SoSAA:
A Framework for Integrating Components & Agents

Mauro Dragone
CLARITY: The Centre for
Sensor Web Technologies
University College Dublin
Belfield, Dublin 4
Ireland
+353 1 716 2491
Mauro.Dragone@ucd.ie

David Lillis
University College Dublin
Belfield, Dublin 4
Ireland
+353 1 716 2908
David.Lillis@ucd.ie

Rem Collier
University College Dublin
Belfield, Dublin 4
Ireland
+353 1 716 2465
Rem.Collier@ucd.ie

G.M.P. O'Hare
CLARITY: The Centre for
Sensor Web Technologies
University College Dublin
Belfield, Dublin 4
Ireland
+353 1 716 2472
Gregory.Ohare@ucd.ie

ABSTRACT

Modern computing systems require powerful software frameworks to ease their development and manage their complexity. These issues are addressed within both Component-Based Software Engineering and Agent-Oriented Software Engineering, although few integrated solutions exist. This paper discusses a novel integration strategy, which builds upon both paradigms to address their shortcomings while leveraging their different characteristics to define a complete software framework.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *Frameworks.*

General Terms

Design

Keywords

Agent oriented Software Engineering; Component based Software Engineering; Agent Programming Toolkit, Component Framework

1. INTRODUCTION

Today's ubiquitous and pervasive applications are typically open and dynamic, as the nature and the availability of both their hardware and software elements are not stable but may change at run-time. In order to adapt to such environments, modern applications must exhibit run-time flexibility, such as an ability to re-organize the interaction patterns of their architectural elements during execution. These issues are being addressed by both the Component-Based Software Engineering (CBSE) and Agent-Oriented Software Engineering (AOSE) paradigms, each offering a modular design by which to encapsulate, integrate and organize the different systems functionalities.

CSBE [15] operates by posing clear boundaries between architectural modules (the *components*) and guiding the developers in assembling these components into a system architecture. Within CBSE, domain analysis captures the principle quality attributes and expresses them in form of a ***component model***. This typically provides an unambiguous description of the different component types: their features and behavioral properties, and the set of their legitimate mutual relationships. These are supported by inter-component communication channels. Such a level of flexibility is deemed essential for implementing

framework-level mechanisms for ensuring that inter-component dependencies evolve along well-defined and anticipated lines so as to guarantee the preservation of system-wide quality attributes.

AOSE provides a method of abstraction and system decomposition based on *agentification*. This transforms a software application into an agent, by building a wrapper around it so it can interoperate with the rest of the system. This results in component-based systems in which traditional components are replaced by agents with reasoning capabilities and Agent Communication Language (ACL) interfaces.

Despite the similarities and commonality of objectives, there is little actual cross-fertilization between CBSE and AOSE. CBSE has begun to tackle reasoning about the assembly and integration of composite systems. In striving to produce self-managing and adaptable architectures, a formal base is usually required to describe the provided and required features of individual components and also important semantic aspects, such as the correct way those features are to be used. Many of the problems and solutions being encountered in CBSE resemble those already addressed in AOSE for multi agent coordination and high-level negotiations for resource provision.

By drawing on work conducted within the Multi-Agent Systems (MAS) community, AOSE can utilise dedicated design methodologies through which analysis and modelling of inter-agent interaction can be performed. However, the emphasis of multi agent toolkits in general is in allowing the coordination of large scale, deliberative MASs, while issues arising from low-level functionalities are often overlooked. The effort in standardizing the ACL-level is not matched by similar efforts in enabling the integration with low-level functionalities. What is missing is a proper mechanism to allow multiple agents to share a fine-grained access to a common functional layer without incurring interference or costly ACL-based coordination. Thus, AOSE architectures are mostly used as high-level, application integration frameworks, leaving the developer with the problem of managing all the agent's functionalities within each part of the application. Consequently, in many hard applicative domains, take up of the AOSE approach is still limited, with a more traditional component-based approach usually preferred.

This paper presents a step toward the solutions of these problems that builds on both CBSE and AOSE to leverage on their distinct characteristics to provide a complete construction process with its associated software framework. The key focus is on the interaction between the component and deliberative layers of the

framework, with overviews provided of the other framework features.

