Exploiting Agent-Oriented Programming for Developing Android Applications

Andrea Santi
University of Bologna
Cesena, Italy
Email: a.santi@unibo.it

Marco Guidi
University of Bologna
Cesena, Italy
Email: marco.guidi7@studio.unibo.it

Alessandro Ricci
DEIS, University of Bologna
Cesena, Italy
Email: a.ricci@unibo.it

***Abstract*—Agent-Oriented Programming (AOP) provides an effective level of abstraction for tackling the programming of mainstream software applications, in particular those that involve complexities related to concurrency, asynchronous events management and context-sensitive behaviour. In this paper we support this claim in practice by discussing the application of AOP technologies – *Jason* and CArtAgO in particular – for the development of smart nomadic applications based on the Google Android platform.**

I. Introduction

The value of Agent-Oriented Programming (AOP) [20] is often remarked and evaluated in the context of Artificial Intelligence (AI) and Distributed AI problems. This is evident, for instance, by considering existing agent programming languages (see [4], [6] for comprehensive surveys) – whose features are typically demonstrated by considering toy problems such as block worlds and alike.

Besides this view, we argue that the level of abstraction introduced by AOP is effective for organizing and programming software applications in general, starting from those programs that involve aspects related to reactivity, asynchronous interactions, concurrency, up to those involving different degrees of autonomy and intelligence. Following this view, one of our current research lines investigates the adoption and the evaluation of existing agent-based programming languages and technologies for the development of applications in some of the most modern and relevant application domains. In this context, a relevant one is represented by next generation nomadic applications. Applications of this kind are getting a strong momentum given the diffusion of mobile devices which are more and more powerful, in terms of computing power, memory, connectivity, sensors and so on. Main examples are smart-phones such as the iPhone and Android-based devices.

On the one side, nomadic applications share more and more features with desktop applications, and eventually extending such features with capabilities related to context-awareness, reactivity, usability, and so on, all aspects that are important in the context for Internet of Things and Ubiquitous Computing scenarios. All this increases – on the other side – the complexity required for their design and programming, introducing aspects that – we argue – are not suitably tackled by mainstream programming languages such as the object-oriented ones.

So, in this paper we discuss the application of an agent-oriented programming platform called JaCa to the development of smart nomadic applications. Actually JaCa is not a new platform, but the integration of two existing agent programming technologies: *Jason* [5] agent programming language and platform, and CArtAgO [17] framework, for programming and running the environments where agents work. JaCa is meant to be a general-purpose programming platform, so useful for developing software applications in general. In order to apply JaCa to nomadic computing, we developed a porting of the platform on top of Google Android, which we refer as JaCa-Android. Google Android is an open-source software stack for mobile devices provided by Google that includes an operating system (Linux-based), middleware, SDK and key applications.

Other works in literature discuss the use of agent-based technology on mobile devices—examples include AgentFactory [13], 3APL [10], JADE [3]. Differently from these works, here we do not consider the issue of porting agent technologies on limited capability devices, but we focus on the advantages brought by the agent-oriented programming level of abstraction for the development of complex nomadic applications.

The remainder of the paper is organised as follows: in Section II we provide a brief overview of the JaCa platform – which we consider part of the background of this paper; then, in Section III we introduce and discuss the application of JaCa for the development of smart nomadic applications on top of Android, and in Section IV we describe two practical application samples useful to evaluate the approach. Finally, in Section V we briefly discuss some open issues related to JaCa and, more generally, to the use of current agent-oriented programming technologies for developing applications and related future works.

II. Agent-Oriented Programming for Mainstream Application Development – The JaCa Approach

An application in JaCa is designed and programmed as a set of agents which work and cooperate inside a common environment. Programming the application means then programming the agents on the one side, encapsulating the logic of control of the tasks that must be executed, and the environment on the other side, as first-class abstraction providing the actions and functionalities exploited by the

agents to do their tasks. It is worth remarking that this is an *endogenous* notion of environment, i.e. the environment here is part of the software system to be developed [18].

More specifically, in JaCa *Jason* [5] is adopted as programming language to implement and execute the agents and CArtAgO [17] as the framework to program and execute the environments.

