

# Practical Development of Hybrid Intelligent Agent Systems with SoSAA

Mauro Dragone<sup>1</sup>, Rem W. Collier<sup>2</sup>, David Lillis<sup>2</sup>, and Gregory M. P. O'Hare<sup>1</sup>

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

<sup>2</sup> 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 and 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 Artificial Intelligence techniques in order to build large-scale intelligent software systems. 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 [?]. However, less attention has been paid to the practical implementation of such systems. Although a number of toolkits exist for the development of MASs [?,?], agents tend to be developed purely from an agent standpoint, with little regard to the underlying apparatus of the system.

---

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

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 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 [?]. Other work has focused on the environment within agents are situated [?]. Here, it is advocated that the environment is an integral part of the MAS and as such should be an essential building block in the development of such systems. 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 combines the concepts of Agent Oriented Software Engineering (AOSE) and Component-Based Software Engineering (CBSE) in the development of MASs. 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.

This paper demonstrates how SoSAA can be used to integrate component-based systems and intelligent agents to provide a standardised approach to the development of intelligent software. The separation of concerns is a key feature of this approach. A brief overview of the framework is provided in Section ?? . Section ?? outlines a number of improvements that were necessary so as to integrate the SoSAA approach into the underlying Agent Factory framework [?]. This integration is implemented via the SoSAA Adaptor, which is discussed in ?? .

An example of the applicability of the SoSAA approach to software development is contained in Section ?? . This takes the form of an implementation of a Transport Manager that implements a hybrid backchannel communication strategy, as advocated in the RETSINA Agent Architecture [?]. Finally, our conclusions and ideas for future work are outlined in Section ?? .

## 2 SoSAA

Popularised by their use in robotics (e.g. in [?]), 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. ?? . 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 [?].

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 computational environment is provided to the

high-level framework, which then contributes its multi-agent organisation, ACL-level interaction, and goal-oriented reasoning capabilities to intelligently perform this system assembly. Agents may also alter the system architecture and/or configuration to reflect changing goals and environmental circumstances. By interacting with the component layer, intelligent agents can access the system's resources (e.g. hardware, CPU, data, network), 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.

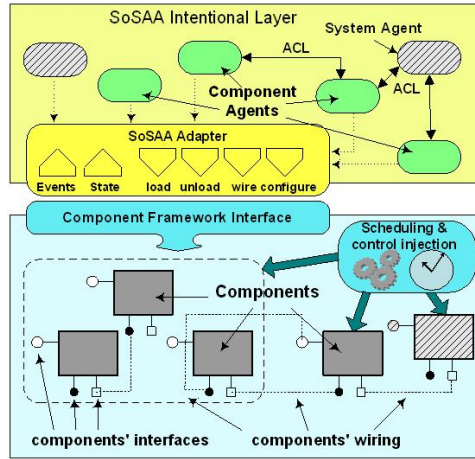


Fig. 1. SoSAA Hybrid Integration Strategy

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

The other mechanism adopted for implementing the SoSAA hybrid strategy is a callback design pattern with which dedicated SoSAA component agents can collaborate with the low-level framework by monitoring it and by registering themselves as controllers for specific events. In this manner, these agents are then able to override the default behaviour of the mechanisms provided by the low-level infrastructure, and thus take more informed, context-sensitive decisions. While this can be done by taking application-specific situations into consideration, as both infrastructure and application-level issues are equally addressed at the ACL-level, these two levels can now be confronted by different agents. Instead of overloading one agent with both types of concerns, such an approach enables the definition of distinct agents that focus only on certain aspects and that address conflicts and inter-dependencies by negotiating at the ACL-level.

Fig. ?? shows the multi agent organisation in a typical SoSAA node. Specifically, the low-level framework provides the interface for operating both at the application and at the infrastructure level. Accordingly, depending on their interests, SoSAA component agents can be distinguished between application and 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 in order to facilitate the adoption 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 ??, and the Java Modular Component Framework (JMCF). JMCF comes with a package of built-in component types and base-class implementations, each providing (i) component-container functionalities (e.g. loading, unloading and configuration of components), (ii) brokering functionalities to register and find component services, and (iii) support for the run-time execution of active (process-type) components. Further details on JMCF can be found in [?].

### 3 Improvements to Agent Factory

Agent Factory is a modular, extensible, open source framework for the development of Multi Agent Systems. [?]. The key components of this framework are a distributed 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 that 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 Adapter 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) [?], an agent oriented programming language that has been successfully applied in a number of significant problem domains [?]. 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 activities 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 [?]. Specifically, this involved the introduction of two

