Separation of Concerns in Hybrid Component and Agent Systems

Mauro Dragone*

CLARITY: Centre for Sensor Web Technologies
E-mail: mauro.dragone@ucd.ie
*Corresponding author

Howell Jordan, David Lillis and Rem W. Collier

School of Computer Science and Informatics,
University College Dublin (UCD),
E-mail: howell.jordan@lero.ie
E-mail: david.lillis@ucd.ie
E-mail: rem.collier@ucd.ie

Abstract: Modularising requirements is a classic problem of software engineering; concerns often overlap, requiring multiple dimensions of decomposition to achieve separation. Whenever complete modularity is unachievable, it is important to provide principled approaches to the decoupling of concerns. To this end, this paper discusses the Socially Situated Agent Architecture (SoSAA) - a complete construction methodology, which leverages existing well established research and associated methodologies and frameworks in both the Agent-oriented and Component-based Software Engineering domains. As a software framework, SoSAA is primarily intended to serve as a foundation on which to build agent based applications by promoting separation of concerns in the development of open, heterogeneous, adaptive and distributed systems. While previous work has discussed the design rationale for SoSAA and illustrated its application to the construction of multiagent systems, this paper focuses on the separation of concerns issue. It highlights concerns typically addressed in the development of distributed systems, such as adaptation, concurrency, fault-tolerance. It analyses how a hybrid agent/component integration approach can improve the separation of these concerns by leveraging modularity constructs already available in agent and component systems, and sets clear guidelines on where the different concerns must be addressed within the overall architecture. Finally, this paper provides a first evaluation of the application of our framework by applying well-known metrics to a distributed information retrieval case study, and by discussing how this initial results can be projected to a typical multiagent application developed with the same hybrid approach.

Keywords: Separation of Concerns, Distributed System, Agent Oriented Software Engineering, Component Oriented Software Engineering

Reference to this paper should be made as follows: Dragone, M., Jordan, H., Lillis, D., Collier, R.W. (2010) 'Separation of Concerns in Hybrid Component and Agent Systems', *Int. J. Communication Networks and Distributed Systems*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Mauro Dragone is a postdoctoral researcher in the CLARITY centre and a postdoctoral fellow at University College Dublin, School of Computer Science and Informatics. He received a PhD from University College Dublin and a Magna Cum Laude laurea degree in Informatics from the University of Bologna (Italy). Prior to his involvement with the CLARITY centre, he worked for more than 12 years in a variety of software companies as a software engineer in the area of distributed systems and enterprise applications. His current research addresses a wide range of theoretical and practical aspects of software engineering for ubiquitous and pervasive computing systems, including the integration between robots and wireless sensor networks.

Howell Jordan is a PhD student in Software Engineering at Lero, University College Dublin, Ireland. He is investigating the design qualities, such as structure, re-use, and maintainability, of agent-oriented programs. He graduated in Physics from Oxford (2000-2003), and received an MSc. in Computer Science from the University of Wales, Aberystwyth (2003-2004). He has 4 years' experience as a software engineer in the telecommunications industry, having previously worked as a Technical Lead (EMF Client) in Subscriber Data Management for Nokia Siemens Networks.

David Lillis is a PhD student in University College Dublin. He is also a Research Assistant investigating the applicability of data fusion in Information Retrieval as part of the SIFT project. His MSc thesis was written while working as part of the HOTAIR Project (Highly Organised Teams of Agents for Information Retrieval), which was affiliated with the IIRG (Intelligent Information Retrieval Group). His principal research interests are in the areas of Information Retrieval and Interaction Protocols in Multi Agent Systems. He is a developer of the Agent Factory Framework.

Rem Collier is a senior lecture at the School of Computer Science at the University College Dublin (UCD). Rem has a primary degree in pure and applied mathematics from the University of Bristol in 1994; a M.Sc. in Computation (Examination and Dissertation) in 1995 and a M.Phil (Research) in 1996 from the University of Manchester Institute for Science and Technology (UMIST); and a Ph.D. in Computer Science in 2001 from University College Dublin. After completing his Ph.D., Rem spent a year working in industry, in the area of web-based e-learning content management platforms for course authoring and localisation. In 2002, he returned to UCD where he worked as a Post-Doctoral Researcher, and in 2003 he was appointed as an Assistant Lecturer. Finally, in 2005, Rem was appointed as a College Lecturer.

This work is supported by Science Foundation Ireland (SFI) under grant 07/CE/I1147.

1 Introduction

Today, Agent-Oriented Software Engineering (AOSE) is regarded as a general-purpose paradigm that facilitates the engineering of complex software by way of Multi Agent Systems (MASs), i.e. loosely-coupled, situated, autonomous and social components (software agents). AOSE, as discussed by Shoham (1993), offers a novel approach to abstraction and system decomposition based on the concept of agentification - the process of transforming a software application (or component) into an agent by building an "agent wrapper" around it to enable it to interoperate with the rest of the system. Zambonelli & Omicini (2004) note that AOSE is well-suited for web-based, heterogeneous and pervasive applications that comprise a set of autonomous processes that cooperate in a decentralised fashion. Specifically, AOSE helps to confront the high degree of variability within such applications, and counteract network delays that would frustrate a more centralised architecture.

The advantages of using agent-based methods are: (i) the resulting architectures can draw from the large body of work conducted within the MAS community, both in terms of theoretical insights as well as engineering methodologies, toolkits and middleware; (ii) system design can be simplified because the designer does not have to specify all interactions at design time as some can be autonomously handled by the agents themselves; and (iii) system robustness potentially increases because these interactions handled at run-time, allowing the system as a whole to better adapt to environmental conditions.

These advantages are realised through the adoption of concepts such as:

1. **Language:** Agent-Oriented Programming is a paradigm for programming agents using languages designed to capture a theory of rational agency.

2. **Goals:** Symbolic descriptions of the expected behaviour of an agent. Low-level decisions about the specific sequence of steps necessary to achieve the goal are identified and carried out autonomously by the agents.

3. **Communication Infrastructures:** Point-to-point communication and communication protocols for: (i) distributing control and data; and (ii) supporting automated coordination and negotiation.or

4. **Naming/Brokering:** Dynamic discovery of other agents, thus decoupling agents and enabling their dynamic fitting or replacement to increase interoperability and fault-tolerance.

5. **Roles:** Abstract specifications of behavioural patterns, which help structuring a system in a set of more manageable sub-systems with given responsibilities and interaction capabilities.

Specifically, Agent-Oriented languages provide syntactic constructs (terms, variables, functions) to represent the domain knowledge, goals and actions of an agent. While there is usually a degree of abstraction between language and architectural issues, languages inevitably impose constraints on the underlying execution layer that realises their semantics. The most common solution is to decompose the execution layer into agents and agent platforms, with the latter providing the functional basis that allows the agents to operate in their environment and interact with one another. Thus, an agent can be seen as an application layer software component that uses the agent platform as middleware to gain access to standardised services and infrastructure, such as life-cycle management, inter-agent communication, directory facilitators and coordination. Agent platforms not only free the developer from low-level details but also promote elementary modularity in the construction of the MAS as the features of the platform are re-used in each agent.

The context-aware, goal-directed reasoning capabilities and the sociality advocated within AOSE address the issue of adapting to the unpredictable dynamics of open environments that arise not only due to the mobility of users and processes, but also to lack of resources, power or communication disruptions. However, while MASs can respond to such unpredictability by re-organising inter-agent interaction patterns and (re-)negotiating for resource provision, they often violate the principle of separation of concerns by forcing first the developer (at design-time), and then the agents (at run-time) to consider both infrastructure-level and application-level issues. This contradicts the modular development process advocated within Component-Based Software Engineering (CBSE), and also results in poorly transferable systems, both in terms of software re-use and portability. As a result, developers are forced to rely on traditional object oriented approaches to define complex, application-specific interfaces for the application and operating system resources.

