TOP)
(ont.assert isa-parent "mammal" animal)
(ont.assert isa-parent "cat" mammal)
(ont.assert isa-parent "lion" cat)
(ont.assert isa-parent "siamese" cat)
(ont.assert isa-parent "dog" mammal)
(ont.assert isa-parent "retriever" dog)
(ont.assert isa-parent "chihauhau" dog)
```

Uses

This is all good, but we note that there are synonyms for some of the terms used in this ontology. For example, we might want to also refer to "dog" as "k9", which we take to be the same thing. The ontology allows for this by allowing one relation to *use* another relation as an equality relation. This is implemented as another decorator on a relation and it is specified as a parameterized relation property. For example, we may revise our previous definition of the `isa-parent` relation to:

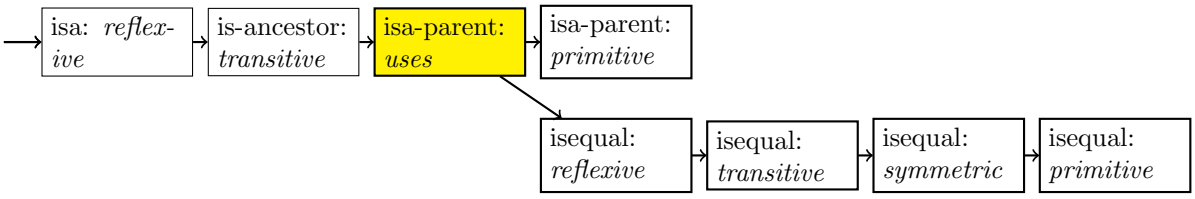



```
(ont.relation "isa-parent" :uses isequal)
```

and add the definition:

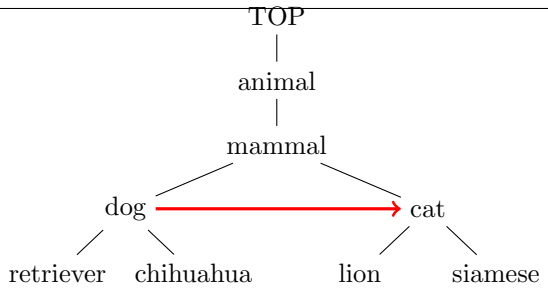
```
(ont.assert isequal "k9" dog)
```

The decorator pattern chain for the `isa` relation now looks like this:



The ontology system is not only capable of representing a type (*isa*) lattice with equality, but also arbitrary relations as well. For example, we may wish to implement a relation *bigger* whose maplet's domain elements are "bigger" than the corresponding range elements. We would expect *bigger* to be transitive, and not reflexive and not symmetric, and we want it the *use* the `isa` lattice:

```
(ont.relation "bigger" :transitive :uses isa)
```



And we can say that dogs are bigger than cats:

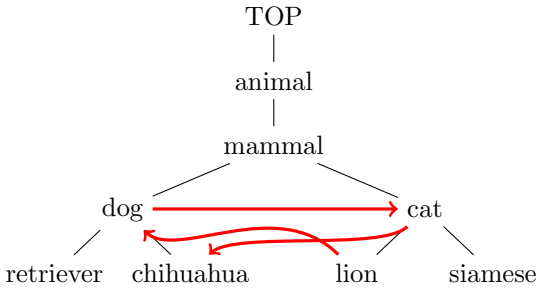
```
(ont.assert bigger dog cat)
```

which gives the following results:

```
(bigger dog cat) → T
(bigger retriever cat) → T
(bigger retriever siamese) → T
(bigger retriever lion) → T
(bigger chihuahua cat) → T
```

which is what we expect, taking into account the `isa` type lattice. However, while the first 3 evaluations above are "normal", the last two aren't really what we want: a lion is an exceptionally big cat, and is usually bigger than a typical dog. And a chihuahua is an unusually small dog, and is really smaller than a cat. We can deal with these anomalous situations using a form of *asymmetry*: where we force a relational query to accept only one of multiple possible inferred conflicting maplets, disambiguating based on which is the *most specific*. To do this, we give the `bigger` relation the asymmetric property and add the more detailed maplets:

```
(ont.relation "bigger" :transitive :uses isa :asymmetric)
(ont.assert bigger dog cat)
(ont.assert bigger lion dog)
(ont.assert bigger cat chihuahua)
```



which gives more realistic evaluations:

```
(bigger retriever lion) → NIL
(bigger lion retriever) → T
(bigger chihuahua cat) → NIL
(bigger cat chihuahua) → T
```

But also the following evaluation holds

```
(bigger dog chihuahua) → T
```

which seems wrong because a chihuahua *is* a dog. It turns out the system *infers* this truth because `(bigger dog cat)` and `(bigger cat chihuahua)` and `bigger` is transitive. This makes sense intuitively too, since a chihuahua is smaller than a typical dog.

Constraints

In addition, modelling often requires restricting the domain and range of relations in several ways. To accommodate this, there are two additional key parameters that `ont.relation` takes: `:domain-constraint` and `:range-constraint`. Both of these take a `Constraint` object as a parameter. In Lisp, you can easily construct a constraint for almost any situation using the `constraint` function:

type (optional) Constraints the parameter (domain/range) to this type as the most general type.

:individual-only Constrains the parameter to be an Individual.

:type-only Constrains the parameter to be a Type.

:exp Constrains the parameter to conform to this Lisp expression. The expression must return a boolean result. The expression will be executed in the context of the following environment variables: $type is the type-name under validation; $0 is the domain type in the relation; $1 is the range type in the relation.

:ontology Sets the ontology in which this constraint exists. If you omit this key, the ontology will be dynamically chosen at run time.

:agent Sets the agent that "owns" this constraint. If you omit this key, the agent will be dynamically chosen at run time.

For example, one may define the relation `smaller-dog`, which is restricted to a domain and range of individual dogs, and is also restricted to being consistent with the relation `bigger`:

```
(ont.relation smaller-dog :asymmetric T
      :domain-constraint (constraint dog :individual-only T :exp '(bigger ?1 ?0))
      :range-constraint (constraint dog :individual-only T))
```

Summary

To summarize, relations are built on top of *primitive relations* using the decorator pattern [Gamma et al., 1994] (a chain of decorators, where each decorator represents a *property* of the relation). The decorators currently available are:

reflexive $\forall x \bullet (rel\ x\ x)$.
Any element is always related to itself. For example, the *equal* relation.

symmetric $\forall x, y \bullet (rel\ x\ y) \rightarrow (rel\ y\ x)$.
If x is related to y, then y must be related to x. For example, the *sibling* relation.

transitive $\forall x, y, z \bullet (rel\ x\ y) \wedge (rel\ y\ z) \rightarrow (rel\ x\ z)$.
If x is related to y, and y is related to z, then x must be related to z. For example, the *bigger* relation.

asymmetric $\forall x, y \bullet (rel\ x\ y) \wedge (rel\ y\ x) \rightarrow (mostSpecific\ (rel\ x\ y)\ (rel\ y\ x))$
where

$$(mostSpecific\ (rel\ x1\ y1)\ (rel\ x2\ y2)) = \\ ((x1 < x2 \wedge y1 <= y2) \vee (x1 <= x2 \wedge y1 < y2)) \rightarrow (rel\ x1\ y1) \wedge \\ ((x2 < x1 \wedge y2 <= y1) \vee (x2 <= x1 \wedge y2 < y1)) \rightarrow (rel\ x2\ y2)$$, otherwise it is an error.

where $<$ and $<=$ are the *uses* relation and the reflexive *uses* relation respectively.
If the relation is found to run in both directions, then take the direction of the *most specific* relational pair as being true and the other direction as being false. Use the *uses* relation to evaluate specificity. If we cannot distinguish one relational pair[11] as being more specific than the others, then it is an error: something was probably misspecified in the assertions.

[11] Either because there is no *uses* relation, or the *uses* relation does not distinguish.

```
(declOntology "actions" ;
  '() ; super ontologies
  '(
    ; ACTIONS
    (ont.assert isa-parent "action" TOP)
    (ont.assert isa-parent   "LAC_closing" action)
    (ont.assert isa-parent   "act" action)
    (ont.assert isa-parent   "chat_message" action)
    ...
  )
)
```

Figure 6: CASA type definitions (excerpt).

uses=*usesRel* $\forall x, y \bullet \exists x', y' \mid ((usesRel\ x\ x') \vee x = x') \wedge ((usesRel\ y\ y') \vee y = y') \bullet (rel\ x'\ y') \rightarrow (rel\ x\ y)$. This relation will use *usesRel* as its equality relation. That is, if x is there is related to x' and y is there is related to y' and x' is related to y' by the *uses* relation, then we shall take x and y to be related by the relation. For example, the *isa* relation *uses* the *is equal* relation.

In addition, the following constraints may be applied:

domain-constraint=*constraint* The constraint on the domain.

range-constraint=*constraint* The constraint on the range.

5.3 CASA Ontologies

CASA agents implicitly load a specific ontology when they are activated. Specifically, the ontology they search for is named by the same name as the agent's name, the same name as the agent's most specific class, and the name for that class's superclass, etc. These ontologies are first checked to see if they are already loaded, and then an attempt is made to load them from a file named "*name*.ont.lisp" or "*name*.owl" depending on if the current ontology engine is CASA's native engine or the OWL2 engine. For an explanation of *where* CASA searches for these filenames, see §A. In the case of the OWL2 ontology, if no named ontology is found on the local machine, a final check is made for a web version at `http://casa.cpsc.ucalgary.ca/ontologies/`.

Because all agents must communicate and these communications demand a shared understanding of message types (*performatives, acts*, etc.) all CASA agent ontologies should have the ontology `casa` as a superontology, which can be loaded from the file `casa.ont.lisp`[12] or `casa.owl`[13]. The `casa` ontology does little more than inherit from the `actions` and the `events` ontologies (also included in the distribution as `actions.ont.lisp` and `events.ont.lisp` respectively). Figure 6 shows an example excerpt from the `actions` ontology.

6 Knowledge Base

The default agent knowledge base is an implementation of the Jade's [Bellifemine et al., 2007, Telecom Italia Lab, 2008, Bellifemine et al., 2003] Semantic Extension [Pautret, 2006] belief base.

[12]The `casa.ont.lisp` file is in the CASA distribution in the `/datafiles` directory

[13]The `casa.owl` file is in the CASA distribution in the `/datafiles` directory, but may also be found at `http://casa.cpsc.ucalgary.ca/ontologies/casa.owl`.

```
( request
 :act register_instance
 :sender Alice
 :receiver LAC
 :reply-by "2008.09.30 15:54:34.004 MDT"
 :reply-with /casa/ChatAgent/Alice--0
 :conversation-id /casa/ChatAgent/Alice--0
 :language casa.*
 :content "[\"(casa.URLDescriptor\\)\\\"Alice\\\"\",
         \"(java.lang.Boolean\\)\\\"true\\\"\"]"
 :priority 10 )
```

Figure 7: A typical *request* message using KQML markup. URLs have been simplified to save space.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CASAmessage SYSTEM "CASAMessage.dtd">
<CASAmessage version="1.0">
  <performative>agree</performative>
  <act>request|register_instance</act>
  <sender>LAC</sender>
  <receiver>Alice</receiver>
  <reply-by>Mon Jun 18 12:14:25 MDT 2012</reply-by>
  <reply-with>LAC--3</reply-with>
  <in-reply-to>Alice--2</in-reply-to>
  <conversation-id>Alice-2</conversation-id>
  <language>casa.*</language>
  <content>["(casa.URLDescriptor)\"Alice\"",
         "(java.lang.Boolean)\"true\""]</content>
  <timeout>Wed Dec 31 17:00:10 MST 1969</timeout>
</CASAmessage>
```

Figure 8: A typical *request* message using XML markup. URLs have been simplified to save space.

7 Messages

CASA deals with messages as the abstract class `MLMessage`, which is little more than a dictionary from keys (`String`) to values (also `String`). CASA provides two concrete subclasses, `KQMLMessage` and `XMLMessage` that provide streaming services for KQML-messages and XML-messages respectively. Programmers are free to add addition message markup languages by subclassing MLMessage similarly. If the subclass is named "<*identifier*>`Message`" and is loaded early after startup, CASA will automatically use the new message format if it's referenced. However, programmers can also manually register it using environment variables, or user or system preferences with the identifier `casa.MLMessage-subclasses` (see §B).

Figure 7 shows a typical KQML message, and Figure 8 shows a typical XML message. By default, CASA uses the KQML message format for outgoing messages, but this default can be changed using the preference `defaultMarkupLanguage` (see §B), or at runtime by calling `MLMessage.setMarkupLanguage()`. Incoming messages are parsed using any of the registered markup languages by attempting them all, starting with the default.

Programmers may easily create a message of the default type by calling `MLMessage.getNewMLMessage()` or `MLMessage.getNewMLMessage(String... keyValuePairs)`. To specify the markup type, use `MLMessage.getNewMLMessageType(String markupIdentifier)` or `getNewMLMessageType(String markupIdentifier, String... keyValuePairs)`. In Lisp, use the function `(agent.message` *performative act receiver ...any key is allowed*`)`.

Messages, whether incoming, outgoing or just observed[14], are treated just as any event, and are queued on the event queue as `MessageEvents` (a subclass or `Event`). For details of event handling see §4.1.

[14] an *observed* event differs from an *incoming* event in that the observed event is not addressed to the agent, but still arrived on the event queue for any number of reasons.

8 Commitments

Social commitments may be thought of as obligations an agent has to other agents, where the agent is obligated to perform some action in the future. A social commitment is a much more complex object than than an event, and social commitments are implemented in CASA using several different events.

The `SocialCommitment` class is used to track a social commitment. Figure 9 shows a social commitment's possible states and transitions. `SocialCommitments` are also `EventObservers`, and can therefore react to CASA events as described in §4.1. `SocialCommitments` are created in association with a set of `Event` objects, which implicitly include prefired *start* events and *perform action* events if they are not originally included in the event set. A `SocialCommitment` starts out life as *created* but inactive. It becomes active when a *start* event moves it into the *started* state. The commitment cannot be executed until it moves into the *perform-action* state. At any time, a commitment may be designated *fulfilled*, or *cancelled*, or one of its violation events may occur, which places it in one of its terminal states. *Persistent* commitments may be executed repeatedly. These may be designated *started* from the *ready-fulfilled* state.

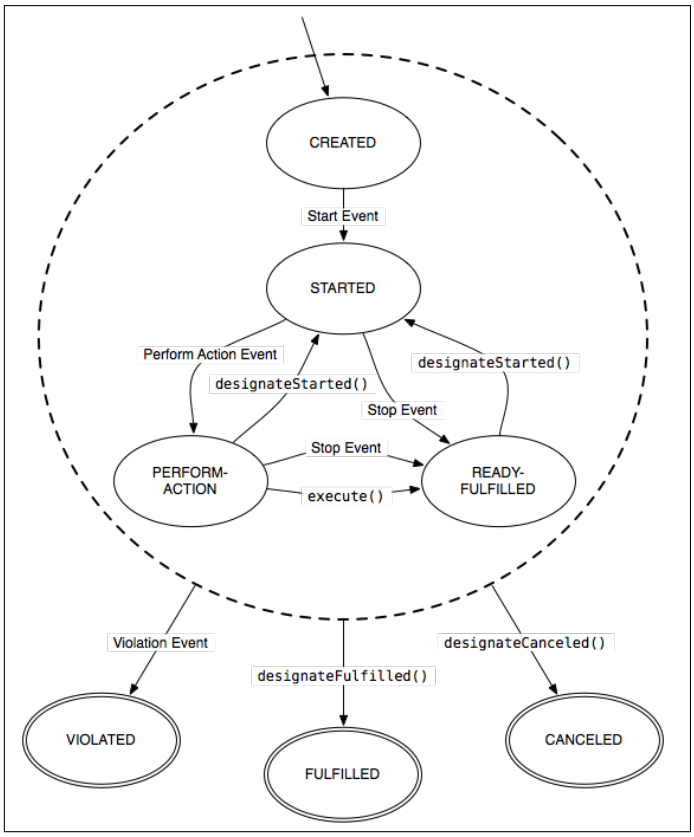


Figure 9: Social Commitment states

One may wonder why a `SocialCommitment` does not move into a terminal state after it is *executed*. The reason for this is that, although an agent may execute a commitment, the intended outcome of the commitment still needs to be verified (for example, by the approval of another agent) before it can be designated *fulfilled*.

The state machine follows the following rules:

- The state machine starts in the *created* state.
- The state machine ends in one of the *fulfilled*, *canceled*, or *violated* states.
- If the *start* event occurs and the current state is *created*, then the current state is set to *started*.
- If the *perform action* event occurs and the current state is *started*, then the current state is set to *perform-action*.
- If the state becomes *started* and the *perform action* event has previously occurred, then the current state is set to *perform-action*.
- If the *stop* event occurs and the current state is *started* or *perform-action*, then the current state is set to *fulfilled*.

- If the state becomes *started* and the *stop* event has previously occurred, then the current state is set to *fulfilled*.
- If `execute(PolicyAgentInterface)` is called and the current state is *perform-action*, then the current state is set to *ready-fulfilled*.
- If `designateFulfilled()` is called and the current state is not an end state, then the current state is set to *fulfilled*.
- If `designateCanceled()` is called and the current state is not an end state, then the current state is set to *canceled*.
- If `designateStarted()` is called and the current state is not an end state, then the current state is set to *started*.
- If the *violation* event occurs and the current state is not an end state, then the current state is set to *violated*.
- If the social commitment does not have a *start* event or *perform action* event, then those events are considered to have occurred.
- If the *start* event or the *perform action* event occur before they have an effect, then the fact that they have occurred is recorded.
- If the social commitment does not have a *stop* event or a *violation* event, then those event never occur.

8.1 Social commitment operators

Although CASA does not necessarily required the use of commitments, commitments are typically created in policies (§9), especially policies following the Social Commitment paradigm. Policies are typically written in Lisp (§11), so social commitments have a Lisp interface. However, commitments in policies are all executed in "parallel", so they are never executed directly, but rather, policies create *social commitment operators* when they fire, and the *operators* are all executed when the policies have all finished firing. The execution of the operators is what actually instantiates, cancels, or fulfills social commitments. The Lisp commitment operators are `sc.add`, `sc.fulfil`, and `sc.cancel` which instantiate and remove social commitments to/from an agent's commitment store.

The key arguments for `sc.add` are:

:DEBTOR (casa.URLDescriptor) The debtor.

:CREDITOR (casa.URLDescriptor) The creditor.

:PERFORMATIVE The performative as defined in the agent's ontology.

:ACT (casa.Act) The act.

:ACTION-CLASS (java.lang.String) A string representing the fully-qualified class name of the java `Action` class (only one of :action and :action-class is allowed).

:ACTION (org.armedbear.lisp.Cons) An expression cons-list of repenting the action to be performed (only one of :action and :action-class is allowed).

:DEPENDS-ON (casa.socialcommitments.SocialCommitmentDescriptor) The social commitment that this one depends on and must be executed before this one can be executed.

:SHARED The commitment is a shared commitment.

:PERSISTENT The commitment is persistent and can only be removed with a CANCEL.

:GETEVENTS We should retrieve the events from the agent.