The remainder of this paper is organized in the following manner: Firstly, Section 2 provides a conceptual overview of our approach, and also discusses the design rationales of the resulting framework – the Socially Situated Agent Architecture (SoSAA). Section 3 describes the implementation of the framework. After that, Section 4 introduces an example application to illustrate its capabilities in more detail. Section 5 discusses this work in relation to related work in both the CBSE and AOSE communities. Finally, Section 6 summarises our experiences with the new framework and points to the directions we intend to explore in our future research.

2. SoSAA Hybrid Framework Strategy

SoSAA is a software framework initially intended to foster and investigate the use of AOSE for the development of modern robot systems. In general, the aim of SoSAA's design is to ease the development, maintenance and extension of complex applications that are specified in terms of elements with agreed-upon responsibilities and interfaces. SoSAA addresses system's modularity by drawing from the analogy with hybrid control architectures for autonomous agents. Popularised for their use in robotics (e.g. [9]), hybrid control architectures are layered architectures combining low-level behaviour-based systems [2] with high-level, deliberative/procedural reasoning apparatus. From a control perspective, such an arrangement enables delegating many of the details of the agent's control to the behaviour system, where they are undertaken by closely monitoring the agent's sensory-motor apparatus, without employing symbolic reasoning. Although hybridization approaches vary, the general trend is trying to eliminate the dominance of some layers over others. In particular, hybrid architectures try as much as possible to exercise an abstract control of the objectives pursued by the reactive/behavioural layer. However, ultimately the latter is never left on its own device, as the higher layers usually intervene on an event basis to re-configure its short-term objectives.

The original solution implemented in the SoSAA framework is to apply such a hybrid integration strategy also to the system's infrastructure. Fig. 1 helps illustrating this point. SoSAA combines a component-based infrastructure framework, with a MAS-based high-level infrastructure framework. The first can be used to instantiate different component-based systems and provide a computational environment to the second, which then augments its capabilities with its multi-agent organization and goal-oriented reasoning. To this end, the SoSAA adapter provides meta-level perceptors and meta-level actuators modules, which collectively define the interface between the two layers in SoSAA. In particular, the SoSAA adapter sets the range of the possible interventions and monitoring capabilities of the intentional layer

by distinguishing between the capabilities of the infrastructure and those of the application. Crucially, it also allows components to be accessed by multiple intentional agents, called *component agents*, whose specializations can be formalized by defining a number of roles, covering either application, infrastructure or cross-level concerns.
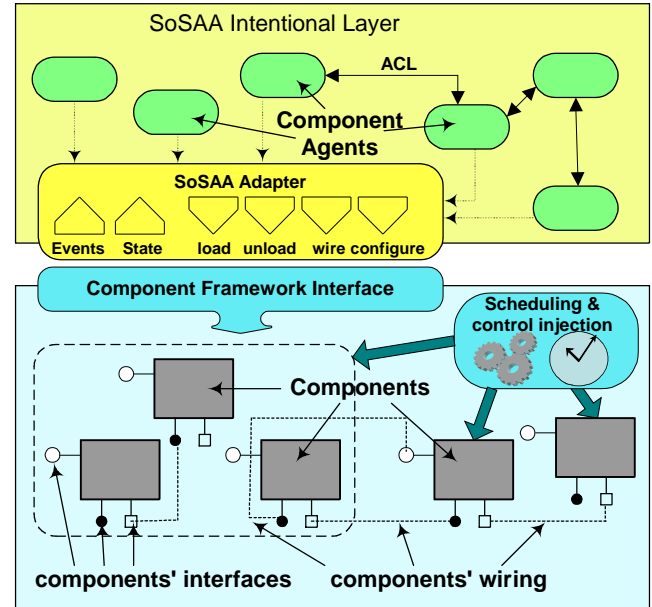

Figure 1. SoSAA's hybrid framework strategy.

SoSAA leverages both the component-model of its low-level infrastructure, and the MAS organization of its intentional layer to define a complete architectural framework. Simply put, SoSAA requires wrapping functional skills within low-level components before they can be administered by component agents in the SoSAA intentional layer. This enables the adoption of a low-level component model that can be oriented toward supporting specific application domains.

SoSAA developers also have flexible control of system granularity. Given the standardized interface toward low-level components, it is easy to implement and test different configurations, for example to tune a particular application or to adapt to pre-existing contexts. Finally, we also highlight how this interface is defined in terms of both standard agent capabilities and common features of component models. As such, SoSAA's design facilitates the replacement of different agent platforms and different component-based frameworks. In addition to easing the extensibility of the framework, this also enables to create heterogeneous deployments of SoSAA system, e.g. to adapt to computational constraints environments.