Recently, new research has emerged that has begun to deal with these issues in more detail. For example, Ricci et al. (2007) have developed the CArtAgO framework, which uses the Agents and Artifacts meta-model to model passive software entities, known as artifacts. These are resources and tools to be utilised in a standardised manner by agents to satisfy their objectives. Similarly, Weyns et al. (2007) argue that the environment in which the agents are situated is an integral part of a MAS and that the creation of an exploitable design abstraction of the environment is considered a key step in its design and implementation.

This increasing emphasis on offering a clear separation of concerns between the intelligent layer on one hand, and the low-level actions on the other is mirrored in recent work on the Socially Situated Agent Architecture (SoSAA) framework, described by Lillis et al. (2009) and Dragone et al. (2009*b*). SoSAA is an open source framework that combines AOSE with CBSE to leverage their distinct strengths and provide a complete construction process. While the original motivation of the work comes from robotics, it is our view that SoSAA has great potential as a general approach to fabricating complex distributed systems.

Furthermore, this use of layering as a mechanism for separating concerns in has gained some traction in the separation of concerns community (see Section 2). To this end, this article investigates how separation of concerns is impacted through the use of a hybrid integration strategy such as SoSAA. Firstly, we discuss in more detail the concerns typically addressed in the development of distributed systems, such as adaptation, concurrency, fault-tolerance, and review how they are addressed in related work. Secondly, we provide an overview of how component based techniques may support highly modular software architectures. Finally, we discuss SoSAA and analyse the case study of an agent-based search engine case study to illustrate how separation of concerns can benefit from the agent/component hybrid approach pursued in SoSAA.

2 Separation of Concerns in Distributed Systems

Separation of concerns is a concept that is at the core of software engineering. It refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concern. Mili et al. (2004) define a concern as a set of related properties/requirements of the software being developed. Melliar-Smith et al. (1997) distinguish between two different levels of separation of concerns: (i) a conceptual level, aimed at identifying the primitive concerns of a specific domain/application, and (ii) an implementation level, aimed at providing an adequate organisation that isolates the concerns. The goal of the implementation level is to separate the blocks of code that address the different concerns, and provide for a loose coupling of them. Unfortunately, they observe how few programming languages allow conceptual level abstractions to actually be separately programmed. In other words, separation of concerns is often practiced at the conceptual level, but not at the implementation level. The resulting code organisation is monolithic, intertwining different concerns.

The latest generation of separation of concerns techniques include: subject-oriented programming, described by Harrison & Ossher (1993); composition filters, described by Aksit et al. (1992); AOP and view-oriented programming, described by Mili et al. (1999). It can be argued that these techniques define new modularisation boundaries for requirements that are necessarily different from those offered by traditional object-oriented approaches. This is because concerns are rarely identifiable with specific architectural elements but are instead cross-cutting (aspects) and scattered over different software artifacts.

Many kinds of concern may be relevant to different developers in different roles, or at different stages of the software life-cycle. Mili et al. (2004) identify three major categories of requirements for software development: (i) **Application-Specific Functional requirements**; **Run-time (non-functional) requirements** (dictating application-independent properties, including distribution, persistence, security; and **Requirements on the software artifacts** (dealing with modularity, reusability, choice of programming language, adherence to specific programming style, etc.). Melliar-Smith et al. (1997) argue for the clear separation of the concerns that underpin each of the first two categories. That is, the functional behaviour of the application should be programmed by programmers who understand the application, while the non-functional behaviour (e.g. fault tolerance, maintaining the consistency of replicated information, assigning objects across a distributed system or load balancing and real-time scheduling) should be undertaken by programmers with expertise in those specific areas. Silva et al. (1995) investigate the set of concerns that arise from non-functional requirements specific to distributed systems by examining the level of transparency offered in modern distribution middleware, and identify seven concerns: *fragmentation*, *replication*, *naming*, *concurrency*, *failure (fault tolerance)*, *configuration*, and *communication*. For Silva et al. (1995), the **naming concern** is associated with location transparency, in which the application is not aware of the resource location. The **communication concern** is associated with access transparency, in which identical operations are used to access local and remote objects. The **concurrency concern** generates and controls access to shared resources ensuring undesirable interferences do not arise. Scaling transparency is not handled by a particular concern but is achieved if all the concerns consider scalable solutions. Finally, performance transparency, the ability of dynamically configuring the software in order to satisfy performance requirements, is associated to the **configuration concern**.

However, Silva et al. (1995) observe that, in most heterogeneous and open distributed systems, transparency is not sufficient for the handling of concerns at different levels of abstraction, namely:

1. ***Model level***, describing the expectation of users about the application or system behaviour;
2. ***Policy level***, defining algorithms which support application models;
3. ***Mechanisms level***, defining functionality which is used by policies to implement their algorithms.

For Silva et al. (1995), mechanisms should be further refined in terms of *abstract* and *concrete mechanisms.* The first fulfil the needs of models and policies in a platform independent manner while second ground them in the functional resources of the specific platform. This concept is illustrated via the concurrency concern by describing how concurrent activities can be implemented in a given platform with operating system threads while mutual exclusion can be supported by synchronisation primitives *Mutex* and *Condition.* Policy independence allows the same model to be simultaneously supported by different policies, so that heterogeneous policies, offering the same model, can coexist in the same application. Moreover, independence between policies and concrete mechanisms allows the application to be supported by heterogeneous platforms.

Atkinson & Kuhne. (2000) investigate the concept of a **stratified architecture** from a methodological standpoint as a means to abstract from aspects to the point where they become relevant. Their key observation is that although it is desirable to base aspects on high level system concepts, such as patterns or architectural styles, much of the perceived "tangling" between software components comes from low level component interactions where only code level abstractions are available. As a result, their methodology is based on refinement design patterns whereas system abstractions are used as anchors for defining the location of aspects, before implementation details are added in a step-wise manner. While they envisage the existence of many refinement patterns for specific aspects, they illustrate their approach by describing how the interaction between a Writer class and a File object in a distributed system can be refined by adding components in charge of data marshalling and distribution.

A key insight offered by Atkinson & Kuhne. (2000) is the realisation that providing different levels of abstraction helps the design and development of large scale systems. For example, in the persistence scenario, someone trying to understand the overall structure of the system is best served with a high level view, where only the Writer and the File objects are visible, while another person, whose task it may be to change the way data is marshalled over a network, gains more from looking at the refined model. This is in line with a number of other works trying to fill a perceived gap in aspect-oriented methodologies, i.e. that they do not explicitly address the need to document, communicate, reuse and systematically handle high-level compositions. For instance, Landuyt et al. (2007)

propose the Theme/UML graphical notation to capture the stakeholder concerns for a software system in a more direct manner and improve the traceability of concerns throughout the software life-cycle. Finally, it is this insight that drives the approach advocated by SoSAA where we believe that a stratified approach that combines AOSE and CBSE offers the potential to improve the separation of concerns in distributed systems.

3 Component Based Software Engineering

The Open Service Gateway Initiative (OSGi), CORBA Component Model, Microsoft Object Model, Enterprise JavaBeans and Fractal are some of the component-enabling technologies used for the creation of many industrial-strength distributed systems. Conceptually, the same technologies also provide a composite model for Service Oriented Architectures, by helping to design systems in terms of application components that can expose their public functionality as services (interfaces) as well as invoke external services.

While Object Oriented Programming (OOP) stresses separation of concerns, it is still fundamentally oriented toward white-box reuse. In the context of object oriented integration frameworks, for example, polymorphism and inheritance are used for the implementation of the customisation and extension mechanisms. Both mechanisms create compile-time dependencies between the various elements of the frameworks. This limits software modularity since a modification in one element may require the modification of other elements in the framework. Consequently, much of the merit of the design (e.g. its ability to stand further iterations), stems from the architectural choices and design patterns realised in each system.

In contrast, Szyperski (1998) notes that within CBSE, components are units of independent development, deployment and reuse that must conform to specific contractual obligations. Domain analysis captures the main quality attributes and expresses them in the form of a component model. This is aimed at providing an unambiguous description of the component types, in terms of their features and behavioural properties, and the set of their legitimate relationships. Together, a component model and its (usually object-oriented) component framework realisation constitute a well-defined and stable architectural frame, which assures that components can be developed independently while preserving system quality attributes like responsiveness, fault tolerance, scalability and performance.

