Autonomic Information Auditing through Electronic Institutions
Technical Report IIIA-TR-2009-07

Héctor G. Ceballos[1], Pablo Noriega[2], Francisco Cantú[1]
ceballos@itesm.mx, pablo@iiia.csic.es, fcantu@itesm.mx
Tecnológico de Monterrey[1], IIIA-CSIC[2]

This report presents the development of an Electronic Institution for auditing information in a research and graduate programs corporate memory. Electronic Institutions are used for formalizing processes performed by human experts and professors providing a platform for gradual automation. New features like a Directory Facilitator and a protocol for instantiating new agents and inviting them to certain scene are implemented. Such features provide self-configuration capabilities to the system.

July 20th, 2009

Contents

A Research and Graduate Program Corporate Memory

The Tecnologico de Monterrey counts with a research and graduate programs corporate memory that consolidates information gathered from institutional transactional systems, information reported by researchers and information gathered from web sites [Cantu et al, 2005]. Meanwhile information proceeding from transactional systems is validated by institutional processes, information feed manually is susceptible of errors like misclassifications, duplicity or missing data. Given that this information is used for calculating institutional indicators used on institutional making decision, consistency and trustworthy is an important issue.

Information feed by professors is associated to catalogs representing organizational units, internal and external people and institutions, accountant information, research and graduate programs, etc., resembling a Research Social Network for our institution. On this way, new information feed by one professor may affect the personal record of another person (a coauthored publication for example). Such changes are notified to related or interested people, allowing them to complement or correct information.

These notifications allows to have auditing checkpoints where expert auditors can revise the information feed by professors and perform the necessary changes in order to maintain the *repository consistency*. This process is made periodically and results of the auditing are notified to the information responsible. The user can reply to the correction and provide additional information for validating the original information.

Next is presented a simplified example of an auditing process on publications information and some issues that motivated the current work.

Publications auditing

The publications repository registers the scientific production of professors organizing it in an institutional taxonomy consisting of about 20 different categories organized at different levels. Every category receives a different weight on every evaluation system. For simplicity we present a simplified taxonomy constituted by: articles in journals, articles in proceedings and thesis. This taxonomy is illustrated in Figure 1.

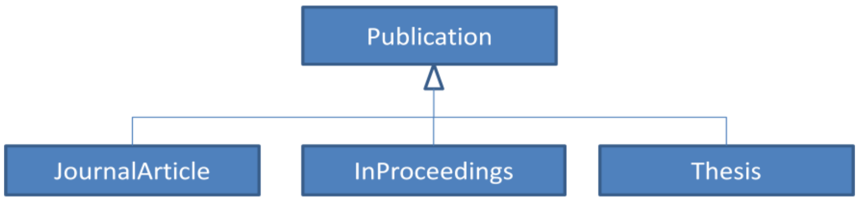


Figure 1. Simplified publications taxonomy

Information stored in the repository is actually the metadata of the publication; hence we have common data elements as shown in Figure 2.

There is additional information and constraints for each type of publication. For example, a journal article is published in a journal, meanwhile that a proceedings article is published in the proceedings of a conference. It is important to maintain a differentiated catalogue of journals and conferences that allows not only quantifying but qualifying professors' scientific production. This qualification is made through the Thomson's Journal Citation Report (JCR) impact factor of the journals on which the article appears.

A thesis for example, is published with the support of an education institution and distinguishes itself of other publications because its author is a student instead of a professor. Professors appear as advisors in a separated field.

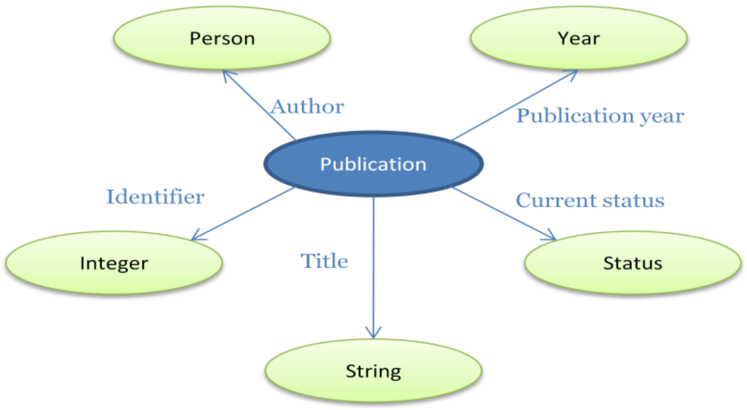


Figure 2. Common metadata for all the publications.

There are some common inconsistencies that expert auditors have already detected and modeled. One of them is the duplicity of the publication in the repository and consists on the existence of two publications in the repository having such a degree of similarity that make the auditor suspect that both are in fact the same publication registered twice.

Other common problem is the inconsistency between the status of the publication and the year of publication. Given that the repository allows that the professor register its production in progress, indicating a tentative publication date and a current status, it is possible that both data become inconsistent with the pass of time. Publication status and the transitions between them are shown in Figure 3.

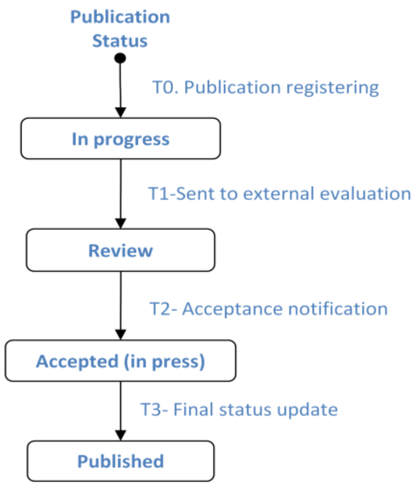


Figure 3. Publication status evolution

Other problem is posed by homonymy in participants' names, which can produce association errors that cause losses on credit assignment. This is hardly detected by an expert auditor.

As can be seen, some of the previous problems cannot be detected online. The first requires a comparison against the entire content of the repository; meanwhile the second requires evaluating the rule periodically. Both verifications are done offline. This auditing can be automated if the outcome of the automatic auditing is revised by a human expert and the results are notified to the responsible professor for a second opinion.

Professors cannot be bothered too often for validating the information. The expert auditor is allowed to gather information from other systems in order to reduce the necessity of asking the professor. On the other hand, the confidence on the information stored depends on the revision of the direct responsible. As can be seen, both objectives are contradictory.

The expert auditor uses web sites (Google) to verify provided information and for gathering missing data or additional information not requested by the system. Auditing rules might be wrong or be too broad. Along time and due to his own experience, the expert updates his *auditing rules* adding new criteria. This causes that auditors consider different criteria for the same auditing rule.

Another issue is *information incompleteness* in the repository, i.e. the existence of publications that have not been reported by professors but that have been published in some conference or journal. We are not attacking this problem right now.

Other Practical issues on the current process

- Manual data extraction of the Thomson's JCR impact factors.
- Taxonomy is not enough for classifying information. Main categories are used for validating the minimal information that must be captured. Nevertheless, categories used for visualization and generation of reports and indicators depend on the type of analysis, which usually doesn't have a direct correspondence with the former. For instance, all the scientific production published in foreign countries.

Autonomic Information Auditing

In order to provide a software solution capable of growing and adapting to the organizational environment of our institution is presented the approach of Autonomic Information Auditing. The objective is having an infrastructure on which expert knowledge is transferred to autonomous agents supervised by human experts. On this way, human intervention is reduced progressively but is available for cases on which the agent cannot make a decision based on current information. Causal discovery is guided by a human expert.

Periodic auditing of the information contained in the publications repository must discover the type of inconsistencies described above. If the solution is evident, i.e. there is a correction rule that can be applied confidently, the correction can be made automatically. The professor is notified of the change in order to allow him/her reply to it.

If the solution requires human supervision, i.e. there are many possible solutions and there is no a single one that had demonstrated to be right in most of the cases, then a human expert is notified in order to select the right option or correct manually the record. If the solution is not evident then the professor is requested for making the correction. This process is illustrated on Figure 4.

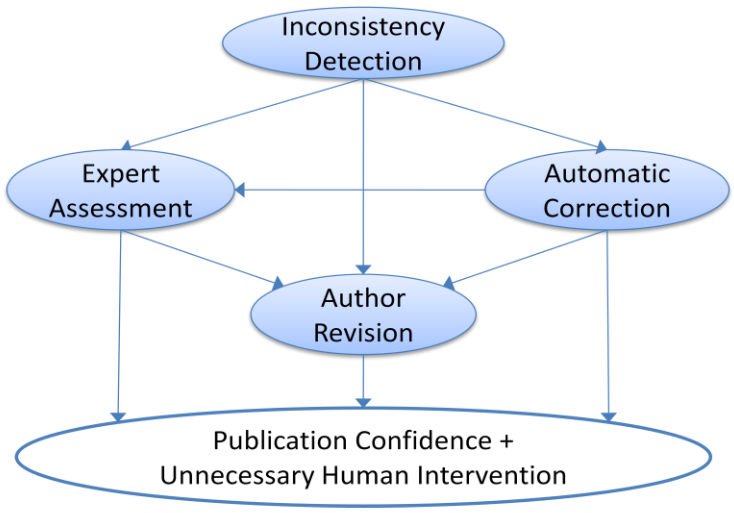


Figure 4. Core of the auditing process.