3. Implementation

The implementation of the SoSAA framework can be summarised by covering both the CBSE and the AOSE aspects, and their integration in the SoSAA Adapter.

3.1 Low-level component-based framework

In order to serve as a general-purpose infrastructure framework, the low-level component framework in SoSAA needs to provide the following features:

R1) Support for the *connection-driven* (procedural calls between clients and service providers), and *data-driven* (based on messaging and/or events) component-composition styles.

R2) Brokering, which is used by components to locate suitable collaboration partners for each of the composition styles supported by the framework. Thus participating components are not statically bound at design/compilation time but can be bound either at composition-time or at run-time.

R3) Container-type functionalities, used to load, unload, activate, de-activate, configure, and query the set of functional components loaded in the system, together with their interface requirements (in terms of provided and required collaborations).

R4) Binding operations, with which the client-side interfaces (e.g. service clients, event listeners, data consumers) of one component can be programmatically bounded to server-side interfaces (e.g. service providers, event sources, data producers) of other components.

R5) Support for life-cycle management and scheduling/control-injection of activity-type components (e.g. encapsulating processes and threads as in data processing routines, sensor drivers, and sensory-motor behaviours)

While these requirements cover features that are commonly agreed within CBSE, SoSAA demands two additional features, respectively:

R6) Support for priority dispatching of consumable events. This is required so that an event handler situated in the SoSAA intentional layer may override handlers registered within the low-level framework by: (i) registering itself as a prioritized event handler, and (ii) declaring the event consumed in order to cancel it from the event bus.

R7) A state repository associated with each component, which is used by the component agents to collect information about the state of the component's inner variables, including: the component's run-time requirements; other functional parameters; and the events it raises during its execution. This information must be translated into first-order predicates in order to be understood by component agents.

Finally, SoSAA also sets two important non-functional requirements on any implementation of its low-level component framework, namely: (i) extensibility to different applicative domains, and (ii) a clear separation between infrastructure and application concerns.

Microsoft's COM+ and Common Language Runtime (CLR) for the .NET platform, Sun's Java language, RMI, J2EE platform and Enterprise JavaBeans (EJB), are all candidate technologies. However, rather than being component frameworks in their own right, these constitute component-enabling technologies that can be used to create domain specific applications. Also, the majority of these initiatives are biased toward business related domains. They usually facilitate the design of multi-tier enterprise systems but provide only limited support for extension and adaptation. A notable exception in the CBSE area is the Fractal component

model [4]. Specifically, Fractal introduces the notion of a component endowed with an open set of control capabilities. These are not fixed in the model but can be extended and adapted to fit the programmer's constraints and objectives.

A similar approach is adopted in the JMCF (Java Modular Component Framework). JMFC is organized in a core package, which describes the framework in the form of a set of domain-independent interfaces, and in an implementation package. The latter includes common abstract implementation of the framework's basic classes as well as their domain-specific specializations. Abstract implementations in JMCF define *component types*, which capture the various characteristics of the specific application domain. Component types simplify the development of applications by providing a set of primitive components that are ready to be specialized by the developer. The other purpose of component types in JMCF is to manage the relationships with framework-type components. These are components offering system-wide services, such as *logging*, *scheduling/control-injection,* and *component repair,* that can be used by the functional components defined at the application level.

In JMCF, once an application's components extend a specific component type, they automatically inherit the framework mechanisms and the features supported by that component type. They are then left to declare the component's name and the Java interfaces by which the component's interfaces may be utilised (e.g. a source or listener event interface, a data consumer or producer interface, or a service interface).
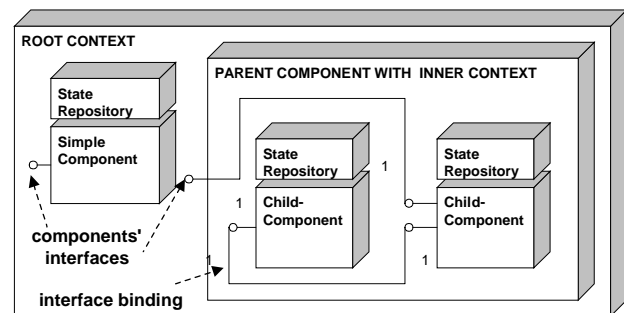

Figure 2. Recursive contexts in JMCF

As in Fractal, JMCF supports recursive components' contexts (see Fig. 2) to hierarchically organize system components and also to provide container and brokering functionalities. While a *root* context provides the main container, each component can also be a composite component by providing its own inner context to organize inter-component functionalities among its children.