3.1 Component Context

The most important common concept among component models and frameworks is the relationship between a component and its environment, wherein a newly instantiated component is provided with a reference to its Container or Component Context. This usually acts as an access point for framework-level functionalities reused across different applications. Reina & Torres (2004) call these functionalities "infrastructural services". The container can be thought of as a wrapper that deals with technical concerns such as synchronisation, persistence, transactions, security and load balancing. The component must provide a technical interface so that all components will have a uniform interface to access the infrastructure services.

OSGi defines a standardised component model and a lightweight container framework, built above the Java Virtual Machine, that is used as a shared platform for network-provisioned components and services specified through Java interfaces. Each OSGi platform facilitates the dynamic installation and management of simple components called *bundles*, by acting as a host environment whereby various applications can be executed and managed in a secured and modularised environment. An OSGI bundle organises the framework's internal state and manages its core functionalities. These include both container and life-cycle operations to install, start, stop and remove components as well as checking dependencies.

The *java.beans.beancontext* package in the Java Beans specifications provides the same kind of container environment. In addition, it introduces the notion of a hierarchical nesting or structure of *BeanContext* and JavaBean instances. This enables logical and/or functional grouping of components and allows the definition of macro-functionalities that can operate on those groups.

In contrast, Fractal supports a fully recursive hierarchical structure, whereby each component can also be a composite component by providing its own inner context to organise inter-component functionalities among its children. Additionally, 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.

3.2 Composition, Distribution and Adaptation

Many of the infrastructural services associated with component contexts act as late-binding mechanisms that can be used to defer inter-component associations by locating suitable collaboration partners for each of the collaboration styles supported by the framework. Through these brokering and naming mechanisms, components do not need to be statically bound at design/compilation time but can be bound either at composition-time or at run-time in order to favour the construction of adaptable software architectures. These features are present in all the frameworks considered in the previous section and also in more traditional distribution middleware, as in the CORBA trading service. The *Activator* class in OSGi, the *BeanContext*, and the component's membrane in Fractal enable components to look up services in the framework's service registry, register services, access other components, and install additional components within the local platform. In these cases, distributed component bindings are usually achieved through port and proxy mechanisms, as in *OSCAR*, a component framework built over OSGi. In *OSCAR*, a port can be viewed as a connection point on the surface of the component where the framework can attach (connect) references to provides-ports provided by other components. The PortsManager component is responsible for returning the correct Java object when a port is requested by a component. It either calls the appropriate methods of the locally available service implementation object or translates the Java method calls to messages, sends them to a remote container (e.g. availing of Java RMI, SOAP, or JXTA), waits for remote execution and then returns the value contained in the received message. In addition, the *PortsManager* also supports intelligent hot swapping of services to implement fault-tolerant and adaptive solutions. Specifically, as every service in OSGi may be given a certain rank which describes its quality and importance, when queried about a particular service, the *PortsManager* automatically tries to locate the highest-ranked implementation. However, in most of the frameworks of this type, run-time architectural adaptation usually leaves the issues of synchronising and coordinating between peer adaptive entities at the application-level to the developer.

4 The SOcially Situated Agent Architecture (SoSAA)

SoSAA incorporates modularity by applying the principles of hybrid control architectures to autonomous agents. Popularised by their use in robotics (e.g. by Gat (1992)), hybrid control architectures are layered architectures combining low-level behaviour-based systems with high-level, deliberative reasoning apparatus. This is a familiar concept. As humans, there are certain menial tasks that we can undertake without investing a significant amount of thought. Indeed we even refer to such activities as "mindless" on a regular basis. Working on a simple production line is one such example, and so it is desirable to separate the important intentional actions from those menial tasks.

From a control perspective, this approach enables the delegation of many of the details of the agent's control to the behaviour system of the agent, which closely monitors the agent's functional layer. The original solution implemented in the SoSAA framework is to also apply such a hybrid integration strategy to the system's software infrastructure, as illustrated in Figure 1, by combining a low-level component-based infrastructure framework with an agent-based high-level infrastructure framework. The low-level framework operates by imposing clear boundaries between architectural modules (the components). It then provides a computational environment to the high-level framework, which augments its capabilities with its multi-agent organisation and goal-oriented reasoning.

A standardised and simple interface between the agent and the component layer allows the use of different agent systems and different component technologies to build heterogeneous distributed and embedded systems. Agent programming languages that have been developed specifically with deliberative reasoning in mind, such as the AF-APL2 language proposed by Collier et al. (2003) as part of the Agent Factory (AF) framework, can be used for the higher-level, goal-based management of components in SoSAA. Agents can decide when it is appropriate to activate or deactivate components according to the needs of the system as a whole. Components are left to carry out lower-level functionalities, which are then driven/configured by the agents..

The key to the implementation of SoSAA is the *SoSAA Component Platform Service*, discussed in more detail later in this section. This service describes the SoSAA Component Model and exposes its core mechanisms to the system's agents. These mechanisms are commonly supported within different component-enabling technologies, such as Java Beans, OSGi or Fractal. These mechanisms are encapsulated in a service for the AF platform and are accessed through a set of interface modules, collectively named the *SoSAA Adapter*. Specifically, query-type modules (perceptors) query the component framework and provide beliefs about events, the set of installed components, their interfaces (based on

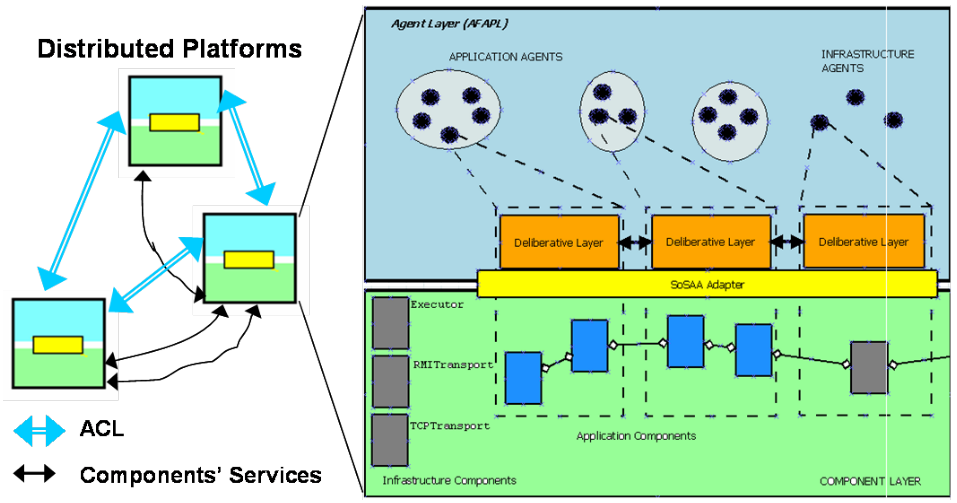

Figure 1 SoSAA Hybrid Agents/Components Integration Strategy

messages, events and/or procedural calls), their actual wiring, and their run-time performance, while actuator-type modules control the loading, unloading, configuration, activation, de-activation, and wiring of components.

The key to supporting the separation of concerns in SoSAA systems is the fact that infrastructure-type mechanisms are realised in the form of infrastructure-type components. By modelling these mechanisms in the agent layer, both the system developer and the agents can then rely on the default behaviour these mechanisms provide whenever possible, and override them only when they do not suffice to reach the system's objectives. To this end, SoSAA uses an event-based collaboration pattern to guide the integration and the run-time interaction between the two heterogeneous software elements it combines. Agents will usually declare their interest in either application or infrastructure types of events, in order to override both the basic functionalities and the framework-wide mechanisms provided by the underlying component framework (e.g. logging, component repair, scheduling), and operate more reasoned structural and behavioural re-configurations to the functional layer through the SoSAA Adapter. There are three levels of separation of concerns in SoSAA, respectively:

1. **Separation between functional and non-functional concerns** in both the agent and the component layer.
2. **Separation between high-level interaction models**, which are defined in terms of multi-agent coordination and goal strategies to attend performance and adaptation concerns, **policies**, supported through the associated coordination and execution plans, and **abstract mechanisms** supported in the SoSAA component model.
3. **Separation of these abstract mechanisms from their actual implementation**, which is grounded in the specific component framework adopted in the component layer.