Being a concrete implementation of an extended version of AgentSpeak(L) [15], *Jason* adopts a BDI (Belief-Desire-Intention)-based computational model and architecture to define the structure and behaviour of individual agents. In that, agents are implemented as reactive planning systems: they run continuously, reacting to events (e.g., perceived changes in the environment) by executing plans given by the programmer. Plans are courses of actions that agents commit to execute so as to achieve their goals. The pro-active behaviour of agents is possible through the notion of goals (desired states of the world) that are also part of the language in which plans are written. Besides interacting with the environment, *Jason* agents can communicate by means of speech acts.

On the environment side, CArtAgO – following the A&A meta-model [14], [19] – adopts the notion of *artifact* as first-class abstraction to define the structure and behaviour of environments and the notion of *workspace* as a logical container of agents and artifacts. Artifacts explicitly represent the environment resources and tools that agents may dynamically instantiate, share and use, encapsulating functionalities designed by the environment programmer. In order to be used by the agents, each artifact provides a usage interface composed by a set of *operations* and *observable properties*. Operations correspond to the actions that the artifact makes it available to agents to interact with such a piece of the environment; observable properties define the observable state of the artifact, which is represented by a set of information items whose value (and value change) can be perceived by agents as percepts. Besides observable properties, the execution of operations can generate signals perceivable by agents as percepts, too. As a principle of composability, artifacts can be assembled together by a link mechanism, which allows for an artifact to execute operations over another artifact. CArtAgO provides a Java-based API to program the types of artifacts that can be instantiated and used by agents at runtime, and then an object-oriented data-model for defining the data structures used in actions, observable properties and events.

The notion of workspace is used to define the topology of complex environments, that can be organised as multiple sub-environments, possibly distributed over the network. By default, each workspace contains a predefined set of artifact created at boot time, providing basic actions to manage the overall set of artifacts (for instance, to create, lookup, link together artifacts), to join multiple workspaces, to print message on the console, and so on.

JaCa integrates *Jason* and CArtAgO so as to make the use of artifact-based environments by *Jason* agents seamless. To this purpose, first, the overall set of external actions that a *Jason* agent can perform is determined by the overall set of artifacts that are actually available in the workspaces where the agent is working. So, the action repertoire is dynamic and can be changed by agents themselves by creating, disposing artifacts. Then, the overall set of percepts that a *Jason* agent can observe is given by the observable properties and observable events of the artifacts available in the workspace at runtime. Actually an agent can explicitly select which artifacts to observe, by means of a specific action called *focus*. By observing an artifact, artifacts' observable properties are directly mapped into beliefs in the belief-base, updated automatically each time the observable properties change their value. So a *Jason* agent can specify plans reacting to changes to beliefs that concern observable properties or can select plan according to the value of beliefs which refer to observable properties. Artifacts' signals instead are not mapped into the belief base, but processed as non persistent percepts possibly triggering plans—like in the case of message receipt events. Finally, the *Jason* data-model – essentially based on Prolog terms – is extended to manage also (Java) objects, so as to work with data exchanged by performing actions and processing percepts.

A full description of *Jason* language/platform and CArtAgO framework – and their integration – is out of the scope of this paper: the interested reader can find details in literature [17], [16] and on *Jason* and CArtAgO open-source web sites[1][2].

III. Programming Android Applications with JaCa

In this section we describe how JaCa's features can be effectively exploited to program smart nomadic applications on top of Android, providing benefits over existing non-agent approaches. First, we briefly sketch some of the complexities related to the design and programming of such a kind of applications; then, we describe how these are addressed in JaCa-Android—which is the porting of JaCa on Android, extended with a predefined set of artifacts specifically designed for exploiting Android functionalities.

A. Programming Nomadic Applications: Complexities

Mobile systems and nomadic applications have gained a lot of importance and magnitude both in research and industry over the last years. This is mainly due to the introduction of mobile devices such as the iPhone[3] and the most modern Android[4]-based devices that changed radically the concept of smartphone thanks to: (*i*) hardware specifications that allow to compare these devices to miniaturised computers, situated – thanks to the use of practically every kind of known connectivity (GPS, WiFi, bluetooth, etc.) – in a computational network which is becoming more and more similar to the vision presented by both the Internet of Things and ubiquitous computing, and (*ii*) the evolution of both the smartphone O.S. (Apple iOS, Android, Meego[5]) and their related SDK.