Within JMCF, components' state repository is implemented as an add-on feature of each component type. Developers of functional components need only write the translation code for the information they wish to export to the SoSAA intentional layer. Finally, the particular implementation of JMCF facilitates the separation of application-type information, originated in the functional components, from infrastructure-level information, originated by the component types and by the framework-type components.

Consequently, each component's state repository can be queried selectively to retrieve either application-level or infrastructure-level information.

3.2 High-level MAS framework

The high-level framework is implemented using the pre-existing Agent Factory (AF) framework [5], a cohesive, FIPA standards compliant framework for building and deploying multi-agent systems. AF offers an open and extensible set of interfaces that supports the creation of a diverse range of agent architectures. It also supports the deployment of platform services, which act as shared platform-level resources, implemented in Java, that agents may bind to and use.

For SoSAA, AF is used in tandem with AFAPL [14], a purpose-built Agent-Oriented Programming language that models agents as mental entities whose internal state consists of beliefs and commitments. Informally, beliefs represent the agent's current state of its environment, while commitments represent the outcome of an underlying reasoning process through which the agent selects what activities it should perform. In AFAPL, an agent has both primitive abilities, in the form of directly executable actions, and composite abilities, in the form of plans built from plan operators such as SEQ (sequential execution), OR/XOR (branching), FOREACH (plan expansion), DO_UNTIL (loop), and DO_WHEN (conditional). Execution of an AFAPL program involves the update of the agent's mental state by repeatedly applying an internal reasoning process that combines: update of the agents beliefs via perception of the environment through a set of auxiliary Java components, known as perceptors; the adoption of new commitments though the evaluation of a set of commitment rules, which map belief states onto commitments that should be adopted should that state arise; and the realisation of commitments by performing actions, which are implemented through a set of auxiliary Java components, known as actuators.

An additional feature of AFAPL that is leveraged in the implementation of SoSAA is its support for code reuse through the introduction of an IMPORT statement that is somewhat similar to the **#include** statement of C. This allows developers to create partial AFAPL programs and reuse them as appropriate. For SoSAA, this is used to specify a minimal AFAPL program that contains the core actuators, and perceptors necessary to manipulate the SoSAA adapter.

3.3 Adapter

The key to our implementation of SoSAA is the adapter layer, which exposes core functions of the underlying component framework to the higher-level agent architecture. In our implementation, this is achieved by encapsulating the JMCF framework within an AF platform service. As was illustrated in Fig. 1, once bound to this service, AFAPL agents are able to interact with the underlying component in certain well-defined ways:

- *Loading / Unloading of components*: the ability to create new / destroy existing primitive and composite components for deployment within the framework.

- *Wiring of components:* the ability to bind components to one another either explicitly or implicitly based on an underlying wiring algorithm. Similar support exists for removing existing bindings.

- *Configure:* the ability of changing components' parameters, i.e. functional properties influencing components' behaviour, both at loading-time and at run-time.

- *Inspecting / Monitoring components*: the ability to inspect part or all of the current component hierarchy; the ability to query the state of the interface of a specific component; and the ability to start / stop monitoring the events and / or state of a component.