As can be seen the decision is not always the same, depends on the information of the case evaluated and the confidence on the possible solutions. Beyond that, we rely on the response of experts and professors for warrant the consistency of the repository, which depends on the consistency of the each publication contained on it.

Automation of the auditing process requires implementing a back-end platform on charge of monitoring, auditing and correcting information feed by professors. Our proposal includes implementing a MultiAgents System for performing the offline auditing. The proposed architecture is illustrated in Figure 5 along the systems and classes of agents necessary for the task.

Expert auditors are on charge of defining the inconsistency and correction rules. These rules must be expressed in a format gentle for experts and that agents can understand and execute. The confidence on the efficiency of the rules must be expressed probabilistically and be modeled through agents' observations (experience).

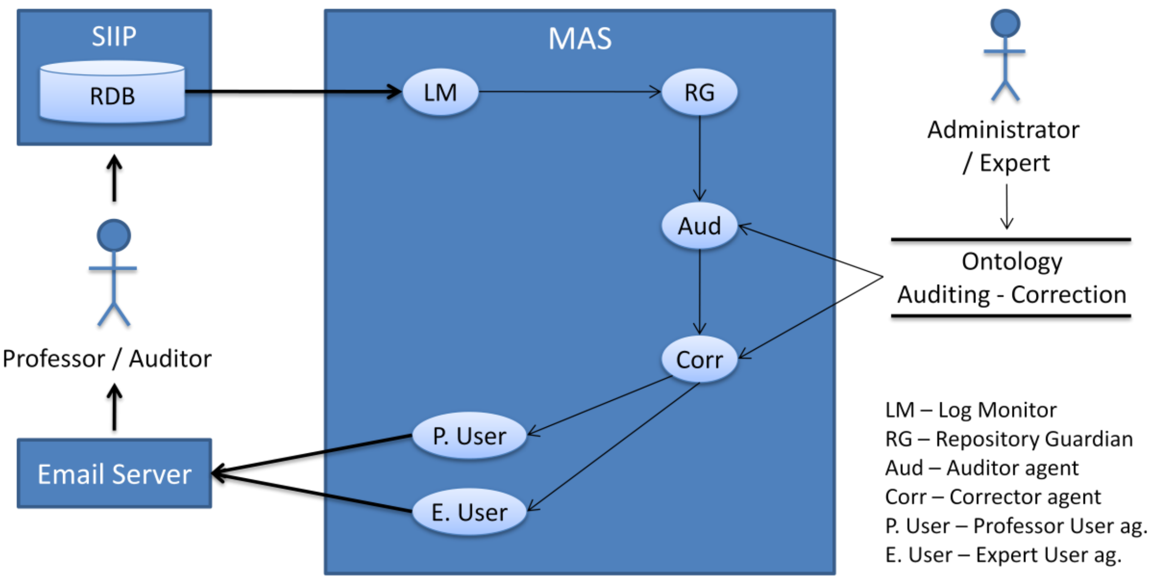


Figure 5. Multiagents System Architecture for publications auditing

The LogMonitor (LM) agent monitors changes in the repository. The RepositoryGuardian (RG) agent is responsible of instantiating service agents required for the auditing process, as well as keep track of the entire process and learn from it. Auditor (Aud) agents evaluate auditing rules on demand; they are responsible for gathering the necessary information for auditing the record and request a correction whenever one is available. Corrector (Corr) agents apply the correction rules if they are trustworthy. User agents communicate the information to its designated user through email messages and waits for their response. Additionally, actions derived of notifications are tracked by the LogMonitor agent in the information system, closing the loop.

The RepositoryGuardian agent must assure that the LogMonitor agent is working and that there is an Auditor agent for each kind of inconsistency reported in the ontology by the administrator. Similarly to Auditors, Corrector agents must be available for every possible correction applicable to the recognized types of inconsistencies. User agents are instantiated on demand.

Electronic Institutions for Autonomic Information Auditing

Electronic Institutions formalism [Sierra and Noriega, 1997] and tools [Esteva et al, 2002] developed at IIIA provide a valuable framework for modeling and development of the Multiagents System proposed in previous section. In this section is described the iterative process of modeling, implementing and testing the MAS. Some ideas or approaches are evaluated and through them the formalism is better understand.

First Modelling Phase (EI specification)

As first step, having valid illocutions in scenes as the only way for exchanging information between agents motivated the introduction of a new type of agent: the PubCarrier. This agent is responsible for transporting the information of the publication through the different scenes in the process allowing capture an entire picture of the auditing process as an observed case. Otherwise, we would think of having an agent accessing the repository at any moment, from any part of the process. With this addition we started to model the process in Islander.

The **first approach** consisted on modeling the auditing process through multiple scenes and controlling the entrance and exit of agents along the process. Each scene has a part of the auditing protocol where the type of agents is restricted. See Figure 6.

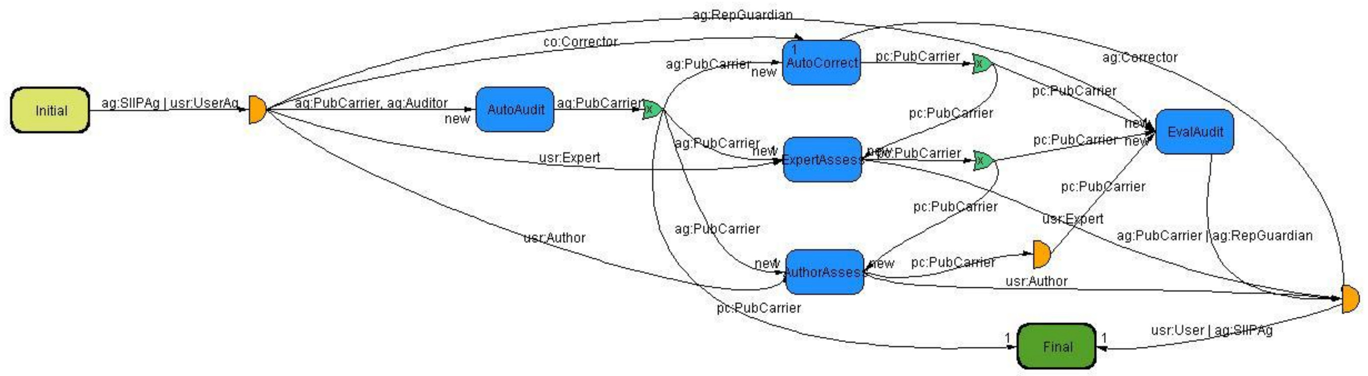


Figure 6. Auditing process codified as performative structure.

The **second approach** consisted on modeling the process like a single scene where any internal or external agent could enter and exit at different points. The scene has the entire auditing protocol. See Figure 7. The performative structure contained a single scene. On this approach experts and authors are considered as external entities and the communication with them is represented in the automata.

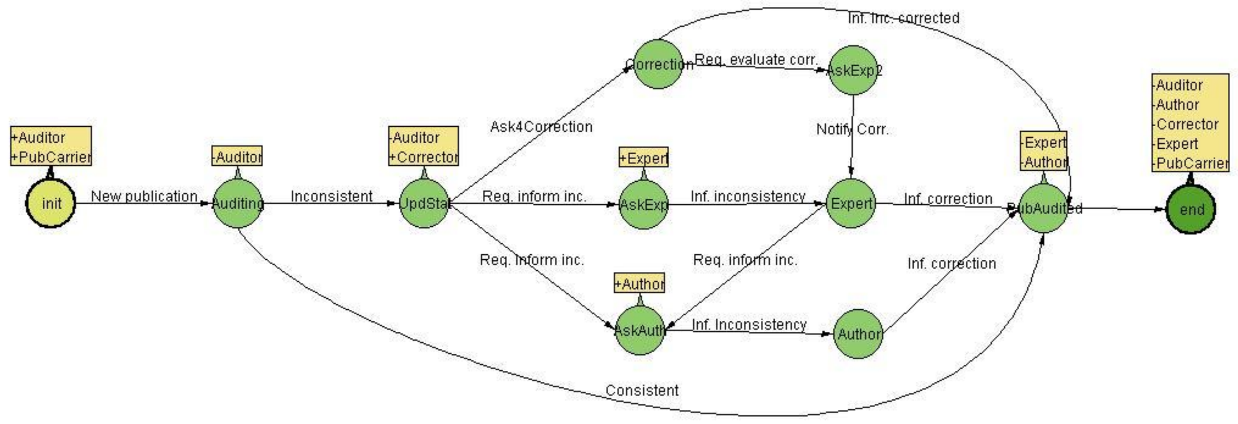


Figure 7. Auditing process codified in one scene (for one inconsistency).

