Augmenting Android with AOSE Principles for enhanced Functionality Reuse in Mobile Applications

Christopher Frantz, Mariusz Nowostawski, Martin K. Purvis
{cfrantz, mnowostawski, mpurvis}@infoscience.otago.ac.nz

Department of Information Science, University of Otago, New Zealand

Abstract. The Android platform has popularized and caused a widespread adoption of its application development approach based on loosely coupled application components. This loose coupling allows for a flexible composition of applications but also enables invocations and reuse of individual components from third-party applications.
One problem with the original Android design is that the rather coarse-grained application components themselves prohibit more fine-grained decomposition. To provide more flexible loosely coupled components and also to foster the reuse of more elementary fine-grained functionality, we suggest the extension of Android application components with our concept of μ-agents. Moreover, the organisational aspects of the μ-agent model introduce means to structure functionality in a more systematic manner.
In this article, we introduce our platform concept called *Micro-agents on Android* (MOA) that realizes the integration of application development principles with a lightweight notion of agency. Several scenarios are used to describe the benefit from functionality reuse across applications enabled by MOA. A performance evaluation demonstrates that μ-agents interact in a more efficient manner than Android services, making them well-suited for fine-grained decomposition.
Our approach serves as an example showing how existing technology can benefit from utilizing the modelling advantages of agent-based technologies.

Keywords: μ-agents, micro-agents, multi-agent systems, mobile applications, agent-oriented software engineering, functionality reuse, agent organisation, android, MOA, intents

1 Introduction

With the increased adoption of smartphones, the continuous trend towards ubiquitous computing has reached the mainstream of users. Smartphones combine the abundance of available sensors (e.g. GPS, compass, accelerometer, gyroscope, light and temperature) with the Internet. The perceived 'smartness' of those devices and their applications derives more from the combination of those different information sources than from particularly intelligent features. Operating systems for smartphones cater for these application characteristics and support notions of loose coupling as well as aspects such as intentionality. While early generation smartphones provided environments to develop monolithic applications, modern mobile operating systems emphasize multi-threading, loose coupling of application parts, along with the use of a wider range of information sources.

One system that has adopted some notions of intentionality is Android [8]. In fact the Android infrastructure and architectural design has a fair degree of similarity with multi-agent systems.

Android provides so-called *application components* that serve as runtime containers for particular behavioural patterns (e.g. running in foreground, running in background, event subscription) that can be combined to realize complex applications.

But beyond those well-specified mechanisms on how to compose applications from application components, Android does not provide distinct mechanisms to organise or manage functionality on lower levels. As a result, coarse-grained application components can combine a wide range of functionality in an application-dependent manner, which limits the reusability of functionality subsets, i.e. the application components as a whole might be accessible but the more elementary functionality is not. This coarse-grained model prohibits the use of functionality beyond the application component level across different applications. To leverage the potential for better application reuse and organisation, while being confined to the processing constraints of mobile systems, we propose the integration of the computationally efficient notion of μ-agents with the Android architecture.

In this article, we first introduce the Android application development principles, followed by the description of our μ-agent concept and its implementation. The concept of **M**icro-agents **o**n **A**ndroid demonstrates how those two technologies can be interlinked, and Android applications be seamlessly backed by a fine-grained cross-application μ-agent organisation. Different scenarios are outlined to exemplify how functionality reuse can be improved using μ-agents.

2 Android and the Concept of μ-agents

2.1 Android architecture and application components

Android [10], developed by Google in collaboration with the Mobile Handset Alliance and released as an open source software platform, is increasingly adopted by smartphone manufacturers. Beyond a Linux-based kernel and the device-specific hardware drivers, it offers a comprehensive software stack of libraries centered around the Dalvik Virtual Machine. Dalvik operates like the Java Virtual Machine (JVM) and provides its own runtime libraries. Semantically related library functionality is controlled via so-called managers (e.g. LocationManager for all location-related functionality) where useful. At the top layer of functionality, applications access both the various managers and library functionality using

Java syntax. A comprehensive insight into the different architectural layers of Android is provided under [10].

The interesting aspect from an architectural point of view is the way applications are composed. Android caters for a concurrent and loosely coupled layout of applications by providing the following application components:

- *Activities* are designed for rather short-running functionality with direct user interaction. Multiple activities can be combined to provide more comprehensive functionality such as wizards.
- *Services* in contrast are designed to be long-running in the background.
- *Broadcast Receivers* are instantiated upon registered (system or application) events and execute a particular behaviour and are destroyed after execution.
- *Content Providers* serve as an abstraction layer for system-wide access to particular persistent storage locations.

All those components (with the exception of content providers) are connected via, so-called, *intents* which represent abstract request specifications, have a unified structure, and allow asynchronous messaging between the aforementioned application components. Intents allow either explicit addressing of target components (by class name) or implicit addressing by matching intent characteristics, such as action (e.g. VIEW to open a viewer application), handled data type, further component-related attributes, or categories (e.g. PREFERENCE indicating that the component is a 'Preferences' panel) against application characteristics which are registered by individual application components (as *intent filters*). Further, the content of intents can be arbitrarily defined by the application developer and is attached as *extras* maintained in a dynamically typed map data structure.

Table 1 shows the structure of Android intents.

Table 1. Description of Android Intent Structure

Android Intent Property	*Description*
Action	Action to be performed, e.g. VIEW, PICK
Data	URI representation of data type and payload to be processed (in conjunction with corresponding action field value), e.g. tel://123454 as telephone number
Category	Specification of the type of component to handle intent, e.g. CATEGORY_BROWSABLE
Extras	dynamically typed map structure holding data specified by developer
Flags	Specification of component behaviour once raised, e.g. always starting new component instance or reusing existing one

The composition of the particular application components and their intent-based interaction are the building blocks for any Android-based application. This also includes core applications such as the caller application, thus giving the application developer the power to access a wide range of system-level functionality. Further information on those architectural principles of Android can be found under [10].

2.2 Comparing Android Application Development Principles with Multi-agent Systems

We consider the Android development approach to share various characteristics of that of multi-agent system development, and this has motivated us to augment Android with some of our μ-agent platform features. The key characteristics in this connection are:

– *Loose coupling* – Application components in both Android and MAS are loosely coupled. The Android coupling mechanisms are equivalent to the addressing mechanisms employed with MAS, as specified by FIPA (see [6]), where target services or activities can be directly addressed by name (i.e. explicit intents) as well as indirectly, using implicit intents, which work similar to a yellow-page lookup. The explicit definition of applications in Android is effectively done in the application manifest; its actual components always rely on runtime binding.
– *Asynchronous communication* – Asynchronous communication plays a key role in both MASs and Android. It is the basis for autonomy in agent systems and a means to participate in concurrent conversations. With respect to Android, asynchronous messaging is used as the default mechanism for inter-component communication.
– *Decentralization* – This is a core principle in both Android and MASs. Although the underlying platform can be controlled to a certain extent, the actual components are not centrally controlled in Android, and can only be activated using intents. Both MAS agents and Android application components have clearly defined lifecycles.
– *Intentionality* – Software notions of intentionality are in line with the notion of the intentional stance (see Dennett [5]). While intentions in MAS represent a concept used for practical reasoning in rational agents, in Android they express the interest of a component to invoke another (unknown) application component (which might succeed or fail at runtime).
– *Open system* – Given the implicit intent resolution mechanism, the Android system is generally open (analogous to MAS) towards newly added applications whose specified intent filters might affect future bindings.

In summary, Android and MAS share the principles of loose coupling, asynchronous communication and concurrency application layout enabled by the decentralized system architecture. Taking the combination of those elements into account, Android and MAS can be similarly non-deterministic; and further, Android's technical foundations allow its use as an infrastructure for the construction of actual MAS systems. However, Android is not a multi-agent system on its own:

– Autonomy – Although all application components in Android define a life cycle and are asynchronously invoked, they do not exhibit characteristics of advanced autonomy in the shape of proactive behaviour or goal-directedness.
– Interaction – Application components in Android only engage in simple request-response interactions and do not provide facilities to model or maintain extended conversations (which are typical for multi-agent systems).

