Practical Development of Hybrid Intelligent Agent Systems with SoSAA

Mauro Dragone[1], Rem W. Collier[2], David Lillis[2], and Gregory M. P. O'Hare[1]

[1] CLARITY: Centre for Sensor Web Technologies
School of Computer Science and Informatics
University College Dublin
{mauro.dragone, gregory.ohare}@ucd.ie*

[2] School of Computer Science and Informatics
University College Dublin
{rem.collier, david.lillis}@ucd.ie

Abstract. The development of intelligent Multi Agent Systems (MAS) is a non-trivial task. While much past research has focused on high-level activities such as co-ordination and negotiation, the development of tools and strategies to address the lower-level concerns of such systems is a more recent focus. SoSAA (Socially Situated Agent Architecture) is a strategy for the integration of high-level MASs on one hand with component-based systems on the other. Under the SoSAA strategy, a component-based system is used to provide the lower-level implementation of agent tasks and capabilities, allowing for the agent layer to concentrate on high-level intelligent co-ordination and organisation. This paper provides a practical perspective on how SoSAA can be used in the development of intelligent MASs, illustrating this by demonstrating how it can be used to manage backchannel transport services.

1 Introduction

Multi Agent Systems (MAS) are often advocated as a method of leveraging new and existing Artifical Intelligence techniques in order to build large-scale intelligent software sytems. In this type of system, autonomous software entities are tasked with reasoning about themselves and their environment in order to achieve individual or system-wide objectives and goals.

To date, a large body of research on MASs has been carried out on developing solutions to such problems as agent co-ordination, negotiation and reasoning, along with the development of standards governing agent communication [1]. However, less attention has been paid to the practical implementation of such systems. Agents have tended to be developed purely from an agent standpoint, with little regard to the underlying apparatus of the system.

In recent years, new research has emerged that deals with the lower-level aspects of intelligent systems in more detail. The CArtAgO framework makes use

* This work is supported by Science Foundation Ireland under grant 07/CE/I1147

of the Agents and Artifacts meta-model in the creation of *artifacts*: resources and tools to be utilised by agents in the satisfaction of their objectives [2]. Other work has focused on the environment within which agents are situated [3], arguing that the environment is an integral part of the MAS. The creation of an exploitable design abstraction of the environment is considered a key step in the design and implementation of a MAS. Both of these approaches emphasise the separation of concerns between the intelligent intentional layer on one hand, and the low-level actions on the other.

This trend is continued by the introduction of the Socially Situated Agent Architecture (SoSAA) framework. SoSAA is an open source framework that combines the concepts of Agent Oriented Software Engineering (AOSE) and Component-Based Software Engineering (CBSE) in the development of MASs[3]. There is a clear separation of concerns between the intelligent functionality of the agents, developed using AOSE concepts, and their lower-level actions, which are encapsulated by components. The background, motivations and underlying low-level functionality of SoSAA have been presented in previous papers [4, 5].

In contrast to this prior work, the key contribution of this paper is to demonstrate the practical usage of the framework to integrate a component-based system and intelligent agents to develop a Transport Manager that facilitates communication between low-level components. A brief overview of the framework is provided in Section 2. Section 2.1 outlines a number of improvements that were necessary so as to integrate the SoSAA approach into the underlying Agent Factory framework [6]. This integration is implemented via the SoSAA Adaptor, which is discussed in Section 2.2.

The implementation of the Transport Manager, which implements a hybrid backchannel communication strategy, is presented in Section 3. Finally, our conclusions and ideas for future work are outlined in Section 4.

2 SoSAA

Popularised by their use in robotics (e.g. in [7]), hybrid control architectures are layered architectures combining low-level behaviour-based systems with intelligent high-level, deliberative reasoning apparatus. The solution implemented in the SoSAA framework is to apply such a hybrid integration strategy to the infrastructure of a MAS, as illustrated in Fig. 1. SoSAA combines a low-level component-based infrastructure framework with a MAS-based high-level infrastructure framework. This section provides only a brief overview to the SoSAA framework. A more complete complete discussion can be found in [4].