In the **third approach**, Expert and Author agents encapsulate the communication with external users, which is the purpose for which were User agents were proposed in Electronic Institutions. See Figure 8. The participation of human agents is represented by User agents: Experts or Authors. An institutional message is sent to the human through a web interface (synchronously) or through an email (asynchronously), controlled by the user agent. The user agent is responsible for monitoring human response and passing it to the Institution through a message. Details on the implementation of these procedures are hidden for the Institution.

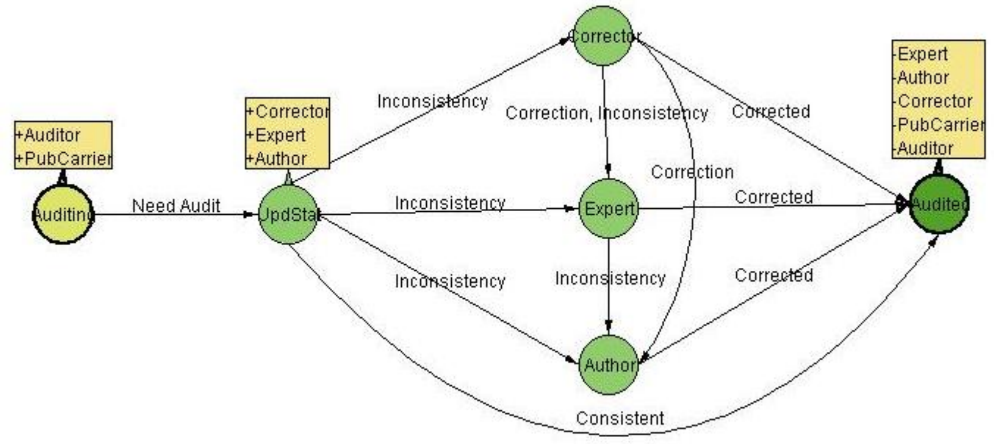


Figure 8. Simplified auditing codified in one scene (for one inconsistency).

Given that the auditing scene only checks/correct a single inconsistency, the performative structure must consider the flow for sending the request for auditing to every auditor agent. On this way, there is generated a scene per each publication-inconsistency. See Figure 9.

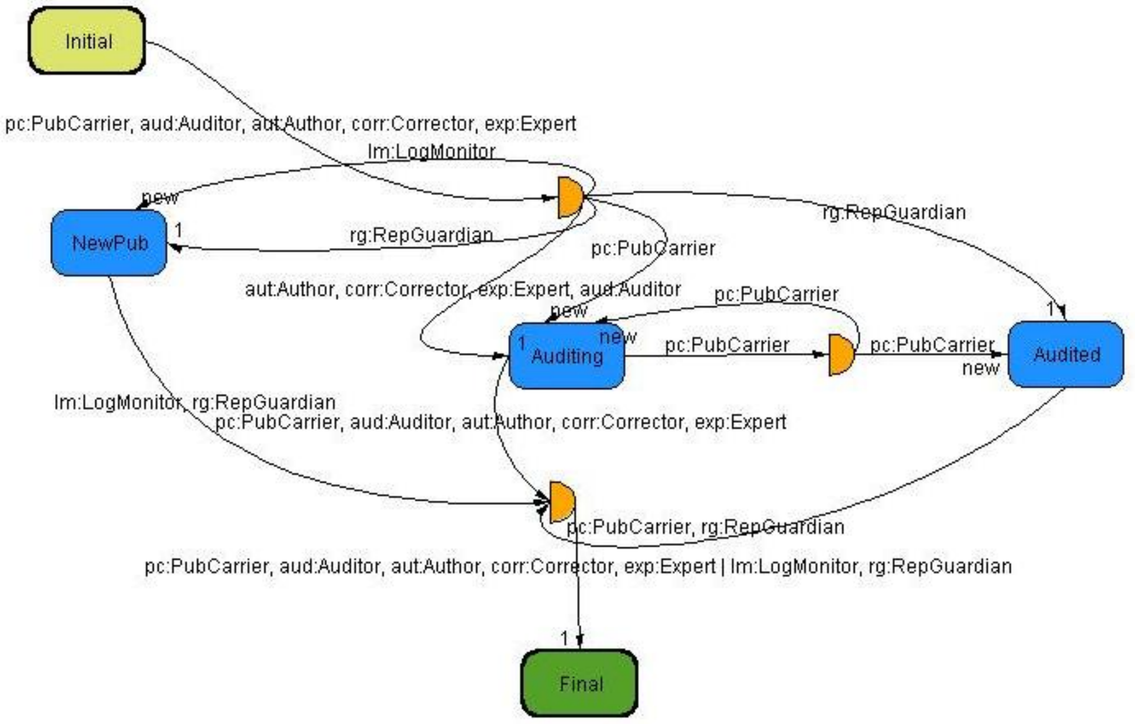

Initial
pc:PubCarrier, aud:Auditor, aut:Author, corr:Corrector, exp:Expert
lm:LogMonitor
new
1
NewPub
rg:RepGuardian
rg:RepGuardian
pc:PubCarrier
aut:Author, corr:Corrector, exp:Expert, aud:Auditor
pc:PubCarrier
new
1
Auditing
new
pc:PubCarrier
pc:PubCarrier
new
1
Audited
lm:LogMonitor, rg:RepGuardian
pc:PubCarrier, aud:Auditor, aut:Author, corr:Corrector, exp:Expert
pc:PubCarrier, rg:RepGuardian
pc:PubCarrier, aud:Auditor, aut:Author, corr:Corrector, exp:Expert | lm:LogMonitor, rg:RepGuardian
1
Final

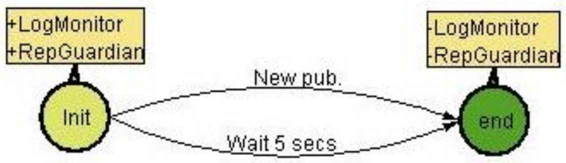
Figure 9. Performative Structure for auditing new publications in the repository.

It was added a scene where the LogMonitor agent detects new publications in the repository. The RepositoryGuardian agent needs to instantiate a PublicationCarrier agent for each new publication. The PublicationCarrier creates auditing scenes for each available Auditor agent and finally summarize the result of the auditing.

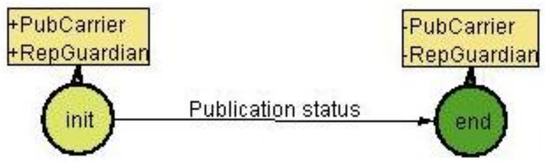
Even when the instantiation of agents is not represented in the EI, we can assume that the instantiated agent gets into the institution on the same way that other agents do, through the starting point in the performative structure. On the other way it should allow that an agent that didn't entered in the scene could get out of it, i.e. an agent is created on the scene.

Another instantiation of agents is made in the Auditing Protocol, where the Auditor agent can instantiate an Expert or Author agent, depending on what is more convenient. If some of these agents already exist, the agent should be informed by the institution that its presence is required in the scene.

Finally, the Audited scene allows that the information gathered by the PubCarrier be concentrated by the RepGuardian. Before reaching this point I thought of extracting the information from the scene object in order to structure a single case of the auditing process. But beyond the privacy violation on which I could incur, as Bruno explained me, the process should be concentrated in a single scene, which would limit the flexibility and growth of the process, as it will be shown later. Figure 10 shows the initial proposal for the NewPublication and Audited protocols.



(a)



(b)

Figure 10. Protocols for (a) detecting new publications and (b) reporting the auditing result.

The specification through the third approach was completed and validated with the tool provided by Islander. Products obtained up to this point were: a performative structure (main workflow), three scenes/protocols with valid illocutions, a dialogical framework (agents classes), and an ontology for the contents of the illocutions.

The validation tool for the EI provided by Islander was of great value. It allowed me to learn through examples, indicating when something was missing, invalid formats for illocutions contents, cuts in the flow of agents, etc. If the validation only had thrown the classic Valid or Invalid output, without a partial error message, the design process would had been tortuous.

Additionally, the Auction House example provided lot of practical examples.