The use of pure Android interaction mechanisms to model a multi-agent system is not practical, especially with "intelligent agents" in mind. However, the integration of Android with other MAS implementations is of interest, since it can offer the advantages of a mobile platform and can facilitate agent-oriented software engineering on such a platform. We think that the notion of μ-agents is particularly suited for this purpose.

2.3 The μ-agent Concept

To provide a context for the suggested augmentation of Android with agent-based technology, we provide below our proposed solution. It relies on the notion of μ-agents and describes their surrounding principles.

μ-agents are derived from the notion of the intelligent software agent concept. As such, the μ-agent concept inherits characteristics such as executional autonomy, and reactivity, as well as proactivity and social behaviour.

In order to realize a lightweight notion of μ-agents, strong assumptions about their particular characteristics are relinquished, e.g. μ-agents do not necessarily need to show proactive behaviour. Along with this, any assumptions about particular internal architectures are abandoned. In the first instance this limits the interoperation between μ-agents to the infrastructural level, but it allows heterogeneous μ-agent internals, e.g. purely reactive behaviour up to the level of sophisticated reasoning. This unconstrained internal architecture is a key mechanism that qualifies '*micro*'-agents and allows them to be effectively of arbitrary complexity. The low threshold of *agenthood* allows developers to build complex systems from the ground up while embracing consistent agent-oriented thinking, even on elementary levels. Using the agent metaphor on all application levels improves the notion of loose coupling of application elements, which in turn eases the maintenance of the application or rewiring the μ-agents to build a different application reusing some functionality. As a minimum of consistency, and as a matter of application performance, μ-agents commit to a common communication infrastructure based on efficient asynchronous message passing that makes strong functionality decomposition affordable. Additionally, μ-agents can use so-called μ-intents that allow request specification and automated dynamic binding of target agents – which will be highlighted at a later stage of this discussion. A key and distinguishing feature of μ-agents is their affordance of organisational hierarchies of μ-agents: the functionality of one μ-agent can be decomposed into a number of more elementary μ-agents. This requires suitable organisational modeling mechanisms to maintain a consistent view of the application structure and embodied abstractions.

The metamodel for a μ-agent organisation of our desktop platform implementation of the μ-agent concept, μ^2 [7], is based on the KEA model [15] and visualized in Figure 1.

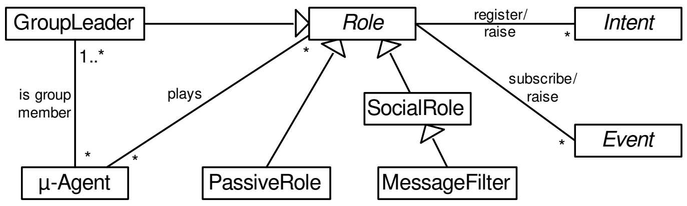

Fig. 1. Core Relationships in μ^2

This metamodel recognizes agents and roles as the first-order entities. From our perspective, roles are characterized as a collection of behaviours applicable to one or more specific contexts. Each agent plays one or more roles which are specializations of the three first-level specializations identified in the metamodel: *Social Roles*, *Passive Roles*, *Group Leaders*. Social Roles represent the most expressive role type, making use of asynchronous message passing and an explicit message container. Passive roles only support blocking communication, which makes them useful for very fine-grained functionality, since the interaction barely involves any performance penalty (compared to a direct method call) while still retaining the advantages of loose coupling between individual agents.

A fundamental aspect of this metamodel is the degree to which it supports the organisation of μ-agents into groups. By playing the *group leader role*, μ-agents can themselves start a group that further agents can be registered with. The group leader has two functions: it controls its group's members, or respectively dispatches control commands from its own group leader, but it can also compose its functionality by combining more fine-grained functionality from its group members. The latter agents can lead groups themselves in order to compose their functionality from further sub-agents. As a result of this cascading structure, a multi-level agent organisation emerges (as schematically shown in Figure 2). However, group leaders need not necessarily compose their functionality from sub-agents but can also simply organize sub-agents to structure the agent organisation by functionality aspects. The hierarchy, however, does not restrict the communication of sub-agents; sub-agents can communicate with agents outside their group, allowing access to their functionality from across the whole agent organisation. Note that for consistency purposes agents which are not assigned to a particular group are members of the SystemOwner group to enforce a consistent control structure.

Mechanisms in the μ-agent context to allow the automated binding of functionality – the key to a loosely coupled composition of more complex applications – are *Intents* (distinguished from Android intents). Roles that can satisfy requested intents (e.g. sending SMS messages) register those as *applicable intents*. Any μ-agent can then raise a request which is automatically delivered to the satisfying role (intent-based dynamic binding).

These intents in the μ-agent context (called μ-intents in the following), derived from the mental concept of intentions, are in fact abstract execution requests and are implemented as Java objects with a freely defined property/operation set. Both requester and requestee need to know and understand the semantics of the μ-intent; for other agents this is not relevant. However, as part of the control mechanism, group leaders can restrict/prohibit the adoption of applicable intents by group members at runtime, if those intents are incompatible with the functionality managed in the corresponding group. The concept of μ-intents seeks to compensate for some aspects that have been sacrificed in the μ-agent concept as a consequence of dropping the assumption of a common agent interior, such as having a similar symbolic representation. As a consequence, μ-intents offer a neutral mechanism to share data in a common representation among μ-agents.

Dynamic binding of message recipients is a core feature of μ-agents but not desirable in cases where the application developer wants to address distinct μ-agents identifiable by ID or name, or wants to employ other `1:n` communication patterns. To achieve this, μ-agents additionally come with the notion of different addressing patterns. Table 2 provides an overview over available patterns.

Table 2. Addressing Patterns for μ-agent Communication in μ^2

Addressing Pattern	***Description***
Unicast	sends message to one specified recipient
Broadcast	sends message to all registered μ-agents (differentiation between local and network-wide broadcast)
Multicast	sends message to event subscribers; modelled via event subscription mechanism
Groupcast	sends message to all μ-agents which are members of a certain group
Rolecast	sends message to all μ-agents playing a given role
Randomcast	sends message to a specified (or random) number of random registered μ-agents

μ-agents can communicate using a wide range of patterns that respect the different modeling artifacts in the concept, such as the addressing of specific roles in the Rolecast, group members via Groupcast, and a Randomcast that allows addressing of random μ-agents (which is particular useful in applications that rely on stochastic elements, such as games of simulations).

The final element of the metamodel to be mentioned here are *Events*. Each role implementation can subscribe to particular events (such as a notification about the initialization of a new agent or a connected platform). Their implementation is realized by extending an abstract class (which enforces the specification of an event source) with arbitrary class structure – similar to the specification of intents. The key difference between event and μ-intents is that μ-intents involve the dynamic resolution to *one* of potentially multiple targets, while events are received by all subscribers. A specific purpose of events is the environmental embedding of μ-agents, e.g. by subscribing μ-agents to system-level events.

The combination of those features in our μ-agent concept enables a fairly direct and clear interpretation of the key characteristics of Software Engineering, and in particular Agent-Oriented Software Engineering (AOSE) as highlighted by Jennings and Wooldridge [13]: *Decomposition* describes means of breaking up coarse-grained functionality into more fine-grained elements, *Abstraction* refers to the necessity of limiting the scope of a developer at a given time in order to limit the overall complexity for a given task. *Organisation*, finally, is the structural specification of an agent society resulting from the application of the aforementioned characteristics. The notion of levels and groups as means to specify them allows an effective decomposition while providing an arbitrarily fine-grained structure of functionality elements, both in a horizontal manner – structured by functionality groups – and vertical manner – breaking it down, hierarchically, to an atomic level. Abstraction is realized by focusing the developer's view on a single level or multiple adjacent levels of this agent organisation at a given time. The application of those principles with this metamodel are visualized in Figure 2.