[1] http://jason.sourceforge.net
[2] http://cartago.sourceforge.net
[3] http://www.apple.com/it/iphone/
[4] http://www.android.com/
[5] http://meego.com

These innovations produce a twofold effect: on the one side, they open new perspectives, opportunities and application scenarios for these new mobile devices; on the other side, they introduce new challenges related to the development of the nomadic applications, that are continuously increasing their complexity [1], [11]. These applications – due to the introduction of new usage scenarios – must be able to address issues such as concurrency, asynchronous interactions with different kinds of services (Web sites/Services, social-networks, messaging/mail clients, etc.) and must also expose a user-centric behaviour governed by specific context information (geographical position of the device, presence/absence of different kinds of connectivity, events notification such as the reception of an e-mail, etc.).

To cope with these new requirements, Google has developed the Android SDK[6], which is an object-oriented Java-based framework meant to provide a set of useful abstractions for engineering nomadic applications on top of Android-enable mobile devices. Among the main coarse-grain components introduced by the framework to ease the application development we mention here:

- Activities: an activity provides a GUI for one focused endeavor the user can undertake. For example, an activity might present a list of menu items users can choose, list of contacts to send messages to, etc.
- Services: a service does not have a GUI and runs in the background for an indefinite period of time. For example, a service might play background music as the user attends to other matters.
- Broadcast Receiver: a broadcast receiver is a component that does nothing but receive and react to broadcast announcements. Many broadcasts originate in system code - for example, announcements that the timezone has changed, that the battery is low, etc.
- Content providers: a content provider makes a specific set of the application's data available to other applications. The data can be stored in the file system, in an SQLite database, etc.

In Android, interactions among components are managed using a messaging facility based on the concepts of Intent and IntentFilter. An application can request the execution of a particular operation – that could be offered by another application or component – simply providing to the O.S. an Intent properly characterised with the information related to that operation. So, for example, if an application needs to display a particular Web page, it expresses this need creating a proper Intent instance, and then sending this instance to the Android operating system. The O.S. will handle this request locating a proper component – e.g. a browser – able to manage that particular Intent. The set of Intents manageable by a component are defined by specifying, for that component, a proper set of IntentFilters.

Generally speaking, these components and the Intent-based interaction model are useful – indeed – to organise and structure applications; however – being the framework fully based on an object-oriented virtual machine and language such as Java – they do not shield programmers from using callbacks, threads, and low-level synchronisation mechanisms as soon as applications with complex reactive behaviour are considered. For instance, in classic Android applications asynchronous interactions with external resources are still managed using polling loops or some sort of mailbox; context-dependent behaviour must be realised staining the source code with a multitude of if statements; concurrency issues must be addressed using Java low-level synchronisation mechanisms. So, more generally, we run into the problems that typically arise in mainstream programming languages when integrating one or multiple threads of control with the management of asynchronous events, typically done by callbacks.

In next section we discuss how agent-oriented programming and, in particular, the JaCa programming model, are effective to tackle these issues at a higher-level of abstraction, making it possible to create more clear, terse and extensible programs.

B. An Agent-oriented Approach based on *JaCa*

By adopting the JaCa programming model, a mobile Android application can be realised as a workspace in which *Jason* agents are used to encapsulate the logic and the control of tasks involved by the mobile application, and artifacts are used as tools for agents to seamlessly exploit available Android device/platform components and services.

From a conceptual viewpoint, this approach makes it possible to keep the same level of abstraction – based on agent-oriented concepts – both when designing the application and when concretely implementing it using *Jason* and CArtAgO. In this way we are able to provide developers a uniform guideline – without conceptual gaps between the abstractions used in the analysis and implementation phases – that drives the whole engineering process of a mobile application.