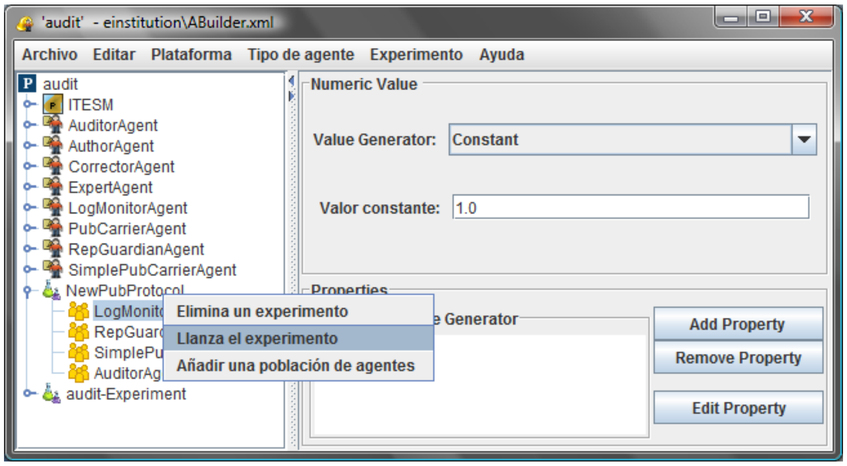
Agents Java Code Generation

Once we have a valid regulatory framework we can continue with the development of agents. The ABuilder tool provided a way of implementing them. ABuilder creates the Java code for event-driven agents that interact with the governor agent, as well as the ontological framework used on illocutions. ABuilder generates java classes for: the agent, the performative structure, and for each scene on which participates (according to the specification), and adds TODO labels indicating that the developer must incorporate logic at that point.

The resulting code was migrated to an Eclipse project facilitating the compilation and execution. Note: additional to the configuration instructions given in the web page, the Eclipse project must include the current source code in the Java Build Path (see project properties). The execution of Islander and Ameli can be configured in Eclipse using the class es.csic.iiia.eide.MainLauncher and passing as parameter islander or ameli, respectively, and indicating the XML specification configuration file. Both files are generated by ABuilder during the generation of the performance project.

Testing the Specification

After a rapid inspection of the generated code I ran the experiment through the ABuilder interface, see Figure 11. The population of agents can be configured and randomized through the design of the experiment. For example, once a variable is defined on the agent, a random numbers generator can be used for filling its value on each instantiated agent. Nevertheless, the generated agent class doesn't stores such values in properties automatically; this must be done by the developer through the AgentInstanceConfiguration object passed as parameter in the agent constructor.

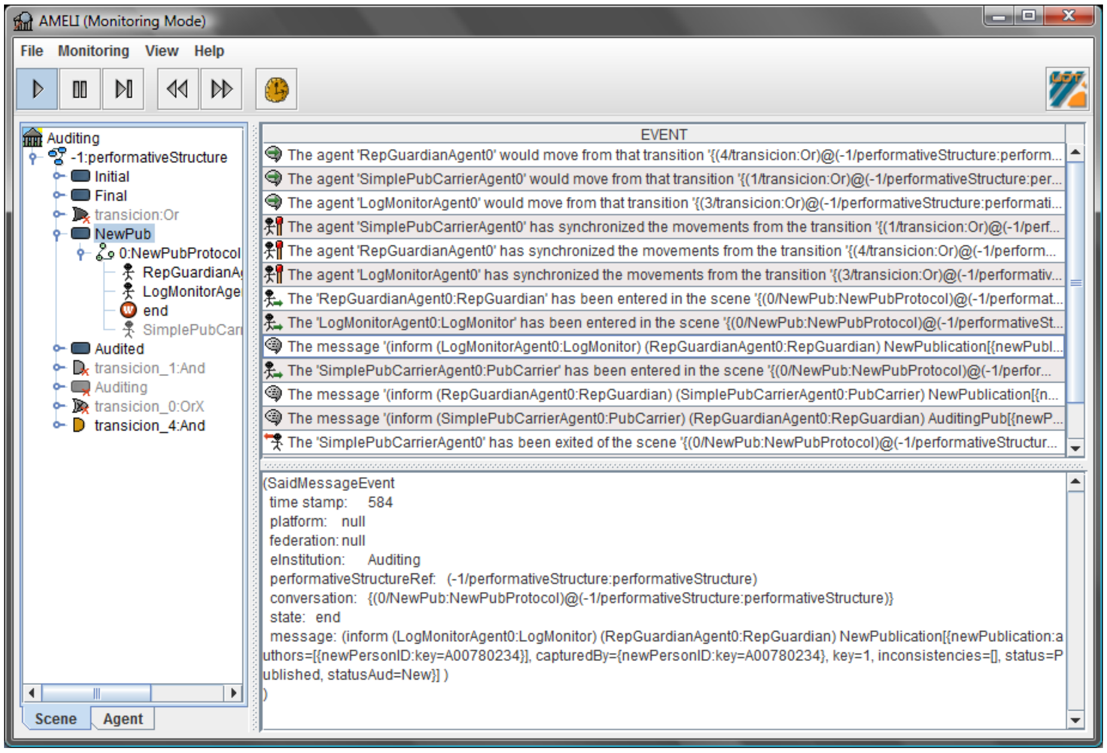


Value Generator: Constant

Valor constante: 1.0

Figure 11. Agents populations of an ABuilder experiment.

The experiment is launched in AMELI and its execution was analyzed from the scenes and agent perspective through the traces of the messages exchanged during the experiment's execution, see Figure 14. Iconography identifies the different types of events occurring on scenes. As soon as I started to see them I became familiar with them. Colors on agents' names and on performative structure elements denote if there is currently some activity on them.



```
The agent 'RepGuardianAgent0' would move from that transition '{(4/transicion:Or)@(-1/performativeStructure:perform...
The agent 'SimplePubCarrierAgent0' would move from that transition '{(1/transicion:Or)@(-1/performativeStructure:per...
The agent 'LogMonitorAgent0' would move from that transition '{(3/transicion:Or)@(-1/performativeStructure:performati...
The agent 'SimplePubCarrierAgent0' has synchronized the movements from the transition '{(1/transicion:Or)@(-1/perf...
The agent 'RepGuardianAgent0' has synchronized the movements from the transition '{(4/transicion:Or)@(-1/perform...
The agent 'LogMonitorAgent0' has synchronized the movements from the transition '{(3/transicion:Or)@(-1/performativ...
The 'RepGuardianAgent0:RepGuardian' has been entered in the scene '{(0/NewPub:NewPubProtocol)@(-1/performat...
The 'LogMonitorAgent0:LogMonitor' has been entered in the scene '{(0/NewPub:NewPubProtocol)@(-1/performativeSt...
The message '(inform (LogMonitorAgent0:LogMonitor) (RepGuardianAgent0:RepGuardian) NewPublication[{newPubl...
The 'SimplePubCarrierAgent0:PubCarrier' has been entered in the scene '{(0/NewPub:NewPubProtocol)@(-1/perfor...
The message '(inform (RepGuardianAgent0:RepGuardian) (SimplePubCarrierAgent0:PubCarrier) NewPublication[{n...
The message '(inform (SimplePubCarrierAgent0:PubCarrier) (RepGuardianAgent0:RepGuardian) AuditingPub[{newP...
The 'SimplePubCarrierAgent0' has been exited of the scene '{(0/NewPub:NewPubProtocol)@(-1/performativeStructur...
```

```
(SaidMessageEvent
 time stamp:    584
 platform:   null
 federation: null
 eInstitution:    Auditing
 performativeStructureRef:   (-1/performativeStructure:performativeStructure)
 conversation:   {(0/NewPub:NewPubProtocol)@(-1/performativeStructure:performativeStructure)}
 state:   end
 message: (inform (LogMonitorAgent0:LogMonitor) (RepGuardianAgent0:RepGuardian) NewPublication[{newPublication:a
uthors=[{newPersonID:key=A00780234}], capturedBy={newPersonID:key=A00780234}, key=1, inconsistencies=[], status=P
ublished, statusAud=New}] )
)
```

Figure 14. AMELI execution.

As result of the execution of the experiment I could observe how scenes were instantiated and where agents get stocked. Adjusting the action selection in the code generated allowed me to simulate the different paths an agent could follow. After some experiments I noticed that some arrangements should be done.

Synchronizing Specification and Implementation

Ideally, EI's specification should consider all the cases and code generation should be done only once. For this reason, an iterative specification-development process becomes cumbersome as long as code generation overrides agent's customizations. For example, some specification changes that require deep understanding of the EIDE framework for being done simultaneously are: adding new vocabulary in the ontology or adding arcs or nodes in the performative structure or protocols.

Further, with a better understanding of the generated classes, I learned which files to overwrite from the new specification. For example, the ontology class, the dialogical framework class, abstract classes representing the performative structure and scenes, as well as the scene classes for each agent. The definition of abstract classes by ABuilder allowed to apply changes along all the agent classes participating on the given performative structure or scene.

Second Modeling Phase

Knowing that a single agent can be simultaneously in different scenes motivated changes on the scenes: making some static (NewPub and Audited), and modifying them for allow that some agents would remain fixed on them (RepGuardian for instance). The new performative structure is shown in Figure 15, the new version of NewPubProtocol is shown in Figure 16, and the new version of AuditedPubProtocol is shown in Figure 17.