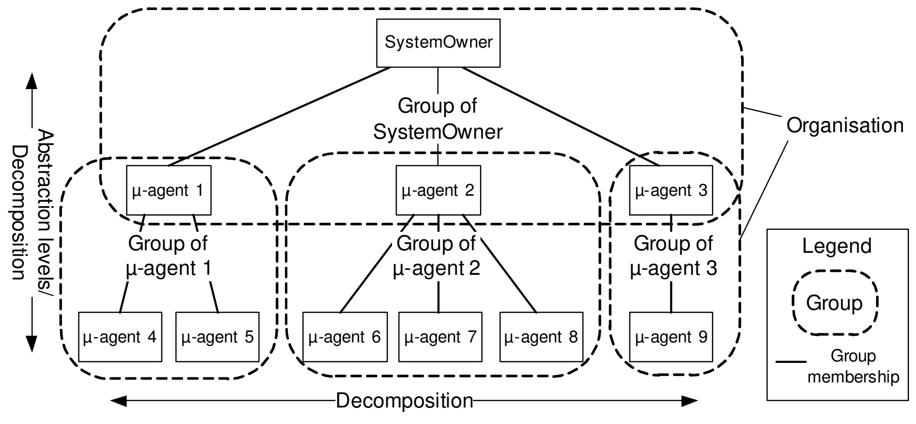

Fig. 2. Representation of AOSE characteristics *Decomposition*, *Abstraction* and *Organisation* with μ-agents

A conceptual advantage of this organisational model is that it allows the distinct application of abstraction levels by simply suppressing lower or higher levels of the agent organisation where appropriate. Agent models without an organisational perspective of this nature limit the possibility of structuring functionality in an explicit vertical manner. Consequently their ability to express decomposition is restricted to more coarse-grained non-hierarchical functionality groups.

Assigning the applicable intents to μ-agents (respectively roles) in this hierarchy allows the definition of an explicit structured functionality repository that allows the flexible use by any other agent on the platform.

At this point the focus of discussion should be turned to performance considerations of the interaction mechanisms. Decomposing functionality using purely agent-based abstractions and a hierarchical organisation will afford the use of μ-agents for even fairly primitive activities. Only by giving up assumptions about a particular internal architecture can μ-agents be used to implement agent internals depending on the application needs, e.g. a short-running μ-agent with generic functionality is used by a reasoning μ-agent composing functionality offered by primitive μ-agents. In order to avoid significant performance impact (compared to other programming paradigms) on the higher levels, efficient inter-agent communication is particularly important on the more primitive levels of decomposition.

An additional aspect of agent interaction is that we consider their efficient operation in distributed environments to be imperative. In our μ-agent scheme this is further supported by the dynamic binding of functionality across different platforms.

2.4 Implementation Aspects of μ-agents

To put the μ-agent concept into context we provide a brief outline of our current implementation of the μ-agent platform named μ^2. We have implemented the platform in Java [2]. The Java language provides strong platform-independence and has become increasingly popular due to the availability of a number of Java Virtual Machine (JVM) languages that build on Java itself, such as Clojure [12]. In the context of μ-agents, this means that μ-agents cannot only be developed with heterogeneous internal architectures but can also easily depend on and integrate different programming languages.

The platform implementation is structured into three layers as visualized in Figure 3. The top layer, the *Agent Logic Layer*, captures the modelling artifacts for application development as described in the previous section, namely μ-agents, roles, μ-intents and events.

The application developer implements roles by extending provided implementations for the different role types (i.e. passive, social and group leader roles). μ-intents and events are implemented as Java objects and can thus encapsulate arbitrary fields and methods.

On the next lower layer, the *Message Routing Layer*, the platform holds a register of all μ-intents and events that are registered on roles instantiated on

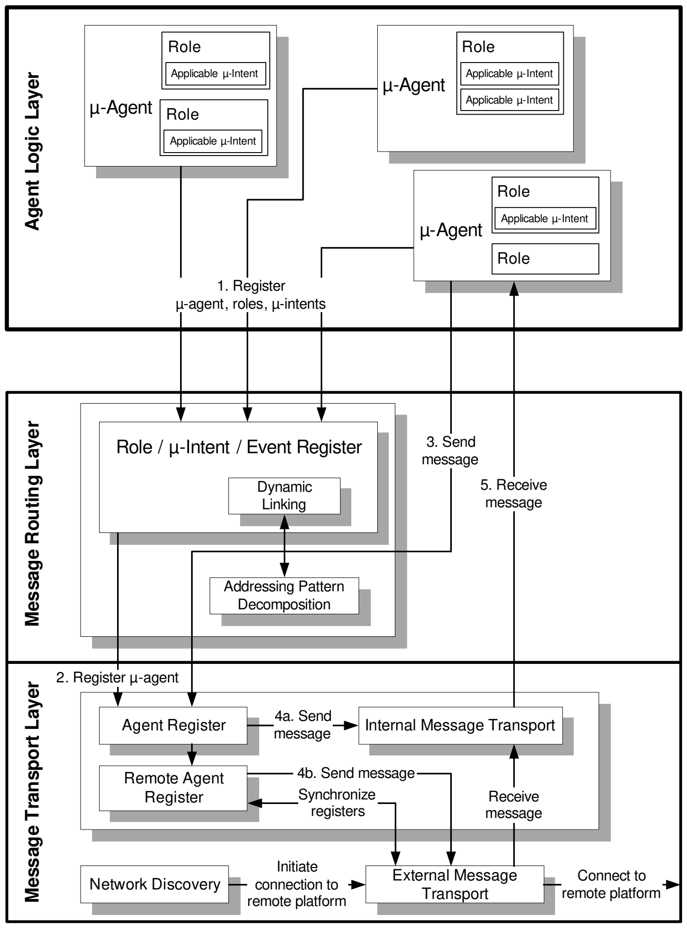

Fig. 3. Architectural Schema of μ^2

that platform. It serves as a basis for the dynamic resolution of roles and μ-agents that can satisfy requested μ-intents. Along with this – and in practice largely an alternative to μ-intent-based communication – the plaform offers the different addressing patterns outlined in subsection 2.3.

The *Message Routing Layer* is connected with the *Message Transport Layer* which resides at the lowest level of the platform architecture. It holds the register containing the IDs of all registered μ-agents, which is used to connect μ-agents with the message passing mechanisms both for local and remote communication. This level does not deal with any other concept than μ-agent identifiers. Requested μ-intents and raised events are merely considered payload and forwarded to μ-agents or platforms specified on the *Message Routing Layer*.

From the perspective of the *Agent Logic Layer* the lower two layers are not differentiated. However, limiting the links between Message Transport Layer and higher layers simplifies the replacement of message passing facilities over time. Along with actual message passing, the lowest layer incorporates network discovery mechanisms to automate the connection to remote platforms.

In Figure 3 the connectors representing the registration process of μ-agents are marked as 1 (Registration of roles, μ-intents and events on the Message Routing Layer) and 2 (Registration of the μ-agent ID on the Message Transport Layer). The Connectors 3–5 show the flow of μ-agent messages across all layers.

2.5 Android Application Components vs. μ-agent Artifacts

Looking at the characteristics of both Android and the introduced μ-agent concept, loose coupling and concurrent communication are core principles in both. *Services* in Android loosely reflect the notion of *Agents*, as they are rather long-running and operate in the background. *Activities* in contrast mediate interaction between service and user and represent visible actions of a service, i.e. *agent operations*. In our μ-agent concept agent operations are not explicitly modelled. *Broadcast receivers* represent an equivalent to an *event subscription mechanism* which, similar to multi-agent systems, integrates agents with events in their surrounding. However, the similarities mentioned here reside on the infrastructural level; services exhibit no motivational autonomy but are purely reactive and additionally do not support complex long-running conversations.