AFAPL agents utilize these features of the platform service via a corresponding set of actuators and perceptors. For example, a *ComponentStatePerceptor* is used to harvest the state information from components that the agent is monitoring and generate a corresponding set of beliefs; a *LoadActuator* is provided that supports the creation of both primitive and composite components that can be destroyed via a corresponding *UnloadActuator*; a *BindActuator* is used to support the various wiring mechanisms; a *InspectActuator* supports the inspection of a component and its interfaces; and a *FocusActuator* is provided to allow agents to specify which components they wish to monitor, i.e. in order to listen to the events they generate and to observe their state. At the AFAPL program level, each actuator is associated with one or more actions. As is highlighted in Fig. 3, another key feature of our implementation of SoSAA is the creation of an ontology that specifies a set of terms that represent information about the component framework. Inferences based on these terms can be implemented in AFAPL via the specification of belief rules [14].

```
ONTOLOGY SoSAAOntology {
    // ?ctxId     = context id
    // ?cId(n)    = component id
    // ?iId(n)    = interface id
    // ?cClass    = component class
    // ?iClass    = component class
    // ?iCategory = [DATA|SERVICE|EVENT]
    // ?iDir      = [CLIENT|SERVER]
    PREDICATE active(?cId);
    PREDICATE suspended(?cId);
    PREDICATE focusingOn(?cId, ?type);
    PREDICATE property(?cId, ?prop, ?val);
    PREDICATE created(?cId);
    PREDICATE removed(?cId);
    PREDICATE component(?cId);
    PREDICATE context(?ctxId);
    PREDICATE contains(?ctxId, ?cId);
    PREDICATE interface(?cId, ?iId, ?iClass,
            ?iCategory, ?iDir)
    PREDICATE bound(?cId1, ?iId1, ?cId2, ?iId2)
    PREDICATE failedBinding(?cId, ?iId);

    ...
}


// Perceive component's state information
// (generates beliefs of predicate property)
PERCEPTOR sosaaStateMonitor {
   USES  sosaa;
   CLASS sosaa.adapter.ComponentStatePerceptor
}


// load a component of given class into a context
// ?id will be the identifier of the new component
ACTION load(?ctxId, ?class, ?cId) {
    PRECONDITION BELIEF(componentClass(?class)
    POSTCONDITION BELIEF(true)
    USES SoSAAOntology
    CLASS sosaa.adapter.LoadComponentActuator
}
```

```
// load a component by assigning a numeric id
ACTION load(?ctxId, ?class)

// remove the component
ACTION remove(?cId) { ... }

// Bind (Explicitly) two interfaces
ACTION bind(?cId1, ?iId1, ?cId2, ?iId2) { ... }

// bind (Implicitly) a client interface (the
// brokering mechanism of the component's context
// is responsible for finding a compatible
// server-side interface.
ACTION bind(?cId, ?iId) { ... }

// change the value of the component's property
ACTION configure(?cId, ?prop, ?value) { ... }

// activate the component
ACTION activate(?cId) { ... }

// de-activate the component
ACTION deactivate(?Id) { ... }

// start focusing on the component
ACTION focus(?cId) { ... }

// stop focusing on the component
ACTION unfocus(?cId) { ... }

// inspect the component
ACTION inspect(?cId) { ... }

LOAD_MODULE sosaa sosaa.module.ComponentStore;
```

Fig. 3. Part of the SoSAA core AFAPL agent program

4. Case Study: Robotics

Today's robot systems constitute a special class of ubiquitous applications, whereby the need for dynamic and self-configurable architectures emerges from the very same requirements for autonomous operations. Robotics is also revelatory of the difficulty of AOSE in making an impact in this type of applicative domains, as the field is clearly dominated by more traditional object-oriented and component-based frameworks (e.g. [3]). It is a not a coincidence that robotics also gives a compelling example of the importance of sub-symbolical inter-component interaction over symbolic, ACL-based, coordination. Behaviour-based robot control architectures [2] explicitly rely upon the interaction between loosely coupled behaviour-producing modules, which may be easily encapsulated within components in component-based robot software frameworks. However, while the majority of these frameworks allow dynamic modification of collaboration patterns among system components, e.g. through late binding and reflection mechanisms, those mechanisms have relative importance at runtime, as once finalized those applications will generally run in a stable run-time environment. Furthermore, the same frameworks do not explicitly incorporate generic mechanisms for context-aware re-configuration of the architecture. Other than a missed opportunity, this is also a problem because application-specific solutions violate the principle of separation of concern and thus crucially result in poorly transferable systems, both in terms of software re-use and portability. In contrast, SoSAA offers a complete infrastructure framework upon which both low-level and high-level functionalities can be organized and integrated. To illustrate the