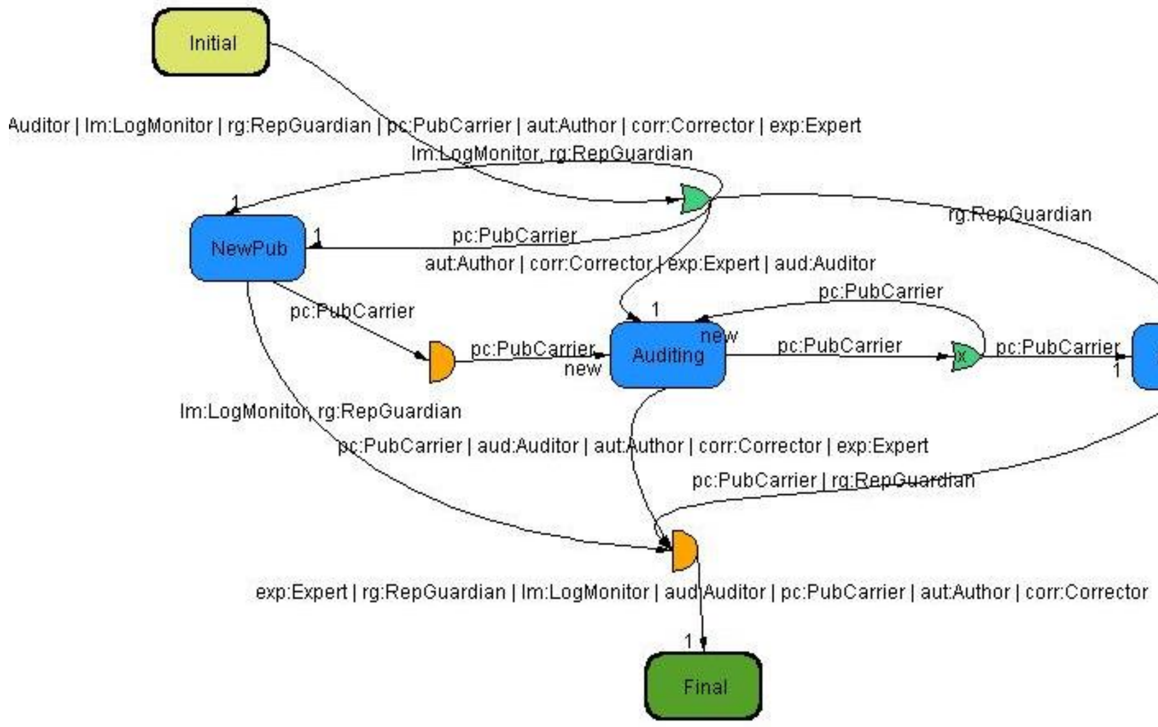

Initial
Auditor | lm:LogMonitor | rg:RepGuardian | pc:PubCarrier | aut:Author | corr:Corrector | exp:Expert
lm:LogMonitor, rg:RepGuardian
rg:RepGuardian
1
NewPub
1
pc:PubCarrier
aut:Author | corr:Corrector | exp:Expert | aud:Auditor
pc:PubCarrier
pc:PubCarrier
1
1
pc:PubCarrier
new
Auditing
new
pc:PubCarrier
pc:PubCarrier
1
Audited
lm:LogMonitor, rg:RepGuardian
pc:PubCarrier | aud:Auditor | aut:Author | corr:Corrector | exp:Expert
pc:PubCarrier | rg:RepGuardian
exp:Expert | rg:RepGuardian | lm:LogMonitor | aud:Auditor | pc:PubCarrier | aut:Author | corr:Corrector
1
Final

Figure 15. Performative Structure afte agent's implementation.

Another lesson learned was the correct use of transitions. In principle I used only ANDs, but through experiments noticed that transitions are not only used for synchronization, but for indicating which ways can/must follow the agent. For instance, in Figure 15, the upper transition was changed to OR to allow the RepGuardian enter to NewPub or Audited, but not necessarily both. On the same way, the XOR transition between Auditing and Audited forced the PubCarrier to move only towards one of the both scenes. Agent's decision is codified in the respective performative structure class.

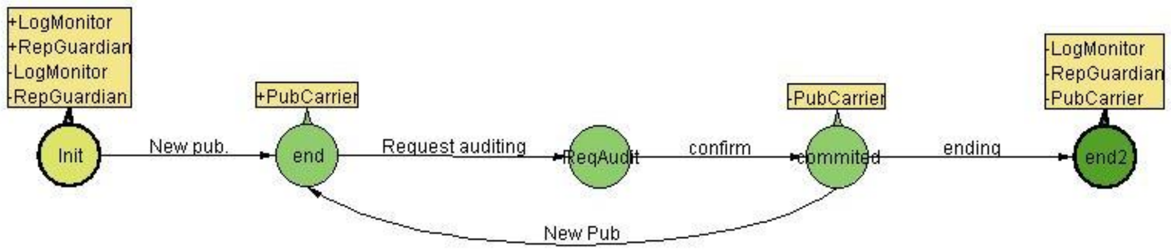


Figure 16. NewPubProtocol after agents' implementation.

The new version of the NewPubProtocol is static, allowing PubCarrier agents entering and exiting the scene meanwhile the RepGuardian and the LogMonitor agents remain on it. Besides, the protocol structure allows controlling the application flow restricting the valid illocutions. For example, even when the LogMonitor agent might be detecting new publications, it is only allowed to communicate one to the RepGuardian agent when a

PubCarrier agent has been dispatched. On this way we can control the information flow in terms of agents events like: an agent entered in the scene or message received.

This scene restrict to having a single RepGuardian agent. This constraint enables identifying easily an agent by its role. For instance, the PubCarrier agent can be sure that it is addressing the same RepGuardian all the time, on this scene.

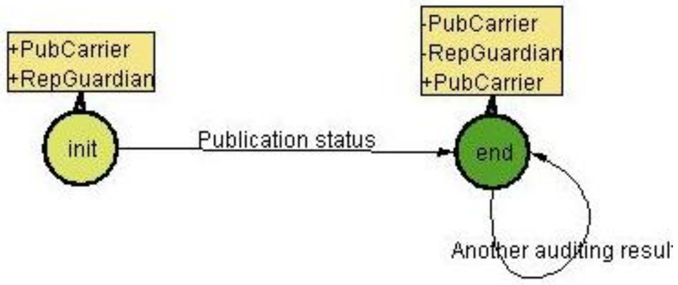


Figure 17. AuditedPubProtocol after agents' implementation.

The AuditedPubProtocol became static too. The cycle in the end node allows to the RepGuardian agent remain in the scene indefinitely. PubCarrier agents can enter and exit in that node. Checking the specification noticed an error in this scene but neglect it and generated the code. The code was generated successfully and ran the experiment. Later was told by Bruno that an ending node cannot have illocutions on it. Nevertheless, the decision for exiting the scene is controlled by the agent (with the method exit in the scene performance), so this validation didn't make too much sense to me.

Agents' implementation for LogMonitor and RepGuardian was completed first as long as they did participate in static scenes only. NewPubProtocol and AuditedProtocol were complete.

Now the auditing scene was revised. Given that the creator and "owner" of the auditing scene would be the PubCarrier, we should allow that Auditor agents would leave it once they had made their evaluation; see Figure 18.

Facing the question of how many publications a PubCarrier would manage simultaneously I prepared this agent creating subclasses of it. On this way it would be possible to compare the performance of the system for PubCarriers capable of auditing several publications at the same time. For now I only prepared the SimplePubCarrier subclass, which can only carry a single publication. Nevertheless, it were introduced general methods in the PubCarrierAgent class for allowing to implement new versions of this class. For example, methods like canAcceptPub() or getNextPubToAuditWith(Auditor) are overridden in SimplePubCarrier using fields proper of the agent implementation. The PubCarrierAgent can be declared as abstract in order to avoid its instantiation. The new class is defined in the ABuilder project for allowing the instantiation of the class SimplePubCarrier instead of PubCarrier.

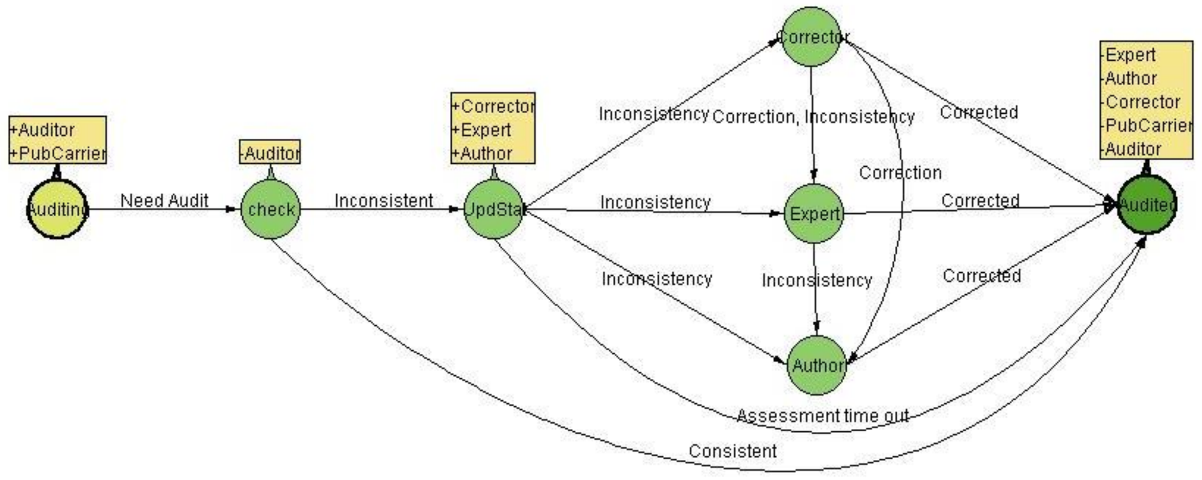


Figure 18. AuditingProtocol during agents' implementation.