Apart from the aforementioned application components, Android's intents and μ-intents have strong conceptual similarities, since both are representing request specifications. Android's intents have a fixed internal structure and can represent a request as well as a message container at the same time. μ-intents do not provide a fixed internal structure and are separated from the message container provided with μ^2; μ-agents do not necessarily use μ-intents to communicate.

A further difference between Android intents and μ-intents is the degree of loose coupling. In Android, intents can be of an explicit nature, addressing a distinct target component, but also be implicit in specifying characteristics of its content or the target application component. In both cases the application developer needs to know at least the target component type, e.g. activity or service. This lowers the degree of abstraction between caller and callee. In the μ-agent context this is not of concern, as addressed entities are always μ-agents.

To emphasize the use of integrating Android functionality with μ-agents, a closer look at the purpose of Android application components is helpful. Application components are powerful means to structure applications by frontend and backend functionality, in the shape of different runtime containers. However, Android does not provide further mechanisms to allow a structured decomposition of functionality maintained in rather long-running services which – especially in the case of more complex applications – are holding the application's core functionality. Although one possible approach to achieve this is the use of numerous services, the performance of intent-based interaction (which is elaborated in a later section) is prohibitive for fine-grained functionality. Moreover, Android does not provide mechanisms to embed services in an organisational structure, which limits the reusability of fine-grained functionality across different applications.

To support the principal idea of composing Android applications from multiple loosely coupled entities, we suggest, and have demonstrated, the general integration of an organisation-centric μ-agent layer. This allows effective modeling of agent-based applications on Android systems, provides organisational modeling facilities to legacy Android applications, and fosters the reuse of functionality across different applications.

3 μ-agents on Android

3.1 Design Aspects

The similarities between Android and μ-agents suggest an integrated approach which facilitates the support of Android applications with agents to encourage reuse of functionality, offering a lightweight explicit organisational scheme, and enabling the modeling of agent-based applications. μ-agents themselves can react to external events and access Android functionality, which allows them to act in a real environment.

The integration of μ-agents with Android, constituting MOA, is established by linking a particular μ-agent with a dedicated Android service. This makes the interaction virtually seamless for both sides; agents make use of the functionality offered by the interfacing agent, while Android application components interact with the interfacing service in the same manner as with other components. Figure 4 shows this linked agent/service entity which represents the core of MOA that will be explained in the following.

In order to link interactions, the different intent concepts of Android and the μ-agent concept are dynamically converted. This approach has limitations, as not all Android capabilities can be directly accessed via intents but require additional code, especially when dealing with Android's *managers* (e.g. TelephonyManager). Depending on this, Android functionality can thus either be directly invoked (e.g. requesting the user to pick one of the existing contacts) or needs to be mediated with an additional mechanism.

The dynamic conversion mechanism further needs to handle the particular differences between μ-intents and Android intents. Android intents have a fixed implementation (class structure) for dynamically typed content; μ-intent implementations are structurally flexible (i.e. their structure is entirely defined by the application developer) and merely need to implement the Intent interface. As a consequence, a μ-intent rebuilding the Android intent structure (AndroidExecutionIntent) is attached to the interfacing μ-agent (AndroidInterfaceAgent). This way μ-agents can directly invoke intents in Android. Android requires the specification of the target component type to be invoked (i.e. Activity or Service), therefore μ-agents need to supply this information as part of the re-modelled Android intent.

Android intents do not allow the specifications of a sender in the case of direct invocations. Thus, the use of a mediating IntentExecuterActivity is necessary to

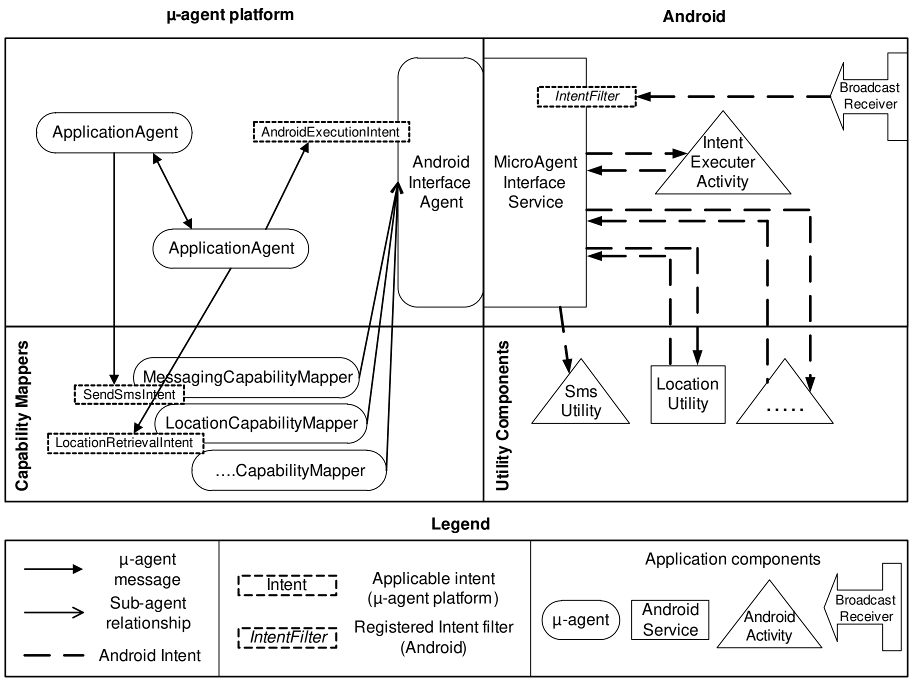

Fig. 4. Architectural Schema of MOA

cache the sending agent, track the execution result of a particular intent, and return eventual responses to the original requester.

In cases where Android functionality cannot be invoked in a direct manner, the conversion mechanism is additionally augmented with *Utility (Application) Components* on the Android side and *Functionality Mapper* agents on the μ-agent side. Those then encapsulate the necessary pre- and post-processing of custom intents and manage the actual functionality. Examples include the subscription to Location services (location updates) which cannot be directly registered via intents but are mediated by the LocationUtility service.

We should emphasize here that the use of agent-based technology is an enabler to better structured applications and to improved functionality reuse. We have introduced this software engineering opportunity into the Android environment without sacrificing or constraining existing system features. Application components can still continue to address the interfacing service and other componenets using all available Android mechanisms, thus by means of either explicit intents (using its class name) or particular intent filters, specified by the application developer. Thus the use of MOA does not have any impact on functionality access by Android application components.

3.2 Application Development and Functionality Reuse with MOA

When designing applications with MOA, functionality is initially separated into frontend and backend components. The frontend dealing with user interaction is developed using legacy Android application components, such as Activities and Broadcast receivers. Those are backed by a μ-agent society living behind the interfacing service. As a consequence application developers need to be aware of both Android concepts and μ-agent concepts, which are connected by MOA's interfacing mechanism that mediates the interaction between both worlds.