kind of organizational and context-aware, goal-oriented, cross-level and dynamic configuration enabled by SoSAA, Fig. 4 shows a sketch of a robot navigation system built with SoSAA (the details of which can be found in [7]).
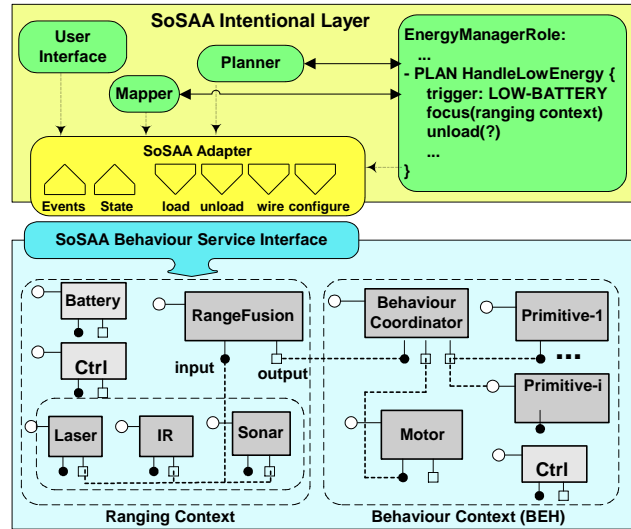

Figure 4. A robot navigation system built with SoSAA.

The specific low-level component framework includes two component contexts. The first groups, respectively: (i) a set of hardware drivers interfacing with range sensors (sonar, laser, infrared), (ii) a range-fusion component, whose duty is to merge their individual outputs to produce an overall picture of the obstacles in the robot's surrounding, and (iii) a component reading the robot's battery level,. The second context groups, respectively: (i) a set of primitive behaviour-producing components, and (ii) a component in charge of their coordination, whose duty is to merge their individual preferences in order to find the actual velocity control to send to the robot's actuators. Both contexts also include a framework-type component in charge of scheduling & control injection. Fig. 4 also singles out one of the component agents in the SoSAA intentional layer of this application. The specific component agent regulates the energy consumption of the system by triggering a plan (part of which is shown in Fig. 5) whenever the level of the robot's onboard batteries falls below a given threshold. In that eventuality, the agent stops the robot, unload every range-sensing driver, re-load and rewire a set of new drivers based on the level of battery charge, and re-start the behaviour context after setting a velocity adequate to the refresh rate obtainable with that set.

Noticeably, the *HandleLowEnergy* plan in Fig. 5 does not enter into the functional details other than in those exported through the SoSAA Adapter by the application's components. As such, the SoSAA Adapter defines the boundary between the responsibilities of the two layers in the system's architectures. Notwithstanding these limitations, such an approach enables the implementation of complex strategies that are defined across multiple functional areas and that account for both computational and functional

requirements. The simple component-based framework is enriched with the self-configuration capabilities enabled by the context aware reasoning conducted in the SoSAA intentional layer. In the example, the component agent in charge of the ranging context may even contact the robot's path planner by notifying the new maximum velocity in order to trigger the re-computation of the robot's path. Similarly, the path planner agent may also ask the interface agent to advise the user in the case the robot will be late for a pre-arranged rendezvous. By sharing the knowledge on the functional components constituting the applicative system, and also on how to affect them through the SoSAA Adapter, component agents can negotiate at the ACL-level only to resolve conflicts.

```
PLAN HandleLowEnergy(?id, ?type)
BODY
  SEQ(
    configure(beh.coordinator, vel, 0), // stop

    focusOn(battery), // monitor batteries

    FOE_EACH( // remove all ranging drivers
       Belief(contains(ranging.drivers, ?c)),
       remove(?c)
    ),

    //Re-load ranging drivers in order of
    //increasing energy consumption
    DO_UNTIL(!Belief(lowEnergy),
     getNextExpensiveRangeSensorClass(?c),
     DO_WHEN(Belief(nextExpensive(?c)),
       SEQ(load(ranging.drivers, ?c),
         DO_WHEN(Belief(created(?cId),
          bind(?cId,output,RangeFusion, input))))))
  ...
  DO_WHEN(Belief(fail),send(Planner,newVel(?v)))
END
```

Figure 5. Part of the HandleLowEnergy Plan.

5. Discussion

The RETSINA MAS [10] employs a hybrid method of communication, separating the coordination of the agents from the simple flow of data. ACL such as KQML and FIPA-ACL have been developed so as to be tailored to the needs of agent coordination: facilitating requests, responses, the flow of information and bearing higher-level conversations such as auctions in mind. However, this is not suitable for certain types of data, for example telemetry data or video. For these, RETSINA MAS makes use of "backchannels", which are specifically designed to cater for the flow of this type of low-level data. These do not employ ACL, as to do so results in inefficiencies in terms of greater processing overheads being invoked, without a tangible benefit being observed. The management of these backchannel streams is, however, carried out via ACL, with agents communicating in this way in order to establish and close channels as necessary.