At this point Pablo suggested separating the auditing scene in two: one for the auditing and another for the correction. This change would allow implementing different types of auditing protocols; some of them could involve multiple the participation of multiple agents. At the same time, cycles would allow the PubCarrier to pass through multiple auditing scenes, one for each type of known inconsistency. See Figure 19.

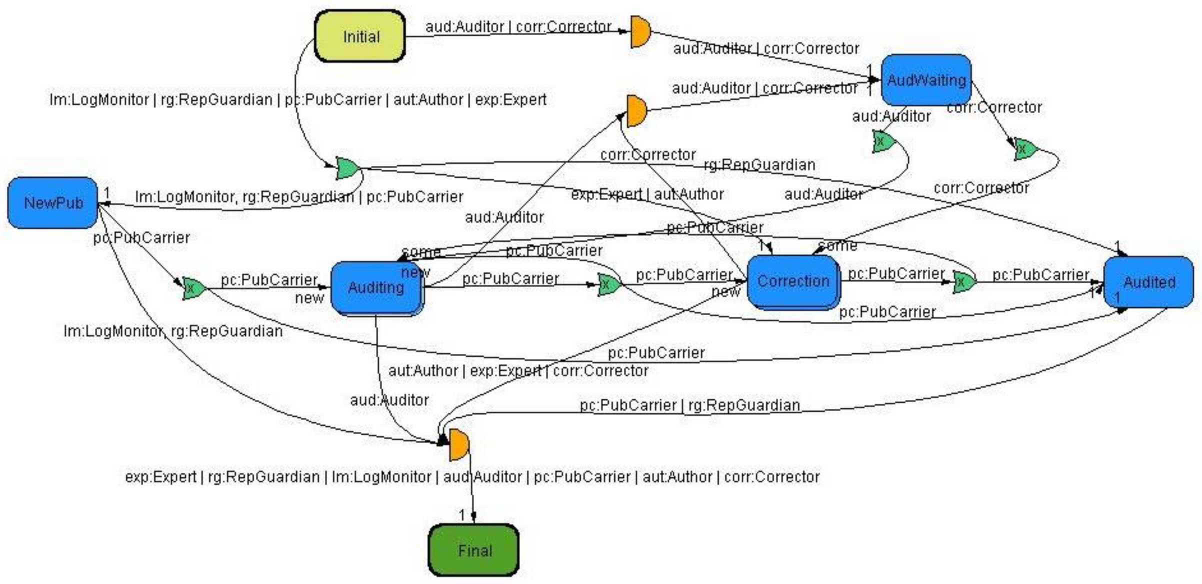

Initial
aud:Auditor | corr:Corrector
aud:Auditor | corr:Corrector
AudWaiting
aud:Auditor | corr:Corrector
lm:LogMonitor | rg:RepGuardian | pc:PubCarrier | aut:Author | exp:Expert
aud:Auditor
corr:Corrector
corr:Corrector
rg:RepGuardian
NewPub
lm:LogMonitor, rg:RepGuardian | pc:PubCarrier
exp:Expert | aut:Author
aud:Auditor
corr:Corrector
pc:PubCarrier
aud:Auditor
pc:PubCarrier
some
pc:PubCarrier
some
pc:PubCarrier
Auditing
pc:PubCarrier
pc:PubCarrier
Correction
pc:PubCarrier
pc:PubCarrier
Audited
new
new
new
pc:PubCarrier
lm:LogMonitor, rg:RepGuardian
pc:PubCarrier
aut:Author | exp:Expert | corr:Corrector
aud:Auditor
pc:PubCarrier | rg:RepGuardian
exp:Expert | rg:RepGuardian | lm:LogMonitor | aud:Auditor | pc:PubCarrier | aut:Author | corr:Corrector
Final

Figure 19. Main performative structure with simultaneous Auditing and Correction scenes.

This modification enabled concurrency during auditing and correction, which was included in the specification marking the Auditing and Correction scenes as lists. The incoming edges of auditors and correctors were changed from ONE to SOME. The institution controlled that auditors and correctors entered to only one scene at the time. Nevertheless, PubCarriers agents created all the scenes as soon as they were allowed. On this way, it can be multiple scenes of auditing and correction, meanwhile auditors and correctors agents can choose to which to enter or change from one to another. Another improvement was introducing the AudWaiting scene on which Auditor and Corrector agents can wait until finding some suitable Auditing or Correction scene.

This change impacted the implementation of agents by splitting the original Auditing protocol in two. The change affected all the agents participating in this scene (Auditor, Corrector, Author, Expert, PubCarrier). But the radical change was passing from a single inconsistency checking and correction to a process that allows correcting simultaneously several inconsistencies with the participation of agents at different stages, which optimize assignment of resources.

In the agent implementation this change required to store in the agent class information that was originally managed on the scene through messages. The PubCarrier's event-based plan now is divided in two parts motivating the implementation of methods in the agent class for continuing the auditing process. PubCarrier decisions for following some path or another uses these new functions; for instance, a method like getNextInconsistencyToCheck() would allow to decide if it needs to create another auditing scene or if it must terminate.

The invitation mechanism

Given that not all Auditor, Corrector, Author and Expert agents need to participate in all the scenes, we require a mechanism for "inviting" to certain agents to specific scenes. This mechanism is represented by the following functionality:

1. A Directory Facilitator (DF) for knowing which agents are present and their capabilities.
2. Instantiation of agents on demand, whenever there is none available for a specific need.
3. Requesting agents for certain scene according to a given agent description.
4. A protocol for inviting agents to join to some scene.

This mechanism was thought in a server-client schema, where the same agent that has control of the DF is responsible for processing invitations and instantiating new agents. Another option would be implementing all this functionality as a service that agents would access through the existing interfaces (see EInstitution class). The advantage of our approach is that it would allow a negotiation phase during the invitation, enabling a ContractNet protocol implementation for instance.

In the next sections is explained the implementation of this mechanism and the problems found during its implementation.

Designing the new functionality

The new functionality was modeled through two roles: CAIDAg for the server side, and SIIPAg for the client side. It was designed a second performative structure containing the new functionality, see Figure 20.

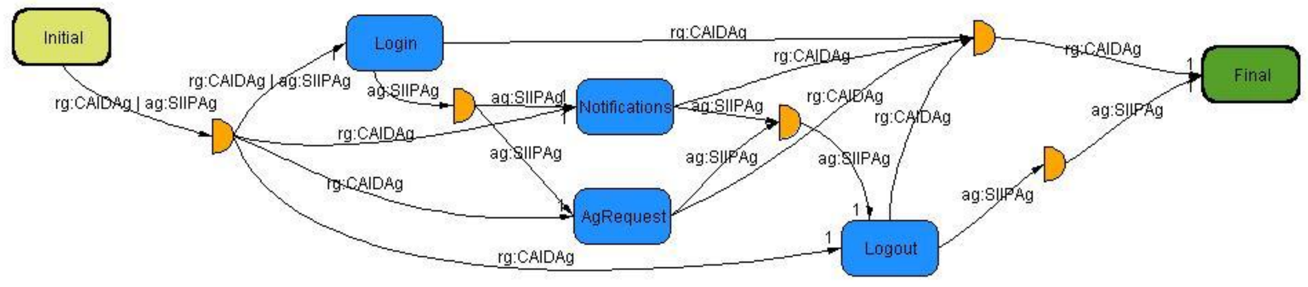


Figure 20. CAID services Performative Structure

In order to add this functionality to every agent, it was added an upper performative structure indicating that every agent simultaneously access both scenes: auditing and CAID; see Figure 21. Observe that the agent identifier indicates feasible changes on role. The OR transition indicates that a SIIPAg can adopt any of the other roles in the other side of the transition: Auditor, Expert, etc.