The low-level framework allows for the development of functional components that encapsulate simple system behaviours and facilitate interaction with the agents' environment. These components are designed so as to be assembled into a system architecture. A run-time environment is provided to the high-level

[3] SoSAA may be downloaded from `http://www.sourceforge.net/projects/agentfactory`

framework, which then contributes its multi-agent organisation, interaction using Agent Communication Languages (ACL), and goal-oriented reasoning capabilities to intelligently perform this system assembly. Agents may also alter the system architecure and/or configuration to reflect changing goals and environmental circumstances. By interacting with the component layer, intelligent agents can access the system's resources, coordinate the components' activities and resolve conflicts. While the high level can be programmed according to different cognitive models, domain and application-specific issues can be taken into account in the engineering of the underlying functional components.

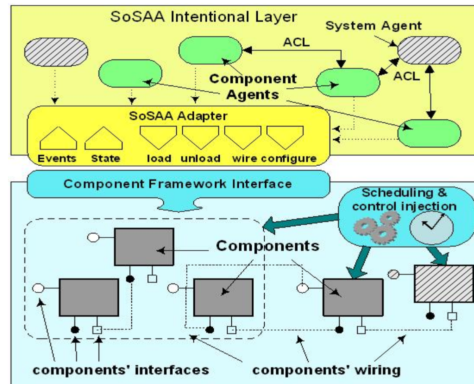

Fig. 1. SoSAA Hybrid Integration Strategy

The SoSAA high-level framework provides meta-level perceptors and actuators that collectively define an interface (delivered as a SoSAA Adaptor Service, discussed in Section 2.2) that can be used to sense and act upon elements and mechanisms of the low-level framework by loading, unloading, configuring and binding components.

Fig. 1 shows the multi agent organisation in a typical SoSAA node. The low-level framework provides the interface for operating both at the application and the infrastructure level. Depending on their interests, SoSAA component agents can be categorised as either application or infrastructure agents. Such a clear separation is fundamental for promoting not only the efficiency and the portability of the resulting systems, but also for separating the different concerns at design time to facilitate the use of a modular development process.

The current implementation of SoSAA is based on two open-source toolkits: the Agent Factory (AF) multi-agent toolkit, which is described briefly in Section 2.1, and the Java Modular Component Framework (JMCF) [4]. JMCF comes with a package of built-in component types and base-class implementations. Further details on JMCF can be found in [5].

[4] Agent Factory and JMCF can be downloaded from `http://www.sourceforge.net/projects/agentfactory`

2.1 Improvements to Agent Factory

Agent Factory is a modular, extensible, open source framework for the development of Multi Agent Systems [6]. The key components of this framework are a Run-Time Environment (RTE) consisting of a FIPA-compliant agent platform along with a number of agent system architectures and a set of development kits that contain implementations of agent interpreters and architectures.

One of the more important features offered by the Agent Factory RTE is the support of platform services. These are shared services that are offered to all agents residing on a particular agent platform. Examples include communication services, such as services that enable agents to exchange FIPA-compliant messages using HTTP, UDP or local message passing. In the context of the work presented here, the SoSAA Adaptor is implemented as a platform service that can be accessed by all agents on a platform. Support is also provided for the Agent Factory Agent Programming Language (AFAPL2) [6], an agent oriented programming language that has been successfully applied in a number of significant problem domains [8]. AFAPL2 employs a commitment-based mental state, whose core components are *beliefs*, *plans* and *commitments*. Beliefs represent an agent's view of the world, according to information gathered by its *perceptors*. Plans combine primitive actions (implemented by *actuators* or *effectors*) into complex activities that may be carried out by the agent. This is done by way of certain plan operators that are made available to agent developers. Commitments represent the activites that an agent has resolved to perform, according to its own reasoning mechanism.

A recent addition to AFAPL2 has been the introduction of goal-based reasoning. This implementation is based on the goal mechanism found in the Procedural Reasoning System [9]. Specifically, this involved the introduction of two new operators specifically designed for goals: ADOPT and MAINTAIN. Whenever an agent is required to satisfy a goal, firstly the postconditions of each of the agent's available activities (simple actions and more complex plans) are examined. Those activities whose postconditions would result in the goal being achieved are put into an option list. Once this option list has been created, the agent then examines the preconditions of each of the candidate activities and chooses the first whose precondition is satisfied by the state of the world as it is currently perceived. If an activity fails without achieving the specified goal, the agent will select another of the candidate activities. If an agent is required to ADOPT a goal, once the goal is achieved, it is dropped. In contrast, when attempting to MAINTAIN a goal, an agent will attempt to re-adopt the goal every time it becomes unsatisfied.