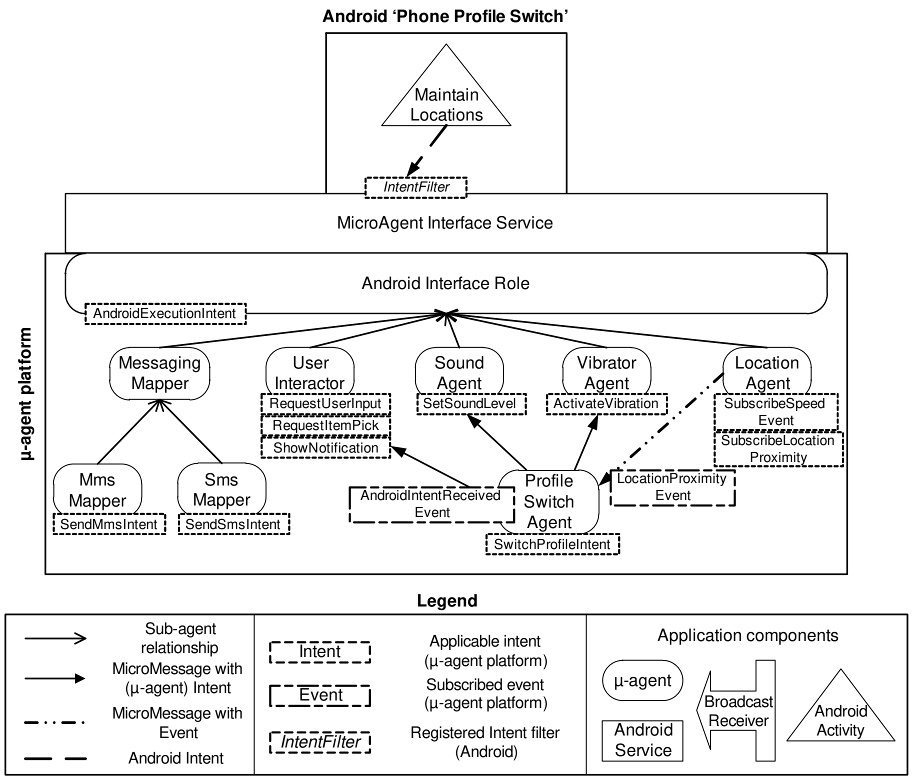

Fig. 5. Basic MOA Application 'Phone Profile Switch'

Figure 5 shows a basic example application following this development principle. The figure visualizes both the application frontend and backend. On the backend, the Android Interface Role, which is directly linked with the Micro-Agent Interface Service, exposes Android functionality to μ-agents in the shape sub-agents, such as the MessagingMapper which manages its own sub-agents (MmsMapper and SmsMapper) to structure its functionality. Further functionality includes simple user interaction, access to the phone's sound settings, the vibration functionality and a μ-agent offering access to location information. The PhoneProfileSwitchAgent is the only actual application-related agent. It composes its functionality from the Android capabilities mapped by MOA, such as subscribing to particular locations and changing phone settings if in proximity to a given location and showing a user notification. The frontend is modelled as an Android activity which allows the user to maintain locations and associate those with particular phone profiles. The ProfileSwitchAgent has subscribed to the AndroidIntentReceivedEvent which notifies the μ-agent once an Android event is received and allows it to extract the relevant information. The ProfileSwitchAgent itself offers its functionality (i.e. switch profiles) as an applicable μ-intent.

This brief example application shows the loosely coupled modelling approach among μ-agents interlinked via μ-intents. It also provides a basis for further applications that can capitalize on the implementation effort.

To give an example for the reuse potential, we extend the previous scenario with the *Driver's Responder* application (see Figure 6[1]) which introduces further context-sensitivity. The additional μ-agent (SpeedResponderAgent) subscribes to the CurrentSpeedEvent offered by the LocationAgent and is thus constantly informed about the current speed. The matching frontend application component is represented as the SpeedManager activity which allows the user to enter some speed thresholds (e.g. 30 kph) in excess of which the SpeedResponderAgent considers the user to be occupied with driving a vehicle. When reaching this threshold, the μ-agent requests the change into silent profile by sending the SwitchProfileIntent offered by the PhoneProfileSwitchAgent. Beyond this, it can compose further functionality, such as an automatic response to an SMS message indicating that the user is busy driving (and eventually showing a notification to the user). The scenario described here is a simplification of the particular application scenario but demonstrates how the backend of MOA applications can reuse functionality across different applications only relying on μ-intents and events.

Given the risk of creating interdependencies between applications, the use of agents (in contrast to other modeling paradigms) is useful, as they are conceptually capable of handling failed binding requests and find alternatives dynamically at runtime (here this would be the case if the PhoneProfileApplication is missing). However, the same would be the case for interdepending legacy Android applications. In order to extend applications by introducing new μ-agents, developers only need to know the internals of relevant μ-intents (e.g. SwitchProfileIntent) in order to use the functionality; the executing μ-agent is automatically resolved when raising this μ-intent.

The features of MOA are not merely constrained to the interaction between Android application components and μ-agents but also allow the development of distributed applications, or simply the extension of existing applications with distributed features. As described for the previous examples, context-sensitivity

[1] μ-agents belonging to MOA and the previous application are greyed out to emphasize the added functionality.

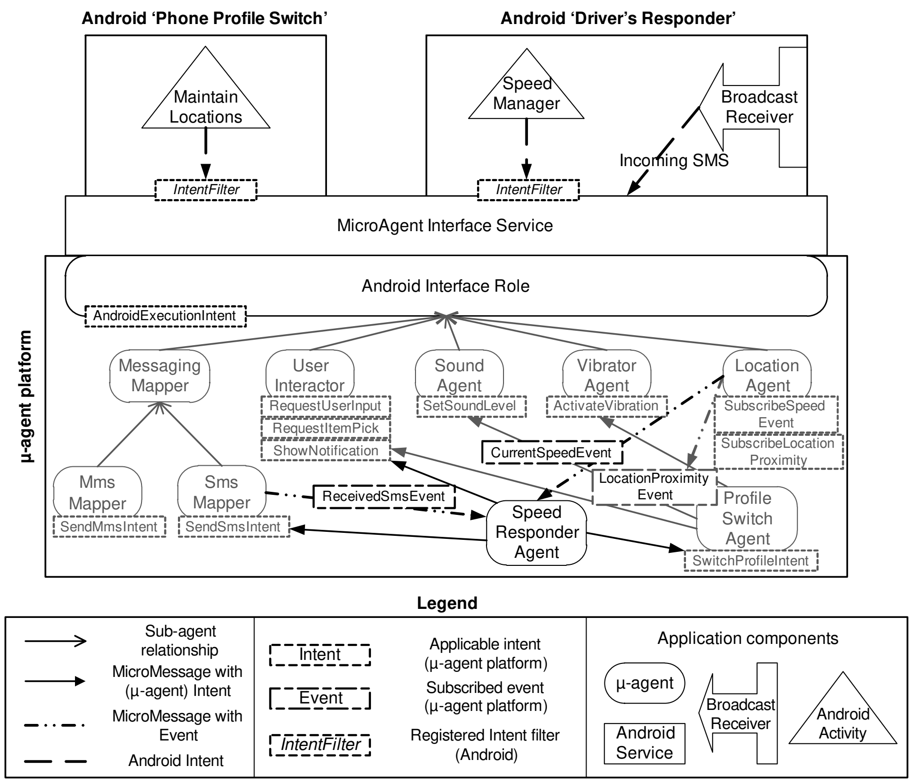

Fig. 6. Reuse of Functionality by MOA Application 'Driver's Responder'

of μ-agents is achieved using event subscription. Along with events specified by the application developer, platform implementations include system events such as a notification upon newly instantiated μ-agents or established connections with remote platforms. Upon connection the realized platform implementation also transmits the current location. The previous scenario can thus easily be extended with a synchronization feature, as visualized in Figure 7[2].

The ProfileSwitchAgent subscribes to the PlatformLocationEvent (which is raised once a remote platform connects). Depending on location, or name of the remote platform, the ProfileSwitchAgent can raise the SynchronizePhoneProfiles intent that tries to synchronize the phone profile for specific locations as specified by the user. As such, this works as a location-sensitive backup mechanism.

In this context it shows that the composition of applications and reuse of functionality by different applications can thus extend across different devices that run MOA or μ^2, the desktop implementation of the μ-agent concept. This allows extended reuse of functionality, specifically the use of functionality which cannot be provided on the local device (e.g. printing mediated via a desktop pc).