For example, to add a social commitment for Alice to send a message to Bob to Alice's message store:

```
(sc.add :debtor Alice :creditor Bob :action '(agent.send
      (agent.message inform nil Bob :content "Hi Bob!")))
```

The key arguments for `sc.cancel` and `sc.fulfil` are:

:DEBTOR (casa.URLDescriptor) The debtor.

:CREDITOR (casa.URLDescriptor) The creditor.

:PERFORMATIVE The performative.

:ACT (casa.Act) The act.

Superficially, `cancel` and `fulfil` have the same impact on the system: they remove a social commitment. Of course, a `fulfil` is only appropriate when the desired actions have been successfully executed by all the agents involved. A `cancel` is appropriate when the action is no longer desired or cannot be executed.

9 Policies

Policies are rules that fire whenever they match the occurrence of an event. Policies dictate what an agent should do whenever an event is processed from its event queue (see Section 4.3). As rules, policies have an *antecedent* and a *consequent*. In a normal rule the antecedent us evaluated for a truth value, and if true, is —emphfired (the consequent is executed), otherwise nothing happens. In CASA is the situation is slightly different: there are two parts to the antecedent. One part, called the antecedent, is a object called an *event descriptor*, which is matched against the event. If the event descriptor and the event match the rule is eligible to be fired. The second part, called the *precondition* is evaluated for a truth value and both the precondition is true and the antecedent matches, the policy is fired.

When a CASA policy fires, the Lisp expressions in the consequent are executed in order. Unless a consequent expressions evaluates to a social comment operator (see §8.1), the results are ignored. However, all of the results of consequent expressions that return a social comment operator are collected (for all the the policies that are executed for the event) and these are all executed only after all the policies have fired for the event. This has the effect of the policies operating and firing in parallel (in the same environment with respect to social commitments).

An agent contains several categories of policies that may or may not be searched depending on the circumstances. They are searched in this order:

1. **Conversation policies** Conversations (see §10) contain policies, and an agent may have zero or more active conversations. Any event that matches one or more active conversations (usually by a *conversation-id* match) will trigger a search for matching policies in those conversations.

2. **Normal policies** If no active conversation has matching policies, then normal policies are searched for matches.

3. **Always-apply policies** These policies are search for matching policies regardless of whether or not there are already matched policies.

4. **Last-resort policies** These policies are search only if no other matching policies are found (with the exception of *ghost* policies (see §9.1. These would typically be used to respond with a "not understood" to a message event, or to log an error or warning in response to some other event.

For each event, the agent searches the policy categories as explained above, but it does not fire the policies until after the search is completed. Then the policies are filtered based on their event descriptors: one policy overrides another if they both have identical *performatives* and its *act* is more specialized than the other. The overridden policy is ignored. Only then are the policies actually fired. In this way, all policies are evaluated in the same environment, independent of effects of other policies firing. It is possible to change the environment by placing environment-changing code within the `:precondition` expression, but this is to be considered side-effect, and is to be discouraged.

To actually register a policy with an agent, the Lisp operator is `agent.put-policy` which registers the argument policy as a normal policy by default, but ask takes the optional boolean key parameters `:always-apply` and `:last-resort` to register the policy as an always-apply or last-resort policy respectively.

9.1 Policy Specification

A `policy` is a Lisp function (see the example in Figure 10), which may take the following parameters:

event-discriptor - An *event descriptor*

actions - A quoted list of social commitment *operators* appropriate to the event (i.e. `add`, `fulfil`, `cancel`, and `addif`)

doc - A Documentation String (optional)

:precondition - A single-quoted list function which must evaluate to true for the policy to be selected (optional)

:postcondition - A single-quoted list function which tests whether the policy succeeded (optional) (currently not used)

:name - A name to be applied to the policy. The default is the type of the event descriptor or "LispExpression" if there is no event descriptor.

:ghost - Marks the policy as not-to-be counted when we count applicable policies. This is meant to model low-level social norms or physical-world assumptions. (optional)

A `policy` function's first parameter, the Event Descriptor, is typically filled by a `event-descriptor` for a `msgEvent-descriptor` function call, both of which take one required parameter and a set of key parameters:

event-type The type of the event, which is expected to be an ontology entry.

other keyed parameters These keys may have any name and any value and they are used to match against the key/value pairs in the target event.

```
(policy
  `(msgevent-descriptor  event_messageReceived
     :performative request
     :act (act ,the-act)
     :sender client
     :receiver server
  )
  `(
    (sc.add
      :Debtor server
      :Creditor client
      :Performative reply
      :Act (act (event.get-msg 'performative)
                (event.get-msg 'act))
      :Action `(agent.reply (event.get-msg)
                            ,',request-decision)
      :Shared
    )
    (conversation.set-state "waiting-request")
  )
  "For a request, instantiate a commitment to reply
   for the receiver."
  :precondition
    '(equal (conversation.get-state) "init")
  :postcondition
    '(equal (conversation.get-state) "waiting-request")
  :name "ask-server-000"
) ; end policy
```

Figure 10: A *request* policy from the server's point of view. It reacts to an incoming *request* message by instantiating a social commitment for the receiver to *reply* to the sender.

The `policy` function's second parameter is made up of a quoted list consisting of any number of Lisp functions. Should this policy's event take place, these functions are executed. Of special significance are functions that return *social commitment operators*, such as `sc.add`, `sc.fulfil`, or `sc.cancel` as described in §8.1. See the middle of Figure 10 for an example `sc.add` clause.

9.2 Policy Execution

Policies are read in at agent initialization (see Section 4.2), but they are not executed then. They are stored in one of a number of `PolicyContainer`s and selectively *applied* whenever an event occurs. The `PolicyContainer`s are *conversation* policies, *agent-global* policies, *always-apply* policies, and *last-resort* policies. In addition, policies may also be tagged as *ghost* policies, which means they don't count as applicable policies, event when they are applied.

Within a specific context (according to the process defined in Section 4.3), all policies that match an event and who's *precondition* succeeds are selected, filtered, and ordered.

selection: A policy is selected if the `event descriptor` in the policy matches. This match is firstly a match if both the event's type is a subtype of the policy's `event descriptor` type and, if there is a `:precondition` clause, the precondition doesn't return false (Nil). A match may be further constrained by the key/value pairs in the policy's `event descriptor`; for example, a `MsgEvent-descriptor` also

checks that the performative of the message is subtype of the performative in the `MsgEvent-descriptor` and the act of the message is a subtype of the act in the `MsgEvent-descriptor`.

filtering: Any policy in the set is eliminated if it is overridden another policy: One policy overrides another If they both have identical *performatives* and its *act* is more specialized than the other. The overridden policy ("super policy") is ignored.

ordering: Policies have no real intrinsic order, but they are fired in order of their ID numbers.

The consequents of the each of these policies is then executed.

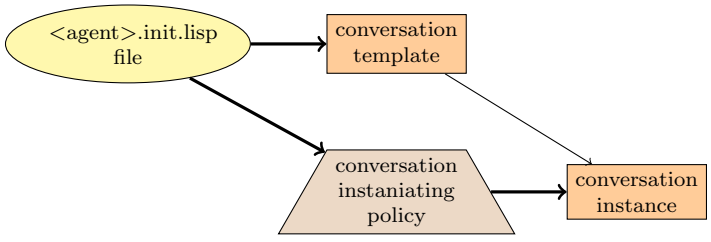
Social commitments (see §8) are handled specially: if a policy consequent generates a `SocialCommitment-Operator` it is saved in a list of `SocialCommitmentOperators`, and this list is executed only after all the policies have been been fired (executed). In this way, policies are guaranteed to be executed in the context of the social commitments that existed before the policies where applied: They appear to operate in parallel with respect to the current social commitments.

10 Conversations

While policies capture an agent's reaction to a particular event, agents more typically need a lot more *context* to make a decision about their behaviour in light of the current event. Especially for message exchanges with other agents, this context is captured by the concept of a *conversation*. We have both conversation *templates* that specify a the overall pattern and restrictions on a *type* of conversation; and conversation *instances* that follow a template, have state, and capture a particular conversation. Conversation templates and instances are analogous to classes and objects in object-oriented programming. In general, two-party conversations have distinct behaviour or each side, which we typically label *server* part and *client* part. See the example in Figure ?? for the server side of a *request* conversation[15]

Figure ?? shows the inner structure of a conversation in the context of an agent and it's own policies and social commitments. When an event is processed, conversation policies are searched first for all applicable conversations, then the outer agent policies are searched, as described in §9. But the situation is actually a bit more complex than that. Conversations have variables (including a *state* variable) which can figure in the selection of applicable policies in the search. In addition, conversations can have any number of sub-conversations, which may have their variables bound to the outer conversation's variables. Furthermore, conversations automatically keep track of commitments they had instantiated in the agent's commitment store.

Conversation templates are usually created at agent initialization in the `<agent>.init-.lisp` file[16] using the `conversation` lisp operator. Conversation instances, on the other hand, are usually created by the firing of a policy to create the conversation instance using the `agent.instantiate-conversation` lisp operator in it's antecedent. These conversation-instantiating policies are using created along side the corresponding conversation template in the `<agent>.init.lisp` file.



[15] In fact, the example in Figure ?? is a Lisp function definition, `(defun ...)`, with default arguments that simplify the creation of a specific type of request server. For example, we might instantiate a conversation template to "request to register an agent".

[16] Where `<agent>` stands for the fully-qualified agent class name, for example, `casa.TransientAgent`.

```
(defun request-server
  ( ;will be named: the-act+"-request-server"
    the-act
    result-action
    &key
    ;the actual request-like performative
    (request-performative request)
    ;called for the error actions
    (exception-handler `(agent.println ...))
    ;setting this gives a chance to AGREE or REFUSE
    (request-decision `(performdescriptor 0 :performative agree))
    ;when we receive a AGREE reply to our PROPOSE
    agree-action
    ;when we receive a REFUSE reply to our PROPOSE
    (refuse-action exception-handler)
    ;when we receive a NOT_UNDERSTOOD reply to our PROPOSE
    (not-understood-action exception-handler)
    (timeout-action exception-handler)
    agree-discharge-action
    (refuse-discharge-action exception-handler)
    (not-understood-discharge-action exception-handler)
    (timeout-discharge-action exception-handler)
    (base-name "request-server")
    nopropose
    transformation
    optional-negotiation
    &aux
    (name (concatenate 'string the-act "-" base-name))
    (ask-name (concatenate 'string name "-ask"))
    (offer-name (concatenate 'string name "-offer"))
    (approver-name (concatenate 'string name "-approver"))
    )
    "A Request server-side conversation" ; doc

    ; incoming request (agent-global policy): create this kind of conversation
    (agent.put-policy
      (policy
        `(msgevent-descriptor event_messageReceived :=performative ,request-performative :act (act ,the-act))
        `(
           (agent.instantiate-conversation ,name (event.get))
           ;Do we really want this? Conversation instantiation sets the state
           ;(conversation.set-state "init") ; this should probably be the default
           )
        (concatenate 'string "Conversation2 global policy: For an incoming request/" the-act
          ", instantiate a server conversation. request-server-000")
        ;:precondition '(equal (get-conversation-state) "not started")
        ;:postcondition '(equal (conversation-state) "started")
        :name (concatenate 'string "request-server-000(" base-name ")." the-act)
        ) ; end policy
      )
```

Figure 11: An example CASA conversation (excerpt).

```
; outgoing propose (agent-global policy): create this kind of conversation
(if nopropose
  ()
  (agent.put-policy ...
    )
  )
(conversation name
  (list
    (ask-server ask-name the-act result-action
      :request-performative request-performative
      :request-decision request-decision
      :optional-negotiation optional-negotiation
      )
    (if nopropose
      ()
      (offer-server offer-name the-act result-action
        :refuse-action refuse-action
        :not-understood-action not-understood-action
        :timeout-action timeout-action
        )
      )
    (discharge-server approver-name the-act ;(act2string (act discharge perform the-act))
      :request-performative request-performative
      :agree-action agree-discharge-action
      :refuse-action refuse-discharge-action
      :not-understood-action not-understood-discharge-action
      :optional-negotiation optional-negotiation
      )
    )
  :bind-state (append `(
                        ("init" ,ask-name "init")
                        ("waiting-request" ,ask-name "waiting-request")
                        ("terminated" ,ask-name "terminated")
                        ("waiting-discharge" ,ask-name "terminated-pending")
                        )
                      (if nopropose
                        ()
                        `(
                          ("init" ,offer-name "init")
                          ("waiting-propose" ,offer-name "waiting-propose")
                          ("terminated" ,offer-name "terminated")
                          ("waiting-discharge" ,offer-name "terminated-pending")
                          )
                        )
                      `(
                        ("init" ,approver-name "blocked-init")
                        ("waiting-request" ,approver-name "blocked-request")
                        ("waiting-propose" ,approver-name "blocked-propopose")
                        ("waiting-discharge" ,approver-name "init")
                        ("waiting-propose-discharge-reply" ,approver-name "waiting-propose")
                        ("terminated" ,approver-name "terminated")
                        )
                    )
```

Figure 11: An example CASA conversation (excerpt). (continued)

```
    :bind-var '(("server"
                 (if (agent.isa (event.get-msg 'performative) request)
                   (new-url (event.get-msg 'receiver))
                   (new-url (event.get-msg 'sender))
                   )
                 )
                ("client"
                  (if (agent.isa (event.get-msg 'performative) request)
                    (new-url (event.get-msg 'sender))
                    (new-url (event.get-msg 'receiver))
                    )
                  )
                )
    :bind-var-to (append `(
                           ("client" ,ask-name "client")
                           ("server" ,ask-name "server")
                           )
                         (if nopropose
                           ()
                           `(
                              ("client" ,offer-name "client")
                              ("server" ,offer-name "server")
                              )
                           )
                         `(
                            ("client" ,approver-name "client")
                            ("server" ,approver-name "server")
                            )
                         )
    ) ;conversation name
) ;defun request-server
```

Figure 11: An example CASA conversation (excerpt). (continued)

10.1 Conversation specification

The list operator `conversation` creates a conversation template, and registers it to the agent in context. The `conversation` operator has the following definition:

```
The function's lambda list is:
(NAME BINDINGS &KEY BIND-VAR BIND-VAR-TO BIND-STATE CLASS)
Declares a conversation.
NAME (java.lang.String) The name of the conversation.
BINDINGS (org.armedbear.lisp.Cons) A Cons list of Lisp functions describing sub-conversations or policies.
:BIND-VAR (org.armedbear.lisp.Cons) A Cons list of pairs of symbol/values pairs (themselves Cons lists) that will
    be bound in the context of the conversation. The expressions are evaluated at the time the conversation is
    created.
:BIND-VAR-TO (org.armedbear.lisp.Cons) A Cons list of triples of symbol/childConversation/childSymbol (themselves
    Cons lists) that will be bound in the context of the conversation.
:BIND-STATE (org.armedbear.lisp.Cons) A Cons list of triples of state/childConversation/childState (themselves
    Cons lists) that will be bound in the context of the conversation.
:CLASS (java.lang.String) The specific subclass of a Conversation.
```

The NAME required parameter is the name of the conversation, which will be used to refer to the conversation template when, for example, we wish to instantiate an instance of a conversation from the template. The

BINDINGS required parameter is a list of Lisp expressions. Those expressions returning a policy will have that policy incorporated in the conversation; those expression returning a conversation will have that conversation incorporated as a subconversation in the conversation; those expressions returning other values will be ignored. The following subsections explain the keyed parameters in detail.

Variables

Variables are declared by listing symbol/value pairs in the `:BIND-VAR` key parameter. The symbols will be available (for read/write) as regular Lisp symbols to the policies and inner Lisp code during the course of the conversation. The initial value of the symbol in the value from the symbol/value pairs. The symbol/value pair `state/init` is automatically initialized.

Bindings

Variables in a subconversation may have their values bound to the values of variables (not necessarily with the same name) in the containing conversation by listing the mapping `:BIND-VAR-TO` parameter. It is as if the variable in the sub conversation were a pointer to the containing conversation's variable. A change in the sub conversation's variable value is instantly reflected in the corresponding change to the containing conversations' variable value, and vice-versa.

The `state` variable is a special case in which the *values* if the state valuable in the subconverstion are

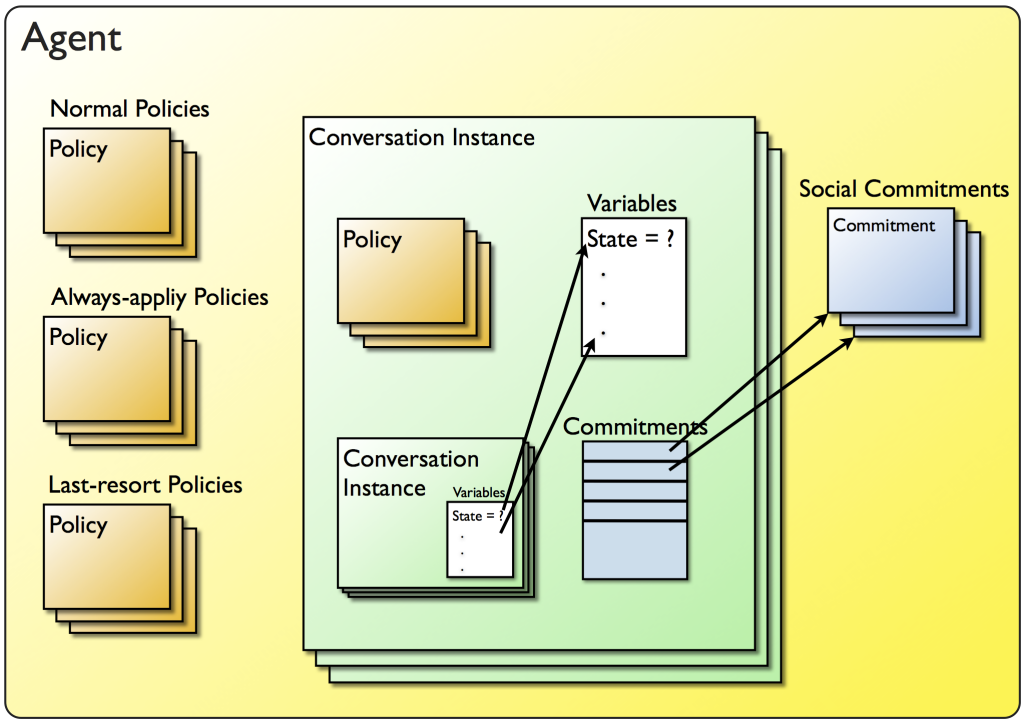


Figure 12: CASA conversations, policies, and social commitments

mapped to *values* of the `state` variable in the containing conversation. This is done with the `:BIND-STATE` key parameter. This is primarily used to like the states between the containing conversation and all subcenversations. For example, a conversation might contain two sub conversations, which should both be initially disabled. One way to do this to set `:BIND-STATE` to `:BIND-STATE '((init subc1 disabled) (init subc2 disabled))`. If later, when the state of the containing conversation becomes, say, "waiting" we want to activate the first subconversation, we could use the following setting:

```
:BIND-STATE '((init subc1 disabled)
              (init subc2 disabled)
              (waiting subc1 init)
              (waiting subc2 disabled))
```

Policies

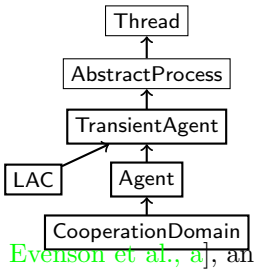
Policies in conversations are just the same as policies in the "global" context of the agent. But the they often play a special role in controlling the state (and other variables) within the conversation as a side effect of the execution of policies. Both the antecedent and the `:precondition` part of the conversation's policies are evaluated in the context of the the conversations variables as explained above. The variables are ordinary lisp symbols in the Lisp environment, so they are easily used to determine if a policy can fire. Most policies in a conversations turn themselves on and off by including a clause in the `:precondition` that evaluates to true only if the conversation is in a certain state (or set of states).