4.1 Component Layer

The *Java Modular Component Framework* (JMCF) is the component framework developed as part of SoSAA and explicitly designed to support a stratified architecture by acting as an interface toward different component-enabling technologies. To this end, JMFC is organised (see Figure 2) in a core package, which describes the minimal SoSAA minimal component model, and in an implementation package. The latter includes common abstract classes defining common component types. Rather than directly implementing each of their features and capabilities, all the component types and the components in JMCF use a proxy design pattern to, respectively: (i) declare that they support a specific interface (feature), and (ii) return a reference to its implementation. Toward the root of the class-inheritance hierarchy, component types feature interfaces to infrastructure-type services, for example, used for configuration, life-cycle management, event support, and also for supporting of recursive contexts.

Once an application's components extend a specific component type, they automatically inherit the infrastructure mechanisms, the interfaces, and all the features supported by that component type. They are then left to declare the component's name, and all the component's interfaces by specifying (via an *InterfaceInfo* object), respectively: their name, their type (*SERVICE*, *DATA*, *EVENT*), and their class. To work properly, an object implementing the interface's class needs to be associated with each server-side interface. This can be done either explicitly or implicitly. In the former case, the method *wire(InterfaceInfo client, InterfaceInfo server)* in the context is called to assign a client-side interface to the reference of its server-side implementation. In the latter case, this is done automatically by the component context for each client-side interface of a newly installed component, by selecting the first server-side implementation compatible with the client within the context.

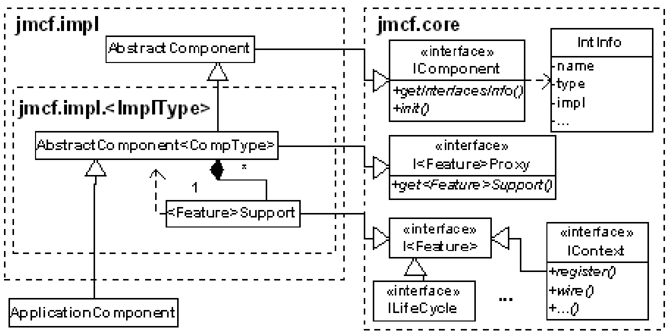

Figure 2 OOP Organization of the Java Modular Component Framework

In addition to the components that are logically part of agents, i.e. taking care of application-specific requirements, each platform also hosts a number of standard infrastructure-type components to handle non-functional concerns by interacting with the corresponding component types defined in the framework.

Basic Infrastructure Services: The most abstract implementations of JMCF components include interfaces enabling configuration, life-cycle management, component repair, and recursive contexts. For instance, if a component declares a *SelfTolerantComponent* interface, it is implicitly bound with a *ComponentRepair* component. This exports a ping() server-side interface, which is used to implement a basic health-monitoring service by: (i) automatically reloading every component that fails to call it for more than a given amount of time, and (ii) raising a *ComponentFailureEvent* within the context.

The *jmcf.impl.beans* package provides an implementation based on the Java-Beans standard, in which context functionalities are implemented using respectively: the *BeanContextServiceProvider* class from the java.beans package, while the *jmcf.impl.j2me* package implements the same functions availing of the the standard Container library. In order to provide a consistent interface to the agent layer, the other responsibility handled at this level is to smooth some of the differences between the different component technologies. For instance, the JMCF core allows the definition of recursive contexts by overriding the service brokering functionalities of contexts that do not natively support them, as in Java Beans. If the JMCF component is defined as composite, every interface declared by the components installed in its internal context, is also declared in the outside context.

Executors and Activities JMCF supports *ExecutorType* components to handle the concurrency concern in the component layer. Each context can have at most one active executor for scheduling active components. Contrary to passive components, which only respond to requests from other components, active components usually need to cyclically attend to their operations. Activity components adhere to the inversion of the control design pattern by subclassing the *ActivityType* component type. This exports a procedural interface that is periodically called by the the framework's executor. Such a design pattern, popular within component-based systems, represents a scalable alternative to multithreading and assists the development of code with high cohesion and low coupling that is easier to debug, maintain and test. In the case of the JMCF, the available implementations of the scheduling component are based either on the *java.util.Timer* class or on the *java.util.concurrent.ScheduledThreadPoolExecutor* class. Further specialisation of the these classes implement different scheduling strategies, such as round-robin or priority based.

Transport Protocols and Network Adapters Similar to what happens in the OSCAR component framework, JMCF uses network adapter components to connect Java client-side interfaces to server-side interfaces operating in remote component contexts. The *JMCF.impl.dist* package includes subpackages implementing a number of common interface adapters and converting the corresponding method calls to calls and messages to be exchanged through standard communication protocols, such as TCP/IP, UDP and HTTP, and popular Java middleware technologies, such as RMI and JMS. The built-in interface adapters are those for the pull (*Object pull()*) and for the push (*void push(Object)*) interfaces while generic adapters can be created at run-time, by availing of Java proxies. Correspondingly, there also exists components for managing the various available network protocols. *TCPTransport* is the component that manages the TCP transport protocol. On setup it configures the socket layer and exports the TCP-IP server socket to which all incoming TCP-IP connections connect.

After a connection is made, the client component (e.g. a *TcpPullClient* or a *TcpPushClient*) sends the name of the component it wants to speak with and the TCPTransport notifies the requested server-side component (e.g. a *TcpPullServer* or a *TcpPushServer*) of the request, by passing the socket it has just accepted.

4.2 Agent Layer

For the SoSAA agent layer, AF is used in tandem with AFAPL, a purpose built BDI-style Agent-Oriented Programming language that models agents as mental entities whose internal state consists of beliefs, goals and commitments. 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 addition to atomic actions, each agents can avail of a library of pre-compiled plans, i.e. procedural knowledge of the steps necessary to achieve given goals in given circumstances. The goal-based nature of these agents constitutes an important source of modularity, as designers can concentrate on writing plans for a subset of essential situations and can construct plans for more specific situations as they debug the system. Plans are invoked either in response to particular situations or based on their purpose, by availing of meta-reasoning heuristics used to select the best possible plan.

Execution of an AFAPL program involves the update of the agent's mental state by repeatedly applying (driven by the *AF Platform Scheduler Service*) an internal reasoning process that combines: (i) update of the agents beliefs via perception of the environment through a set of auxiliary Java components, known as perceptors; (ii) the adoption of new commitments through the evaluation of a set of PROLOG-like commitment rules, which map belief states onto commitments that should be adopted should that state arise; and (iii) the realisation of commitments by performing actions, which are implemented through a set of auxiliary Java components, known as actuators. Finally, the special actions *adoptGOAL(condition)*, *achieveGOAL(condition)* are used to post new goals and consequently trigger the nested activation of sub-plans with matching post-conditions, respectively in asynchronous and synchronous modality. In addition, *mantainGOAL(condition)* is used to define homoeostatic goals, that are automatically re-posted by the AF interpreter whenever they become unsatisfied.

The AF platform includes standard infrastructure agents, such as the *HealthMonitor* Agent, in charge of supervising the liveness of the agents in the system and cause their re-initialisation in case of detected failure, and a *DirectoryFacilitator* Agent, in charge of naming and brokering functionalities. In addition to application-specific agents, SoSAA adds a number of infrastructure-type agents to supervise or override the basic infrastructure mechanisms implemented in the component layer.

Executor Agent: An executor agent in SoSAA can be used to supervise each executor component used in the component layer. This executor agent monitors the state of the activities running in the corresponding context, as well as the frequency of their execution. The underlying executor component may be configured to use a thread pool to inject control in the active components installed on that context. If at run-time one activity requires more CPU, the agents can negotiate the urgency of that activity among themselves.