2.2 SoSAA Adaptor

The SoSAA Adaptor bridges the low-level component framework and the higher-level agent programming language. In the context of Agent Factory, support for this is implemented through a combination of a platform service, an agent module, a set of actuators and perceptors and a partial agent program that links

```
PERCEPTOR sosaaEventMonitor { ... }
LOAD_MODULE sosaa sosaa.module.ComponentStore;

ACTION create(?id, ?type) { ... }
ACTION remove(?id) { ... }
ACTION bind(?id1, ?iface1, ?id2, ?iface2) { ... }
ACTION configure(?id, ?param, ?value) { ... }
ACTION de/activate(?id) { ... }
ACTION focus(?id) { ... }
ACTION lookup(?id) { ... }
```

Fig. 2. SoSAA Adaptor Code

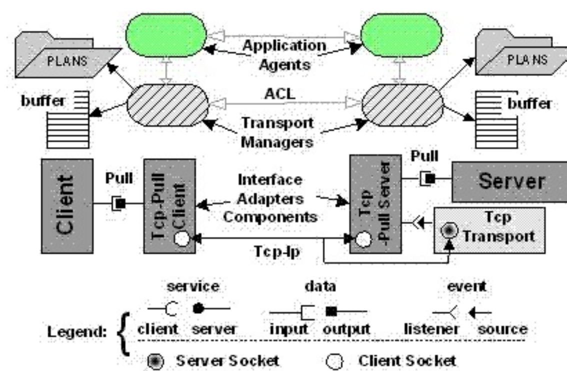

Fig. 3. SoSAA Backchannel Management

together all the pieces and provides a basis for developing SoSAA agents. Specifically, the platform service encapsulates the underlying component framework and provides an interface through which that framework may be manipulated, including the loading/unloading, activation/deactivation, binding, inspection, monitoring, and configuration of components.

Access to these operations is supported through the provision of a set of actuator units. Fig. 2 illustrates their declaration as part of a partial AFAPL2 agent program that can be reused as a basis for creating SoSAA agents. As can be seen in this figure, this partial agent program also makes use of an agent module. Agent modules are provided by AFAPL2 to support the creation of resources that are private to a given agent. In this case, the module provides a mechanism for the agent to keep track of the components that it is interested in and also a way of accessing the events and properties that are generated by those components. To achieve this, the sosaaEventMonitor perceptor has been created. This perceptor converts events and properties into beliefs that can be used at the agent level.

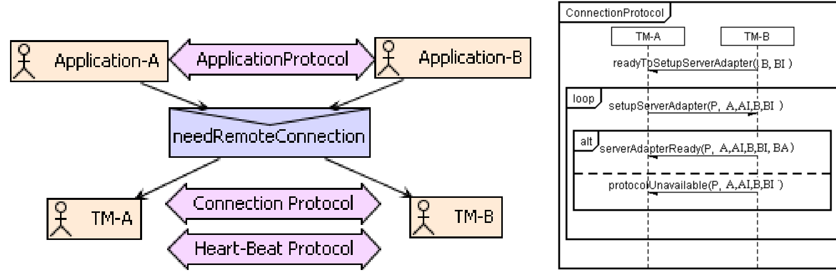

Fig. 4. (a) Interaction Diagram (left), and (b) AUML TransportManager's setupRemoteConnection protocol (Legend: P: Communication Protocol;A: Name of component A; AI: Name of interface for component A; B: Name of component B; BI: Name of interface for component B; BA: Name of network adapter for component B) (right)

3 Example: Transport Manager

To showcase how SoSAA can be used to construct intelligent software, this section focuses on the design and implementation of a hybrid backchannel management infrastructure service [4]. This service provides support for the transmission of diverse types of data between internal systems nodes using heterogeneous transport mechanisms, such as raw TCP-IP, RMI, JMS, and CORBA.