From a programming point of view, the agent paradigm makes it possible to tackle quite straightforwardly some of the main challenges mentioned in previous sections, in particular: (*i*) task and activity oriented behaviours can be directly mapped onto agents, possibly choosing different kinds of concurrent architectures according to the need—either using multiple agents to concurrently execute tasks, or using a single agent to manage the interleaved execution of multiple tasks; (*ii*) agents' capability of adapting the behaviour on the basis of the current context information can be effectively exploited to realise context-sensitive and context-dependent applications; (*iii*) asynchronous interactions can be managed by properly specifying the agents' reactive behaviour in relation to the reception of particular percepts (e.g. the reception of a new e-mail).

To see this in practice, we developed a porting of JaCa on top of Android – referred as JaCa-Android – available as open-source project[7]. Fig. 1 shows an abstract representation of the levels characterising the JaCa-Android platform and of a generic applications running on top of it.

[6]http://developer.android.com/sdk/index.html

[7]http://jaca-android.sourceforge.net

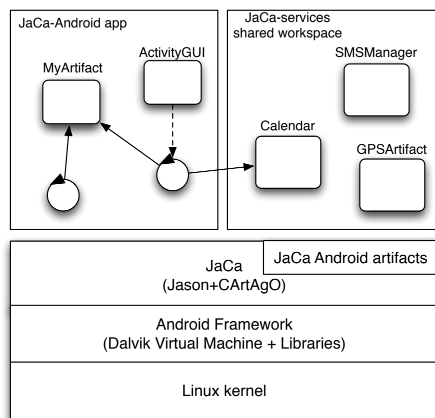

Fig. 1. Abstract representation of the JaCa-Android platform – with in evidence the different agent technologies on which the platform is based – and of generic applications running on top of it.

The platform includes a set of predefined types of artifacts (`BroadcastReceiverArtifact`, `ActivityArtifact`, `ContentProviderArtifact`, `ServiceArtifact`) specifically designed to build compliant Android components. So, standard Android components become fully-fledged artifacts that agents and agent developers can exploit without worrying and knowing about infrastructural issues related to the Android SDK. This makes it possible for developers to conceive and realise nomadic applications that are seamlessly integrated with the Android SDK, possibly interacting/re-using every component and application developed using the standard SDK. This integration is fundamental in order to guarantee to developers the re-use of existing legacy – i.e. the standard Android components and applications – and for avoiding the development of the entire set of functionalities required by an application from scratch.

Besides, the platform also provides a set of artifacts that encapsulate some of the most common functionalities used in the context of smart nomadic applications. In detail these artifacts are:

- `SMSManager/MailManager`, managing sms/mail-related functionalities (send and receive sms/mail, retrieve stored sms/mail, etc.).
- `GPSManager`, managing gps-related functionalities (e.g. geolocalisation of the device).
- `CallManager`, providing functionalities for handling – answer/reject – phone calls.
- `ConnectivityManager`, managing the access to the different kinds of connectivity supported by the mobile device.
- `CalendarArtifact`, providing functionalities for managing the built-in Google calendar.

These artifacts, being general purpose, are situated in a workspace called jaca-services (see Fig. 1) which is shared by all the JaCa-Android applications—being stored and executed into a proper Android service installed with the JaCa-Android platform. More generally, any JaCa-Android workspace can be shared among different applications—promoting, then, the modularisation and the reuse of the functionalities provided by JaCa-Android applications.

In the next section we discuss more in detail the benefits of the JaCa programming model for implementing smart nomadic applications by considering two samples that have been developed on top of JaCa-Android. These applications are quite simple and their aim is to show how it is possible to address *some* – e.g concurrency issues are not addressed – of the relevant aspects of smart nomadic applications development with JaCa-Android.

IV. Evaluation through Practical Examples

The first example aims at showing how the approach allows for easily realising context-sensitive nomadic applications. For this purpose, we consider a JaCa-Android application inspired to Locale[8], one of the most famous Android applications and also one of the winners of the first Android Developer Contest[9]. This application (JaCa-Locale) can be considered as a sort of intelligent smartphone manager realised using a simple *Jason* agent. The agent during its execution use some of the built-in JaCa-Android artifacts described in Section III and two application-specific artifacts: a `PhoneSettingsManager` artifact used for managing the device ringtone/vibration and the `ContactManager` used for managing the list of contacts stored into the smartphone (this list is an observable property of the artifact, so directly mapped into agents beliefs). The agent manages the smartphone behaviour discriminating the execution of its plans on the basis ofa comparison among its actual context information and a set of user preferences that are specified into the agent's plans contexts. TABLE I reports a snipped of the *Jason* agent used in JaCa-Locale, in particular the plans shown in TABLE I are the ones responsible of the context-dependent management of the incoming phone calls.