[2] Various μ-agents that are not relevant in this context are omitted in the figure.

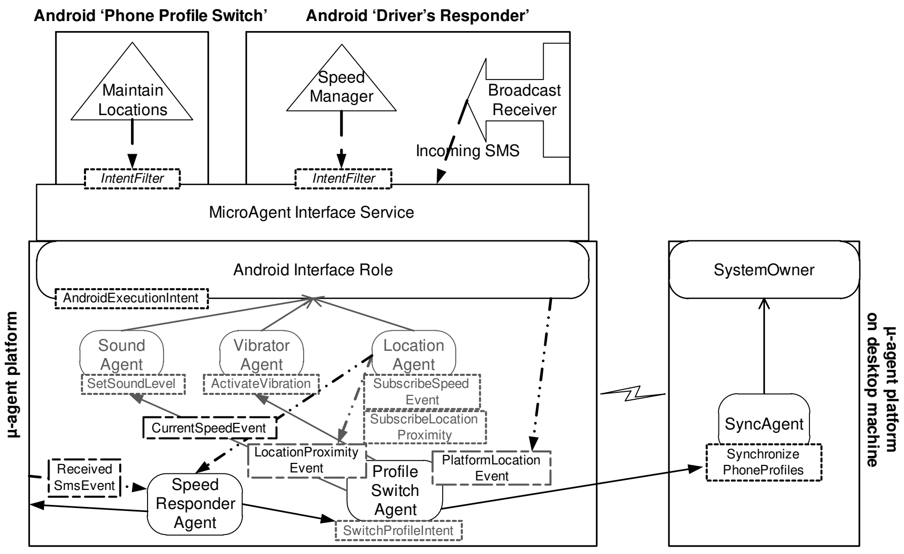

Fig. 7. Synchronizing of Phone Profiles across desktop and mobile platform

This equally enables desktop pc's to use the functionality of the mobile device (e.g. sending SMS messages).

Application development with MOA can thus be realized in a consistently agent-based manner involving the provision and implementation of intent functionality as well as events in a location-independent manner. The entire handling of all the network connections is delegated to the MOA (or, respectively, its desktop equivalence, μ^2).

Apart from the concrete reusability aspects of functionality as described above, the decomposition into μ-agents enables a more emergent view on applications.

Android applications are generally defined by an application manifest that specifies all related applications components. This feature is generally desirable to ensure coherent applications and address security concerns by clearly associating application components with applications. In principle μ-agents allow a dynamic composition of applications, since μ-agent functionality can be added at any time during an application's runtime, thereby allowing the development of more adaptive applications, e.g. changing application behaviour depending on usage. This aspect will become even more interesting and powerful once Android allows just-in-time compilation, which will enable the development of μ-agents at runtime.

3.3 μ-agents as Event Sources

Android application development characteristics and the conceptual fit of μ-agents also facilitates addressing another aspect that is relevant for applications that are not backed with the MOA approach.

μ-agents make functionality of Android *managers* more accessible, since it is now uniformly expressed in the notion of μ-intents. Referring back to the example application scenario of combining PhoneProfileAgent with the Speed-ResponderAgent automatically adjusting phone profiles according to speed (see Figure 6 in the previous subsection), the SpeedResponderAgent incorporates the mechanisms to identify the movement type of its user[3]. In this example the information about the current state was only used for internal application purposes, but could equally be offered to other μ-agents using a μ-intent (similar to the PhoneProfileAgent that offers its capabilities as μ-intents). Beyond that, the high-level description of the current state could be offered as an event that is raised once the state changes (e.g. from DRIVING to WALKING). To offer those events to applications that are building on the MOA development principles, they can be exposed as events in the Android realm and caught by application components that can benefit from this functionality using broadcast receivers. This way MOA can deliver functionality reuse for legacy Android applications which is useful for enriching raw sensor data with semantic information instead of potentially having to reimplement this functionality repeatedly.

An example for this is the phone's positioning information. Android data received from the gyroscope can be used to determine the phone's orientation. However, this information is provided in the shape of rotation matrix values which demand conceptual understanding even if only used for a small task, such as determining whether the phone is lying, tilted to either side, or held upright. μ-agents can take over the task of converting this detailed information into more coarse semantic categories by approximating the device orientation of the device from the data, and offer this information on a pragmatic abstraction level. Android intents containing this information, e.g. indicating the device as upright (DEVICE_UPRIGHT), can be registered by broadcast receivers of any Android application and thus increase the reuse of this functionality across any number of applications running on the device. To mediate the registration of such events (and to raise them as Android intents), MOA uses a specialized μ-agent that handles the forwarding of generated events to the broadcast receiver mechanism. Figure 8 visualizes the interaction schema to register and raise events in both Android and in the μ-agent realm.

Events (here the SimplePositioningEvent) are offered by the capability mapper μ-agent that directly interacts with a corresponding utility component counterpart. A mediating agent, the Micro-to-AndroidEventAgent, subscribes to the event on the MOA side (Message 1 in the figure) and raises it in Android (Messages 2-4), so that it can be captured by any Android application component

[3] In the given example the identification of movement types is parameterized by the user. One could imagine other more sophisticated approaches, e.g. by classifying sensor data.

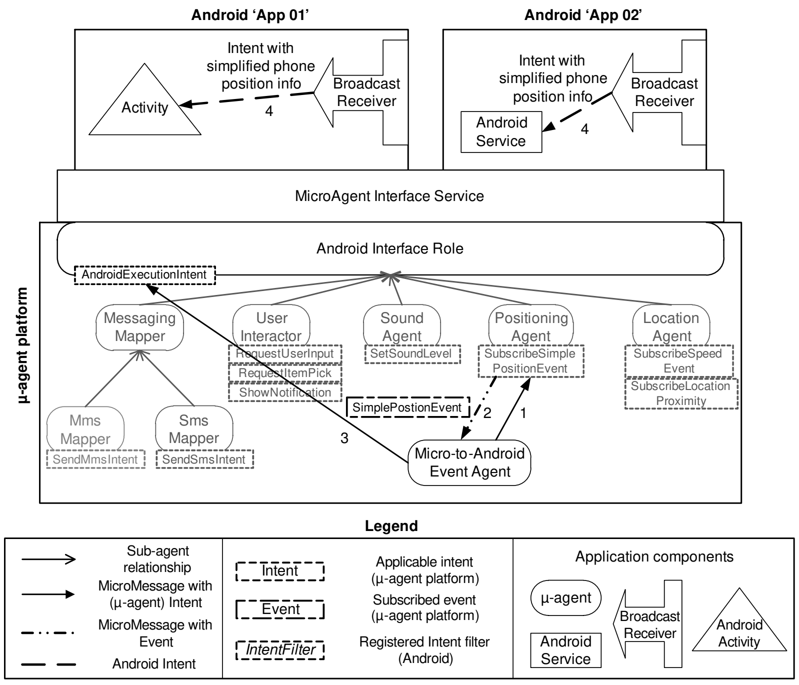

Fig. 8. μ-agents as Event Sources

that is capable of handling this particular event. As a consequence, Android developers can use the information about the device positioning without a detailed concern for the actual functionality involved with transforming sensor data to semantically more expressive information. The developer only needs to know the (Android) intent internals; he can ignore any μ-agent implementation aspects. But as the generated intents are fully convertable and operate both in the μ-agent realm as well as in conjunction with legacy Android application components, this functionality is accessible to both sides.

A future aspect of this functionality is to automate the usage from the Android side, i.e. enabling the subscription to particular events transparently from Android without relying on the mediating agent managing subscription and broadcasting of events to Android.

3.4 Performance Evaluation

To quantify some benefits of the use of μ-agents on Android, we developed a benchmark measuring the interaction performance for both Android-based services and a version realizing this functionality with μ-agents. It simulates a simple context-aware application, automatically responding to incoming SMS text messages and is shown in Figure 9.