In addition, when a policy is fired it may also re-assign values to conversation variables (including the conversation's `state` variable), thus affecting the outcome of future policy applications. Typically, conversation policies will change the conversation's `state` variable.

Subconversations (Conversaionlets)

Subconversations are really no different from top-level conversations. They have their own sets of variables and, in particular the value of the `state` variable can be tied to specific values of the containing conversation using the `:bind-state` parameter in the containing conversation. Thus, if all the policies within within the policies have preconditions (`:precondition`) that require specific states, and the parent `state` value corresponds to a non-existent `state` variable in the subconversation, then the subconversation is completely disabled. In a like manner, subsets of the subconversation's policies can also be turned off and on.

11 Scripting language: Lisp



Common Lisp, as incorporated and applied in CASA, offers the programmer a great deal of design and implementation flexibility when creating agents. In terms of governing agent behaviour, policies, ontologies, and initialization scripts may all be defined with Common Lisp. As well, `casaLispOperators` may be implemented by the programmer to facilitate the interaction between the behaviour-defining scripts and the actual agent software written in Java.

CASA's Lisp implementation is Armed Bead Common Lisp [Common-Lisp.net, 2011, Evenson et al., a], an open-source project. This implementation fails roughly only 20 out of 21,702 tests in the ANSI test suite for Common Lisp[Americal National Standards Institute (ANSI), 2004].

Common Lisp is a very simple language whose entire syntax is based on parenthesis-delimited lists. For a on-line Common Lisp references, see Guy Steel's *Common Lisp the Language, 2nd Edition*[Steele, 1990] at http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html.

11.1 Ontologies

The there a 4 functions are:

`ont.relation` - define a relationship

`ont.type` - declare a terminological type

`ont.assert` - set the relationship between elements

The `ont.relation` function determines how children and ancestors on the subsumption lattice relate to each other. It takes the following arugments:

The name of the relationship

`:base` - the relation upon which the newly declared relationship is based

`:inverse` - boolean

`:reflexive` - boolean

`:symmetric` - boolean

`:transitive` - boolean

`:assignable` - boolean

For example, the definition of a *parent* relationship may take this form:

```
(ont.relation "isa-parent")
```

The programmer can now define other relationships using `isa-parent` relationship as a base class:

```
(ont.relation "isa-child"
  :base isa-parent
  :inverse :assignable)
```

Note that an `:assignable` relationship is one that can be used as the first argument of the `ont.assert` function. The definitions of the other boolean parameters for this function correspond to the common mathematical ones.

Before the `ont.assert` function can be called to define the ontology, the programmer must declare at least one terminological type using the `ont.type` function. It takes one argument: a string describing the new type.

```
(ont.type "TOP")
```

Assuming `TOP` is the root element, the programmer can now use the `ont.assert` function to build the agents' ontology:

```
(ont.assert isa-parent "action" TOP)
(ont.assert isa-parent "LAC_closing" action)
(ont.assert isa-parent "act" action)
(ont.assert isa-parent "chat_message" action)
```

See Figure 6 for a more complete example.

11.2 Initialization

Initialization Scripts CASA initialization scripts may be written entirely in Common Lisp. As such, the programmer may utilize the full breadth of the Common Lisp language. What follows is a description of a few key functions required to initialize a typical multi-agent system.

The first, and most important, function is `new-agent`. It takes the following parameters:

type - the type (Java class) of agent to be created

name - name of the agent (java.lang.String)

port - port of the agent (-ve value indicates to 'hunt') (java.lang.Integer)

:process - specifies the process in which to run the agent (i.e., "LAC", "CURRENT", or "INDEPENDENT")

:lacport - specifies the port of the LAC the agent registers with (-1 indicates not to register) (java.lang.Integer)

:shortcutting - set shortcutting in message protocols on or off (java.lang.Boolean)

:ack - turn acknowledge (ack) requirement on or off (java.lang.Boolean)

:markup - specify the markup language for inter-agent messages (affects the entire process globally) (i.e., "KQML" or "XML")

:persistent - turn persistent saving of agent data on or off (java.lang.Boolean)

:root - root directory for the CASA files (LAC only) (java.lang.String)

:debug - turn CASA debugging on or off for the agent's entire process (java.lang.Boolean)

:trace - turn set the trace flags. Bits are: 1=off, 2=on, 4=monitor, 8=toFile) (java.lang.Integer)

:tracefile - turn file tracing on or off (only effective if trace=true) (java.lang.Boolean)

:tracewindow - turn the trace window on or off (only effective if trace=true) (java.lang.Boolean)

:tracetags - a java.lang.String list of trace tags (identifiers) to add—remove. Remove a tag by preceding it with a '-'. Current valid tags: calls, msg, msgHandling, warning, info, policies, commitments

:interface - the fully-qualified Java class name of the interface for the agent to use. Defaults to an appropriate window interface. The special [name] 'text' yields a default text interface; 'none' specifies the agent should have no interface

:guard - turn the guard (secure) on or off for the agent (java.lang.Boolean)

:proxies - add proxies to the agent (a semi-colon separated list of fully-qualified class names) (java.lang.String)

:strategy - the desired strategy: SC, reactive, BDI, or SC3 (java.lang.String)

:security - the desired security package. Currently, "casa.security" or "none"

:ontologyengine - the Java class for the ontology engine (currently, either casa.ontology.CASAOntology or casa.util.TypeHierarchy)

:ontologyfile - the file from which the ontology engine should read initialization data (java.lang.String)

Agent-specific keys - any other key/value pairs read by a subclass of `Agent` or `TranientAgent`

The `new-agent` function takes the following form:

```
(new-agent "casa.LAC" "ResearchLAC" 9000
  :process "CURRENT" :markup "KQML"
  :trace :traceFile :strategy "sc3"
  :traceTags
    "warning,msg,msgHandling,commitments,
    policies5,lisp,conversations")
```

Given CASA's threaded nature, it is often necessary to allow certain critical agents (the Local Area Coordinator, especially) time to fully initialize. If the LAC is not given enough times, it may be the case that the agents that depend on the LAC will initialize before it is ready. The simplest way to solve this problem is to delay execution of the script long enough so that critical agents will be fully initialized before their dependents. The `sleep` function allows the programmer to delay execution. The following example shows how `sleep` may be called to delay script execution for five seconds:

```
(sleep 5)
```

Besides the LAC, there are two other types of agents that are of interest here: the `TransientAgent` and the `CooperationDomain`. `TransientAgent` is the base class from which CASA agents typically extend (including the `LAC` and `CooperationDomains`). In the example that follows, they will take the place of the more specialized agents the programmer may create. The `CooperationDomain` agent, as its name suggests, provides a proxy through which agents communicate and form partnerships. The following example demonstrates how to initialize two `TransientAgents` and a `CooperationDomain`:

```
(new-agent "casa.TransientAgent" "Alice" 6700
  :LACPORT 9000 :markup "KQML"
  :persistent :trace :traceFile :strategy "sc3"
  :traceTags
    "warning,msg,msgHandling,commitments,
    policies,conversations")

(new-agent "casa.TransientAgent" "Bob" 6701
  :LACPORT 9000 :markup "KQML"
  :persistent :trace :traceFile :strategy "sc3"
  :traceTags
    "warning,msg,msgHandling,commitments,
    policies,conversations")

(new-agent "casa.CooperationDomain" "CD" 8700
  :LACPORT 9000 :markup "KQML"
  :persistent :trace :traceFile :strategy "sc3"
  :traceTags
    "warning,msg,msgHandling,commitments,
    policies,conversations")
```

Once initialized, the agents can be scripted to join the `CooperationDomain` in the following manner:

```
(sleep 5)
(tell "6700" "(join \"8700\")")
(tell "6701" "(join \"8700\")")
```

The `tell` function sends a `request/execute` message to the agent URL specified (here, the agents' port numbers). The content of that message specifies the requested action. Note that the content contains string data, which necessitates the use of escaped quotes.

11.3 Custom Lisp Operators

All of the Common Lisp functions seen in the previous section were defined with the abstract `casa.abcl.casaLispOperator` class in Java. These classes may be declared anywhere the programmer feels appropriate. Once declared `static` with the necessary method implemented, they may be used in the same way as any Common Lisp command. A `casaLispOperator` object takes four parameters:

1. The name of the Common Lisp function (java.lang.String)
2. The list of arguments required by the function (java.lang.String)
3. The class of the agent to which this function is registered (extends casa.TransientAgent)
4. Any number of optional, comma-separated function *synonyms* (java.lang.String...)

Of the four parameters the Common Lisp argument list requires the most explanation. As specified, the list is simply an ordinary `String`. In its simplest form it states the names of the parameters, all separated by spaces. For instance, the following would be an acceptable to `casaLispOperator`'s second parameter:

```
"PERFORMATIVE ACT RECEIVER"
```

A `casaLispOperator` with an argument list specified this way requires three parameters. The names provided to these parameters are used by the abstract `casaLispOperator` class to retrieve the data passed to the Common Lisp function. For example:

```
String receiver = params.getJavaObject("RECEIVER");
```

In addition to the parameter names, the argument list can accept two types of documentation strings: *type data*, which is prefixed with a *@*, and a simple description, which is prefixed with a *!*. Each documentation string must be enclosed in its own set of double quotes. For example:

```
"\"!Sends a message to another agent.\" "
  + "MESSAGE \"@casa.MLMessage\" \"!The message to send\" "
```

Here, the overall purpose of the function is described (i.e., "Sends a message to another agent") , the parameter name is declared (i.e., "MESSAGE"), and the parameter's expected type and description are provided (i.e., type: `casa.MLMessage`, description: "The message to send").

`casaLispOperator` also allows the programmer to specify `&optional`, `&rest`, `&key`, and `&allow-other-keys` parameters usual to Common Lisp. Without touching on the purpose of each parameter-type, the following demonstrates how they are implemented (further to the example above):

```
"\"!Sends a message to another agent.\" "
  + "MESSAGE \"@casa.MLMessage\"  \"!The message to send\" "
  + "&KEY "
  + "PROXY \"!Send the message through the indicated proxy (optional)\" "
```

Once the new `casaLispOperator` has been instantiated, it is good practice to immediately define the object's abstract `execute` method, which takes three parameters:

1. The agent object (casa.TransientAgent)
2. A parameter list (casa.abcl.ParamsMap)
3. The agent's user interace (casa.ui.AgentUI)

Since `casaLispOperators` are declared `static` (as mentioned previously), the *agent* object is necessary to make method calls. The agent's user interface allows the programmer to output information to the agent window wherever appropriate. The parameter list contains the parameters passed to the function declared. The `tell` function's `execute` method is shown below:

```
1  @Override
2  public Status execute(TransientAgent agent, ParamsMap params, AgentUI ui) {
3    try {
4      URLDescriptor url = new URLDescriptor((String)params.getJavaObject("AGENT"));
5      String content = (String)params.getJavaObject("COMMAND");
6      return agent.sendMessage(ML.REQUEST,ML.EXECUTE, url, ML.CONTENT, content);
7    } catch (URLDescriptorException e) {
8      return new Status(-6,"Bad URL: "+e.toString());
9    }
10 }
```

`execute` returns a `Status` or `StatusObject`. The latter extends the former. The `Status` object contains an integer execution code (negatives indicate an error, by convention) and a `String` *explanation*. There are times, however, when it is appropriate to return an object upon execution. The `StatusObject` meets this need. For instance, if a `Boolean` value is desired, `execute`'s `return` statement may look like this:

```
return new StatusObject<Boolean>(0, false);
```

What follows is a compete example (from `TransientAgent`) showing the coding of the Lisp command `agent.show-conversations`:

```
1    /**
2     * Lisp operator: (AGENT.SHOW-CONVERSATIONS)<br>
3     * Show the agent's current or known conversations.
4     */
5    @SuppressWarnings("unused")
6    private static final CasaLispOperator AGENT__SHOW_CONVERSATIONS =
7    new CasaLispOperator("AGENT.SHOW-CONVERSATIONS", "\"!Show the conversations -- known, current, or a "+
8        + "specific (known) one.\" "
9        +"&OPTIONAL NAME \"@java.lang.String\" \"!If present shows the named template, otherwise all are shown.\" "
10       +"&KEY "
11       +"CURRENT \"!Show the current conversations for the current agent; ignores the NAME parameter.\" "
12       +"(VERBOSE 1) \"@java.lang.Integer\" \"!0=only names; 1=state with no policies; 2=all\" "
13       , TransientAgent.class, "SCONV")
14   {
15     @Override public Status execute (TransientAgent agent, ParamsMap params, AgentUI ui, Environment env) {
16       int verbose = (Integer)params.getJavaObject("VERBOSE");
17       boolean current = params.containsKey("CURRENT");
18       String name = (String)params.getJavaObject("NAME");
19
20       agent.conversations.purge();
21
```

```
22        if (current || name==null) {
23          Pair<Integer, String> ret = agent.getConversationsReport(current, verbose);
24          ui.println(ret.getSecond());
25          return new StatusObject<Integer>(0,ret.getFirst());
26        }
27
28        int count = 0;
29        Conversation conv = Conversation.findConversation(name);
30        if (conv==null)
31          ui.println("No such conversation");
32        else {
33          if (verbose==0)
34            ui.println(conv.getName()+(conv.getId()==null?"":(", id:"+conv.getId()+", state:"+conv.getState())));
35          else
36            ui.println(conv.toString(0,verbose==1));
37          count++;
38        }
39        return new StatusObject<Integer>(0,count);
40      }
41    };
```

Lines 7&8 demonstrate a documentation line, line 9 demonstrates an optional (unnamed) parameter, lines 10-12 demonstrate the use of (optional) keyword parameters, line 12 demonstrates the use of a default value (1 in this case), and line 13 demonstrates the declaration of synonym.

12 Observers

CASA uses the Observer pattern [Gamma et al., 1994] for several different purposes. The most important of which is to enable flexible user interfaces: User interfaces are normally not know to agents, but rather, interfaces are *observers* of agents. This allows agents to have a variety of interfaces, and even multiple interfaces simultaneously.

A problem arises in CASA's agent implementation though: The agent classes inherit from `Thread`, and Java's implantation of *Observer* from Gamma et al. makes the `Observable` class a class, so the agent classes cannot also inherit from `Obervable`. CASA gets around that problem by implementing the interface `casa.CASAObservable` (which can stand in for `Observable`) and the class casa.CASAObservableObject (which extends `Observable` and implements `CASAObservable`.

`CASAObservable` also adds the capability of remote (inter-process and inter-computer) observation via CASA's inter-agent message passing capability.

To observe an agent's events you should implement the interface `java.util.Observer`. The `update (Observable, Object)` method you will implement will be called for each agent event. Unlike a typical `update()`, the first (`Observable`) parameter will NOT be a pointer to the agent (it will be a pointer to an `ObservableObject`, which you don't usually want. However the second (`Object`) argument will be an `ObserverNotification` object, through which you can access the agent, event type (which will be subtype of event in the ontology), and any relevant auxiliary object, depending on the event type. These are accessed through the `ObserverNotification` object's `getAgent()`, `getType()`, and `getObject()` methods respectively.

13 User Interfaces

CASA's interfaces are usually not known to an agent at all. Instead, they are observers (see §12) of an agent. In this way, CASA agents can support as many and as varied interfaces as one would want.

However, CASA agents usually support one or more default user interfaces. By default, agents create a graphical user interface of a subclass of `casa.ui.TransientAgentInternalFrame`. The agent constructor parameter "INTERFACE" allows one to switch to the default text-only interface using the value "text" (see Appendix §E). The default text user interface is a subclass of `casa.ui.TextInterface.TextInterface`.

The default text user interface is nothing more than an Lisp interpreter in the context of the agent. The default graphical user interface is more complex, but quite flexible, and is explained in the next subsection.

13.1 The Default GUI

Figure 13.1 shows the default GUI as produced by the `casa.ui.TransientAgentInternalFrame` class. You can replace it by overriding the method `makeDefaultInternalFrame(TransientAgent agent, String title, Container aFrame)`, which returns a `casa.ui.TransientAgentInternalFrame`. If you want to replace it with an interface that is not a subclass of `casa.ui.TransientAgentInternalFrame`, then override method `makeDefaultGUI (String[] args))` which must only return a class implementing the `casa.ui.AgentUI` interface.

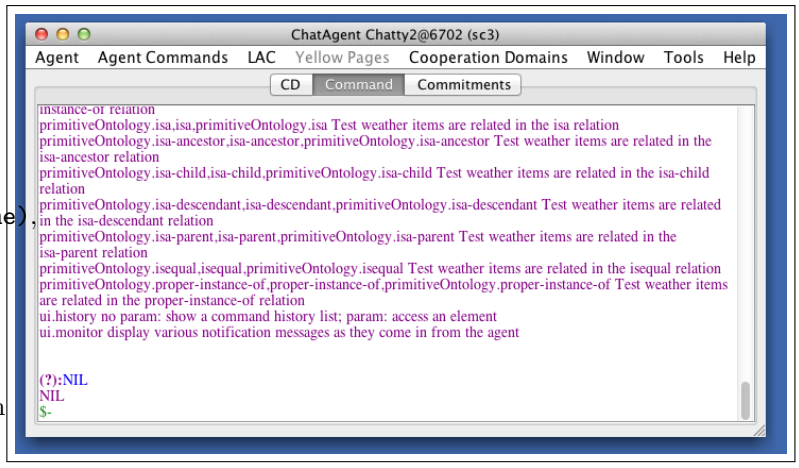


Figure 13: The default GUI produced by class `casa.ui.TransientAgentInternalFrame`.