The idea of intelligent backchannel management using agents is not new, and was previously proposed for the RETSINA architecture [10]. Backchannels are designed to allow the flow of low-level data between agents or components. The motivation behind this type of communication is that it does not affect the decision-making of agents and as such, the use of expensive ACL communication is undesirable. In SoSAA, backchannels allow components to pass data between one another. In certain situations (e.g. the failure of a line of communication), components may raise events to inform agents of the circumstances. Thus, since the processing of backchannel data does not require any deliberation on the part of an agent, it can be separated from the intentional layer of the agent.

Using SoSAA, we have designed and implemented a backchannel management service that consists of an infrastructure agent, known as the *Transport Manager (TM)* and a set of *adaptor components* that implement the data transfer functionality for various transport mechanisms. As illustrated in Fig. 3, a TM is deployed on each node (agent platform) of the distributed system. Application agents, upon agreeing to make use of a backchannel, contact their local TM and request that two components (on different nodes) be wired together using a backchannel. Details of which transport mechanism to use and the setup and configuration of the associated adaptor components that implement that backchannel is delegated to the TMs.

Our backchannel management service supports two types of backchannel: (1) *push* backchannels, wiring remote components where the component that generates the data controls when the next unit of data is transmitted; and (2)

pull backchannels, wiring components where the component that receives the data controls when the next unit of data is transmitted. Fig. 3 shows a pull type of backchannel that has been set up based on a raw TCP-IP transport mechanism.

Fig. 4(a) provides a high-level view of the four protocols that are required to implement the service. Here, we ignore the application-level protocol as it is not part of the backchannel management service, but rather represents the application-level interaction that results in the agreement to set up of a backchannel.

Once this decision is made, the two application agents contact their local TM using the *setupRemoteConnection* message. This begins the connection protocol shown in Fig. 4(b). In making the initial request, the application agents include the information necessary to set up the backchannel, namely the name of the local and remote components; the interfaces that are to be bound together; and the name of the platform on which the remote component resides so that the TM can locate it. As can be seen in the partial AFAPL2 code outlined in Fig. 5 the TM responds to this request by looking up local component. The *lookup(...)* action is a generic action that is part of the SoSAA adaptor described in Section 2.2. Once invoked, it generates beliefs indicating whether or not the given component implements a client or a server interface. Based on this, the agent adopts a maintenance goal to set up a backchannel. The parameters associated with this goal include the Java interface (?jI) of the component; the local platform name (?lP), component name (?lC), and interface name (?lI); and the remote platform name (?rP), component name (?rC) and interface name (?rI); the adaptor name (?a); and the transport protocol (?p) (for example TCP, RMI, or JMS). The latter two parameters are not set when the maintenance goal is adopted (i.e. they are unbound variables). This allows the goal to be satisfied for any given protocol and its associated adaptor.

Ultimately, the specific belief that satisfies the goal is adopted once the relevant adaptor has been set up, as is shown in the setupClientConnection(..) plan in Fig. 6. This plan implements the client side of the interaction protocol presented in Fig. 4(b). Here, TM-A is the *Transport Manager* responsible for the client side while TM-B is responsible for the server side. The plan forces TM-A to

```
BELIEF(platformName(?lP)) &
BELIEF(message(request,?s,setupRemoteConnection(?lC,?lI,?rC,?rI,?rP))) =>
COMMIT(?self,?now,BELIEF(true), SEQ(achieve_goal(GOAL(platform(?rP, ?a))), lookup(?lC),
      OR(DO_WHEN(BELIEF(clientInterface(?lC, ?lI, ?type, ?jI, ?m, ?ib, ?mb, ?b)),
          MAINTAIN(GOAL(clientConnection(?jI, ?lP, ?lC, ?lI, ?rP, ?rC, ?rI, ?a, ?p)))),
        DO_WHEN(BELIEF(serverInterface(?lC, ?lI, ?t, ?jI)),
           MAINTAIN(GOAL(serverConnection(?jI, ?lP, ?lC, ?lI, ?rP, ?rC, ?rI, ?a, ?p)))))));
```