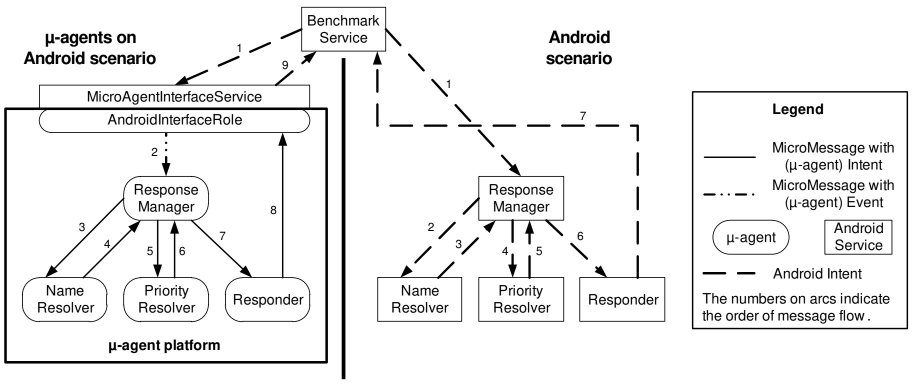
μ-agents on Android scenario
Benchmark Service
Android scenario
MicroAgentInterfaceService
AndroidInterfaceRole
Response Manager
Name Resolver
Priority Resolver
Responder
μ-agent platform
Legend
MicroMessage with (μ-agent) Intent
MicroMessage with (μ-agent) Event
μ-agent
Android Service
Android Intent
The numbers on arcs indicate the order of message flow.

Fig. 9. Benchmark Scenario for Performance Comparison

An incoming text message is forwarded to a responding entity (ResponseManager) which coordinates the resolution of the sender's name (via NameResolver), the identification of the priority (PriorityResolver) of the sender, and finally responds to this message (Responder). The functionality is standardized, and in each case a response message is generated to measure the pure interaction performance for both benchmark implementation variants. This scenario has been executed for increasing numbers of rounds to show the scalability of MOA. Each configuration has been executed ten times, with an initial warm-up run of 5 rounds. Table 3 shows the average durations along with standard deviation and relative performance factor of Android services in comparison to μ-agents.[4] Figure 10 shows the graph of those results.

Table 3. Selected Benchmark Results per Scenario Rounds

Rounds	MOA (ms)	σ	native Android (ms)	σ	Factor[a]
5	231	67.62	639	43.57	2.77
10	390	88.93	950	61.07	2.44
25	850	65.12	1875	30.57	2.21
50	1637	142.90	3466	132.30	2.12
100	3027	68.95	6789	106.77	2.24
250	7387	117.64	16948	735.71	2.29
500	14407	256.74	33777	350.04	2.34
1000	28404	219.85	70088	379.47	2.47
2500	77451	984.75	201685	1493.96	2.60

[a] Relative performance of Android intents to μ-agents.

[4] The benchmark has been run on a HTC Magic smartphone running Android 2.2.1. In both scenarios all entities run in the same process, avoiding computationally expensive Inter-Process Communication (IPC).

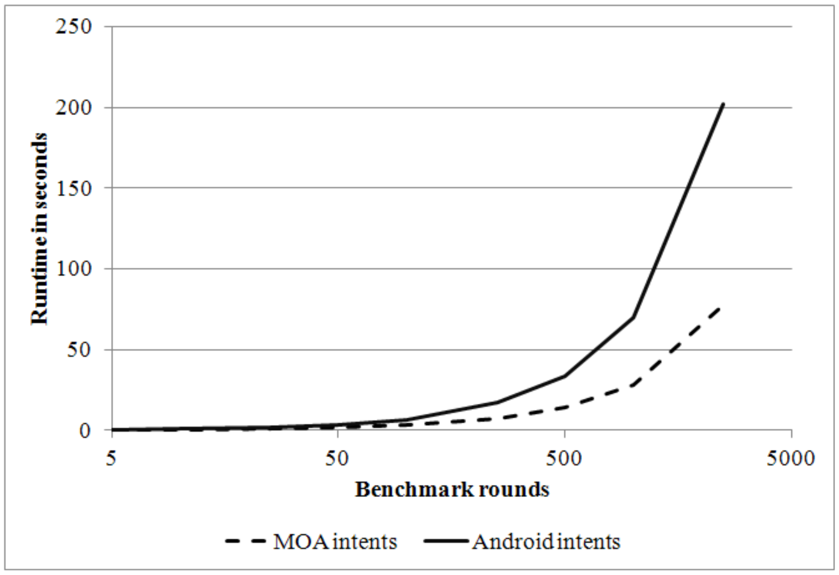

Fig. 10. Benchmark Results Graph

Despite the additional two Android intents necessary to realize the MOA variant of the scenario, it still significantly outperforms the purely Android-based interaction. The performance difference is surprising, but we attribute it to the fact that Android's application components are more featureful (potentially allowing IPC, providing a more comprehensive life cycle) and thus demand a heavier implementation (and processing) than the μ-agents. μ-agents are directly built on the provided libraries but themselves do not use any of the Android application components for their internals; their purpose is to allow efficient communication between numerous less featureful entities.

Beyond the qualitative argument for modeling benefits from an explicit agent organisation, this gives a clear indication that a strong decomposition into μ-agents can be achieved without performance loss, and might – depending on the number and degree of decomposition into multiple μ-agents – even result in faster applications.

4 Related Work

This work is not the first targeting the comparatively young Android platform but takes a different approach than existing efforts to run agent-based systems on this platform.

The mobile version of the popular agent platform JADE [19], JADE-LEAP, is available in an Android version, JADE ANDROID [11]. It enables the integration of an Android-based software agent into the comprehensive and mature JADE infrastructure. JADE organises individual agents by the containers they are associated with. For distributed use JADE ANDROID relies on a main container provided by the connected full JADE version. The number of agents running in one JADE ANDROID instance is currently restricted to one. So agent development with JADE ANDROID focuses on the one-to-one assignment of application and user.

Another approach is presented by Agüero et al. [1], who use Android as a basis to implement their Agent Platform Independent Model (APIM), which is derived from the analysis of commonalities in various AOSE methodologies. Their implementation is directly based on the full Android infrastructure (e.g. extension of Services as Agents). In contrast to the μ-agent concept argued here, the APIM puts the focus on agent internals. Organisational modeling is not of primary concern.

JaCa-Android [18] implements the Agents and Artifacts model [17] on Android. The Agents and Artifact model puts forth the notion of agents and artifacts as modeling entities to describe application functionality. For the implementation of agent internals, JaCa-Android relies on the AgentSpeak interpreter Jason [3]. Organisational aspects are modelled using the notion of workspaces to structure an agent's environment. Agents can participate and collaborate in various workspaces in a distributed manner. Android capabilities, such as sensor information or messages (e.g. SMS message, GPS coordinates) are modelled as artifacts. Artifacts expose specific attributes and operations to agents that are operating on those artifacts across different workspaces.

A last approach – seemingly similar to our concept – is Jadex micro agents [4]. Originally, Jadex [16] was developed as a BDI layer working on top of the JADE platform. With the recent version 2, this dependency was given up, making Jadex an agent platform on its own. Along with this, the notion of micro agents was introduced. In Jadex they act as a counterpart to conventional Jadex agents and consequently focus on performance, allow the handling of their own lifecycle and cater for the execution of primitive tasks. Jadex micro agents avoid the representation of organisational concepts such as roles or groups; introducing any organsational structure is left to the developer. As a consequence Jadex micro agents have a very small memory footprint. Interaction can be modelled using a generic message type that allows unconstrained communication among different micro agents.