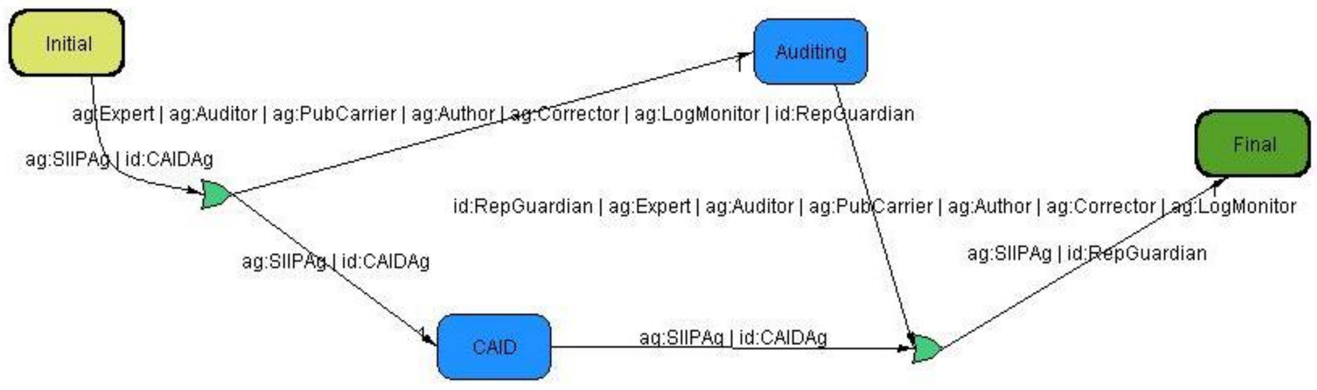

Initial
Auditing
CAID
Final
ag:Expert | ag:Auditor | ag:PubCarrier | ag:Author | ag:Corrector | ag:LogMonitor | id:RepGuardian
ag:SIIPAg | id:CAIDAg
id:RepGuardian | ag:Expert | ag:Auditor | ag:PubCarrier | ag:Author | ag:Corrector | ag:LogMonitor
ag:SIIPAg | id:RepGuardian
ag:SIIPAg | id:CAIDAg
ag:SIIPAg | id:CAIDAg

Figure 21. Main performative structure.

ABuilder's code generation provided an implementation of every agent. SIIPAg implemented the client version of the CAID functionality, the RepGuardian implemented both functionality as long as it participates explicitly in Auditing and CAID performative structures, and finally the rest of the agents were only implement with the functionality described in the Auditing PS.

This functionality is implemented through a new EInstitutionService called AgentInstantiator and four static scenes where every agent participates until its disposal, see Figure 20. These static scenes are:

- **LogInProtocol**: Every new agent informs to the CAIDAg of its entrance in the system, as well as its essential role and a list of actual accidents. See Figure 22. Additionally, the CAIDAg should notify the invitation that motivated its instantiation in order to validate that the agent is capable of achieving the task (this isn't included in the Figure 22).

- **LogOutProtocol**: The agent notifies the CAIDAg when is leaving the institution. The CAIDAg updates its list of available agents and releases slots for instantiating new agents. See Figure 23.
- **AgsRequestProtocol**: An agent that requires the presence of some agent on its current scene, before entering or once it has entered on it, requests to the CAIDAg for the agent through an agent description. The agent description may include a role, a name and a set of potential or actual properties. If there is no agent available with the given description and the CAIDAg have free slots then instantiate the required agents. Simultaneously, the CAIDAg invites the required agents through the invitation protocol. See Figure 24.
- **NotificationsProtocol**: Similarly to a chat room, this protocol allows the CAIDAg to invite other agents to participate in a given scene with a given role. The invited agent can accept or refuse the invitation. See Figure 25.

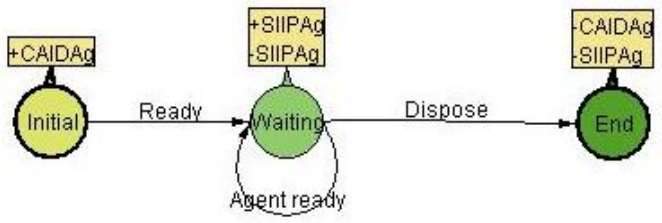


Figure 22. CAID Functionality: LogIn protocol.

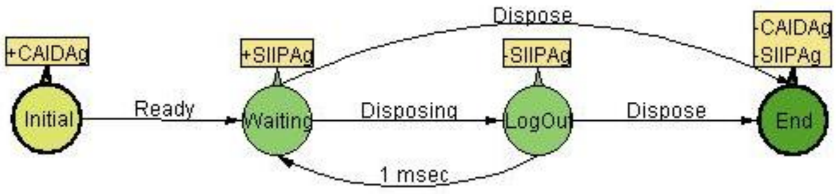


Figure 23. CAID Functionality: LogOut protocol.

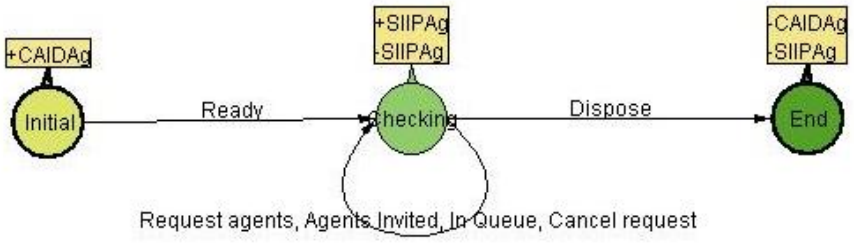


Figure 24. CAID Functionality: AgentsRequest protocol.

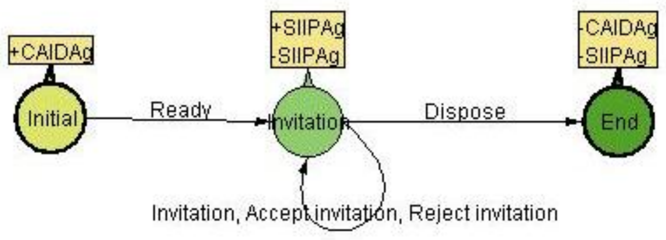


Figure 25. CAID Functionality: Notifications protocol.

As can be seen in the diagrams, protocols designed for CAID functionality contain a main node on which agents can perform certain illocutions. Even when these are not protocols properly, this was a way of implementing exchange of messages between all the agents in the system, for inviting them to a scene for instance. The single state protocol allows agents to issue any valid illocution as soon as they can. The work of organizing and give coherence to the protocol relies on the logic of a single agent representing the server side. Otherwise, agents would need to wait for saying a valid illocution when it was allowed to, and very complex or extended protocols would delay request from other agents.

Implementing the new functionality

In order to incorporate the client side of the CAID functionality in an agent, the SIIPAg was declared as superclass of every class, except the RepGuardian. The CAIDAg agent was declared as superclass of the RepGuardian. Both classes, SIIPAg and CAIDAg where declared abstract. Methods used for merging both performative structures giving a different treatment to every agent class where declared abstract too, forcing the implementation in subclasses. For instance, the original role of the agent is obtained through the getEssentialRole() method.

Initially, there were defined some parameters for controlling this functionality:

- The maximum number of agents that can be instantiated by role.
- The maximum number of invitations that can accept each agent type, or 0 if it doesn't accept invitations.

Additionally it was codified in methods a default behavior that can be overridden on each agent implementation. On this customization it can be included a condition for accepting an invitation. For example, the Auditor might not accept invitations if it is currently auditing some publication.

Additionally, it was implemented the mechanism for assuring that agents exiting of the Auditing performative structure would automatically exit of the main PS and from the CAID PS. The mechanism included a method for forcing the termination of all the agents in the system, in the last two performative structures.

An issue was that once that the agent split itself and enter in both sub performative structures, it was necessary to control manually that the agent wait until be logged in and participating in the notification and agent request protocols. Evidently it was necessary to integrate in other way both functionalities. Figure 26 shows the new main performative structure. On it, scenes originally defined in the CAID PS were organized around the auditing scene.

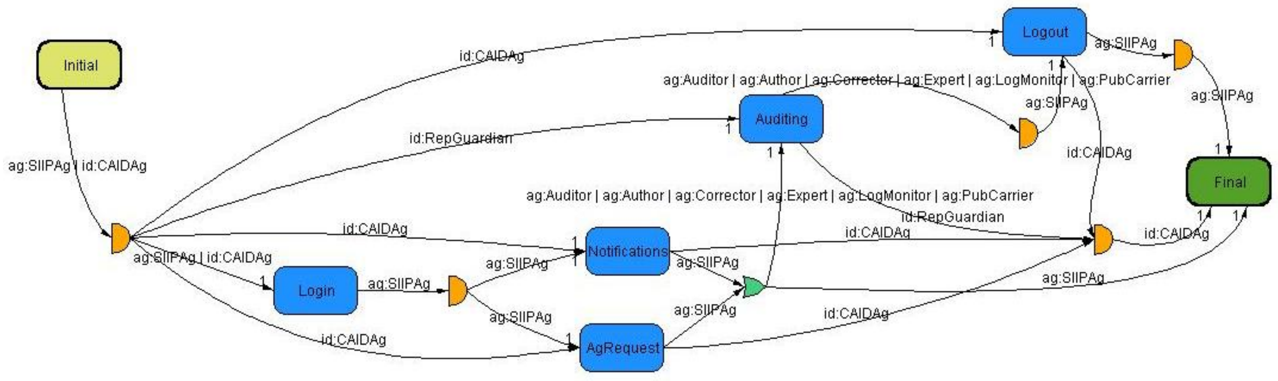


Figure 26. Integration of CAID functionality around original function (Auditing).