You can add your own tab to a default interface by calling its `addTab(String title, Component component, boolean visible)` method. The easiest way to do this is to subclass and override the agent's makeTabbedPane() method. For example, to add a tab labeled "My Tab" containing a `JPanel`:

```
1  @Override
2  protected JTabbedPane makeTabPane() {
3    JTabbedPane ret = super.makeTabPane();
4    addTab("My Tab", new JPanel());
5    return ret;
6  }
```

Likewise, you can add or remove items from the the menu bar using the following methods:

```
public JMenu getMenuBarMenu(String name)
public void insertMenuBar(JMenu menu, int location)
public void insertMenuBarBefore(JMenu menu, String name)
public void insertMenuBarAfter(JMenu menu, String name)
public void replaceMenuBar(JMenu menu)
```

To use the above methods, it's easiest to subclass and override the `makeMenuBar()` method:

```
1  @Override
2  protected JMenuBar makeMenuBar () {
```

```
3     JMenuBar ret = super.makeMenuBar();
4     <your code>
5     return ret;
6   }
```

14 Security

15 Persistence

An agent is persistent if it inherits from the `casa.Agent` class and sets the attribute `CasaPersistent` to true (which can be done either in the agent code of via the command line). The agent stores the information in a file according the the LAC's setup. All you have to do to make an attribute persistent is to mark it's declaration with the `@CasaPersistent` annotation. For example:

```
// a simple persistent boolean attribute that will
// be stored in the properties under "myFlag"
@Persistent
boolean myFlag;

//a persistent object that will be stored in the
// properties under "options.x" and "options.y"
// because it has at least one @CasaPersistent
// attribute itself.
public class Options {
  @CasaPersistent int x = 4;
  @CasaPersistent double y = 7.5;
}

@CasaPersistent
Options options;

// a persistent object that will be stored in the
// properties under "stuff" in the standard CASA
// serial format (because none of it's properties
// are marked @CasaPersistent).  Note that
// the class Stuff MUST have a toString()
// method and a corresponding constructor
// that takes a single string.
public class Stuff {
  int x = 4;
  double y = 7.5;
  public Stuff() {...}
  public Stuff(String persistData) {...}
  @Override
  public toString() {...}
}

@CasaPersistent
Stuff stuff = new Stuff();
```

An object that is not part of the agent but can access the agent (such as UI object) can store and retrieve information with the agent. For example, to store the boolean variable `isIcon` with the agent in the variable `agent` under the name "isIcon":

```
agent.setBooleanProperty("isIcon",isIcon);
```

Later, or in a new invokation of the agent, the stored value of `isIcon` can be retrieved:

```
Boolean isIcon = false; //establish a default
try { // if it isn't stored, it will throw
  isIcon = lac.getBooleanProperty("isIcon");
} catch (PropertyException e) {
  // just accept the default
}
```

16 Debugging/Logging (Trace)

Due to it's dynamic and interpretive nature, CASA agents can be difficult to debug. To facilitate tracing what's going on in the program, CASA provides a flexible logging system. If the agent is tracing (see §16.3), it will record information in any of several places. One of the places is a file name with the agent's name and the extension ".log" in the current directory. The agent will write status information to this file as it runs. See §16.4 for details about the format of this file. Trace logging also generates *observer events*, so any interface observing the agent may pick up this information. For example, one can create a *trace window* that shows a live listing of these messages. In another example, the default agent interface's "Command" tab can display these messages in real time (use the command-tab's right-click popup menu to turn this on).

The information that gets saved to the trace file is controlled at run time by setting *trace tags*: a particular message is printed *only if* the associated trace tag is switched on. Trace tags are in the form of a string *topic name* immediately followed optionally by a single-digit severity specifier. For example, "warning9" is trace tag that causes an agent to print all warning messages to the greatest detail on topic "warning". Trace tag topic names can be any alpha-numeric string not containing white space and not ending in a digit. Trace tag severity specifiers can be any single digit from 0 to 9. If the trace tag severity specifier is missing, it is taken to the 0. Therefore, "msg" is taken to be the same as "msg0".

16.1 Specifying Trace Tags

Users specify tags either at the time an agent is instantiated, or anytime at run time. If an agent is persistent (see §15) trace tags may also be recovered from the persistence file between invocations. To specify trace tags at agent instantiation, use the `:trace-tags` keyword parameter on the `agent.new-agent` Lisp command. To specify trace tags at run time, use one of:

- the `Agent | Options` dialog box
- the run-time Lisp command `(agent.options :options.traceTags ...)`,
- the Java method `AbstractAgent.addTraceTags(String)`
- the Java method `Trace.addTraceTags(String)`.

The syntax of setting trace tags in all of these ways is a comma-separated list of tags, optionally preceded by a "-" sign to indicate "remove it" instead of "add it":

[-] <name><digit> [, [-] <name><digit> ...]

For example:

```
(agent.options :options.traceTags "error,warning6,-msg,-commitments")
```

The special trace tag `error` is always implicitly on.

Programmers specify trace tags and severities merely by mentioning them in a call to any of `AbstractProcess`'s `print()` methods. For example,

```
println("msg5", "Malformed message");
```

will log an entry with the string "Malformed message" to the trace file only if the tag "msg" is set to severity 5 or greater ("msg5", "msg6", ...).

16.2 Using `println()`

CASA agents have several forms of `println()`. These are listed here:

```
1 println(<tag>, <message>);
2 println(<tag>, <message>, <Throwable>);
3 println(<tag>, <message>, <Status>);
4 println(<tag>, <message>, <option-flags>);
5 println(<tag>, <message>, <Throwable>, <option-flags>);
6 println(<tag>, <message>, <Status>, <option-flags>);
```

The message will also always be logged if `<tag>` is null, an empty string or "`error`". Otherwise, the message will be ignored unless the appropriate tag has been set to the appropriate level.

In forms 2 and 5, the message will be conditionally logged followed by the message and stack trace found in the Throwable object. As a special case, if `<Throwable>` is null, then a stack trace will be constructed (starting with the caller of the `println()` method). Note that `println(<tag>, <message>, null)` is ambiguous between forms 2 and 3, so you will need to make the call as `println(<tag>, <message>, (Throwable)null)`.

Form 3 conditionally logs the message along with an appropriately formatted Status object.

The behaviour of the `println()` methods may be modified by using the `option-flags` parameter. Option flags include the following:

- `Trace.OPT_SUPRESS_STACK_TRACE`: Default for print methods that do not include an Exception parameter or method calls that have a null Exception parameter.
- `Trace.OPT_FORCE_STACK_TRACE`: Default for print methods that include an Exception parameter where the Exception is non-null. Allowable, but expensive for other operations.
- `Trace.OPT_INCLUDE_CODE_LINE_NUMBER`: Default for error and warning tags. And expensive operation.
- `Trace.OPT_SUPPRESS_AGENT_LOG`: Only applicable to the process-wide Trace; forces output to sysout rather than trying to find an agent.
- `Trace.OPT_COPY_TO_SYSOUT`: Copies output to sysout.
- `Trace.OPT_COPY_TO_SYSERR`: Copies output to syserr.
- `Trace.OPT_SUPPRESS_HEADER_ON_SYSOUT`: Suppresses the header block and ¿¿¿¿ flagging when printing to sysout or syserr

If you want to log to an agent's file, but there is no agent is scope, you can use the state `Trace` family of methods that exactly mirrors the `println()` family of methods. For example, `Trace.log(<tag>, <message>)`

or `Trace.log(<tag>, <message>, <Throwable>)`. These methods search for the appropriate in-scope agent using the current Thread as a clue, and then call that agent's `println()` method. If an agent is not found, the message is similarly logged either the process-wide trace file ("casaOut¡*digits*¿.log") or to `System.out` or to both.

16.3 Turning on tracing

Logging can be turned on or off at the time an agent is instantiated, or anytime at run time. If an agent is persistent (see §15) tracing behaviour may also be recovered from the persistence file between invocations. To specify tracing at agent instantiation, use the `:trace` keyword parameter on the `agent.new-agent` Lisp command. To specify trace tags at run time, use either the `Agent | Options` dialog box, or the run-time Lisp command `(agent.options :options.trace ...)`.

The value used in all these cases is an integer interpreted as a bit vector with the following interpretations, as per Table 4.

Bit	Value	Meaning
0	1	turn tracing off
1	2	turn tracing on
2	4	trace to a separate logging window
3	8	trace to a file in the current directory named `<agent-name>.log`

Table 4: Bit meanings for the `:trace` keyword.

16.4 Trace file format

Figure 14 shows a typical log file output. The first line starts with "* * *" and lists the date. After that, there are several lines similarly starting with "* * *" that give further information. Every log entry may have multiple lines but always starts with a *header*, which is bracketed by "[*" and "*]". The entry terminates when either another header is encountered, or a *special line* is encountered. A special line is a line that starts with either ">>>" or "<<<". These special lines are merely flags for human readability to delineate error messages (long lines of ">"s) and warning messages (shorter lines of ">"s), and these may be safely ignored by parsers.

Headers are always between "[*" and "*]" brackets and contain the following colon-delineated information:

```
header  :-  [* <time> : <agent-name> : <trace-tag> : <tread-name> *]
time    :-  <hour> . <min> . <sec> . <millisec>
```

The error and warning starting lines (">>>") also contain the complete method name and source code file and line number of the caller of the `println()`. (Or at least CASA's best guess...)

17 Adding extensions

CASA has three types of extensions: Code extensions, tab extensions and Lisp-script extensions. The first two are java code extensions that exist in jar files. Lisp-script extension are Lisp files that can be conveniently executed form the `Tools|List Scripts` menu. All extensions reside in the either the directory `~/.casa/extensions/` or the CASA executable jar file in the directory `/extensions/`.

```
*** Started log at Fri Apr 04 16:23:00 MDT 2014 ***
*** header format: [* <timestamp> : <traceName> : <tagNameAndDigit> : <TreadName> *]
***   where <tracename> is usually the agent name.
[*16.23.01.502:coolness:msg:coolness-subthread0*] Sending message (TCP/IP):
( request
  :act register_instance
  :sender casa://kremer@10.0.1.2:8700/casa/CooperationDomain/coolness
  :receiver casa://10.0.1.2:9000
  :reply-by "Fri Apr 04 16:24:01 MDT 2014"
  :reply-with casa://kremer@10.0.1.2:8700/casa/CooperationDomain/coolness--0
  :conversation-id casa://kremer@10.0.1.2:8700/casa/CooperationDomain/coolness--0
  :language casa.*
  :language-version 1.0
  :content "[\"(casa.URLDescriptor)\\\"casa://kremer@10.0.1.2:8700/casa/CooperationDomain/coolness\\\"\", \"(...
  :priority 10
  :timeout "Wed Dec 31 17:00:10 MST 1969"
)
>>>>>>>>>>>>>>>>>>>>>>>> casa.TransientAgent$63.run(TransientAgent.java:4182)
[*16.23.11.506:coolness:error:coolness-subthread0*] TransientAgent failed to register with LAC: request timed...
<<<<<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>>>> casa.CasaObservableObject.deleteObserver(CasaObservableObject.java:377)
[*16.23.11.900:coolness:warning:coolness-subthread0*] Attempt to delete Observer "casa.AbstractProcess$1Abst...
        casa.AbstractProcess$1AbstractProcess_MsgObserverEvent-38(event_SCPerformAction), coolness, 0 observers,
        casa.AbstractProcess$1AbstractProcess_MsgObserverEvent-36(event_SCPerformAction), coolness, 0 observers,
        casa.AbstractProcess$1AbstractProcess_MsgObserverEvent-37(event_SCPerformAction), coolness, 0 observers,
        casa.AbstractProcess$1AbstractProcess_MsgObserverEvent-39(event_SCPerformAction), coolness, 0 observers,
Stack trace::
        casa.CasaObservableObject.deleteObserver(CasaObservableObject.java:377)
        casa.AbstractProcess.deleteObserver(AbstractProcess.java:2168)
        casa.event.MessageObserverEvent.cancel(MessageObserverEvent.java:111)
        casa.AbstractProcess.sendRequestAndWait(AbstractProcess.java:1462)
        casa.TransientAgent$60.run(TransientAgent.java:3610)
        java.lang.Thread.run(Thread.java:680)
<<<<<<<<<<<<<
```

Figure 14: An example log file listing (edited).

You can see the extensions loaded by using the standard agent interface's menu item `Tools|Extensions` to display a dialog box listing the tools.

17.1 Code extensions

This extension constitutes a java class that will be added into the class path at runtime. If the attribute `autoload` is true, then that class will be loaded into memory, causing any statics in the class to be initialized. All code extensions are found in a jar file, and are defined in the jar's manifest file in sections named `code` and optionally followed by a string of digits (which allows one to define several different code extensions in a single jar file).

Code extensions are not intrinsically associated with any particular agent (but subtypes of code extensions may be).

This extension is affected by the following attributes defined in manifest sections beginning with `code`:

Main-Class (required) The fully-qualified class name in the jar file. If `autoload` is true, then this class static

initialization will occur on loading the CASA application. If this attribute is omitted, and there is a `Main-Class` attribute defined in the manifest's main section, then this value will be used by default.

autoload (optional) iff "true" the class will be loaded into memory and it's static initialization will occur when the CASA application loads. The default is "true".

Extension-Name (optional) This is the display name of the extension. The default is the name of the class.

doc (optional) A short documentation string that may be used for mouse-overs, etc.

17.2 Tab extensions

Tab extensions specializations of *code extensions* that are UI elements that appear as tabs on the agents' default GUI. The class (`Main-Class`) that defines a tab extension is a Swing `Component`. The implementation is "lazy" in that the `Main-Class` is never loaded and instantiated until the tab is actually displayed.

Tab extensions are associated with a particular agent and a particular agent GUI frame. The association is restricted to using attributes `agenttype` and `frametype`.

This extension is affected by the following attributes defined in manifest sections beginning with `tab`:

Main-Class (required) The fully-qualified class name in the jar file that defines a subtype of `java.awt.Component` with a constructor with two parameters: (`TransientAgent`, `AbstractInternalFrame`). This will be invoked as the main component in the tab. If this attribute is omitted, and there is a `Main-Class` attribute defined in the manifest's main section, then this value will be used by default.

autoload (optional) iff "true" the tab will be will be displayed in the GUI, otherwise the user can choose `Tools | Tabs | <Extension-Name>` to display the tab. The default is "false".

TabName (optional) The name to appear on the tab itself. The default is the `Extension-Name` or the `Main-Class` if necessary.

AgentType (optional) The fully qualified class name of that restricts the type of the agent this tab will apply to. The default is "casa.TransientAgent".

frametype (optional) The fully qualified class name of that restricts the type of the GUI frame this tab will apply to. The default is "casa.ui.AbstractInternalFrame".

doc (optional) A short documentation string that may be used for mouse-overs, etc.

Note that by default, the jar will NOT be loaded and tabs will NOT be displayed when the interface is initialized. To display the tab interface use the menu bar "Windows — Tabs — [tabName]". To actually load the jar and run the tab code, choose the tab to display it – the jar will only be loaded and run at that point.

Thus, a manifest might look like Figure 15. A target in an ANT script to build a jar with this manifest would look like Figure **??**:

17.3 Lisp-Script Extensions

Lisp script extensions are merely Lisp code files that can be executed in one of three contexts: at process startup, at agent startup, or during agent runtime in the context of a GUI (typically by the use selecting some element from the menu `Tools|Lisp Scripts|...`).

This extension is affected by the following attributes defined in manifest sections beginning with `Lisp-Script`:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.1
Created-By: 20.65-b04-462 (Apple Inc.)
Main-Class: CASA
casa-build-time: 2014/02/23 08:38 PM
Class-Path: .

Name: tabs0
Main-Class: casaTools.SCTab
TabName: SCs
agenttype: casa.TransientAgent
frametype: casa.ui.AbstractInternalFrame
autoload: false
```

Figure 15: An example manifest file in an CASA extension jar file.

```
<target name="distGenericBin" description="builds distribution binary jar" depends="setProperties">
  <jar destfile="${distDir}/${distName}.jar" filesetmanifest="skip">
    <manifest>
      <attribute name="Main-Class" value="CASA"/>
      <attribute name="Class-Path" value="."/>
      <attribute name="casa-build-time" value="${build.time}"/>
      <section name="tabs0">
        <attribute name="Main-Class" value="casaTools.SCTab"/>
        <attribute name="TabName" value="SCs"/>
        <attribute name="agenttype" value="casa.TransientAgent"/>
        <attribute name="frametype" value="casa.ui.AbstractInternalFrame"/>
        <attribute name="autoload" value="false"/>
      </section>
    </manifest>
    <fileset dir="/Apple/CASATools/bin"/>
  </jar>
</target>
```

Figure 16: An example Ant target to build the manifest in Figure 15.

AgentType (optional) The fully qualified class name of that restricts the type of the agent this script will apply to. The default is "casa.TransientAgent".

frametype (optional) The fully qualified class name of that restricts the type of the GUI frame this script will apply to. The default is "casa.ui.AbstractInternalFrame".

once (optional) If this attribute is set to true, the script will be executed at application startup time and will NOT appear in any menus, etc.

oncePerAgent (optional) If this attribute is set to true, the script will be executed at agent startup time (at the end of thread initialization) and will NOT appear in any menus, etc.

doc (optional) A short documentation string that may be used for mouse-overs, etc.

Appendices

Identifier	Path	Type	Default	Explanation
LACdefaultport	org/ksg/casa	env, user, system	9000	The TCP/IP port for the LAC an agent will use if it is not specified on in the agent's startup.
synonymURLs	org/ksg/casa	user		
knownTraceTags	org/ksg/casa	user, system		The tr35 ace tags known to the system; these are built up automatically over time.
router	org/ksg/casa	env, user, system		The InetAddress in the form ¡int¿.¡int¿.¡int¿.¡int¿ that CASA will use as the local host in URLs. If this is not included, CASA will use the value returned by `InetAddess.getlocalHost()`
rootDirectory	org/ksg/casa	env, user, system	∼/.casa/	The root directory under which all the CASA persistent data is stored.
casa.MLMessage-subclasses	org/ksg/casa	env, user, system		Users should usually never use this preference, but if you are having trouble loading a MLMessage subtype class or it has an irregular name (other than "<markup-specifier>Message", then you can use this dictionary to specify it explicitly. Otherwise, CASA will automatically detect the subtype and add it to this preference (as long as the subtype class is loaded by the loader early enough in the startup process).
defaultMarkup-Language	org/ksg/casa	env, user, system	KQML	The default markup language to use when sending messages.

Table 5: CASA preference identifiers

A Searching for Resources

File resources (such as ontology files, lisp initialization files, etc) are searched as follows:

1. the directory specified in the system property *casa.home* (if *casa.home* is defined)
2. the directory specified in the system property *casa.home*, subdirectroy `dataFiles` (if *casa.home* is defined)
3. the directory specified in the system property *user.home*, subdirectory `casa/dataFiles`
4. the current working directory (".")
5. through the Java `ClassLoader.getSystemResource()` method, which picks up files included with the CASA distribution.

B Preference Settings

This section documents the preferences CASA uses from the system-dependent preferences system. CASA typically looks in the system environment variables, then the user preferences file (or registry), then the system preferences file (or registry), but some values are only sought in a specific subset of these as documented in Table 5. CASA's environment variable names are always prefixed with "CASA_".

Most systems have environment variables, and CASA can access them (read only) through the Java System interface. Most systems also have a way to store user- and system- preferences, and CASA can access them (read/write) through the Java System interface as well. User and System preferences are stored differently on different systems. For example, in Windows, they are stored on in the system registry. On Mac OS/X the users preferences file is stored in their home directory (∼`/Library/Preferences/`); the system preferences are stored

in `/Library/Preferences/`[17] and are only persisted to disk if the user is an administrator. The files are named `org.ksg.casa.plist`.

The search for these preferences is encapsulated by static method `org.ksg.casa.CASA.String getPreference (String prefName, String defaultValue, int operation)`, where operation is a disjunct of `CASA.ENVIRONMENT`, `CASA.USER`, and `CASA.SYSTEM` (a zero will default to search all).

C Matching Operators (event descriptors)

Matching operators are usually 1-character symbols which appear between the colour (":") and identifier in key specs in `EventDescriptors` and `MsgEventDescriptors`. For example,