Fig. 5. Partial *Transport Manager* AFAPL2 code: Handling Backchannel Connection Requests

```
PLAN setupTcpClientConnection(?jI, ?lP, ?lC, ?lI, ?rP, ?rC, ?rI) {
  PRECONDITION BELIEF(platform(?rP, ?a)) & BELIEF(transport(?lP,TCP)) &
      BELIEF(transport(?rP,TCP)) & !BELIEF(transportFailure(?rP,TCP));
  POSTCONDITION BELIEF(clientConnection(?jI, ?lP, ?lC, ?lI, ?rP, ?rC, ?rI, ?a, ?p));
  BODY setupClientConnection(TCP, ?jI, ?lP, ?lC, ?lI, ?rP, ?rC, ?rI);
}

PLAN setupClientConnection(?p, ?jI, ?lP, ?lC, ?lI, ?rP, ?rC, ?rI) {
  BODY
    DO_WHEN(BELIEF(message(inform, ?agentID, readyToSetupServerAdapter(?lC, ?lI))),
      PAR(request(?agentID, setupServerAdapter(?p,?rC, ?rI)),
        DO_WHEN(BELIEF(message(inform,?agentID,serverAdapterReady(?p,?rC,?rI,?nRA,?rip))),
          PAR(createPullClientName(?lC, ?nRA),
            DO_WHEN(BELIEF(uniqueName(?cAN, ?self)),
              FOREACH(BELIEF(clientTransportAdapter(?p, ?jI, ?aC, ?i)),
                SEQ(create(?cAN, ?aC), bind(?cAN, ?i, ?lC, ?lI), focus(?cAN),
                  configure(?cAN, list(SERVER(?rip), SERVER_CONNECTION(?nRA))),
                  ADOPT(ALWAYS(BELIEF(clientConnection(?jI,?lP,?lC,?lI,?rP,?rC,?rI,?cAN,?p)))),
                  activate(?lC))))))));
}
```

Fig. 6. Partial *Transport Manager* AFAPL2 code: Client-Side Adaptor Creation

wait for a readyToSetupServerAdaptor(...) inform message from its counterpart. Upon receipt of this message, TM-A requests the setup of the actual server-side adaptor and waits to be given connection details. When the second message is received, TM-A generates a unique name for the client-adaptor which it then uses to create to adaptor. The adaptor is then bound to the local component; configured based on the information given; and monitored, via the focus action for events, such as transport failures. Finally, the plan activates the local component, causing the backchannel connection to be established. These final steps are achieved through the use of SoSAA adaptor (see Section 2.2) actions. The decision as to which adaptor should be created is based on knowledge that is stored in a set of clientTransportAdapter(...) beliefs, which map a given protocol (e.g. TCP) and Java Interface (e.g. a pull client) to a component type (i.e. a Java component implementation) and an interface. A similar plan exists for the server-side of the connection protocol.

Finally, the intelligent selection of which transport protocol to use is achieved through a set of custom plans that deal with each potential transport protocol. For example, Fig. 6 shows the setupTcpClientConnection(...) plan, which simply calls the setupClientConnection(...) plan with the protocol parameter (?p) bound to TCP. In AFAPL2, all plans whose post-condition matches the given goal are selected as options, and the first plan whose pre-condition is satisfied is chosen from the set of options. Here, the precondition states that: (1) both the local and remote platforms should have the transport protocol, and (2) that the transport protocol should not have previously failed. Failure of a given transport mechanism is detected through the monitoring of the adaptor, and the raising of error events. This results in the activation of a plan that: (1) records the

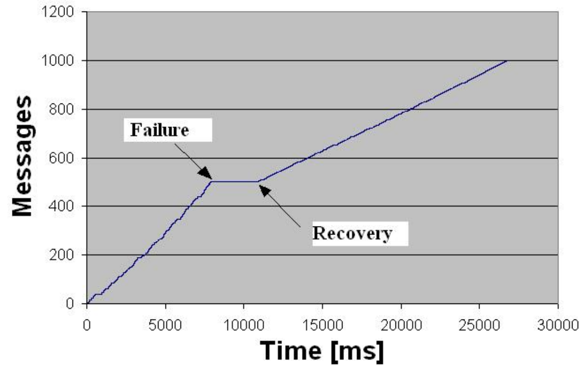

Fig. 7. SoSAA Backchannel Management

failure of the transport mechanism; and (2) causes the original goal to become unsatisfied, kicking off a new attempt by the agent to satisfy the goal.