SoSAA also makes use of such a hybrid communication model. Individual components can make use of backchannels in order to share information amongst themselves and to coordinate their own efforts. This has the effect of reducing the quantity of ACL communications being generated, as these are reserved for coordination at the deliberative layer of the agents. RETSINA, however, is limited solely to coordinating communication, by negotiating different types of communication for various types of data. In contrast, SoSAA extends this hybrid approach to a component-based framework attending both inter-component communication and components' execution needs.

A component-based approach in the construction of multi agent systems has been supported by numerous researchers in the past. This typically considers the components to be simply the building blocks from which agents are constructed. The interfaces by which these components may be used are described using a language such as DAML-S (as in [1]) and an agent composition service is charged with building agents from the library of re-usable components that are available. Once assembled, these components comprise the entirety of the agent. An advantage of this approach is the ability to take domain-specific issues into account at the component level. Domain analysis done at the time components are written allows the types of components, their interfaces and inter-component rules to be tailored accordingly. The decisions made on these issues can therefore be separated from the task of constructing the multi agent system as a whole, thus simplifying the process.

Although not specific to the agent's domain, a similar approach to system construction is adopted in [11], where a Prolog-based "kernel" is used to construct relationships and facilitate communication and method invocation between the system components. This kernel utilises the available descriptions of the components' interfaces to match up components based on the services they provide or require. This type of system allows dynamic rewiring of the system at run-time so as to react to any exceptions that may occur.

The principal difference in SoSAA is that the capabilities of the individual components are augmented by the goal-driven reasoning capabilities of the deliberative layer of the multi agent system. This allows us to leverage programming languages that have been developed specifically with deliberative reasoning in multi agent systems in mind, such as AFAPL or Jason. These can be used for the higher-level management of components, such as deciding when it is appropriate to active or deactivate components according to the needs of the system as a whole. Components are left to automatically carry out lower-level behaviours, with the deliberative layer making decisions about when such behaviours are necessary or desirable in order to satisfy overall system goals. As such, SoSAA is an original construction methodology, going beyond the mere composition of components into an agent.

SoSAA also shares some of the motivations of multi agent systems based on the Agents & Artifacts (A&A) meta-model such as CARTAGO [13]. These systems are based on activity theory and operate by using tools or artifacts to cope with the scaling up of complexity. These artifacts provide a consistent usage interface by which agents may interact with their environment. These interactions can take the form of operations that the artifacts may carry out, or perceptions that are created by the artifacts monitoring their environment for the benefit of the agents. As with component-based systems, domain-specific issues are taken into account at the stage the artifacts are created.

A component-based framework such as SoSAA may also use components in the same way as artifacts. However, the most significant difference between the systems is that artifacts are strictly passive entities. They do not carry out any operations unless instructed to do so by the deliberate layer of the agent. In

contrast, SoSAA leverages existing well-established research in the CBSE domain, specifically as it applies to Robotics. Here, components are not merely passive, but play an essential role in managing the reactive behaviour of an agent. A component will react to events according to a particular behaviour until it is instructed to do otherwise by the agent's deliberative layer. Additionally, individual components may communicate amongst one another at the sub-symbolic level using the backchannels mentioned above. This results in the deliberative layer being free to concentrate on higher-level reasoning. Additionally, CARTAGO currently lacks a well-defined ontology to aid agents in discovering how artifacts may be utilised. In contract, SoSAA can make use of pre-existing research on Architecture Description Languages (ADLs) and contract-based Quality of Service component specification.

6. Conclusions and Future Work

This work described an ongoing effort in combining the strength of AOSE and CBSE through the SoSAA framework. We believe the novelty or this work arises from the way we have combined aspects of other work in the area to provide a cohesive strategy for integrating component-based and agent-based approaches.