---

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

LOAD_MODULE sosaa sosaa.module.ComponentStore;
```

---

**Fig. 2.** SoSAA Adaptor Code

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. In choosing this activity, actions are prioritied over plans, so as to avoid unnecessary complexity. 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.

## 4 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 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. ?? 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

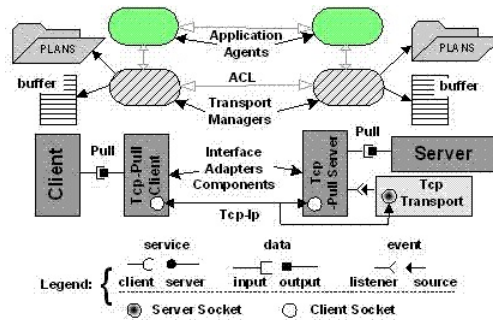


Fig. 3. SoSAA Backchannel Management

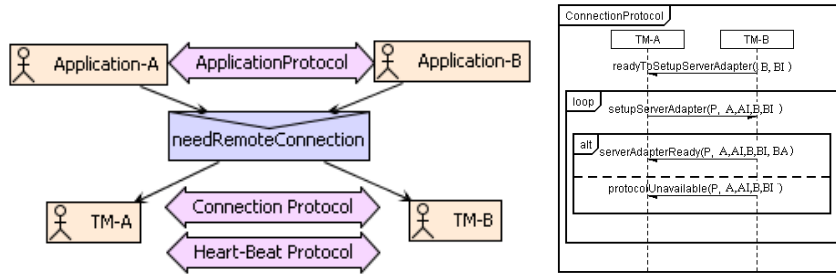
in and also a way of accessing the events and properties that are generated by those components. To achieve this, the `sosaaEventManager` perceptor has been created. This perceptor converts events and properties into beliefs that can be used at the agent level.

To create a SoSAA agent, you simply import the above AFAPL2 program into your own agent program and then write your own code, using the SoSAA support where relevant. Additionally, the `SoSAAService` platform service must also be installed on the agent platform, Details of how this code is used to manage the component layer are provided in the next section.

## 5 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 [?]. 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 agent is not new, and was previously proposed for the RETSINA architecture [?]. However, a limitation of the RETSINA approach is that it provides only ACL-level, implementation-agnostic specifications of the recommended coordination mechanisms. As such, it is left to the developer to implement both the underlying backchannels and also the coordination protocols. In contrast, using SoSAA, we have designed and implemented a simple 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. ??, 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.



**Fig. 4.** (a) Interaction Diagram (left), and (b) A UML 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)

Our backchannel management service supports two types of backchannel: (1) *push* backchannels, which provide support for the wiring of remote components where the component that generates the data controls when the next unit of data is transmitted; and (2) *pull* backchannels, which support the wiring of components where the component that receives the data controls when the next unit of data is transmitted. Fig. ?? shows a pull type of backchannel has been set up based on a raw TCP-IP transport mechanism.

In designing our backchannel management service, we initially focused on the protocols that underpin the service. Fig. ??(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 kicks off the connection protocol shown in Fig. ??(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

---

```

BELIEF(platformName(?IP)) &
BELIEF(message(request,?s,setupRemoteConnection(?IC,?II,?rC,?rI,?rP))) =>
COMMIT(?self,?now,BELIEF(true),
  SEQ(achieve_goal(GOAL(platform(?rP, ?a))),
    lookup(?IC),
    OR(DO_WHEN(BELIEF(clientInterface(?IC, ?II, ?type, ?jI, ?m, ?ib, ?mb, ?b)),
      MAINTAIN(GOAL(clientConnection(?jI, ?IP, ?IC, ?II, ?rP, ?rC, ?rI, ?a, ?p)))),
    DO_WHEN(BELIEF(serverInterface(?IC, ?II, ?t, ?jI)),
      MAINTAIN(GOAL(serverConnection(?jI, ?IP, ?IC, ?II, ?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),
                                configure(?cAN, SERVER, ?rip),
                                configure(?cAN, SERVER_CONNECTION, ?nRA),
                                focus(?cAN),
                                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

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. ?? 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 ?. Once invoked, it generates beliefs indicating whether 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. ?. This plan implements the client side of the interaction protocol presented in Fig. ??(b). Here, TM-A is the *Transport Manager* responsible for the client side while TM-B is responsible for the server side. The plan basically forces TM-A to wait for a *readyToSetupServerAdapter(...)* 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 ??) 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. ?? 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 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.

## 6 Conclusions and Future Work

In this paper, we have presented an overview of ongoing work aimed at investigating the integration of two software engineering paradigms, namely AOSE and CBSE. Our rationale for this is simple: CBSE offers tried and tested technologies for implementing applications that are modular, extensible, and which facilitated the creation of applications that can easily be adapted at run-time. Conversely, AOSE offers the promise of intelligent distributed systems that are well-suited to problems that require run-time coordination. Further, many AOSE toolkits lack explicit support for the development of and integration with low-level code that is required to implement core features of many proposed applications. As a result, this often leads to ad-hoc integration strategies that result in low quality code that is not reusable.

To counter this, we have previously proposed SoSAA, a conceptual framework for integrating AOSE and CBSE. Further, we have developed an incarnation of SoSAA based on a combination of the Agent Factory framework and the JMCF component toolkit. Here, we have attempted to demonstrate how SoSAA, in its current incarnation, 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. This service provides a generic and reusable solution that has been

applied to both an agent-based information retrieval engine [?] and multi-agent robotics [?], and which, is currently being using in connection with ongoing work on Wireless Sensor Networks. Further, we believe that our approach leverages the benefits and strengths of both AOSE and CBSE in a way that promotes the development of high-quality applications.