TransportManager Agent: A similar overriding mechanism is used to enable dynamic data distribution in SoSAA systems. While the component context's brokering system can be used to connect component's interfaces within the same context, a *TransportManager* agent must negotiate (at run-time) the actual transport mechanism employed for interface distribution based on the capabilities of each node involved in a remote communication. The *TransportManager* is activated by application agents through a set-connection request carrying the details of both the local and the remote interface, whenever a need for their connection appears in the application layer. This request is done by way of an ACL (Agent Communication Language) message. Once a *TransportManager* has received such a request from an application agent, it contacts corresponding *TransportManager* agent in the remote platform to exchange information about the network adapters supported by each platform. Once they agree on a common transport mechanism, both agents (i) install the interface adapters for the interface to be wired (respectively a client and a server-side adapter), and (ii) bind it with the respective functional component. In addition, the *TransportManager* on the client-side also configures its network adapter with the details necessary to contact the server's node (e.g. by passing its RMI/JMS reference or a network address). Once a connection between functional components is established, they can use their interfaces exactly as if they were in the same process. Besides the initial connections, the other important responsibility of the *TransportManager* agents is to try to maintain the connection between components despite network disruptions. This is done by exploiting AFAPL's goal handling mechanism.

The TransportManager and the DirectoryFacilitator agents are re-used in all applications. In contrast, the Executor and the HealthMonitor agents offer standard behaviours that often need to be adapted to the specific application. In this case, AF offers customisable points in the form of agent roles, that need to be implemented by the application developer and associated with these agents, e.g. to implement particular priority mechanisms and component re-booting policies.

4.3 Summary

Figure 3 summarises all the modules (agents and components) discussed in the previous section, distinguishing between infrastructure-type modules and application-dependent modules. It also highlights the interaction mechanisms used within SoSAA and, in particular, those used to decouple modules across the infrastructure/application boundary. In the component layer, basic mechanisms dealing with communication, concurrency, performance, scalability, and naming concerns are handled through dedicated infrastructure-type components, while application components handle all the functional concerns. In the agent layer, a number of agents delivered with SoSAA augment the basic infrastructure mechanisms implemented in the component layer while infrastructure agents delivered with the standard AF platform take care of naming and fault-tolerance concerns. The platform also includes application-specific modules addressing system adaptation, in the form of agent roles involving both ACL dialogue plans and coordination protocols. However, these modules need to be instantiated within application-dependent agents, which are ultimately in charge of adaptation, scalability and performance concerns. Both infrastructure and application agents interact with their respective component counterparts via the SoSAA Adapter. Infrastructure agents interact via ACL, which is also used between application agents and the standard modules delivered in the AF platform. However, the interaction between application agents and SoSAA infrastructure agents is supported through the plan/goal deliberation. Application agents post goals that are then achieved by infrastructure agents aware of the necessary technical details. In the component layer, infrastructure and application components interact via standard components' interfaces (messages, events, and procedure calls). However, these interfaces and their associated interactions are hidden in the implementation of the JMCF component types that are sub-classed by the application components. Together, the AF plan/goal deliberation and the JMCF organisation in component types enable the decoupling between application and infrastructure concerns, both at run-time and design-time, as the application developers can just focus on application-dependent development.

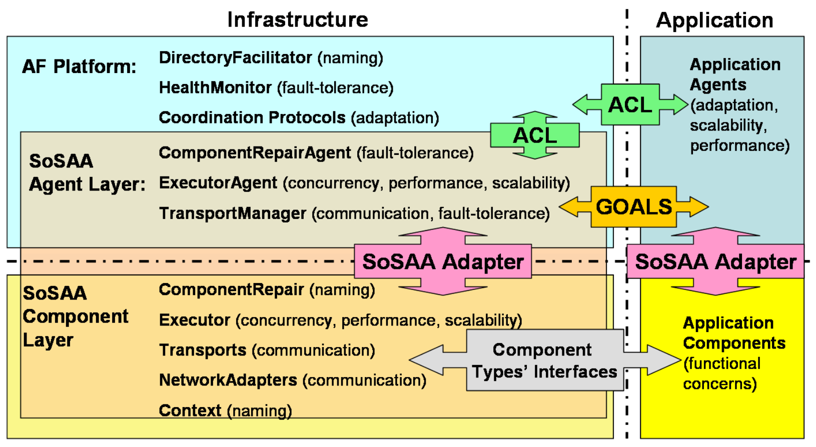
Infrastructure
Application
AF Platform:
DirectoryFacilitator (naming)
HealthMonitor (fault-tolerance)
Coordination Protocols (adaptation)
ACL
ACL
Application Agents (adaptation, scalability, performance)
SoSAA Agent Layer:
ComponentRepairAgent (fault-tolerance)
ExecutorAgent (concurrency, performance, scalability)
TransportManager (communication, fault-tolerance)
GOALS
SoSAA Adapter
SoSAA Adapter
SoSAA Component Layer
ComponentRepair (naming)
Executor (concurrency, performance, scalability)
Transports (communication)
NetworkAdapters (communication)
Context (naming)
Component Types' Interfaces
Application Components (functional concerns)

Figure 3 Separation and Decoupling between concerns in SoSAA

5 The HOTAIR Case Study

To illustrate the effect of SoSAA on the separation of concerns, and provide an initial evaluation of its application to the design of MASs, this section presents a case study of the HOTAIR (Highly Organised Teams of Agents for Information Retrieval) system. HOTAIR is a large-scale Information Retrieval (IR) (i.e. search) engine, built as a MAS, and described by Peng et al. (2005). Specifically, we compare an old implementation of HOTAIR, which did not take advantage of SoSAA, with its new SoSAA-based implementation.

5.1 Methodology

A procedure for measuring separation of concerns in software has already been established by Sant'Anna et al. (2007). However, this is recent work, and suitable thresholds ('good' and 'bad' value regions) for these metrics have yet to be identified. We therefore adopt the benchmarking methodology proposed by Vaishnavi & Kuechler (2007)[p.167], and compare two 'solutions' (one using SoSAA, the other without) in a particular 'scenario' (the HOTAIR application).

HOTAIR was selected as an example of an application that can benefit from the separation of concerns afforded by a hybrid agent/component integration approach. However, the way modularity is achieved in HOTAIR is representative of a large class of multiagent applications, for which the application developer needs to provide application-specific constructs, such as different sets of plans and roles, to address all the functional requirements of the application. The main limitation of comparing just two architectures in one scenario is that our results do not necessarily generalise to other development contexts: further work to establish wider applicability may be required in future.

Table 1 Metrics for Architectural Separation of Concerns

Question	Measured Entity	Measured Attribute	Metric Definition	Unit
1(a)i	Concern	Concern Diffusion over Architectural Elements (CDAE)	Number of architectural elements which contribute to the realization of this concern	Elements
1(a)ii	Concern	Architectural Interlacing Between Concerns (AIBC)	Number of other concerns which share at least one architectural element with this concern	Concerns
1(b)i	Element	Afferent Coupling Between Elements (AC)	Number of elements which require service from this element	Elements
1(b)ii	Element	Efferent Coupling Between Elements (EC)	Number of elements from which this element requires service	Elements
1(c)i	Element	Lack of Concern-based Cohesion (LCC)	Number of concerns addressed by this element	Concerns

From the suite of metrics proposed by Sant'Anna et al. (2007), we select those appropriate to our scenario using the Goal/Question/Metric (GQM) measurement approach of Basili et al. (1994). In this approach, the overall goal of the study is first identified, then a set of questions to address that goal are defined. These questions are then progressively refined until they can be answered by direct measurement of the artifact using appropriate metrics.

For each version of the HOTAIR system, we ask the following questions:

1. How are the important concerns of the HOTAIR system allocated among its architectural elements?
   (a) Where is each concern realised?
      i. Over how many architectural elements is this concern scattered?
      ii. How many other concerns are also realised by the same architectural element(s)?
   (b) How tightly coupled is each architectural element?
      i. How many other elements require service from this element?

   ii. To how many other elements does this element provide service?
   (c) How cohesive is each architectural element?
      i. How many different concerns are realised by this element?