In contrast to existing efforts, the μ-agent approach advocated in this paper offers an agent-based organisational extension to Android's infrastructure which both allows modelling in an agent-based manner, while also increasing the reuse of application functionality across different applications and platforms. The goal of our implementation is not only to run agents on Android but also to provide an interface for the seamless interoperation of agents with legacy application components on Android devices. Given that the specific application landscape on Android device instances can vary significantly, the potential of μ-agents to formulate Android intents in a proactive manner allows them to treat Android itself as an open system.

Table 4. Overview on Existing Android-based Multi-agent Platforms

Approach	*Metamodel*	*Agent Architecture*	*interoperable desktop version*	*direct interaction with Android components*
Agüero et al.	Agent Platform Independent Model (APIM)	intelligent agents[a]	no	no
JaCa-Android	Agents & Artifacts model	intelligent agents	no	no
JADE Android	FIPA Abstract Architecture	intelligent agents	yes	no
Jadex Micro Agents	–	reactive agents	yes	no
MOA	μ^2 Model	architecture-independent	yes	yes

[a] The metamodel it not restricted to a particular internal architecture but the implementation is realized with Jason.

Table 4 summarizes this short survey on the different efforts to run agents on Android based on the available documentation.

5 Future Development of MOA

Future research will include the extension of the current system towards a more comprehensive agent-based ad hoc middleware, integrating a wider range of mobile system services and sensing, together with Internet features (e.g. web services). Part of this work is also to address the potentially harmful bottleneck of MOA when interacting with numerous legacy application components. The development of applications using this blended approach further needs to be harmonized with existing AOSE methodologies. μ-agents are mediators for access to low-level functionality on one side, and intelligent agent notions on the other. Beyond this general strengthening and enhancing of the MOA platform, there will be a further, significant development in connection with the integration of MOA with *Web Intents* [14].

Web Intents are a new client-side browser equivalent to Android intents that support the client-side discovery of services for particular tasks, such as playing music or sharing files. Their functionality includes a subset of the functionality offered by Android intents. To invoke Web Intents it is necessary to provide an *action* which is a verb describing the desired action, such as VIEW or PICK. Optional fields include *type*, which acts as a filter on data types. A last optional parameter is the specification of an URI pointing to an action target.

As a result of the runtime binding of web intents, the integration between different applications or services is delegated to the end user rather than the developer. The user registers client-side services/applications that satisfy specific requests.

The entire concept is similar to the runtime binding realized in MOA. As a consequence, and supported by the more primitive resolution mechanisms of web intents than in Android, web intents provide a useful mediator to integrate phone functionality with the web browser, both for desktop machines and mobile devices. In this context MOA can act as an infrastructural mechanism to mediate network-level aspects, but also enrich the available functionality set for browsers. Given the composition aspects of μ-agents, MOA can serve as a basis to provide more complex smart services, e.g. by adding location-awareness, accessing personalized data, etc. On the other hand, MOA can take over some of the actual binding functionality which reduces the browser-centrism of this approach and – with its different degrees of functionality decomposition – allows a more fine-grained functionality resolution.

With the availability for a wide range of web browsers and the advent of Google's ChromeOS [9], we believe that web intents will rapidly gain increased attention and put a stronger emphasis on ad-hoc composition of functionality to improve the customization of application environments.

6 Conclusion

The mobile application development platform Android offers comprehensive capabilities for a wide range of smart applications and an infrastructure that shows characteristics related to multi-agent systems. Its applications are composed using loosely coupled asynchronously communicating application components.

However, the degree of loose coupling in Android shows limitations and does not offer an organisational scheme for more fine-grained functionality decomposition patterns. We have proposed here the integration of efficiency-oriented μ-agents with Android application components. This enables the comprehensive maintenance of developed functionality and makes it available for reuse across different applications at an abstraction level convenient for the developer. This offers a low-threshold approach to compose required functionality in a consistently agent-oriented manner across a dynamically changing device landscape.

Applications backed with μ-agents can easily coexist with legacy applications; so developers should consider both Android and μ-agent concepts when modelling applications. To separate concerns when using both conceptual architectures, developers can, if they so choose, use μ-agents merely as event sources or for primitive functionality while building the actual application functionality using legacy application components. Beyond this MOA's unique direct interaction with Android application components using Android intents – in conjunction with the varying and changing application landscape on different devices – allows μ-agents to act in open systems.

To provide a strong degree of functionality decomposition – as a prerequisite for the eventual reuse – we deem performance a critical aspect. A benchmark testing the interaction performance of both Android intents and μ-agent interaction mechanism has demonstrated the favorable performance of μ-agent intents.

The increasing provision of technology supporting runtime binding of application functionality supports the conceptual approach taken with MOA. In this context, the consideration of Web Intents will be of particular interest for our next iteration.

Overall the unique approach to interface agent-based modeling principles with legacy technology described here is an example of how agent-oriented software engineering principles can facilitate application development in a practical and cross-paradigmatic manner.

References

1. J. Agüero, M. Rebollo, C. Carrascosa, and V. Julián. Does Android Dream with Intelligent Agents? In J. Corchado, S. Rodríguez, J. Llinas, and J. Molina, editors, *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, volume 50 of *Advances in Soft Computing*, pages 194–204. Springer Berlin / Heidelberg, 2009.
2. K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.
3. R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley, 2007.
4. L. Braubach and A. Pokahr. Micro User Guide. http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/Micro+User+Guide/01+Introduction. Accessed on: 15th August 2011.
5. D. Dennett. *The Intentional Stance*. MIT Press, Cambridge, Massachusetts, 1987.
6. Foundation for Intelligent Physical Agents. FIPA Specifications. http://www.fipa.org/specifications/index.html. Accessed on: 15th August 2011.
7. C. Frantz. Micro-agent platform μ^2. http://www.micro-agents.net. Accessed on: 15th August 2011.
8. Google. Android. http://www.android.com/. Accessed on: 25th January 2011.
9. Google. Chromium OS. http://www.chromium.org/chromium-os. Accessed on: 15th August 2011.
10. Google. What is Android? http://developer.android.com/guide/basics/what-is-android.html. Accessed on: 15th August 2011.
11. D. Gotta, T. Trucco, M. Ughetti, S. Semeria, C. Cucè, and A. M. Porcino. JADE Android Add-on Guide. http://jade.tilab.com/doc/tutorials/JADE_ANDROID_Guide.pdf. Accessed on: 15th August 2011.
12. R. Hickey. Clojure. http://clojure.org/. Accessed on: 15th August 2011.
13. N. R. Jennings and M. Wooldridge. Agent-Oriented Software Engineering. *Artificial Intelligence*, 117:277–296, 2000.
14. P. Kinlan. Web Intents. http://webintents.org/. Accessed on: 15th August 2011.
15. M. Nowostawski, M. Purvis, and S. Cranefield. KEA - Multi-Level Agent Architecture. In *Proceedings of the Second International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS 2001)*, pages 355–362. Department of Computer Science, University of Mining and Metallurgy, Krakow, Poland, 2001.
16. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In G. Weiss, R. Bordini, M. Dastani, J. Dix, and A. F. Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, And Simulated Organizations*, pages 149–174. Springer US, 2005.

17. A. Ricci, M. Viroli, and A. Omicini. Give agents their artifacts: the A&A approach for engineering working environments in MAS. In E. H. Durfee, M. Yokoo, M. N. Huhns, and O. Shehory, editors, *AAMAS*, page 150. IFAAMAS, 2007.
18. A. Santi, G. Marco, and A. Ricci. JaCa-Android: An Agent-based Platform for Building Smart Mobile Applications. In *In Proceedings of LAnguages, methodologies and Development tools for multi-agent systemS (LADS-2010)*, 2010.
19. Telecom Italia. JADE - Java Agent DEvelopment Framework. http://jade.tilab.com, October 2011. Accessed on: 15th August 2011.