```
(msgevent-descriptor event_messageSent :performative subscribe
          :act (act ,the-act)
          :sender client
          :receiver server
          :*language "FIPA-SL"
          )
```

declares that the language field should match "FIPA-SL" using a regular expression match. The operators are:

= equal

! not equal

< isa (as per the agent's ontology)

!< not isa

* regular expression comparison (as per `java.util.regex.Pattern`)

The default operator depends on the key. Defaults are:

performative: < (isa)

act: < (isa)

type: < (isa)

all others: = (equal)

D Properties, Parameters. Persistence, and Options

CASA's `PropertiesMap` and `ParamsMap` classes look very similar. Both are dictionaries from String identifiers to objects.`ParamsMap` is targeted at providing an interface between Lisp and Java, and always includes both Lisp values and (more or less) equivalent Java values. It is also used to pass the many possible (and optional) qualifiers to an agent constructor as an argument. `PropertiesMap`, on the other hand, is used to store values destined to the persistent store in a flexible way. When a persistent CASA agent starts, it reads from the

[17] Note that on Mac OS/X Lion, only the root user may write to `/Library/Preferences/` so CASA will only write user preferences normally.

persistent store in to a `PropertiesMap` before further processing, and when it exits, it stores persistent data to a `PropertiesMap` before writing it out to the persistent store.

A related data structure is the `Options` class which more-or-less mirrors the `PropertiesMap` but contains direct, Java-native structures for much faster access. Since options "real" Java data structures they lack the flexibility of dictionaries and every class that needs to additional options must create a new class that `extend`s the superclass's `Options` class with it's own. To do this, every class that uses an extended `Options` class needs to override the `makeOptions` method and to return its correct version of the `Options` class. The options

E Command Line Qualifiers

If you are running CASA from the command line (front the .jar file), the command line is:

```
java -jar casa.jar options run-time-command
```

or, more specifically:

```
java -jar casa.jar casa.jar [-lLtT?] [-LAC [<port>]] [-NOLAC [<port>]] [-PROCESS [<port>]]
  [-NOPROCESS] [-TAGS <tags>] [-HELP] [<lisp-command>]
```

where:

- l: Suppress automatically stating a LAC if there isn't one (same as -NOLAC).
- L: Automatically start a LAC at port 9000 if there isn't one (same as -LAC).
- t: turn tracing-to-file on.
- T: Use a text interface if there's no command on the command line.
- ?, HELP: Prints this help text.
- LAC [*port*]: Automatically start a LAC at port *port* if there isn't one (defaults to 9000).
- NOLAC [*port*]: Do not start a LAC, but expect a LAC at port *port* (defaults to 9000).
- PROCESS [*port*]: Automatically start a CASAProcess at port *port* (defaults to 9010).
- NOPROCESS [*port*]: Do not start a PROCESS, but run a simple agent that executes the *lisp-command*.
- TAGS [*tag-list*]: A comma-delimited list of tags that the process-wide trace should track. May contain "-" prefixes and digit suffixes.
- *lisp-command*: any legal agent run-time command (runs dialogue mode if this is missing).

L, -LAC, -PROCESS, and -NOPROCESS are mutually exclusive. If none of -L, -LAC, -PROCESS, and -NOPROCESS are present then if a LAC exists at -NOLAC (or 9000), then a CASAProcess is started at 9010 or above, otherwise a LAC is started. Qualifiers can be abbreviated to the shortest unique truncation.

The main class is `casa.CASAProcess`.

Agents are designed to be started up from the Lisp command `(agent.new-agent` *`type name port ...`*`)`, which allows for a large number of optional, named parameters. These are listed in Table 6. Most often the command line startup's Lisp command is a `agent.newagent` command. For example:

```
java -jar casa.jar -LAC (agent.newagent \"casa.TransientAgent\" \"Fred\" 8760)
```

starts up both a `LAC` agent and a `TransientAgent` agent named *Fred* on port 8760.

Parameter	Default	Type	Notes
TYPE			The type (Java class) of agent to be created
NAME			Name of the agent
:PORT	0	java.lang.Integer	Port of the agent; 0: will choose; -ve: indicates to 'hunt' (more than 0 proxies will affect the actual port number)
:PROCESS	CURRENT		Specifies the process in which to run the agent. The value can be: • LAC (the LAC's process); • CURRENT, THIS, PROCESS (the process of the command line); • INDEPENDENT, NEW (A newly generated process); • a port number of an agent in another process; or • a URL of an agent in another process.
:LACPORT	9000	java.lang.Integer	Specifies the port of the LAC the agent registers with (-1 indicates not to register)
:SHORT-CUTTING			Set shortcutting in message protocols on or off
:ACK	NIL	java.lang.Boolean	Turn acknowledge (ack) requirement on or off
:MARKUP	KQML		Specify the markuplanguage for inter-agent messages (effects the entire process globally). KQML or XML
:PERSISTENT	T	java.lang.Boolean	Turn persistent saving of agent data on or off. This only works for agent inheriting from `Agent`.
:ROOT	casa		Root directory for the casa files (LAC only)
:DEBUG		java.lang.Boolean	Turn CASA debugging on/off for the agent's entire process
:TRACE	0	java.lang.Integer	Turn set the trace flags. Bits are (1=off, 2=on, 4=monitor, 8=toFile)
:TRACETAGS	error		A list of trace tags(identifiers) to add—remove. Remove a tag by preceding it with a'-'. Current valid tags: calls, msg, msgHandling, warning, info, policies, commitments.
:INTERFACE			A fully-qualified java class name of the interface for the agent to use. Defaults to an appropriate window interface. The special [name] 'text' yields a default text interface; 'none' specifies the agent should have no interface.
:GUARD		java.lang.Boolean	Turn the guard (secure) on or off for the agent
:PROXIES		java.lang.String	Add proxies to the agent (a semi-colon separated list of fully-qualified class names)
:STRATEGY	sc3		Choose a strategy. sc, reactive, BDI, or sc3.
:SECURITY	none		Choose security package. Currently, 'casa.security' or 'none'
:ONTOLOGY-ENGINE	casa.ontology.v3.-CASAOntology	java.lang.String	The Java class for the ontology engine (currently, either casa.ontology.CASAOntology or casa.util.TypeHierarchy)
:ONTOLOGY-FILE	ontology.lisp		The file from which the ontology engine should read initialization data
:NOWAIT			Don't wait for the agent to start before returning (it doesn't matter what the value is, if :nowait is present it doesn't wait)
:PERSISTENT	NIL	java.lang.Boolean	if this is NIL then this agent will not be persistent, otherwise it will be (applies only the subtypes of the `Agent` class.

Table 6: Agent command line parameters. (Hyphens in parameter names are line wraps – there are not hyphens or spaces in parameter names.)

F Lisp Commands

F.1 ?

? is a synonym for HELP (Section F.83).

F.2 A2L

A2L is a synonym for ACT2STRING (Section F.7).

F.3 ACT

ACT is an external symbol in the COMMON-LISP-USER package. It is an undefined variable; its value is "act". Its function binding is #<SPECIAL-OPERATOR ACT>.

Lambda list

(&REST ACTIONS)

Function documentation

Return a new Act object. Defined in class class casa.Act.

Parameter	Default	Type	Notes
ACTIONS	def="&REST"		one or more action types as strings.

F.4 ACT.ACTION-AT

ACT.ACTION-AT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ACT.ACTION-AT>. Its synonyms are *A2L*.

Lambda list

(ACT INDEX)

Function documentation

Return the ACTION at index INDEX, where the first is index 0. Defined in class class casa.Act.

Parameter	Default	Type	Notes
ACT			The Act object or String to index into.
INDEX		java.lang.Integer	The index, beginning with 0

F.5 ACT.SIZE

ACT.SIZE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ACT.SIZE>. Its synonyms are *A2L*.

Lambda list

(ACT)

Function documentation

Return the number of ACTIONS in the ACT. Defined in class class casa.Act.

Parameter	Default	Type	Notes
ACT			The Act object or String reprsentation of the Act object.

F.6 ACT2LIST

ACT2LIST is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ACT2LIST>. Its synonyms are *A2L*.

Lambda list

(ACT)

Function documentation

Convert an Act JavaObject to a Cons list. Defined in class class casa.Act.

Parameter	Default	Type	Notes
ACT		casa.Act	The Act object to return as a Cons list.

F.7 ACT2STRING

ACT2STRING is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ACT2STRING>. Its synonyms are *A2L*.

Lambda list

(ACT)

Function documentation

Convert an Act JavaObject to its string representation. Defined in class class casa.Act.

Parameter	Default	Type	Notes
ACT		casa.Act	The Act object to return as a Cons list.

F.8 ADD-SINGLE-NUM-VALUE-KBFILTER

ADD-SINGLE-NUM-VALUE-KBFILTER is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ADD-SINGLE-NUM-VALUE-KBFILTER>.

Lambda list

(PREDICATE)

Function documentation

Adds a single-value int filter to the KB. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
PREDICATE		java.lang.String	The name of the predicate to filter.

F.9 AGENT-IDENTIFIER

AGENT-IDENTIFIER is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT-IDENTIFIER>.

Lambda list

(&KEY NAME TYPE PORT HOST USER DATA FRAGMENT)

Function documentation

return a URL based on the FIPA-like agent-identifier expression. Defined in class class casa.agentCom.URLDescriptor.

Parameter	Default	Type	Notes
:NAME		java.lang.String	The name (file) of the agent (NOT including the path or type).
:TYPE		java.lang.String	The type (path) of the agent.
:PORT		java.lang.Integer	The port.
:HOST		java.lang.String	The host.
:USER		java.lang.String	The user.
:DATA		java.lang.String	The data.
:FRAGMENT		java.lang.String	The fragment.

F.10 AGENT.ASYNC

AGENT.ASYNC is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.ASYNC>.

Lambda list

(COMMAND)

Function documentation

Executes command in a separate thread. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
COMMAND			The command to execute as either a String or a quoted Cons.

F.11 AGENT.CREATE-EVENT-OBSERVER-EVENT

AGENT.CREATE-EVENT-OBSERVER-EVENT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.CREATE-EVENT-OBSERVER-EVENT>.

Lambda list

(EVENT-TYPE WATCHED-EVENT-TYPES &KEY CONVERSATION-ID)

Function documentation

Creates an EventObserverEvent to manage subscriptions. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
EVENT-TYPE		java.lang.String	The type of this event.
WATCHED-EVENT--TYPES		org.armedbear.-lisp.LispObject	The types of events to watch.
:CONVERSATION-ID		java.lang.String	An (optional) conversation ID if this event is to be associated with a particular conversation.

F.12 AGENT.DELETE-POLICY

AGENT.DELETE-POLICY is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.DELETE-POLICY>.

Lambda list

(NAME)

Function documentation

Returns the agent's policies as a string. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
NAME		java.lang.String	The name of the policy to be deleted.

F.13 AGENT.EXIT

AGENT.EXIT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.EXIT>.

Lambda list

NIL

Function documentation

Request this agent to exit. Note that this is only a request: the agent may refuse to exit or delay exiting. For example, it may finish up pending requests before exiting.. Defined in class class casa.TransientAgent.

F.14 AGENT.FIND-FILE-RESOURCE-PATH

AGENT.FIND-FILE-RESOURCE-PATH is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.FIND-FILE-RESOURCE-PATH>.

Lambda list

(&OPTIONAL FILENAMEONLY)

Function documentation

Searchs for and returns a filename in system-specific directories. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
FILENAMEONLY	optional		string name file to find

F.15 AGENT.GET-AGENT

AGENT.GET-AGENT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.GET-AGENT>.

Lambda list

NIL

Function documentation

Returns the agent as a java object. Defined in class class casa.TransientAgent.

F.16 AGENT.GET-AGENTS-REGISTERED

AGENT.GET-AGENTS-REGISTERED is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.GET-AGENTS-REGISTERED>.

Lambda list

(&OPTIONAL (PROPERTY all))

Function documentation

Sends a message to the LAC requesting the registered agents. The output will be printed on agents log (see .release_getAgentsRunning()). Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
PROPERTY	optional def="all"		string name of the system property to show (or 'all' to display and return NIL)

F.17 AGENT.GET-AGENTS-RUNNING

AGENT.GET-AGENTS-RUNNING is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.GET-AGENTS-RUNNING>.

Lambda list

(&OPTIONAL (PROPERTY all))

Function documentation

Sends a message to the LAC requesting the running agents. The output will be printed on agents log (see .release_getAgentsRunning()). Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
PROPERTY	optional def="all"		string name of the system property to show (or 'all' to display and return NIL)

F.18 AGENT.GET-CLASS-NAME

AGENT.GET-CLASS-NAME is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.GET-CLASS-NAME>.

Lambda list

NIL

Function documentation

Returns the agent class name as a string. Defined in class class casa.TransientAgent.

F.19 AGENT.GET-POLICIES

AGENT.GET-POLICIES is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.GET-POLICIES>.

Lambda list

(&KEY ID (VERBOSE T))

Function documentation

Returns the agent's policies as a string. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
:ID		java.lang.Integer	retrieve only the policy with this ID.
:VERBOSE	def=T	java.lang.Boolean	set to Nil to get only the names of the policies.

F.20 AGENT.GET-URL

AGENT.GET-URL is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.GET-URL>.

Lambda list

(&KEY OBJECT)

Function documentation

Returns the agent's URL as a string. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
:OBJECT		java.lang.Boolean	Indicates that the URL should be returned as a URLDescriptor object. (optional)

F.21 AGENT.GETUSEACKPROTOCOL

AGENT.GETUSEACKPROTOCOL is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.GETUSEACKPROTOCOL>.

Lambda list

NIL

Function documentation

Return T if the agent is using the ACK parotocol, otherwise return Nil. Defined in class class casa.TransientAgent.

F.22 AGENT.INSTANTIATE-CONVERSATION

AGENT.INSTANTIATE-CONVERSATION is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.INSTANTIATE-CONVERSATION>.

Lambda list

(NAME EVENT)

Function documentation

Instantiates a conversation from a template conversation. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
NAME		java.lang.String	The name of the conversation template.
EVENT		casa.event.-MessageEvent	The initial message event from which to instantiate the conversation.

F.23 AGENT.ISA

AGENT.ISA is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.ISA>.

Lambda list

(CHILD ANCESTOR)

Function documentation

Determines if the child act is a descendent of the given ancestor. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
CHILD		java.lang.String	The child act.
ANCESTOR		java.lang.String	The ancestor act.

F.24 AGENT.JOIN

AGENT.JOIN is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.JOIN>.

Lambda list

(CD-URL)

Function documentation

Attempt to join the cooperation domain specified by the parameter URL. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
CD-URL			The URL of the cooperation domain to join.

F.25 AGENT.LOAD-FILE-RESOURCE

AGENT.LOAD-FILE-RESOURCE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.LOAD-FILE-RESOURCE>. Its synonyms are *LOAD-FILE-RESOURCE*.

Lambda list

(FILE)

Function documentation

Locates and loads a file, returning the path it was loaded from or NIL if it failed. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
FILE		java.lang.String	The file name (not the path) to load.

F.26 AGENT.MAKE-CONVERSATION-ID

AGENT.MAKE-CONVERSATION-ID is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.MAKE-CONVERSATION-ID>.

Lambda list

NIL

Function documentation

Returns a new, unique conversation ID string. Defined in class class casa.TransientAgent.

F.27 AGENT.MESSAGE

AGENT.MESSAGE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.MESSAGE>.

Lambda list

(PERFORMATIVE ACT RECEIVER &KEY CONTENT LANGUAGE LANGUAGE-VERSION REPLY-BY &ALLOW-OTHER-KEYS)

Function documentation

Create a new message object. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
PERFORMATIVE			Any performative subtype of 'inform' or 'request'
ACT			An act name
RECEIVER			The URL of the destination agent
:CONTENT			The content part of the message (optional)
:LANGUAGE			The language in which the content field is written (optional)
:LANGUAGE-VERSION			The version of the language in which the content field is written (optional)
:REPLY-BY		java.lang.Integer	Milliseconds to wait before giving up on a reply (optional) Default=options.timeout

F.28 AGENT.NEW-AGENT

AGENT.NEW-AGENT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.NEW-AGENT>.

Lambda list

(TYPE NAME &OPTIONAL (PORT 0) &KEY (PROCESS CURRENT) (LACPORT 9000) SHORTCUTTING (ACK NIL) (PERSISTENT T) (ROOT /casa/) (TRACE 0) (TRACETAGS error) INTERFACE GUARD PROXIES (STRATEGY sc3) (SECURITY none) (ONTOLOGYENGINE casa.ontology.owl2.OWLOntology) ONTOLOGYFILE ONTOLOGYFILTER NOWAIT FIPA-URLS &ALLOW-OTHER-KEYS)

Function documentation

Command that creates an agent. Defined in class class casa.TransientAgent$NewAgentLispCommand.

Parameter	Default	Type	Notes
TYPE			The type (Java class) of agent to be created
NAME			Name of the agent
PORT	optional def=org.armedbear.lisp.Fixnum@0	java.lang.Integer	Port of the agent; 0: will choose; -ve: indicates to 'hunt' (more than 0 proxies will affect the actual port number)
:PROCESS	def="CURRENT"		Specifies the process in which to run the agent. LAC, CURRENT, or INDEPENDENT.
:LACPORT	def=org.armedbear.lisp.Fixnum@2328	java.lang.Integer	Specifies the port of the LAC the agent registers with (-1 indicates not to register)
:SHORTCUTTING			Set shortcutting in message protocols on or off
:ACK	def=NIL	java.lang.Boolean	Turn acknowledge (ack) requirement on or off
:PERSISTENT	def=T	java.lang.Boolean	Turn persistent saving of agent data on or off (default to on)
:ROOT	def="/casa/"		Root directory for the casa files (LAC only)
:TRACE	def=org.armedbear.lisp.Fixnum@0	java.lang.Integer	Turn set the trace flags. Bits are (1=off,2=on,4=monitor,8=toFile)
:TRACETAGS	def="error"		A list of trace tags(identifiers) to add—remove. Remove a tag by preceding it with a'-'. Current valid tags:calls,msg,msgHandling,warning,info,policies,commitments.-
:INTERFACE			A fully-qualified java class name of the interface for the agent to use. Defaults to an appropriate window interface. The special [name] 'text' yeilds a default text interface; 'none' specifies the agent should have no interface.
:GUARD		java.lang.Boolean	Turn the guard (secure) on or off for the agent
:PROXIES		java.lang.String	Add proxies to the agent (a semi-colon separated list of fully-qualified class names)
:STRATEGY	def="sc3"		Choose a strategy. sc, reactive, BDI, or sc3.
:SECURITY	def="none"		Choose security package. Currently, 'casa.security' or 'none'
:ONTOLOGYENGINE	def="casa.ontology.owl2.OWLOntology"		The Java class for the ontology engine (currently, either casa.ontology.v3.CASAOntology or casa.util.TypeHierarchy or casa.ontology.owl2.OWLOntology)
:ONTOLOGYFILE			The file from which the ontology engine should read inialization data
:ONTOLOGYFILTER		java.lang.String	The ontology filter
:NOWAIT			Don't wait for the agent to start before returning (it doesn't matter what the value is, if :nowait is present it doesn't wait)
:FIPA-URLS		java.lang.Boolean	true: (agent-idenifier :name "fred" ...); false: casa://10.0.1.12/casa/TransientAgent/fred:5400.

F.29 AGENT.NEW-INTERFACE

AGENT.NEW-INTERFACE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.NEW-INTERFACE>.

Lambda list

(CLASSNAME)

Function documentation

Adds a new interface to this agent. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
CLASSNAME		java.lang.String	The fully-qualified java class name of the new interface.

F.30 AGENT.OPTIONS

AGENT.OPTIONS is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.OPTIONS>.

Lambda list

(&KEY &ALLOW-OTHER-KEYS)

Function documentation

Show or set options. No keys lists all options. A key with no value returns it's value. A key with a value sets the value. Defined in class class casa.TransientAgent.

F.31 AGENT.PAUSE

AGENT.PAUSE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.PAUSE>.

Lambda list

NIL

Function documentation

Pause the agent just after processing a message. Defined in class class casa.TransientAgent.

F.32 AGENT.PING

AGENT.PING is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.PING>. Its synonyms are *PING*.

Lambda list

(URL &KEY (TIMEOUT 2000))

Function documentation

Pings another agent and returns it's URL or NIL if the ping failed. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
URL		java.lang.String	The URL of the agent to ping.
:TIMEOUT	def=org.-armedbear.lisp.-Fixnum@7d0	java.lang.Integer	The time in milliseconds to wait.

F.33 AGENT.PRINTLN

AGENT.PRINTLN is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.PRINTLN>.

Lambda list

(TAG &REST MESSAGE)

Function documentation

Log the message to the agent's reporting mechanism. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
TAG			The tag for conditional logging. 'error' or NIL will always print.
MESSAGE	def="&REST"		Objects to be converted to type String, concatonated, and used as the log message.

F.34 AGENT.PUT-POLICY

AGENT.PUT-POLICY is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.PUT-POLICY>.

Lambda list

(POLICY &KEY ALWAYS-APPLY LAST-RESORT)

Function documentation

Insert a policy into the agent's global policy repetoire. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
POLICY		casa.policy.Policy	The policy to be added.
:ALWAYS-APPLY		java.lang.Boolean	Apply this policy to every conversation, if set.
:LAST-RESORT		java.lang.Boolean	Apply this policy only if no other non-last-resort policies are applicable, if set.

F.35 AGENT.REPLY

AGENT.REPLY is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.REPLY>.

Lambda list

(TO-MESSAGE &OPTIONAL PERFORM-DESCRIPTOR &ALLOW-OTHER-KEYS)

Function documentation

Sends a message to another agent constructed from the argument message, with fields being replaced from the PerformDescriptor and any other keys/value pairs given. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
TO-MESSAGE		casa.MLMessage	The message to which the reply is being made
PERFORM--DESCRIPTOR	optional		Normally either the result of a call to the agent's 'consider' method or a legal reply performative string or NIL (taken as 'agree'.

F.36 AGENT.RESET-DEF-FILE-SYSTEM-LOCATIONS

AGENT.RESET-DEF-FILE-SYSTEM-LOCATIONS is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.RESET-DEF-FILE-SYSTEM-LOCATIONS>.

Lambda list

NIL

Function documentation

resets the default file sysetm locations from system properties casa.home and user.home. Defined in class class casa.TransientAgent.

F.37 AGENT.RESUME

AGENT.RESUME is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.RESUME>.

Lambda list

NIL

Function documentation

Resume the agent from a pause state. Defined in class class casa.TransientAgent.

F.38 AGENT.SEND

AGENT.SEND is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.SEND>.

Lambda list

(MESSAGE &KEY PROXY)

Function documentation

Sends a message to another agent. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
MESSAGE		casa.MLMessage	The message to send
:PROXY			Send the message through the indicated proxy (optional)

F.39 AGENT.SEND-QUERY-AND-WAIT

AGENT.SEND-QUERY-AND-WAIT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.SEND-QUERY-AND-WAIT>. Its synonyms are *QUERYW*.

Lambda list

(MESSAGE &KEY (TIMEOUT 2000))

Function documentation

Sends a request-type message and waits for the response. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
MESSAGE		casa.MLMessage	The message to send.
:TIMEOUT	def=org.-armedbear.lisp.-Fixnum@7d0	java.lang.Integer	The time in milliseconds to wait.

F.40 AGENT.SEND-REQUEST-AND-WAIT

AGENT.SEND-REQUEST-AND-WAIT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.SEND-REQUEST-AND-WAIT>. Its synonyms are *REQUESTW*.

Lambda list

(MESSAGE &KEY (TIMEOUT 2000))

Function documentation

Sends a request-type message and waits for the response. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
MESSAGE		casa.MLMessage	The message to send.
:TIMEOUT	def=org.-armedbear.lisp.-Fixnum@7d0	java.lang.Integer	The time in milliseconds to wait.

F.41 AGENT.SHOW-COMMITMENTS

AGENT.SHOW-COMMITMENTS is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.SHOW-COMMITMENTS>. Its synonyms are *SCOM*.

Lambda list

(&KEY CURRENT VIOLATED)

Function documentation

Show the social commitments. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
:CURRENT			Show only the commitments that are not filfilled, violated, etc.
:VIOLATED			Show only the commitments that are violated.

F.42 AGENT.SHOW-CONVERSATIONS

AGENT.SHOW-CONVERSATIONS is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.SHOW-CONVERSATIONS>. Its synonyms are *SCONV*.

Lambda list

(&OPTIONAL NAME &KEY CURRENT (VERBOSE 1))

Function documentation

Show the conversations – supported, current, all, or a specific one. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
NAME	optional	java.lang.String	If present shows the named template, otherwise all are shown.
:CURRENT			Show the current conversations for the current agent; ignores the NAME parameter.
:VERBOSE	def=org.-armedbear.lisp.-Fixnum@1	java.lang.Integer	0=only names; 1=state with no policies; 2=all

F.43 AGENT.SHOW-EVENT-QUEUE

AGENT.SHOW-EVENT-QUEUE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.SHOW-EVENT-QUEUE>. Its synonyms are *SEQ*.

Lambda list

NIL

Function documentation

Show the events on the event queue, returning a count. Defined in class class casa.AbstractProcess.

F.44 AGENT.SHOW-EVENTQUEUE

AGENT.SHOW-EVENTQUEUE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.SHOW-EVENTQUEUE>.

Lambda list

NIL

Function documentation

List the current event queue. Defined in class class casa.TransientAgent.

F.45 AGENT.STEP

AGENT.STEP is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.STEP>.

Lambda list

NIL

Function documentation

Resume the agent from a pause state until just after processing the next message. Defined in class class casa.TransientAgent.

F.46 AGENT.STOP-EVENT-OBSERVER-EVENT

AGENT.STOP-EVENT-OBSERVER-EVENT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.STOP-EVENT-OBSERVER-EVENT>.

Lambda list

(CONVERSATION-ID)

Function documentation

Stop the event with the matching owner conversation ID. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
CONVERSATION-ID		java.lang.String	The event's owner conversation ID.

F.47 AGENT.TELL

AGENT.TELL is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.TELL>.

Lambda list

(AGENT COMMAND)

Function documentation

Sends a REQUEST/EXECUTE message with the specified command. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
AGENT		java.lang.String	The URL of the agent to execute the command.
COMMAND		java.lang.String	The command to execute.

F.48 AGENT.TRANSFORM

AGENT.TRANSFORM is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.TRANSFORM>.

Lambda list

(DESCRIBABLE)

Function documentation

Returns the transformed object, or the argument argument if no transformation is applicable. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
DESCRIBABLE		casa.interfaces.-Describable	The object to transform.

F.49 AGENT.TRANSFORM-STRING

AGENT.TRANSFORM-STRING is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR AGENT.TRANSFORM-STRING>.

Lambda list

(STRING)

Function documentation