Fig. 7 illustrates the results of a simple benchmarking test in which we measured the reaction time required by the TransportManager agent to switch between a TCP-IP connection to one based on JMS once we injected a failure in TCP-IP network adapter. This simple experiment is based on the core workflow that underpins the HOTAIR Information Retrieval test-bed [11]. The figure shows the rate of fixed-length messages exchanged between two components deployed on the same machine over time. Over 10 runs, the system was able to recover from failure with an average of 312ms.

4 Conclusions and Future Work

In this paper, we have presented an overview of ongoing work aimed at leveraging the benefits and strengths of both AOSE and CBSE in a way that promotes the development of high-quality applications. We have described both the rational behind our approach and its current incarnation in the SoSAA software framework, based on a combination of the Agent Factory and the JMCF component frameworks. In particular, we have attempted to demonstrate how SoSAA can be used to effectively construct intelligent software. By way of illustration, we have presented the design and implementation of an intelligent backchannel management service that has been developed using SoSAA, and showed how this service can provide a separation of concerns between infrastructure and application-level aspects. The service has been already used in a number of practical systems, including an agent-based information retrieval test-bed [11] and in a multi-agent robotics scenario [12]

Future work will investigate how an agent's plan-selection, plan-abortion and meta-level reasoning can be employed to implement an intelligent transport selection strategy addressing Quality of Service optimisation and system adaptation in response to changing environmental conditions. The first step will be to cache

the transport manager usage statistics concerning the adapter components, such as the total number of interfaces established toward each collaborating node, their latest throughput, and the last time they were successfully (or unsuccessfully) used. The HOTAIR test-bed [11] will continue to be used to demonstrate the effectiveness of our approach to a large scale, dynamic system.

References

1. Wooldridge, M.: Introduction to multiagent systems. John Wiley & Sons, Inc. New York, NY, USA (2001)
2. Ricci, A., Piunt, M., Acay, L.D., Bordini, R.H., Hubner, J.F., Destani, M.: Integrating Heterogeneous Agent Programming Platforms within Artifact-Based Environments. In: International Joint Conference on Agents and Multi Agent Systems (AAMAS08), Estoril, Portugal (2008)
3. Weyns, D., Omicini, A., Odell, J.: Environment as a first class abstraction in multiagent systems. Autonomous agents and multi-agent systems **14**(1) (2007) 5–30
4. Lillis, D., Collier, R.W., Dragone, M., O'Hare, G.M.P.: An Agent-Based Approach to Component Management. In: Proceedings of the 8th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-09), Budapest, Hungary (May 2009)
5. Dragone, M., Lillis, D., Collier, R.W., O'Hare, G.M.P.: SoSAA: A Framework for Integrating Agents & Components. In: Proceedings of the 24th Annual Symposium on Applied Computing (ACM SAC 2009), Special Track on Agent-Oriented Programming, Systems, Languages, and Applications, Honolulu, Hawaii, USA (March 2009)
6. Collier, R., O'Hare, G., Lowen, T., Rooney, C.: Beyond Prototyping in the Factory of Agents. Multi-Agent Systems and Application III: 3rd International Central and Eastern European Conference on Multi-Agent Systems, Ceemas 2003, Prague, Czech Republic, June 16-18, 2003: Proceedings (2003)
7. Gat, E.: Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In: Proceedings of the National Conference on Artificial Intelligence, JOHN WILEY & SONS LTD (1992) 809–809
8. Collier, R.W., O'Hare, G.: Modeling and Programming with Commitment Rules in Agent Factory. In: Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches. IGI Publishing (2009)
9. Ingrand, F., Georgeff, M., Rao, A.: An architecture for real-time reasoning and system control. IEEE Expert **7**(6) (1992) 34–44
10. Berna-Koes, M., Nourbakhsh, I., Sycara, K.: Communication efficiency in multiagent systems. In: 2004 IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. Volume 3.
11. Lillis, D., Collier, R., Toolan, F., Dunnion, J.: Evaluating Communication Strategies in a Multi Agent Information Retrieval System. In: Proceedings of the 18th Irish Conference on Artificial Intelligence and Cognitive Science (AICS 2007), Dublin, Ireland, Dublin Institute of Technology (2007) 81–90
12. Dragone, M.: SoSAA: An Agent-Based Robot Software Framework. PhD thesis, University College Dublin (2007)