The behaviour of the agent, once completed the initialisation phase (lines 00-07), is governed by a set of reactive plans. The first two plans (lines 9-15) are used for regulating the ringtone level and the vibration for the incoming calls on the basis ofthe notifications provided by the `CalandarArtifact` about the start or the end of an event stored into the user calendar. Instead, the behaviour related to the handling of the incoming calls is managed by the two reactive plans `incoming_call` (lines 17-28). The first one (lines 17-19) is applicable when a new incoming call arrives and the phone owner is not busy, or when the incoming call is considered critical. In this case the agent normally handles the incoming call – the ringtone/vibration settings have already been regulated by the plans at lines 9-15 – using the `handle_call` operation provided by the `CallManager` artifact. The second plan instead (lines 21-28) is applicable when the user is busy and the call does not come from a relevant contact. In this case the phone call is automatically

[8]http://www.twofortyfouram.com/

[9]http://code.google.com/intl/it-IT/android/adc/

```
00  !init.
01
02  +!init
03    <- focus("SMSManager"); focus("MailManager");
04       focus("CallManager"); focus("ContactManager");
05       focus("CalendarArtifact");
06       focus("PhoneSettingsManager");
07       focus("ConnectivityManager").
08
09  +cal_event_start(EventInfo) : true
10    <- set_ringtone_volume(0);
11       set_vibration(on).
12
13  +cal_event_end(EventInfo) : true
14    <- set_ringtone_volume(100);
15       set_vibration(off).
16
17  +incoming_call(Contact, TimeStamp)
18    : not busy(TimeStamp) | is_call_critical(Contact)
19    <- handle_call.
20
21  +incoming_call(Contact, TimeStamp)
22    : busy(TimeStamp) & not is_call_critical(Contact)
23    <- get_event_description(TimeStamp,EventDescription);
24       drop_call;
25       .concat("Sorry, I'm busy due
26         to", EventDescription, "I will call you back
27         as soon as possible.", OutStr);
28       !handle_auto_reply(OutStr).
29
30  +!handle_auto_reply(Reason) : wifi_status(on)
31    <- send_mail("Auto-reply", Reason).
32
33  +!handle_auto_reply(Reason): wifi_status(off)
34    <- send_sms(Reason).
```

TABLE I
SOURCE CODE SNIPPET OF THE JACA-LOCALE JASON AGENT

rejected using the `drop_call` operation of the `CallManager` artifact (line 24), and an auto-reply message containing the motivation of the user unavailability is send back to the contact that performed the call. This notification is sent – using one of the `handle_auto_reply` plans (lines 30-34) – via sms or via mail (using respectively the `SMSManager` or the `MailManager`) depending on the current availability of the WiFi connection on the mobile device (availability checked using the `wifi_status` observable property of the `ConnectivityManager`). It is worth remarking that `busy` and `is_call_critical` refer to rules – not reported in the source code – used for checking respectively: (*i*) if the phone owner is busy – by checking the belief related to one of the `CalendarArtifact` observable properties (`current_app`) – or (*ii*) if the call is critical – by checking if the call comes from one of the contact in the `ContactManager` list considered critical: e.g. the user boss/wife.

Generalising the example, context-sensitive applications can be designed and programmed in terms of one or more agents with proper plans that are executed only when the specific context conditions hold.

The example is useful also for highlighting the benefits introduced by artifact-based endogenous environments: (*i*) it makes it possible to represent and exploit platform/device functionalities at an agent level of abstractions – so in terms of actions and perceptions, modularised into artifacts; (*ii*) it provides a strong *separation of concerns*, in that developers can fully separate the code that defines the control logic of the application (into agents) from the reusable functionalities (embedded into artifact) that are need by the application, making agents' source code more dry.