Returns the transformed string, or the argument string if no transformation is applicable. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
STRING		java.lang.String	The string to transform in id—id—... form.

F.50 CALL-GC

CALL-GC is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR CALL-GC>.

Lambda list

NIL

Function documentation

Requests the Garbage Collector to clean up. Defined in class class casa.TransientAgent.

F.51 COMPARETO

COMPARETO is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR COMPARETO>.

Lambda list

(p1 p2)

Function documentation

Calls the java compareto() operator on the 1st parameter with the second parameter as an argument. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
P1			P1
P2			P2

F.52 CONSEQUENT-CLASS

CONSEQUENT-CLASS is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR CONSEQUENT-CLASS>.

Lambda list

(CLASS-NAME)

Function documentation

Return a new object that is a subclass of Concequent. Defined in class class casa.policy.Policy.

Parameter	Default	Type	Notes
CLASS-NAME			The fully-qualified name if a subclass of Rule.

F.53 CONSTRAINT

CONSTRAINT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR CONSTRAINT>.

Lambda list

(&OPTIONAL TYPE-NAME &KEY INDIVIDUAL-ONLY TYPE-ONLY EXP ONTOLOGY The ontology in which this constraint resides.)

Function documentation

set a type-to-type relationship in the specified relation. Defined in class class casa.ontology.v3.ConstraintSimple.

Parameter	Default	Type	Notes
TYPE-NAME	optional	java.lang.String	The type to constrain to.
:INDIVIDUAL-ONLY		java.lang.Boolean	Constraint to being an individual only and not a type.
:TYPE-ONLY		java.lang.Boolean	Constrain to being a type only and not an individual.
:EXP		org.armedbear.-lisp.Cons	The expression that must evaluate to true for the validation to pass. The expression is evaluation of an environment with variables: ?type (the type under consideration), ?0 (the domain), ?1 (the range), etc.
:ONTOLOGY			:ONTOLOGY

F.54 CONVERSATION

CONVERSATION is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR CONVERSATION>.

Lambda list

(NAME BINDINGS &KEY BIND-VAR BIND-VAR-TO BIND-STATE CLASS)

Function documentation

Declares a conversation. Defined in class class casa.conversation2.Conversation.

Parameter	Default	Type	Notes
NAME		java.lang.String	The name of the conversation.
BINDINGS		org.armedbear.-lisp.Cons	A Cons list of Lisp functions describing sub-conversations or policies.
:BIND-VAR		org.armedbear.-lisp.Cons	A Cons list of pairs of symbol/values pairs (themselves Cons lists) that will be bound in the context of the conversation. The expressions are evaluated at the time the conversation is created.
:BIND-VAR-TO		org.armedbear.-lisp.Cons	A Cons list of triples of symbol/childConversation/-childSymbol (themselves Cons lists) that will be bound in the context of the conversation.
:BIND-STATE		org.armedbear.-lisp.Cons	A Cons list of triples of state/childConversation/childState (themselves Cons lists) that will be bound in the context of the conversation.
:CLASS		java.lang.String	The specific subclass of a Conversation.

F.55 CONVERSATION.EXECUTE-ACTION

CONVERSATION.EXECUTE-ACTION is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR CONVERSATION.EXECUTE-ACTION>.

Lambda list

(EVENT &KEY SUCCESS FAILURE)

Function documentation

Executes ONE of the current conversation's actions. Defined in class class casa.conversation2.Conversation.

Parameter	Default	Type	Notes
EVENT		casa.event.Event	The event in scope,
:SUCCESS		java.lang.Boolean	The success action.
:FAILURE		java.lang.Boolean	The failure action.

F.56 CONVERSATION.GET-STATE

CONVERSATION.GET-STATE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR CONVERSATION.GET-STATE>.

Lambda list

NIL

Function documentation

Gets the state of the current conversation. Defined in class class casa.conversation2.Conversation.

F.57 CONVERSATION.SET-ACTION

CONVERSATION.SET-ACTION is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR CONVERSATION.SET-ACTION>.

Lambda list

(&KEY SUCCESS FAILURE)

Function documentation

Sets one or more actions of the current conversation. Defined in class class casa.conversation2.Conversation.

Parameter	Default	Type	Notes
:SUCCESS		java.lang.String	The success action.
:FAILURE		java.lang.String	The failure action.

F.58 CONVERSATION.SET-STATE

CONVERSATION.SET-STATE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR CONVERSATION.SET-STATE>.

Lambda list

(STATE)

Function documentation

Sets the state of the current conversation. Defined in class class casa.conversation2.Conversation.

Parameter	Default	Type	Notes
STATE		java.lang.String	The new state.

F.59 DECLINDIVIDUAL

DECLINDIVIDUAL is a synonym for ONT.IS-TYPE (Section F.111).

F.60 DECLMAPLET

DECLMAPLET is a synonym for ONT.ASSERT (Section F.103).

F.61 DECLONTOLOGY

DECLONTOLOGY is a synonym for ONTOLOGY (Section F.116).

F.62 DECLRELATION

DECLRELATION is a synonym for ONT.RELATION (Section F.113).

F.63 DECLTYPE

DECLTYPE is a synonym for ONT.TYPE (Section F.115).

F.64 ECHO

ECHO is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ECHO>.

Lambda list

(first &OPTIONAL (second 2 second_default) third &REST rest &KEY NAMED BOOL &ALLOW-OTHER-KEYS)

Function documentation

Merely echos the parameters for casa operators. Defined in class class casa.abcl.CasaLispOperator.

Parameter	Default	Type	Notes
FIRST			first param
SECOND	optional def=org.armedbear.lisp.Fixnum@2		second optional param with default
THIRD	optional		third optional param
REST	def="&REST"		the rest of the line, including keys
:NAMED			named doc
:BOOL		java.lang.Boolean	bool doc

F.65 EVENT-DESCRIPTOR

EVENT-DESCRIPTOR is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR EVENT-DESCRIPTOR>.

Lambda list

(EVENT-TYPE &ALLOW-OTHER-KEYS)

Function documentation

Return a new EventDescriptor that matches fields according the the keys and values. Defined in class class casa.event.EventDescriptor$EventDescriptorLispOperator.

Parameter	Default	Type	Notes
EVENT-TYPE		java.lang.String	The most specific event to match

F.66 EVENT.GET

EVENT.GET is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR EVENT.GET>.

Lambda list

NIL

Function documentation

Return the Event object. Defined in class class casa.TransientAgent.

F.67 EVENT.GET-MSG

EVENT.GET-MSG is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR EVENT.GET-MSG>.

Lambda list

(&OPTIONAL TAG &KEY UNSERIALIZE)

Function documentation

Return the value of the specified message tag, Nil if the tag is missing and throws an exception if the MSG isn't in scope. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
TAG	optional	java.lang.String	The tag for which to return the value.
:UNSERIALIZE			If included, an attempt to unserialize the value will be made.

F.68 EVENT.GET-MSG-OBJ

EVENT.GET-MSG-OBJ is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR EVENT.GET-MSG-OBJ>.

Lambda list

NIL

Function documentation

Return the value of the specified message tag, Nil if the tag is missing and throws an exception if the MSG isn't in scope. Defined in class class casa.TransientAgent.

F.69 EVENT.GET-OWNER-CONVERSATION-ID

EVENT.GET-OWNER-CONVERSATION-ID is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR EVENT.GET-OWNER-CONVERSATION-ID>.

Lambda list

NIL

Function documentation

Return the Event's owner conversation ID if it has one. Defined in class class casa.TransientAgent.

F.70 FIPA-FOLLOWS

FIPA-FOLLOWS is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR FIPA-FOLLOWS>.

Lambda list

(DOMAIN &OPTIONAL RANGE &KEY ONTOLOGY)

Function documentation

Test weather items are related in the FIPA-follows relation. Defined in class class casa.ontology.owl2.OWLOntology.

Parameter	Default	Type	Notes
DOMAIN		java.lang.String	
RANGE	optional	java.lang.String	
:ONTOLOGY		java.lang.String	The ontology use (default is the agent current ontology)

F.71 FIPA-PRECEEDS

FIPA-PRECEEDS is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR FIPA-PRECEEDS>.

Lambda list

(DOMAIN &OPTIONAL RANGE &KEY ONTOLOGY)

Function documentation

Test weather items are related in the FIPA-preceeds relation. Defined in class class casa.ontology.owl2.OWLOntology.

Parameter	Default	Type	Notes
DOMAIN		java.lang.String	
RANGE	optional	java.lang.String	
:ONTOLOGY		java.lang.String	The ontology use (default is the agent current ontology)

F.72 FIRE-EVENT

FIRE-EVENT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR FIRE-EVENT>.

Lambda list

NIL

Function documentation

Fire the current event. Defined in class interface casa.event.Event.

F.73 GET-HOST-NAMES

GET-HOST-NAMES is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR GET-HOST-NAMES>.

Lambda list

NIL

Function documentation

Show the host names for this computer. Defined in class class casa.AbstractProcess.

F.74 GET-INETADDRESSES

GET-INETADDRESSES is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR GET-INETADDRESSES>.

Lambda list

NIL

Function documentation

Show all the InetAddresses for this computer. Defined in class class casa.AbstractProcess.

F.75 GET-OBJECT

GET-OBJECT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR GET-OBJECT>.

Lambda list

(NAME)

Function documentation

(DEPRECATED) Return the value of the specified message tag, Nil if the tag is missing and throws an exception if the MSG isn't in scope. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
NAME		java.lang.String	The key the object to return.

F.76 GET-ONTOLOGY

GET-ONTOLOGY is a synonym for ONT.GET (Section F.105).

F.77 GET-RESIDENT-ONTOLOGIES

GET-RESIDENT-ONTOLOGIES is a synonym for ONT.GET-RESIDENT (Section F.106).

F.78 GET-SYSTEM

GET-SYSTEM is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR GET-SYSTEM>.

Lambda list

(&OPTIONAL (PROPERTY all))

Function documentation

Either return the string value of a system property, or display all system properties (returning NIL). Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
PROPERTY	optional def="all"		string name of the system property to show (or 'all' to display and return NIL)

F.79 GET-THREAD

GET-THREAD is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR GET-THREAD>.

Lambda list

NIL

Function documentation

Returns the current tread name as a string. Defined in class class casa.TransientAgent.

F.80 GETONTOLOGY

GETONTOLOGY is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR GETONTOLOGY>.

Lambda list

NIL

Function documentation

Return the agent's current default ontology. Defined in class class java.lang.reflect.Method.

F.81 GETONTOLOGYENGINE

GETONTOLOGYENGINE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR GETONTOLOGYENGINE>.

Lambda list

NIL

Function documentation

Function's help is undefined. Defined in class class java.lang.reflect.Method.

F.82 HASCONVERSATION

HASCONVERSATION is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR HASCONVERSATION>.

Lambda list

(convID)

Function documentation

Returns T iff the agent has a conversation with a specific conversationID; returns NIL otherwise. Defined in class class java.lang.reflect.Method.

Parameter	Default	Type	Notes
CONVID		java.lang.String	The conversation ID to look up.

F.83 HELP

HELP is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR HELP>. Its synonyms are *?*.

Lambda list

(&OPTIONAL APROPOSE &KEY (LATEX NIL))

Function documentation

Displays list of casa functions. Defined in class class casa.abcl.CasaLispOperator.

Parameter	Default	Type	Notes
APROPOSE	optional	java.lang.String	A string to match to select a subset of casa functions.
:LATEX	def=NIL	java.lang.Boolean	Print the output in LaTex format (not a full document, just a set of subsection's).

F.84 instance-of

instance-of is a synonym for PRIMITIVEONTOLOGY.INSTANCE-OF (Section F.124).

F.85 isa

isa is a synonym for PRIMITIVEONTOLOGY.ISA (Section F.125).

F.86 isa-ancestor

isa-ancestor is a synonym for PRIMITIVEONTOLOGY.ISA-ANCESTOR (Section F.126).

F.87 isa-child

isa-child is a synonym for PRIMITIVEONTOLOGY.ISA-CHILD (Section F.127).

F.88 isa-descendant

isa-descendant is a synonym for PRIMITIVEONTOLOGY.ISA-DESCENDANT (Section F.128).

F.89 isa-parent

isa-parent is a synonym for PRIMITIVEONTOLOGY.ISA-PARENT (Section F.129).

F.90 isequal

isequal is a synonym for PRIMITIVEONTOLOGY.ISEQUAL (Section F.130).

F.91 ISPARENT

ISPARENT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ISPARENT>.

Lambda list

(DOMAIN &OPTIONAL RANGE &KEY ONTOLOGY)

Function documentation

Test weather items are related in the isparent relation. Defined in class class casa.ontology.owl2.OWLOntology.

Parameter	Default	Type	Notes
DOMAIN		java.lang.String	
RANGE	optional	java.lang.String	
:ONTOLOGY		java.lang.String	The ontology use (default is the agent current ontology)

F.92 KB.ARG-DESC

KB.ARG-DESC is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR KB.ARG-DESC>.

Lambda list

(VARIABLE &KEY SUBSUMPTION)

Function documentation

Description of arguments. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
VARIABLE		java.lang.String	PATTERN of the Argument
:SUBSUMPTION			:SUBSUMPTION

F.93 KB.ASSERT

KB.ASSERT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR KB.ASSERT>.

Lambda list

(FORMULA)

Function documentation

Asserts the formula into the KB. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
FORMULA		java.lang.String	The formula.

F.94 KB.DEFINE-ONT-FILTER

KB.DEFINE-ONT-FILTER is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR KB.DEFINE-ONT-FILTER>.

Lambda list

(PATTERN &REST ARGUMENTS)

Function documentation

Defining the ontology filter. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
PATTERN		java.lang.String	PATTERN of the FILTER
ARGUMENTS	def=”&REST”		Set to non-NIL to have PATTERN interpreted as a regular expression occuring in the expression displayed.

F.95 KB.GET-VALUE

KB.GET-VALUE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR KB.GET-VALUE>.

Lambda list

(ATTRIBUTE)

Function documentation

Return the (unique) value of an attribute in an (attribute value) pair. Defined in class class java.lang.reflect.Method.

Parameter	Default	Type	Notes
ATTRIBUTE		java.lang.String	An attribute name in an (attribute value) pair.

F.96 KB.QUERY-IF

KB.QUERY-IF is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR KB.QUERY-IF>.

Lambda list

(FORMULA &KEY REPLY-EXP BOOLEAN TO)

Function documentation

Queries the formula from the KB returning a Nil (if the formula doesn’t exist), T (if it does), or a list of string (if it does and it contains meta variables). Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
FORMULA		java.lang.String	The formula.
:REPLY-EXP			Return a string containing an expression appropriate for a reply instead of the default.
:BOOLEAN			Return either T or NIL instead of the default.
:TO		java.lang.String	Send the request to another agent and return a boolean.

F.97 KB.QUERY-REF

KB.QUERY-REF is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR KB.QUERY-REF>.

Lambda list

(FORMULA)

Function documentation

Return the answer to to JSL-like query-ref. Defined in class class java.lang.reflect.Method.

Parameter	Default	Type	Notes
FORMULA		java.lang.String	A JSL-like identifying expression: iota, any, all, some.

F.98 KB.SHOW

KB.SHOW is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR KB.SHOW>.

Lambda list

(&OPTIONAL PATTERN &KEY REGEX STRICT FACTS QUERIES ASSERTS)

Function documentation

Dispalys the formulas from the KB. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
PATTERN	optional	java.lang.String	If specified, only expressions containing PATTERN will be displayed.
:REGEX			Set to non-NIL to have PATTERN interpreted as a regular expression occuring in the expression displayed.
:STRICT			Set to non-NIL to have PATTERN interpreted as a regular expression that must match the entire expression to be displayed.
:FACTS			Set to non-NIL to return only a list of facts in the KB.
:QUERIES			Set to non-NIL to return only a list of query filters on the KB.
:ASSERTS			Set to non-NIL to return only a list of assert filters on the KB.

F.99 LOAD-FILE-RESOURCE

LOAD-FILE-RESOURCE is a synonym for AGENT.LOAD-FILE-RESOURCE (Section F.25).

F.100 LOAD-JAR

LOAD-JAR is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR LOAD-JAR>.

Lambda list

(JARNAME)

Function documentation

Loads a jar into the classpath. Defined in class class casa.util.JarLoader.

Parameter	Default	Type	Notes
JARNAME		java.lang.String	The fully-qualified pathname of jar file (including the .jar extension).

F.101 MSGEVENT-DESCRIPTOR

MSGEVENT-DESCRIPTOR is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR MSGEVENT-DESCRIPTOR>.

Lambda list

(&OPTIONAL EVENT-TYPE &ALLOW-OTHER-KEYS)

Function documentation

Return a new MessageEventDescriptor that matches fields according the the keys and values. Defined in class class casa.event.EventDescriptor$EventDescriptorLispOperator.

Parameter	Default	Type	Notes
EVENT-TYPE	optional	java.lang.String	The specific subtype of EVENT_MESSAGE_EVENT of to be matched

F.102 NEW-URL

NEW-URL is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR NEW-URL>.

Lambda list

(STRING)

Function documentation

Attempt to construct a URL from a string representation. Defined in class class casa.agentCom.URLDescriptor.

Parameter	Default	Type	Notes
STRING			The URL of the cooperation domain to join.

F.103 ONT.ASSERT

ONT.ASSERT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONT.ASSERT>. Its synonyms are *DECLMAPLET*.

Lambda list

(RELATION-NAME DOMAIN RANGE &KEY ONTOLOGY VERBOSE)

Function documentation

set a type-to-type relationship in the specified relation. Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
RELATION-NAME		java.lang.String	The name of the relation.
DOMAIN		java.lang.String	An element in the domain of this relation.
RANGE			A element or list of elements to be set as the range in this relation.
:ONTOLOGY		java.lang.String	The ontology to put the new type in (default is the agent current ontology)
:VERBOSE			echo the command if verbose/=NIL

F.104 ONT.DESCRIBE

ONT.DESCRIBE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONT.DESCRIBE>.

Lambda list

(ITEM &KEY VERBOSE SYNTAX INDIVIDUAL TYPE RELATION)

Function documentation

Prints out a description of the item. Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
ITEM		java.lang.String	The item to describe.
:VERBOSE			echo the command if verbose/=NIL
:SYNTAX		java.lang.String	manchester, functional, tutorial, XML, DLHTML, DL, KRSS2, KRSS, LatexAxiomsList, latex, OBO, prefix ...
:INDIVIDUAL			print out a lisp description of an individual in the ontology, returning the individual object.
:TYPE			print out a lisp description of a type in the ontology, returning the type object.
:RELATION			print out a lisp description of a relation in the ontology, returning the relation object.

F.105 ONT.GET

ONT.GET is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONT.GET>. Its synonyms are *GET-ONTOLOGY*.

Lambda list

(&OPTIONAL ONTOLOGY &KEY VERBOSE INDIVIDUAL TYPE RELATION (IMPORTS NIL))

Function documentation

Retrieves ontology either from the shared memory or from a file of the same name ([name].ont.lisp). If none of :relation, :type, or :individual is specified the ontology is printed and returned. Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
ONTOLOGY	optional	java.lang.String	The name of the ontology.
:VERBOSE			echo the command if verbose/=NIL
:INDIVIDUAL			print out a lisp description of an individual in the ontology, returning the individual object.
:TYPE			print out a lisp description of a type in the ontology, returning the type object.
:RELATION			print out a lisp description of a relation in the ontology, returning the relation object.
:IMPORTS	def=NIL	java.lang.Boolean	print out the imported ontologies (ignored if INDIVIDUAL, TYPE, or RELATION is present).

F.106 ONT.GET-RESIDENT

ONT.GET-RESIDENT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONT.GET-RESIDENT>. Its synonyms are *GET-RESIDENT-ONTOLOGIES*.

Lambda list

(&KEY VERBOSE)

Function documentation

Retrieves a list of the names of the ontologies in shared memory. Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
:VERBOSE			echo the command if verbose/=NIL

F.107 ONT.IMPORT

ONT.IMPORT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONT.IMPORT>. Its synonyms are *DECLTYPE*.

Lambda list

(NAME &KEY ONTOLOGY VERBOSE)

Function documentation

Import another ontology (ie: make the other ontology a superontology of the agent's ontology). Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
NAME		java.lang.String	The name super (imported) ontology.
:ONTOLOGY		java.lang.String	The ontology to import the new onotlogy into (default is the agent's ontology)
:VERBOSE			echo the command if verbose/=NIL

F.108 ONT.INDIVIDUAL

ONT.INDIVIDUAL is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONT.INDIVIDUAL>. Its synonyms are *DECLINDIVIDUAL*.

Lambda list

(NAME &OPTIONAL TYPE The type of the invidiual. &KEY ONTOLOGY VERBOSE)

Function documentation

Declare an individual in the A-box. Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
NAME		java.lang.String	The name of the new individual.
TYPE	optional	java.lang.String	

F.109 ONT.IS-INDIVIDUAL

ONT.IS-INDIVIDUAL is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONT.IS-INDIVIDUAL>. Its synonyms are *DECLINDIVIDUAL*.

Lambda list

(NAME &KEY ONTOLOGY)

Function documentation

Return true iff the parameter is an individual in the ontology. Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
NAME		java.lang.String	The name of the item.
:ONTOLOGY		java.lang.String	The ontology to look in (default is the agent current ontology)

F.110 ONT.IS-OBJECT

ONT.IS-OBJECT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONT.IS-OBJECT>. Its synonyms are *DECLINDIVIDUAL*.

Lambda list

(NAME &KEY ONTOLOGY)

Function documentation

Return true iff the parameter is a type or an individual in the ontology. Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
NAME		java.lang.String	The name of the item.
:ONTOLOGY		java.lang.String	The ontology to look in (default is the agent current ontology)

F.111 ONT.IS-TYPE

ONT.IS-TYPE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONT.IS-TYPE>. Its synonyms are *DECLINDIVIDUAL*.

Lambda list

(NAME &KEY ONTOLOGY)

Function documentation

Return true iff the parameter is a type (not an individual) in the ontology. Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
NAME		java.lang.String	The name of the item.
:ONTOLOGY		java.lang.String	The ontology to look in (default is the agent current ontology)

F.112 ONT.RELATED-TO

ONT.RELATED-TO is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONT.RELATED-TO>.

Lambda list

(RELATION DOMAIN &OPTIONAL RANGE &KEY ONTOLOGY VERBOSE)

Function documentation

If RANGE is specified, return T iff the DOMAIN is related to the RANGE by the specified RELATION, otherwise return the set of elements in range of DOMAIN by RELATION. Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
RELATION		java.lang.String	The name of the relation.
DOMAIN		java.lang.String	An element in the domain of this relation.
RANGE	optional	java.lang.String	A element in the range in this relation.
:ONTOLOGY		java.lang.String	The ontology in which to check (default is the agent current ontology)
:VERBOSE			echo the command if verbose/=NIL

F.113 ONT.RELATION

ONT.RELATION is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONT.RELATION>. Its synonyms are *DECLRELATION*.

Lambda list

(RELATION-NAME &KEY (DOMAIN-CONSTRAINT NIL) (RANGE-CONSTRAINT NIL) BASE INVERSE REFLEXIVE SYMMETRIC ASYMMETRIC TRANSITIVE USES ASSIGNABLE ONTOLOGY VERBOSE)

Function documentation

Define a relation in the agent's casa ontology. Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
RELATION-NAME			The name of the relation.
:DOMAIN--CONSTRAINT	def=NIL	casa.ontology.-Constraint	The constraint for the domain elements of this relation.