The new main performative structure generated by ABuilder was easily integrated to the current implementation. It was necessary to incorporate a StayAndGo access of the SIIPAg in the Notifications and AgRequest for allowing the simultaneous participation in Auditing and both previous protocols. See state Invitation in Figure 27. The StayAndGo method in the agent implementation produced a "splitting" of the agent.

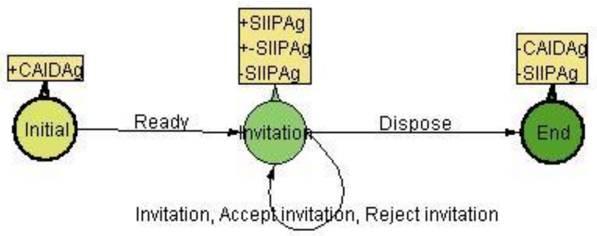


Figure 27. Stay and go for SIIPAg in the invitations protocol.

Another important issue was on synchronization. Given that agents participate simultaneously in several scenes the access to common objects produce synchronization problems. This kind of problem can be solved through the implementation of synchronization locks.

Discussion

As it was shown in our application, not all the processes occur on line and directing the participation of agents towards certain scenes simplify the design of the system. The mechanism of invitation, extended to ContractNet, would enable a good tradeoff between simplicity in the specification and optimal resource allocation. The proposed extension could be implemented as an institutional agent in order to assure the correct functioning of the institution.

Not all the scenes should follow a structure based on graphs. A simple chat room with valid illocutions would be useful to in some applications (see Figures 24 and 25). The conversation control would be done internally by the participants. The disadvantage of having a rigorous protocol on which each agent must wait to issue an illocution poses the necessity of having a queue of illocutions to say on every scene. This problem was tackled extending the ScenePerf class with two methods: addMessageToSend(content, agent), and sendPendingMessages(agent). See class `mx.mty.itesm.caid.ExtendedScenePerf`.

The agent architecture provided by ABuilder doesn't allow carrying out a plan that requires access to different scenes. The work of controlling the flow among scenes according to a plan is left to the agent developer. Information processed in one scene is lost in the transition to another scene and a plan inter-scenes is unfeasible with scene information. For example, once the PubCarrier agent sends to auditing a publication and is informed of an inconsistency, it can choose to enter in the correction scene. If it does, how does it know which was the publication and the inconsistency it was to correct? In order to overcome this limitation, information needs to be uploaded to the agent, as it was done through an API of public methods developed in agent classes. Entering and exiting from scenes would be part of the plan.

The institution acquires information through the post-conditions of illocutions and transitions, and validates these through this gathered information. In principle, I thought that properties declared in the agent specification were public, not only used for agent's initialization, as I was told later by Bruno. If properties declared in the agent specification were considered public, and access methods were provided in the agent implementation, it would be feasible to use *publicly internal agent's properties* for validating transitions. For instance, if an agent is doing some task and its status is Busy, it would be constrained some illocution in some other scene. In terms of privacy, the institution would be constraining the access of an agent to having free access to certain information. In real life, a foreign visitor is obligated to show his passport to enter a country.

ABuilder tool generates an agent template that considers changes in roles. For instance, the RepGuardian agent enters in the institution as CAIDAg and then adopts the RepGuardian role in Auditing scenes. The agent template contains code for the agent on each scene for each role it can play. The agent was codified with the name of CAIDAgAgent. Nevertheless, as I explained before, the original role was RepGuardian, not CAIDAg. My point is that the role doesn't identify the type of agent. In my opinion the agent class should be expressed explicitly and the type of roles an individual of an agent class should be associated explicitly too. Even though, our approach proposes that the publicly available definition of the agent would be enough for determining the kind of roles an agent can play, whenever scenes specify the attributes (including actions) an agent should have.

Conclusions

This experience allowed constructing a Multiagent System capable of supporting the auditing process presented. The autonomic aspect of instantiating additional agents was implemented in a first approach; it needs to be formalized in depth and complemented with ContractNet or other resource allocation algorithm. The motivation and scope of Electronic Institutions presented to me by Pablo Noriega, Marc Steva and Juan Antonio Rodríguez was extremely useful as long as showed me the facilities the approach proposes rather than the limitations that imposes to the agent developer, which is the first impression must people have.

Future Work

Finally I present some research directions that might be interesting on my particular project and for Electronic Institutions in general.

For Automic Information Auditing:

- Incorporate a distributed probabilistic model for agents' decision.
    - The model is built from a global perspective describing the entire process workflow.
    - On this model are included utility nodes that take values depending on the rest of the model; these nodes are called *finality nodes*, as long as they represent final causes of the system.
    - Agents exchange causal information in order to update their model or inform changes on its effectiveness performing some task.
    - The CAID Agent is used for collecting a global vision of the model and to optimize it through the addition or disposal of agents.
- Implementing a new type of inconsistency evaluation based on SNA for detecting homonymy associations errors.
    - Considering two graphs: one for homonymy strength and another weighted for joint authorship.
    - Given three authors A1, A2, and A3, such that homonym(A1, A2), authorship(A1, A3) is high and authorship(A2,A3) is low, would indicate that A2 was associated to a publication by mistake.
- Applications in Knowledge Management (KM). The EI specification could be used for designing, validating and updating manual for institutional roles in a real organization.
    - The list of scenes would indicate the services a person playing a given role can offer.
    - Valid illocutions, the minimal information it must be provided to the other part.
    - States description could contain the description of the task that must perform before issuing an illocution. An atomic or composite action. Maybe a flow diagram.
    - At a given state, it could be useful having links to other protocols that could be started in order to fulfill the current action. I know this is such a privacy invasion, but this is proposed for internal agents only.
- The response of authors and experts can be modeled in terms of Reputation.
    - An auditor can be more accurate on his diagnose than another, for certain inconsistency.
    - Professors that neglects system's request can be sanctioned. Would this sanction improve repository consistency?

For Electronic Institutions:

- Create a Protégé plugin for importing and exporting ontologies to the Islander format.
- Using Description Logic formalisms for specifying an Electronic Institution.
    - It would be necessary to have a mechanism for proving partially the consistency of the elements, instead of only saying if the specification is valid or not.
- Implementing a Directory Facilitator (DF) through a Description Logics system that loads the Institution ontology and enables the following operations:
    - Keep track of registered agents through an agent definition containing an essential role and its actual accidents (DF).
    - Calculating potential accidents from its essential role definition.
    - Express or calculate which institutional roles can play an agent with a given essential role.
    - Load enumerates and constants from the ontology.
    - Express agents' descriptions through SPARQL queries for identifying current agents.

    - Use this agent description for generating all the possible agent definitions that would satisfy the description.
- Using agent definition for determining the kind of roles an agent can play in the Institution.
    - Scenes should be annotated with minimal requisites about participants' capabilities.
    - The public definition of agents should include potential attributes, including actions.
- Implement a ContractNet protocol [Smith 1980] for negotiating *invitations* to agents when there is more than one candidate. Currently we only have direct contracts.
    - It would be used the original approach that hasn't a cost associated to the bid. Instead, the contractor assigns contracts according a plan that consider the information provided by the participants.
    - Information that could be exchanged on this format would be the causal effect (likelihood) of achieving certain goal for the given case. For instance, choosing the Auditor agent that can assess with the highest rate a given inconsistency type.

Acknowledgements

This stay was sponsored by CONACyT and Tecnologico de Monterrey. We'd like to thank to the IIIA for they support on this work; in particular to Marc Steva, Juan Antonio Rodríguez and Bruno Rosell for their explanations on Electronic Institutions theory and tools.

References

[Cantu et al, 2005] Francisco J. Cantú, Héctor G. Ceballos, Silvia P. Mora, Miguel A. Escoffié. A Knowledge-based information system for managing research programs and value creation in a university environment. Americas Conference on Information Systems - AMCIS. Internacional. USA. pp: 781-791. August 2005.

[Sierra and Noriega, 1997] Carles Sierra and Pablo Noriega. A formal framework for accountable agent interactions. In *Fifth Bar-Ilan Sympossium on Foundations of Artificial Intelligence*, pages 23-24, Ramat-Gan, Israel, June 1997.

[Esteva et al, 2002] Marc Esteva, David de la Cruz, and Carles Sierra. ISLANDER: an electronic institutions editor. In *First International Conference on Autonomous Agents and Multiagent systems*, pages 1045-1052, Bologna, July 2002. ACM Press.

[Smith 1980] Reid G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. IEEE Transactions on Computers , 1104 Vol. C-29, No. 12, December 1980.