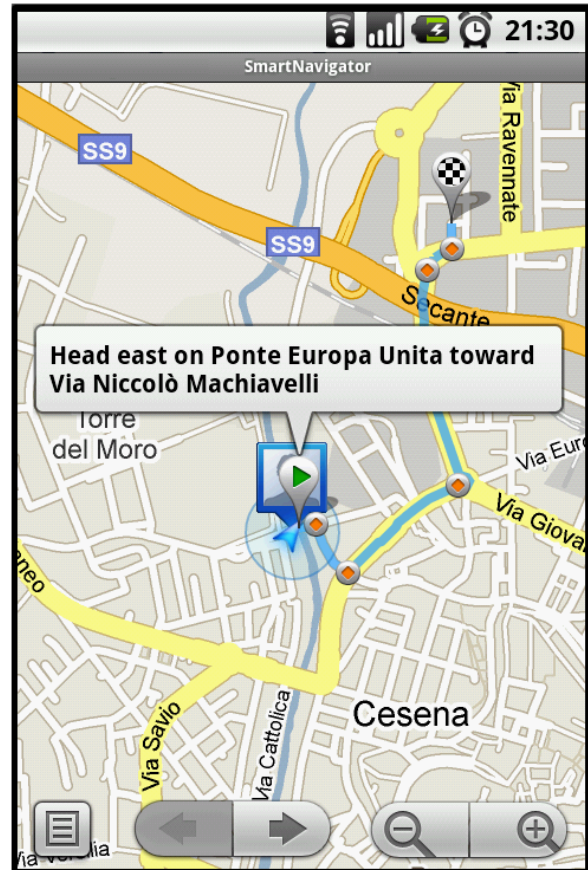

Fig. 2. Screenshot of the SmartNavigator application that integrate in its GUI some of the Google Maps components for showing: (*i*) the user current position, (*ii*) the road directions, and (*iii*) the route to the designed destination.

The second application sample – called `SmartNavigator` (see Fig. 2 for a screenshot) – aims at showing the effectiveness of the approach in managing asynchronous interactions with external resources, such as – for example – Web services. This application is a sort of smart navigator able to assist the user during its trips in an intelligent way, taking into the account the current traffic conditions.

The application is realised using a single *Jason* agent and four different artifacts: (*i*) the `GPSManager` used for the smartphone geolocalisation, (*ii*) the `GoogleMapsArtifact`, an artifact specifically developed for this application, used for encapsulating the functionalities provided by Google Maps (e.g. calculate a route, show points of interest on a map, etc.), (*iii*) the `SmartNavigatorGUI`, an artifact developed on the basis ofthe `ActivityArtifact` and some other Google Maps components, used for realizing the GUI of the application and (*iv*) an artifact, `TrafficConditionsNotifier`, used for managing the interactions with a Web site[10] that provides real-time traffic information.

TABLE II shows a snippet of the agent source code.

[10]http://www.stradeanas.it/traffico/

The agent main goal `assist_user_trips` is managed by a set of reactive plans that are structured in a hierarchy of sub-goals – handled by a set of proper sub-plans. The agent has a set of initial beliefs (lines 00-01) and an initial plan (lines 5-9) that manages the initialisation of the artifacts that will be used by the agent during its execution. The first plan, reported at lines 11-12, is executed after the reception of an event related to the modification of the `SmartNavigatorGUI route` observable property – a property that contains both the starting and arriving locations provided in input by the user. The handling of this event is managed by the `handle_navigation` plan that: (*i*) retrieve (line 15) and updates the appropriate agent beliefs (line 16 and 19), (*ii*) computes the route using an operation provided by the `GoogleMapsArtifact` (`calculate_route` lines 17-18), (*iii*) makes the subscription – for the route of interest – to the Web site that provides the traffic information using the `TrafficConditionsNotifier` (lines 20-21), and finally (*iv*) updates the map showed by the application (using the `SmartNavigatorGUI` operations `set_current_position` and `update_map`, lines 22-23) with both the current position of the mobile device (provided by the observable property `current_position` of the `GPSManager`) and the new route.