:RANGE-CONSTRAINT	def=NIL	casa.ontology.-Constraint	The constraint for the range elements of this relation.
:BASE			The relation on which this is based.
:INVERSE			This relation is to be an inverse relation (x->y => y->x) of the base relation.
:REFLEXIVE			This relation is a reflexive (x->x) version of the base relation.
:SYMMETRIC			This relation is a symmetric (x->y => y->x) version of the base relation.
:ASYMMETRIC			This relation is an asymmetric (x->y => (y->x)) version of the base relation.
:TRANSITIVE			This relation is a transitive (x->y & y->z => x->y) version of the base relation.
:USES		java.lang.String	This relation uses relation [arg] as an equivalence relation.
:ASSIGNABLE			This relation is assignable (that is one can use it as first argument in a (declMaplet ...) operator
:ONTOLOGY		java.lang.String	The ontology to put the new type in (default is the agent current ontology)
:VERBOSE			echo the command if verbose/=NIL

F.114 ONT.SET-DEFAULT

ONT.SET-DEFAULT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONT.SET-DEFAULT>. Its synonyms are *SET-DEFAULT-ONTOLOGY*.

Lambda list

(NAME)

Function documentation

Sets the agent's default ontology either from the shared memory or from a file of the same name ([name].ont.lisp). Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
NAME		java.lang.String	The name of the ontology.

F.115 ONT.TYPE

ONT.TYPE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONT.TYPE>. Its synonyms are *DECLTYPE*.

Lambda list

(NAME &OPTIONAL TYPE The type of the invidiual. &KEY ONTOLOGY VERBOSE)

Function documentation

Declare a type in the T-box. Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
NAME		java.lang.String	The name of the new type.
TYPE	optional	java.lang.String	

F.116 ONTOLOGY

ONTOLOGY is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR ONTOLOGY>. Its synonyms are *DECLONTOLOGY*, *WITH-ONTOLOGY*.

Lambda list

(NAME SUPER-ONTOLOGIES RELS-AND-TYPES &KEY VERBOSE)

Function documentation

Declare a new Ontology or extends an existing Ontology. Defined in class interface casa.ontology.Ontology.

Parameter	Default	Type	Notes
NAME		java.lang.String	The name of the ontology.
SUPER-ONTOLOGIES			A list of existing ontology names to append to the super--ontologies of this ontology.
RELS-AND-TYPES			A list of decl* type declarations.
:VERBOSE			echo the command if verbose/=NIL

F.117 PERFORMDESCRIPTOR

PERFORMDESCRIPTOR is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR PERFORMDESCRIPTOR>.

Lambda list

(&OPTIONAL (STATUSVALUE 0) &ALLOW-OTHER-KEYS)

Function documentation

Creates a new PerformDescriptor with the status value if the parameter (def=0) and the fields specified in the keys on the parameter PD's feilds. Defined in class class casa.PerformDescriptor.

Parameter	Default	Type	Notes
STATUSVALUE	optional def=org.armedbear.lisp.Fixnum@0	java.lang.Integer	The integer status value of the new PerformDescriptor.

F.118 PERFORMDESCRIPTOR.GET-STATUS

PERFORMDESCRIPTOR.GET-STATUS is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR PERFORMDESCRIPTOR.GET-STATUS>.

Lambda list

(PD)

Function documentation

Retrieves this PerformDescriptor's Status object. Defined in class class casa.PerformDescriptor.

Parameter	Default	Type	Notes
PD		casa.-PerformDescriptor	The PerformDescriptor from which the Status object will be retrieved.

F.119 PERFORMDESCRIPTOR.GET-STATUS-VALUE

PERFORMDESCRIPTOR.GET-STATUS-VALUE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR PERFORMDESCRIPTOR.GET-STATUS-VALUE>.

Lambda list

(PD)

Function documentation

Retrieves this PerformDescriptor's Status's value. Defined in class class casa.PerformDescriptor.

Parameter	Default	Type	Notes
PD		casa.-PerformDescriptor	The PerformDescriptor from which the Status's value will be retrieved.

F.120 PERFORMDESCRIPTOR.GET-VALUE

PERFORMDESCRIPTOR.GET-VALUE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR PERFORMDESCRIPTOR.GET-VALUE>.

Lambda list

(PD KEY)

Function documentation

Retrieves this PerformDescriptor's value for KEY or nil if KEY is not defined. Defined in class class casa.PerformDescriptor.

Parameter	Default	Type	Notes
PD		casa.-PerformDescriptor	The PerformDescriptor from which the value will be retrieved.
KEY		java.lang.String	The key.

F.121 PERFORMDESCRIPTOR.OVERLAY

PERFORMDESCRIPTOR.OVERLAY is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR PERFORMDESCRIPTOR.OVERLAY>.

Lambda list

(PD &ALLOW-OTHER-KEYS)

Function documentation

Overlays the fields specified in the keys on the parameter PD's fields. Defined in class class casa.PerformDescriptor.

Parameter	Default	Type	Notes
PD		casa.-PerformDescriptor	The PerformDescriptor whose fields are to be overlayed.

F.122 PING

PING is a synonym for AGENT.PING (Section F.32).

F.123 POLICY

POLICY is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR POLICY>.

Lambda list

(EVENT-DESCRIPTOR ACTIONS &OPTIONAL DOC &KEY PRECONDITION POSTCONDITION NAME GHOST)

Function documentation

Return a new Policy object. Defined in class class casa.policy.Policy.

Parameter	Default	Type	Notes
EVENT-DESCRIPTOR			The antecedent - a description of the event; either an EventDescriptor or a Cons that will evaluate to an EventDescriptor.
ACTIONS		org.armedbear.-lisp.Cons	The result - a Cons list of actions to take if the event happens.
DOC	optional	java.lang.String	The documentation string for the policy.
:PRECONDITION			A (backquoted) boolean expression acting as a guard on this policy.
:POSTCONDITION			A (backquoted) boolean expression for the post condition of this policy (currently not used).
:NAME		java.lang.String	The name to be used as the short name of this policy.
:GHOST		java.lang.Boolean	Sets this policy to be a ghost (not counted as a 'real' policy application).

F.124 PRIMITIVEONTOLOGY.INSTANCE-OF

PRIMITIVEONTOLOGY.INSTANCE-OF is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR PRIMITIVEONTOLOGY.INSTANCE-OF>. Its synonyms are *instance-of*, *primitiveOntology.instance-of*.

Lambda list

(DOMAIN &OPTIONAL RANGE &KEY ONTOLOGY)

Function documentation

Test weather items are related in the instance-of relation. Defined in class class casa.ontology.v3.CASAOntology.

Parameter	Default	Type	Notes
DOMAIN		java.lang.String	
RANGE	optional	java.lang.String	
:ONTOLOGY		java.lang.String	The ontology use (default is the agent current ontology)

F.125 PRIMITIVEONTOLOGY.ISA

PRIMITIVEONTOLOGY.ISA is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR PRIMITIVEONTOLOGY.ISA>. Its synonyms are *isa*, *primitiveOntology.isa*.

Lambda list

(DOMAIN &OPTIONAL RANGE &KEY ONTOLOGY)

Function documentation

Test weather items are related in the isa relation. Defined in class class casa.ontology.v3.CASAOntology.

Parameter	Default	Type	Notes
DOMAIN		java.lang.String	
RANGE	optional	java.lang.String	
:ONTOLOGY		java.lang.String	The ontology use (default is the agent current ontology)

F.126 PRIMITIVEONTOLOGY.ISA-ANCESTOR

PRIMITIVEONTOLOGY.ISA-ANCESTOR is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR PRIMITIVEONTOLOGY.ISA-ANCESTOR>. Its synonyms are *isa-ancestor*, *primitiveOntology.isa-ancestor*.

Lambda list

(DOMAIN &OPTIONAL RANGE &KEY ONTOLOGY)

Function documentation

Test weather items are related in the isa-ancestor relation. Defined in class class casa.ontology.v3.CASAOntology.

Parameter	Default	Type	Notes
DOMAIN		java.lang.String	
RANGE	optional	java.lang.String	
:ONTOLOGY		java.lang.String	The ontology use (default is the agent current ontology)

F.127 PRIMITIVEONTOLOGY.ISA-CHILD

PRIMITIVEONTOLOGY.ISA-CHILD is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR PRIMITIVEONTOLOGY.ISA-CHILD>. Its synonyms are *isa-child*, *primitiveOntology.isa-child*.

Lambda list

(DOMAIN &OPTIONAL RANGE &KEY ONTOLOGY)

Function documentation

Test weather items are related in the isa-child relation. Defined in class class casa.ontology.v3.CASAOntology.

Parameter	Default	Type	Notes
DOMAIN		java.lang.String	
RANGE	optional	java.lang.String	
:ONTOLOGY		java.lang.String	The ontology use (default is the agent current ontology)

F.128 PRIMITIVEONTOLOGY.ISA-DESCENDANT

PRIMITIVEONTOLOGY.ISA-DESCENDANT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR PRIMITIVEONTOLOGY.ISA-DESCENDANT>. Its synonyms are *isa-descendant*, *primitiveOntology.isa-descendant*.

Lambda list

(DOMAIN &OPTIONAL RANGE &KEY ONTOLOGY)

Function documentation

Test weather items are related in the isa-descendant relation. Defined in class class casa.ontology.v3.CASAOntology.

Parameter	Default	Type	Notes
DOMAIN		java.lang.String	
RANGE	optional	java.lang.String	
:ONTOLOGY		java.lang.String	The ontology use (default is the agent current ontology)

F.129 PRIMITIVEONTOLOGY.ISA-PARENT

PRIMITIVEONTOLOGY.ISA-PARENT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR PRIMITIVEONTOLOGY.ISA-PARENT>. Its synonyms are *isa-parent*, *primitiveOntology.isa-parent*.

Lambda list

(DOMAIN &OPTIONAL RANGE &KEY ONTOLOGY)

Function documentation

Test weather items are related in the isa-parent relation. Defined in class class casa.ontology.v3.CASAOntology.

Parameter	Default	Type	Notes
DOMAIN		java.lang.String	
RANGE	optional	java.lang.String	
:ONTOLOGY		java.lang.String	The ontology use (default is the agent current ontology)

F.130 PRIMITIVEONTOLOGY.ISEQUAL

PRIMITIVEONTOLOGY.ISEQUAL is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR PRIMITIVEONTOLOGY.ISEQUAL>. Its synonyms are *isequal*, *primitiveOntology.isequal*.

Lambda list

(DOMAIN &OPTIONAL RANGE &KEY ONTOLOGY)

Function documentation

Test weather items are related in the isequal relation. Defined in class class casa.ontology.v3.CASAOntology.

Parameter	Default	Type	Notes
DOMAIN		java.lang.String	
RANGE	optional	java.lang.String	
:ONTOLOGY		java.lang.String	The ontology use (default is the agent current ontology)

F.131 PRIMITIVEONTOLOGY.PROPER-INSTANCE-OF

PRIMITIVEONTOLOGY.PROPER-INSTANCE-OF is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR PRIMITIVEONTOLOGY.PROPER-INSTANCE-OF>. Its synonyms are *proper-instance-of*, *primitiveOntology.proper-instance-of*.

Lambda list

(DOMAIN &OPTIONAL RANGE &KEY ONTOLOGY)

Function documentation

Test weather items are related in the proper-instance-of relation. Defined in class class casa.ontology.v3.CASAOntology.

Parameter	Default	Type	Notes
DOMAIN		java.lang.String	
RANGE	optional	java.lang.String	
:ONTOLOGY		java.lang.String	The ontology use (default is the agent current ontology)

F.132 proper-instance-of

proper-instance-of is a synonym for PRIMITIVEONTOLOGY.PROPER-INSTANCE-OF (Section F.131).

F.133 QUERY-REF

QUERY-REF is a synonym for QUERYREF (Section F.134).

F.134 QUERYREF

QUERYREF is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR QUERYREF>. Its synonyms are *QUERY-REF*.

Lambda list

(FORMULA &KEY REPLY-EXP TO REPLY-LIST)

Function documentation

Queries the formula for the reference from the KB returning a ListOfTerm structure. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
FORMULA		java.lang.String	The formula.
:REPLY-EXP			Return an expression appropriate for a reply instead of a ListOfTerm structure
:TO		java.lang.String	Send the request to another agent and return a Collection<Term>.
:REPLY-LIST			Return a list of the individual results as Strings.

F.135 QUERYW

QUERYW is a synonym for AGENT.SEND-QUERY-AND-WAIT (Section F.39).

F.136 REQUESTW

REQUESTW is a synonym for AGENT.SEND-REQUEST-AND-WAIT (Section F.40).

F.137 SC.ADD

SC.ADD is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR SC.ADD>.

Lambda list

(&KEY DEBTOR CREDITOR PERFORMATIVE ACT ACTION-CLASS ACTION DEPENDS-ON SHARED PERSISTENT GETEVENTS)

Function documentation

Return a new ADD operator. Defined in class class casa.policy.Policy.

Parameter	Default	Type	Notes
:DEBTOR		casa.agentCom.-URLDescriptor	The debtor.
:CREDITOR		casa.agentCom.-URLDescriptor	The creditor.
:PERFORMATIVE			The performative.
:ACT		casa.Act	The act.
:ACTION-CLASS		java.lang.String	The action class, which needs to be a subclass of Action.
:ACTION		org.armedbear.-lisp.Cons	The action data, which needs to be a cons list.
:DEPENDS-ON		casa.-socialcommitments.-SocialCommitmentDescriptor	The social commitment that this one depends on.
:SHARED			The commitment is a shared commitment.
:PERSISTENT			The commitment is persistent and can only be removed with a CANCEL.
:GETEVENTS			We should retrieve the events from the agent.

F.138 SC.CANCEL

SC.CANCEL is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR SC.CANCEL>.

Lambda list

(&KEY DEBTOR CREDITOR PERFORMATIVE ACT)

Function documentation

Return a new CANCEL operator. Defined in class class casa.policy.Policy.

Parameter	Default	Type	Notes
:DEBTOR		casa.agentCom.-URLDescriptor	The debtor.
:CREDITOR		casa.agentCom.-URLDescriptor	The creditor.
:PERFORMATIVE			The performative.
:ACT		casa.Act	The act.

F.139 SC.FULFIL

SC.FULFIL is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR SC.FULFIL>.

Lambda list

(&KEY DEBTOR CREDITOR PERFORMATIVE ACT)

Function documentation

Return a new FULFIL operator. Defined in class class casa.policy.Policy.

Parameter	Default	Type	Notes
:DEBTOR		casa.agentCom.-URLDescriptor	The debtor.
:CREDITOR		casa.agentCom.-URLDescriptor	The creditor.
:PERFORMATIVE		java.lang.String	The performative.
:ACT		casa.Act	The act.

F.140 SCDESCRIPTOR

SCDESCRIPTOR is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR SCDESCRIPTOR>.

Lambda list

(&KEY PERFORMATIVE ACT &ALLOW-OTHER-KEYS)

Function documentation

Return a new SocialCommitmentDescriptor that matches fields according the the keys and values. Defined in class class casa.socialcommitments.SocialCommitmentDescriptor.

Parameter	Default	Type	Notes
:PERFORMATIVE		java.lang.String	The performative.
:ACT		casa.Act	The act.

F.141 SCOM

SCOM is a synonym for AGENT.SHOW-COMMITMENTS (Section F.41).

F.142 SCONV

SCONV is a synonym for AGENT.SHOW-CONVERSATIONS (Section F.42).

F.143 SEQ

SEQ is a synonym for AGENT.SHOW-EVENT-QUEUE (Section F.43).

F.144 SERIALIZE

SERIALIZE is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR SERIALIZE>.

Lambda list

(OBJECT)

Function documentation

Returns the casa serialization string for the object. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
OBJECT			The object to serialize.

F.145 SET-DEFAULT-ONTOLOGY

SET-DEFAULT-ONTOLOGY is a synonym for ONT.SET-DEFAULT (Section F.114).

F.146 SET-SYSTEM

SET-SYSTEM is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR SET-SYSTEM>.

Lambda list

(PROPERTY VALUE)

Function documentation

Set the string value of a system property. Defined in class class casa.TransientAgent.

Parameter	Default	Type	Notes
PROPERTY			string name of the system property to change
VALUE			the new value of the property

F.147 SHOULDDOEXECUTEREQUEST

SHOULDDOEXECUTEREQUEST is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR SHOULDDOEXECUTEREQUEST>.

Lambda list

(MESSAGE)

Function documentation

Function's help is undefined. Defined in class class java.lang.reflect.Method.

Parameter	Default	Type	Notes
MESSAGE		casa.MLMessage	The message to decide to execute or not.

F.148 SLEEP-IGNORING-INTERRUPTS

SLEEP-IGNORING-INTERRUPTS is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR SLEEP-IGNORING-INTERRUPTS>.

Lambda list

(&optional SECONDS)

Function documentation

Similar to the lisp (sleep [seconds]) operator, but will ingore itnerrupts. Defined in class class casa.abcl.Lisp.

Parameter	Default	Type	Notes
SECONDS	optional	java.lang.Integer	The number of seconds to wait.

F.149 SOCIALCOMMITMENT

SOCIALCOMMITMENT is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR SOCIALCOMMITMENT>.

Lambda list

(&KEY (DEBTOR NIL) (CREDITOR NIL) (PERFORMATIVE NIL) (ACT NIL) (EVENT NIL) (ACTION NIL) (EVENTS NIL) The events associated with this commitment (may be mil)!)

Function documentation

Create and return a new social commitment. Defined in class class casa.socialcommitments.SocialCommitment.

Parameter	Default	Type	Notes
:DEBTOR	def=NIL	casa.agentCom.-URLDescriptor	The deptor
:CREDITOR	def=NIL	casa.agentCom.-URLDescriptor	The creditor
:PERFORMATIVE	def=NIL	java.lang.String	The performative of this commitment
:ACT	def=NIL	casa.Act	The act of this commitment
:EVENT	def=NIL	casa.event.Event	The originating event of this commitemnt
:ACTION	def=NIL	casa.-socialcommitments.-Action	The aciton that this commitment embodies
:EVENTS	def=NIL	org.armedbear.-lisp.Cons	

F.150 SUBSCRIBE-CONVERSATION

SUBSCRIBE-CONVERSATION is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR SUBSCRIBE-CONVERSATION>.

Lambda list

(NAME TO EXPRESSION &KEY SAY)

Function documentation

Declares and instantiates a subscribe conversation. Defined in class class casa.conversation2.SubscribeClientConversation.

Parameter	Default	Type	Notes
NAME		java.lang.String	The name of the conversation.
TO		java.lang.String	A Cons list of Lisp functions describing sub-conversations or policies.
EXPRESSION		java.lang.String	A Cons list of pairs of symbol/values pairs (themselves Cons lists) that will be bound in the context of the conversation. The expressions are evaluated at the time the conversation is created.
:SAY		java.lang.String	Message to print as a warning message when the event occurs.

F.151 TOSTRING

TOSTRING is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR TOSTRING>.

Lambda list

(OBJECT &KEY PRETTY)

Function documentation

prints a java object through it's toString() method. Defined in class class casa.abcl.Lisp.

Parameter	Default	Type	Notes
OBJECT			The object to convert to a String.
:PRETTY		java.lang.Boolean	Call toString(true) instead of toString() for MLMessage's.

F.152 TRANSFORM-STRING

TRANSFORM-STRING is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR TRANSFORM-STRING>.

Lambda list

(TRANSFORMATION STRING)

Function documentation

returns the argument string transformed in id—id—... format. Defined in class class casa.PerfActTransformation.

Parameter	Default	Type	Notes
TRANSFORMATION		casa.-PerfActTransformation	The Transformation o perform.
STRING		java.lang.String	The string to translate in id—id—... format.

F.153 TRANSFORMATION

TRANSFORMATION is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR TRANSFORMATION>.

Lambda list

(FROM TO)

Function documentation

Instantiates a transformation object that translates over the PERFORMATIVE/ACT of any Describable object. Defined in class class casa.PerfActTransformation.

Parameter	Default	Type	Notes
FROM		java.lang.String	The from part in id—id... form.
TO		java.lang.String	The to part in id—id... form.

F.154 TRANSFORMATION.GET-FROM

TRANSFORMATION.GET-FROM is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR TRANSFORMATION.GET-FROM>.

Lambda list

(TRANSFORMATION)

Function documentation

returns the FROM part of the argument TRANSFORMATION in id—id—... format. Defined in class class casa.PerfActTransformation.

Parameter	Default	Type	Notes
TRANSFORMATION		casa.-PerfActTransformation	The Transformation.

F.155 TRANSFORMATION.GET-TO

TRANSFORMATION.GET-TO is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR TRANSFORMATION.GET-TO>.

Lambda list

(TRANSFORMATION)

Function documentation

returns the TO part of the argument TRANSFORMATION in id—id—... format. Defined in class class casa.PerfActTransformation.

Parameter	Default	Type	Notes
TRANSFORMATION		casa.-PerfActTransformation	The Transformation.

F.156 UI.HISTORY

UI.HISTORY is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR UI.HISTORY>.

Lambda list

(&OPTIONAL INDEX &KEY MAX)

Function documentation

no param: show a command history list; param: access an element. Defined in class class casa.ui.ObservingAgentUI.

Parameter	Default	Type	Notes
INDEX	optional	java.lang.Integer	The element of the history list to execute.
:MAX		java.lang.Integer	The the maximum number of history elements to save.

F.157 UI.MONITOR

UI.MONITOR is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR UI.MONITOR>.

Lambda list

(&KEY EVENTS MESSAGES INFO UNKNOWN TRACE)

Function documentation

display various notification messages as they come in from the agent. Defined in class class casa.ui.ObservingAgentUI.

Parameter	Default	Type	Notes
:EVENTS		java.lang.Boolean	show agent events as they come in.
:MESSAGES		java.lang.Boolean	show agent CASA messages as they come in.
:INFO		java.lang.Boolean	show information messages as they come in.
:UNKNOWN		java.lang.Boolean	show unknown notifications as they come in.
:TRACE		java.lang.Boolean	show trace messages as they come in.

F.158 URL.GET

URL.GET is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR URL.GET>.

Lambda list

(URL &KEY FILE NAME HOST DATA DIRECTORY FRAGMENT HOSTANDPORT LACPORT PATH PORT SHORTESTNAME)

Function documentation

return the url as a string, or extract a component; up to ONE key is allowed (result is undefined if more than one key). Defined in class class casa.agentCom.URLDescriptor.

Parameter	Default	Type	Notes
URL			The URL as a URL object or a string.
:FILE			The FILE or NAME component.
:NAME			The FILE or NAME component.
:HOST			The DATA component.
:DATA			The DATA component.
:DIRECTORY			The DIRECTORY component.
:FRAGMENT			The FRAGMENT component.
:HOSTANDPORT			The HOSTANDPORT component.
:LACPORT			The LACPORT component.
:PATH			The PATH component.
:PORT			The PORT component.
:SHORTESTNAME			The SHORTESTNAME component.

F.159 URLS.GET

URLS.GET is an external symbol in the COMMON-LISP-USER package. Its function binding is #<SPECIAL-OPERATOR URLS.GET>.

Lambda list

(&KEY STRING FORMAT)

Function documentation

return a Cons list of all known URLs as Java URLDescriptors. Defined in class class casa.agentCom.URLDescriptor.

Parameter	Default	Type	Notes
:STRING			If this is non-NIL, the returned Cons list will be URL Strings.
:FORMAT			If this is non-null, the return will be a string containing the String urls separated by newlines.

F.160 WITH-ONTOLOGY

WITH-ONTOLOGY is a synonym for ONTOLOGY (Section F.116).

Acknowledgments

The author wishes to acknowledge the Candian Nationial Science and Engineering Research Council (NSERC) for financial support of the research.

Bibliography

[Americal National Standards Institute (ANSI), 2004] Americal National Standards Institute (ANSI) (2004). Lisp Programming Language - ANSI standard document ANSI INCITS 226-1994 (R2004), (formerly X3.226-1994 (R1999)).

[Bellifemine et al., 2007] Bellifemine, F., Caire, G. and Greenwood, D. (2007). Developing multi-agent systems with JADE. Wiley series in agent technology, John Wiley.

[Bellifemine et al., 2003] Bellifemine, F., Caire, G., Poggi, A. and Rimassa, G. (2003). JADE: A White Paper. Technical Report Volume 3, n. 3 Telecom Italia Lab Italy. Available: http://jade.cselt.it/.

[Common-Lisp.net, 2011] Common-Lisp.net (2011). Armed Bear Common Lisp (ABCL) - Common Lisp on the JVM. http://common-lisp.net/project/armedbear/.

[Evenson et al., a] Evenson, M., Hulsmann, E., Stalla, A. and Voutilainen, V. A Manual for Armed Bear Common Lisp.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley, Reading, Mass.

[Motik et al., 2011] Motik, B., Patel-Schneider, P. F. and Grau, B. C. (2011). OWL 2 Web Ontology Language: Direct Semantics (Second Edition). W3c recommendation W3C. Available: http://www.w3.org/TR/owl2-direct-semantics/.

[Motik et al., 2012] Motik, B., Patel-Schneider, P. F. and Parsia, B. (2012). OWL 2 Web Ontology Language: Document Overview (Second Edition). W3c recommendation W3C. Available: http://www.w3.org/TR/owl2-overview/.

[Pautret, 2006] Pautret, V. (2006). Jade Semantics Add-on Programmer's guide. http://jade.tilab.com/doc/tutorials/SemanticsProgrammerGuide.pdf.

[Stanford Center for Biomedical Informatics Research, 2013] Stanford Center for Biomedical Informatics Research (2013). Protégé. http://protege.stanford.edu.

[Steele, 1990] Steele, G. L. (1990). Common Lisp the Language. Number ISBN 1-55558-041-6, 2nd edition edition, Digital Press.

[Telecom Italia Lab, 2008] Telecom Italia Lab (2008). JADE (Java Agent Development Environment). http://jade.cselt.it/.