Table 1 summarises the metrics selected to answer the identified questions, modified for hybrid component-agent systems as follows. The systems under comparison are composed of roles, plans, components, and platform modules - elementary units developed to address the application's functional requirements and access infrastructure services - and we make no distinction between these categories at the architecture level. For this reason, we replace the term "component" with the more general "element", to indicate any of these units of development. For all these metrics, a lower value indicates a higher degree of separation and decoupling of concerns. All have a minimum value of zero, except CDAE which has a minimum value of one.

In order to calculate values of these metrics, we require:

1. the architectural elements of the application, and the relationships between them, to be identified; and
2. the concerns of the application to be known and clearly defined.

5.2 Architectural Analysis

The goal of HOTAIR is to maximise the throughput of documents that are processed by the system and stored in a common index supporting subsequent queries. Compared to a more straightforward component-based implementation, HOTAIR's multiagent nature accounts for an open computational environments, where agents and nodes must be capable of being added and removed at run-time, and all available computational resources are leveraged opportunistically by reacting to the unpredictable content of the files processed through the system. Firstly, this is achieved by encapsulating each of the processing stages in a specific *agent type*. In particular, (i) *Data Gatherer Agents* (web-crawlers) identify and download documents from the Internet, (ii) *Translator Agents* translate these documents (parsing their content, extracting hyperlinks, and reducing them to to a common format), and (iii) *Indexing Agents*, store the resulting documents into the index. Secondly, ACL-based coordination is used to maximise system throughput, by changing the number, the distribution, and the collaborations path-ways between the agents instantiated in the system, while also reacting to run-time performance and unforeseen circumstances, such as failure and platforms joining or leaving the system. For instance, as each data producer (i.e. gatherer and translators) writes its data to a queue, it broadcasts (via ACL over UDP) the length of this queue across the system. Whenever a consumer agent (i.e. translators and indexers) needs more documents to process, (i) it decides where it wishes to take documents from, (ii) makes a request to that effect to the selected producer, and (iii) then reads the data available from the producer's queue. Each agent has its own view of the state of the system, which is used to decide where documents are to be taken from. Each agent makes this decision with only itself in mind and reassigns itself when the queue it has been using becomes empty. In addition, a Performance Manager agent also exists to maintain a view of the overall state of the system and attempt to optimise the system for maximum throughput. It has the power to request that agents take their documents from specific agents, so as to balance the load throughout the system.

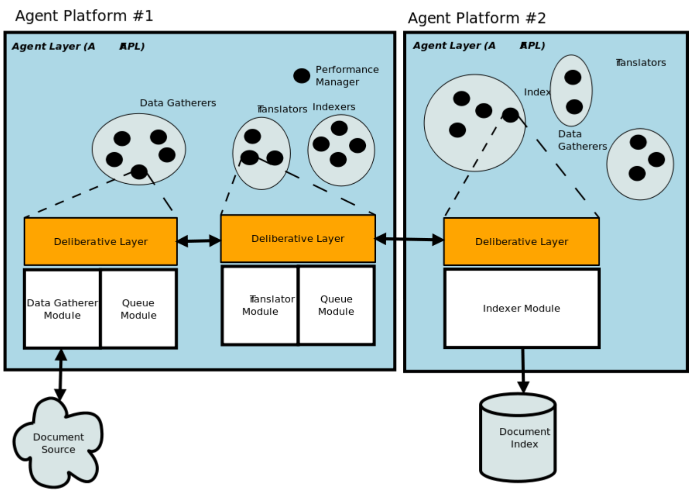

Figure 4 HOTAIR Architecture before SoSAA

The main difference between the two implementations (which were detailed by Lillis et al. (2009)), is that in the pre-SoSAA HOTAIR, application functions were carried out by object-oriented modules encapsulated by AF actuators and perceptors, without the stratified approach introduced with SoSAA. Before the introduction of SoSAA, all functions of agents were at the AFAPL level. This architecture is illustrated in Figure 4. This shows an example of two agent platforms (though the number of platforms is not limited), each hosting groups of Data Gatherer, Translator and Indexer agents. The broadcast messages are done using the FIPA ACL over UDP. Once an agent has decided where it wishes to take documents from, it then switches to direct ACL communication using TCP connections. Once two agents have agreed to cooperate, further communication remains on the ACL level, unlike in the SoSAA-based architecture. Whenever an agent needs more documents to process, it makes a request to that effect. The agent providing the documents responds with the list of documents to process.

Figure 5 shows the architecture of the modified version of HOTAIR designed to make use of SoSAA. Like the pre-SoSAA version, this shows an example of two agent platforms (though the number of platforms is not limited), each hosting groups of Data Gatherer, Translator and Indexer agents. These are shown in the Agent Layer at the top of each platform. The Performance Manager is also present. For the SoSAA-based version, a Transport Manager agent is also present on each platform. This agent is responsible for setting up cross-platform communication links at the component layer, which will be discussed below.

One agent of each type is magnified to show its internal structure. Each has a deliberative layer, written in AFAPL, which fulfils many of the same functions as in the non-SoSAA architecture. This includes broadcasting the queue status

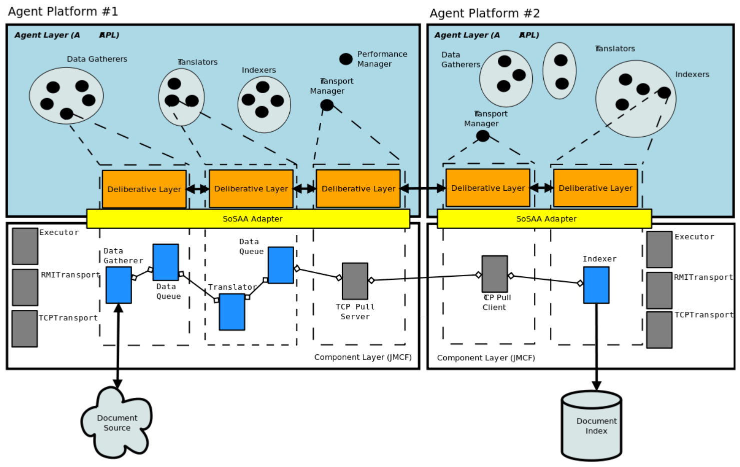

Figure 5 HOTAIR Architecture using SoSAA

and negotiating with other agents to arrange an association for the processing of documents. All communication at this level is still done by means of ACL.

Below this deliberative layer lies the SoSAA Adapter, to expose the component layer to the agents. The principal difference between the non-SoSAA architecture and the SoSAA-based architecture is that lower-level functions are moved from the agent layer to the component layer. In the context of HOTAIR, these low-level functions are the actual processing of documents and the transfer of documents between agents. In Figure 5, dashed lines in the component layer indicate the logical grouping of components with the agents whose functions they carry out.

Each agent that acts as a provider of documents for others has two components performing its low-level functions. One component (with the same name as the agent itself) performs the actual processing of documents and these are then passed to a Data Queue component. The Data Queue maintains the list of documents that have been processed but have not yet undergone the next stage of processing. Indexers do not have Data Queue components, as no further processing is required after this stage.

When the agents have negotiated a working relationship, their appropriate components are wired together. In Agent Platform #1, the Translator and Data Gatherer agents show how this is done when the agents are hosted on the same agent platform. Here, the processing component (a Translator in this case) is wired to the Data Queue of the relevant agent. Documents can continually be fetched from the queue until the deliberative layer decides otherwise.

The situation is more complex when two agents on different platforms wish to collaborate. This is shown in the relationship between the Translator agent on Agent Platform #1 and the Indexer on Agent Platform #2. Here, intermediate components must be introduced that are capable of communicating using network protocols. In Figure 5, this is illustrated by the TCP Pull Server and TCP Pull Client components that bind the Indexer to the appropriate Data Queue. In this configuration, the Indexer can pull documents from the TCP Pull Client as if it were connected directly to the Data Queue.