We have discussed general guide-lines for supporting our integration strategy, and presented one incarnation of SoSAA. SoSAA and its constituent software systems can be freely download at http://www.agentfactory.com. Preliminary versions of the software have been already employed for the construction of autonomous agent architectures in both simulated [6] and real (robotic) settings [7]; for the construction of agent-based ubiquitous systems [8]; and also for the re-factoring of agent-based applications that did not previously availed of CBSE principles [12]. In the later applicative context, SoSAA allowed a clearer separation of concerns between the underlying functionality and the agent-layer coordination mechanisms. This has improved the readability of the code base and crucially also led to more efficient implementations.

In general, the range of these applications displays the flexibility of the SoSAA framework in supporting the development of both low-level functions and high-level (e.g. goal directed and context aware) capabilities. As different people collaborated on the different applications, it was found that SoSAA eases the design, the implementation, the maintenance and the extension of applications that are specified in terms of elements with agreed-upon responsibilities and interfaces. Once they use the SoSAA Adapter or conform to the underlying component framework, developers can focus on exercising their core expertise in the implementation of the internals of either low-level components or intentional agents.

While working on different application scenarios has been crucial in collecting valuable experience and in shaping the current design, future work is necessary to identify a set of benchmarks in order to carry out a more formal validation of SoSAA.

7. REFERENCES

[1] Amor, M., Fuentes, L., and Troya, J. M., Putting Together Web Services and Compositional Software Agents, J.M. Cueva Lovelle et al. (Eds.): ICWE 2003, LNCS 2722, pp. 44–53, 2003.

[2] Brooks, R. A. (1991) *New Approaches to Robotics*. Science (253), September 1991, pp. 1227–1236.

[3] Brooks, A. *et al.,* Towards Component-Based Robotics, *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS* 2005), August 2-6 2005, Edmonton, Alberta.

[4] Bruneton, E. T. *et al.*, *The Fractal Component Model and Its Support in Java*. Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems. 36(11-12), 2006.

[5] Collier, R.W. *et al.*, Beyond Prototyping in the Valley of the Agents, *in Multi-Agent Systems and Applications III, Lecture Notes in Computer Science (LNCS 2691), Springer-Verlag*

[6] Dragone, M. , Lillis, D. , Muldoon, C. , Tynan, R., Collier, R. W., and O'Hare, G. M. P. *Dublin Bogtrotters: Agent Herders*, In Post-proceedings of the Sixth international Workshop on Programming Multi-Agent Systems, ProMAS, 2008.

[7] Dragone, M.: *An agent-based robot software framework*. *PhD Thesis*, Dept. of Computer Science, Univ. College Dublin, 2007. [online] http://csserver.ucd.ie/~mdragone/pubs/MauroDragonePhdThesis.pdf.

[8] Dragone, M., Holz, T. and O'Hare, G.M.P. Using Mixed Reality Agents as Social Interfaces for Robots, *In IEEE International Workshop on Robot and Human Interactive Communication. ROMAN*, 2007.

[9] Gat, E., ATLANTIS: Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. *In Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*. pp. 809-815, 1992.

[10] Koes, *M.,* Nourbakhsh, *L., and* Sycara *K.,* Communication Efficiency in Multi-agent Systems, *In Proceedings of ICRA 2004*, Vol. 3, May, 2004, pp. 2129 – 2134, 2004.

[11] Natali, A., Oliva, E., Ricci, A., Viroli, M., A Framework for Engineering Interactions in Java-based Component Systems. *Electr. Notes Theor. Computer Science. 154*(1): 43-6, 2006.

[12] Peng, L, Collier, R., Mur, A., Lillis, D., Toolan, F., and Dunnion, J., A Self-Configuring Agent-Based Document Indexing System, in *Multi-Agent Systems and Applications IV, Lecture Notes in Artificial Intelligence (LNAI), Volume 3690, Springer-Verlag*

[13] Ricci, A., Piunti, M., Acay, L. D., Bordini, R. H., Hübner, J. F., Dastani, M., Integrating heterogeneous agent programming platforms within artifact-based environments. AAMAS (1) 2008: 225-232.

[14] Ross, R., Collier, R., O'Hare, G.M.P., (2004), AF-APL – Bridging Principles & Practice in Agent-Oriented Languages, *Programming Multi-Agent Systems, Lecturer Notes in Artificial Intelligence (LNAI), Volume 3346, Springer-Verlag.*

[15] Szyperski, C., *Component Software: Beyond Object–Oriented Programming*. Addison-Wesley, 1999