In the example shown, the Transport Manager on Agent Platform #1 sets up a TCPPullServer that transparently provides a TCP interface to the Data Queue component of the Translator agent. On the other side, the Transport Manager on Agent Platform #2 creates a corresponding TCPPullClient, which allows the Indexer agent gain access to the appropriate documents. The Transport Manager agent is described in more detail by Dragone et al. (2009*a*).

5.3 HOTAIR Concerns

This section highlights the application-specific functional concerns in HOTAIR in order to separate them from concerns addressed through infrastructural services (both agent and component based) and thus help estimating the separation of concerns likely to be achieved by applying the framework in other applications.

While we do not claim to be exhaustive, we centre our initial evaluation on the non-functional concerns of distributed applications discussed in Section 2. However, that section reported generic definitions for these concerns, which did not capture the stratification advocated by Atkinson & Kuhne. (2000) and Silva et al. (1995), and which is implemented within SoSAA for all the concerns considered. Grounding a model/policy/mechanism architecture into (i) our hybrid framework and (ii) a specific application, helps with this step, as we are now in a position to describe - although not in a rigorously formal manner - each non-functional concern at a different level of abstraction. We first distinguish the concurrency concern by considering the concurrent execution of the agent deliberation functionalities, which are responsible for the goal-based deliberation and symbolic coordination at the model level of our architecture, from the concurrent execution of application functionalities. In the old HOTAIR, these activities were synchronously driven by the scheduler service installed in the AF platform, with the crucial drawback that the function's execution was subjected to the latency dictated by slow symbolic manipulation techniques used for agent reasoning. The SoSAA implementation successfully managed to decouple the two types of executions by introducing a separate scheduler for the application activity components, as discussed in Section 5.2. Next, we distinguish between three levels of failure: failure of deliberation, failure of data transfer, and failure of application functionalities. While nothing has changed at the deliberative level, the SoSAA implementation has lightened the overall responsibility of the application agent, by using the TransportManager to handle failure at the data transfer level, and the ComponentRepair Component, to handle failure at the component layer. Next, we distinguish between communication deliberation, that is the management of communication models, with the actual data transfer. While the non-SoSAA implementation addressed all the concerns in application agents, by transmitting data via ACL, the SoSAA implementation leaves the agent to deal with the model level, and introduces the TransportManager agent and the Network Adapters Components to deal respectively with the policy level and the mechanism levels. Finally, we do not include either the adaptation concern, as it is addressed by application agents in both implementations, nor the naming concern, as although

Table 2 Locations of Concerns

	Pre-SoSAA	With SoSAA
Concurrency:Deliberation	AF Scheduler$_P$	AF Scheduler$_P$
Concurrency:Function	AF Scheduler$_P$	Executor$_C$
Failure:Deliberation	Health Monitor$_A$	Health Monitor$_A$
Failure:Function	$C\times$Application$_A$	Component Repair$_C$
Failure:Data Transfer	$C\times$Application$_A$	Transport Manager$_A$
Communication:Deliberation	$C\times$Application$_A$	$C\times$Application$_A$
Communication:Policy	$C\times$Application$_A$	Transport Manager$_A$
Communication:Mechanism	$C\times$Application$_A$	Network Adapter$_C$

Table 3 Concern Metrics in HOTAIR Versions ($C = 3$)

	Pre-SoSAA		With SoSAA	
	CDAE	AIBC	CDAE	AIBC
Concurrency:Deliberation	1	1	1	0
Concurrency:Function	1	1	1	0
Failure:Deliberation	1	0	1	0
Failure:Function	C	4	1	0
Failure:Data Transfer	C	4	1	1
Communication:Deliberation	C	4	1	0
Communication:Policy	C	4	1	1
Communication:Mechanism	C	4	1	0

SoSAA handles naming of functions via the component context, function and agents are strictly associated in both HOTAIR implementations.

Table 2 shows the architectural elements that implement these concerns in both versions of HOTAIR. We use subscripts A, C, and P to denote the type of each element: agent, component, or platform module. For simplicity, only architectural elements that exhibit significant differences between the two versions are shown.

In order to aid generalisation to different applications, the Data Gatherer, Translator and Indexer agents have been grouped in a single "application agent" category; each agent is counted as a single element. To emphasise how the metric values depend on the complexity of the application, we introduce the constant C, the value of which is 3 for HOTAIR. C counts the number of application-specific elements in the design (not the number of the actual agents deployed, which does not affect the development effort).

5.4 Evaluation

Performance and fault-tolerance improvements related to the SoSAA implementation of HOTAIR have been described by Dragone et al. (2009*b*,*a*). In this section, we quantitatively compare the separation of concerns in the SoSAA and non-SoSAA HOTAIR architectures.

Tables 3 and 4 show the values of the five selected metrics, calculated manually by inspecting the source code, architecture diagrams, and data shown above. The calculated values are minima, since for clarity we have only considered a subset of the HOTAIR application; the implementations have more architectural elements and other concerns.

The values for the CDAE and AIBC metrics in Table 3, and the LCC metric in table 4 can be computed directly from the locations of concerns in Table 2.

In calculating AC and EC, we only considered dependencies that need to be addressed by the application developer, and ignored dependencies that are resolved

Table 4 Architectural Element Metrics in HOTAIR Versions

	Pre-SoSAA			With SoSAA		
	AC	EC	LCC	AC	EC	LCC
AF Scheduler$_P$	0	0	2	0	1	1
Health Monitor$_A$	0	0	1	0	1	1
Gatherer$_A$	1	0	5	1	0	1
Translator$_A$	1	1	5	1	1	1
Indexer$_A$	0	1	5	0	1	1
Executor$_C$	-	-	-	0(1)	0(1)	1
Component Repair$_C$	-	-	-	0(1)	0(1)	1
Transport Manager$_A$	-	-	-	0(2)	0(2)	2
Network Adapter$_C$	-	-	-	0(1)	0(1)	1

transparently by the framework. For instance, application agents do not deal directly with the AF Scheduler, which drives their deliberation transparently; thus for the pre-SoSAA version, AC and EC in the first row of Table 4 are 0. However, in the SoSAA implementation, the developer may extend the behaviour of the AF Scheduler, by adding plans to control the priority mechanism implemented in the Executor component. The same is true for the Health Monitor: in the SoSAA version, the developer can include application specific strategies for component repair.

There is little difference in the values of AC and EC metrics for the three types of application agents (Gatherer, Translator, Indexer). However, in the pre-SoSAA version, these agents handle all the concerns addressed by the new architectural elements in the SoSAA-based implementation, which explains the significant differences in the LCC values.

In the SoSAA-based implementation, all the dependencies between the new infrastructure elements are handled transparently within the framework, availing of the component, and agent-based de-coupling mechanisms summarised in Figure 3. As a result, all AC and EC metrics are null for the new elements. However, the same rows also report these dependencies within brackets, indicating the likely effort necessary to modify the current architecture.

The application developer does not have to worry about the Transport Manager agent, as the application only deals with communication goals, which are automatically resolved via goal-plan deliberation in Agent Factory. However, the TransportManager entertains relationships with the managers located in other platforms, and also with the component adapters. The latter are pure infrastructure elements, which acts as client (server) to a partner adapter, and server (client) toward an application component.

With the introduction of SoSAA to the HOTAIR architecture, the values for the concern-based metrics CDAE, AIBC, and LCC show significant improvement. Noticeably, we observe how these improvements would be even more significant in case of more complex applications developed with the same model of HOTAIR but with more application elements (i.e. C greater than 3).

For HOTAIR, SoSAA both reduces the scattering and increases the cohesion of the selected concerns. This comes at the expense of slight increases in the coupling between elements, as evidenced by the values of the coupling-based metrics AC and EC - increases which are partly due to the larger number of architectural elements present in the second design. Again, however, this does not reflect in added development effort for the application developer, while presenting clear benefits in terms of efficiency, as demonstrated by Dragone et al. (2009*a*,*b*).

6 Conclusion and Future Work

This paper has discussed the motivations and illustrated how modularity constructs available in both CBSE and AOSE can be hybridised to provide for various levels of separation of concerns. We have illustrated this through SoSAA and shown that: (i) 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; (ii) the particular realisation of the component framework used in SoSAA defines a stratified approach to separation of concerns by capturing commonly agreed mechanisms among modern component technologies and by shielding the agents from the underlying technological aspects; (iii) SoSAA provides different levels of abstraction for system analysis and design; and (iv) by integrating and promoting the interplay between methods that are commonly accepted within AOSE and CBSE, SoSAA provides a clear architectural framework to address cross-cutting concerns while preserving system modularity.

Since SoSAA is only one example of a hybrid integration technology, and HOTAIR is just one scenario, the results shown here are not generalizable to other frameworks and applications. However, this study shows how SoSAA's two-layer design captures widely-agreed strengths of the CBSE and AOSE domains. SoSAA also promotes a separation between business logic, service orchestration/choreography, and service details, similar to that of BPEL, suggesting that these results could be easily replicated using different component/service and agent-enabling technologies.

Our previous work has demonstrated the effectiveness of SoSAA-based implementations in limited case studies (see Dragone et al. (2009*a*,*b*)). The scalability of similar hybrid approaches has been demonstrated by other works, e.g. the integration of heterogeneous communication mechanisms with the hybrid backchannel management in the RETSINA MAS of M. Berna-Koes & Sycara (2004). However, the capability of our framework to enable large scale, open, heterogeneous and dynamic environments, such as those targeted in HOTAIR, remains an open question.

In practice, the advantages of a hybrid component/agent technology such as SoSAA should be balanced against the cost of its introduction. SoSAA requires developers to be familiar with both agent-oriented and component-based development paradigms, and thus has a steep learning curve; it also adds extra run-time complexity. In very small systems, and in rapid prototyping environments where systems may have short lifecycles, these disadvantages may outweigh the benefits.

Future work will focus both on a rigorous formalisation of the mapping of a model/policy/mechanism stratified approach to the requirements of our test-bed application HOTAIR, and on testing our approach in a range of diverse applications in the pervasive computing domain.

This work has shed some light on the difficulty of evaluating separation of concerns for multiagent systems, especially those of a hybrid agent/component design, as in SoSAA. Our evaluation required us to translate currently accepted metrics to our domain, and also make some assumptions regarding how to count architectural elements. These assumptions will need further investigation and validation, in particular by conducting more extensive empirical studies (following the example of R. Burrows & Taiani (2006), Kulesza (2006), Garcia et al. (2004)) of the maintainability of hybrid, pure agent, and pure component-based solutions.

References

Aksit, M., Bergmans, L. & Vural, S. (1992), An object-oriented language-database integration model: The composition-filters approach, *in* 'Proceedings of ECOOP', Vol. 92, Citeseer, pp. 372–395.

Atkinson, C. & Kuhne., T. (2000), Separation of Concerns through Stratified Architectures, *in* 'Proc. International Workshop on Aspects and Dimensions of Concerns (ECOOP. 200)', Citeseer.

Basili, V., Caldiera, G. & Rombach, H. (1994), 'The goal question metric approach', *Encyclopedia of software engineering* **1**, 528–532.

Collier, R., O'Hare, G., Lowen, T. & Rooney, C. (2003), '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* .

Dragone, M., Lillis, D., Collier, R. W. & O'Hare, G. M. P. (2009*a*), Practical Development of Hybrid Intelligent Agent Systems with SoSAA, *in* 'Proceedings of the 20th Irish Conference on Artificial Intelligence and Cognitive Science', Dublin, Ireland.

Dragone, M., Lillis, D., Collier, R. W. & O'Hare, G. M. P. (2009*b*), 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.

Garcia, A., Sant'Anna, C., Chavez, C., da Silva, V., de Lucena, C. & von Staa, A. (2004), 'Separation of concerns in multi-agent systems: An empirical study', *Software Engineering for Multi-Agent Systems II* pp. 343–344.

Gat, E. (1992), Atlantis: Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling realworld mobile robots, *in* 'In Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)', pp. 809–815.

Harrison, W. & Ossher, H. (1993), 'Subject-oriented programming: a critique of pure objects', *ACM Sigplan Notices* **28**(10), 411–428.

Kulesza, U. e. a. (2006), Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study, *in* 'In Proc. ICSM (2006) 223-233'.

Landuyt, D. V., Jackson, A., de beeck, S. O., Grgoire, J., Scandariato, R., Joosen, W. & Clarke, S. (2007), Aspectual vs. component-based decomposition: A quantitative study, *in* 'Iin the Workshop on Aspects in Architectural Description (AARCH) at AOSD 2007'.

Lillis, D., Collier, R. W., Dragone, M. & O'Hare, G. M. P. (2009), 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.

M. Berna-Koes, I. N. & Sycara, K. (2004), Communication Efficiency in Multi-agent Systems, *in* 'in Proceedings of ICRA 2004. New Orleans, LA. April 26-May 1'.

Melliar-Smith, P., Moser, L. & Narasimhan, P. (1997), Separation of Concerns: Functionality vs. Quality of Service, *in* 'Proceedings of the 3rd Workshop on Object-Oriented Real-Time Dependable Systems-(WORDS'97)', IEEE Computer Society, p. 272.

Mili, H., Dargham, J., Cherkaoui, O., Godin, R. & Mili, A. (1999), View Programming for Decentralized Development of OO Programs, *in* 'Proceedings of the Technology of Object-Oriented Languages and Systems', IEEE Computer Society Washington, DC, USA.

Mili, H., Elkharraz, A. & Mcheick, H. (2004), 'Understanding separation of concerns', *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design* pp. 75–84.

Peng, L., Collier, R., Mur, A., Lillis, D., Toolan, F. & Dunnion, J. (2005), 'A self-configuring agent-based document indexing system', *Multi-Agent Systems and Applications IV, Lecture Notes in Artificial Intelligence* **3690**, 624–627.

R. Burrows, A. G. & Taiani, F. (2006), Coupling Metrics for Aspect-Oriented Programming: A Systematic Review of Maintainability Studies, *in* 'Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)'.

Reina, A. & Torres, J. (2004), 'Components+ Aspects: A General Overview', *Revista Colombiana de Computación. Publicado por la Universidad Autónoma de Bucaramanga* **5**(1), 77–95.

Ricci, A., Viroli, M. & Omicini, A. (2007), 'CArtAgO: A framework for prototyping artifact-based environments in MAS', *Lecture Notes in Computer Science* **4389**, 67.

Sant'Anna, C., Figueiredo, E., Garcia, A. & Lucena, C. (2007), On the Modularity Assessment of Software Architectures: Do my architectural concerns count, *in* 'Proc. International Workshop on Aspects in Architecture Descriptions (AARCH. 07), AOSD', Vol. 7, Citeseer.

Shoham, Y. (1993), 'Agent-oriented programming', *Artificial intelligence* **60**(1), 51–92.

Silva, A. R., Silva, A. O. R., Sousa, P., Marques, J. A. & N, R. A. R. (1995), Development of distributed applications with separation of concerns, *in* 'In IEEE Asia-Pacific Software Engineering Conference', IEEE Computer Society Press, pp. 168–177.

Szyperski, C. (1998), 'Component Software: beyond object-oriented software', *Reading, MA: ACM/Addison-Wesley* .

Vaishnavi, V. & Kuechler, W. (2007), *Design science research methods and patterns: innovating information and communication technology*, Auerbach Pub.

Weyns, D., Omicini, A. & Odell, J. (2007), 'Environment as a first class abstraction in multiagent systems', *Autonomous agents and multi-agent systems* **14**(1), 5–30.

Zambonelli, F. & Omicini, A. (2004), 'Challenges and Research Directions in Agent-Oriented Software Engineering', *Autonomous Agents and Multi-Agent Systems 9(3): 253-283 (2004)* **9(3)**, 253–283.