In the case that no meaningful changes occur in the traffic conditions and the user strictly follows the indications provided by the `SmartNavigator`, the map displayed in the application GUI will be updated, until arriving to the designed destination, simply moving the current position of the mobile device using the plan reported at lines 34-38. This plan, activated by a change of the observable property `current_position`, simply considers (using the sub-plan `check_position_consistency` instantiated at line 36, not reported for simplicity) if the new device position is consistent with the current route (retrieved from the agent beliefs at line 35) before updating the map with the new geolocation information (line 37-38). In the case in which the new position is not consistent – i.e. the user chose the wrong direction – the sub-plan `check_position_consistency` fails. This fail is handled by a proper *Jason* failure handling plan (lines 40-42) that simply re-instantiate the `handle_navigation` plan for computing a new route able to bring the user to the desired destination from his current position (that was not considered in the previous route).

Finally, the `new_traffic_info` plan (lines 25-32) is worth of particular attention. This is the reactive plan that manages the reception of the updates related to the traffic conditions. If the new information are considered relevant with respect to the user preferences (sub-plan `check_info_relevance` instantiated at line 28 and not shown) then, on the basis ofthis information, the current route (sub-plan `update_route` instantiated at lines 29-30), the Web site subscription (sub-plan `update_subscription` instantiated at line 31), and finally the map displayed on the GUI (line 32) are updated.

So, this example shows how it is possible to integrate the reactive behaviour of a JaCa-Android application – in this example the asynchronous reception of information from a certain source – with its pro-active behaviour — assisting the user during his trips. This integration allows to easily modify and adapt the pro-active behaviour of an application after the reception of new events that can be handled by proper reactive plans: in this example, the reception of the traffic updates can lead the `SmartNavigator` to consider a new route for the trip on the basis ofthe new information.

```
00  preferences([...]).
01  relevance_range(10).
02
03  !assist_user_trips.
04
05  +!assist_user_trips
06    <- focus("GPSManager");
07       focus("GoogleMapsArtifact");
08       focus("SmartNavigatorGUI");
09       focus("TrafficConditionsNotifier").
10
11  +route(StartLocation, EndLocation)
12    <- !handle_navigation(StartLocation, EndLocation).
13
14  +!handle_navigation(StartLocation, EndLocation)
15    <- ?relevance_range(Range); ?current_position(Pos);
16       -+leaving(StartLocation);-+arriving(EndLocation);
17       calculate_route(StartLocation,
18         EndLocation, OutputRoute);
19       -+route(OutputRoute);
20       subscribe_for_traffic_condition(OutputRoute,
21         Range);
22       set_current_position(Pos);
23       update_map.
24
25  +new_traffic_info(TrafficInfo)
26    <- ?preferences(Preferences);
27       ?leaving(StartLocation); ?arriving(EndLocation);
28       !check_info_relevance(TraffincInfo,Preferences);
29       !update_route(StartLocation, EndLocation,
30         TrafficInfo, NewRoute);
31       !update_subscription(NewRoute);
32       update_map.
33
34  +current_position(Pos)
35    <- ?route(Route);
36       !check_position_consistency(Pos, Route);
37       set_current_position(Pos);
38       update_map.
39
40  -!check_position_consistency(Pos, Route)
41    : arriving(EndLocation)
42    <- !handle_navigation(Pos, EndLocation).
```

TABLE II
SOURCE CODE SNIPPET OF THE SMARTNAVIGATOR JASON AGENT

V. Open Issues and Future Work

Besides the advantages described in previous section, the application of current agent programming technologies to the development of concrete software systems such as nomadic applications have been useful to focus some main weaknesses that these technologies currently have to this end. Here we have identified three general issues that will be subject of future work:

(i) Devising of a notion of type *for agents and artifacts* — current agent programming languages and technologies lack of a notion of type as the one found in mainstream programming languages and this makes the development of large system hard and error-prone. This would make it possible to detect many errors at compile time, allowing for strongly reducing the development time and enhancing the safety of the developed system. In JaCa we have a notion of type just for artifacts: however it is based on the lower OO layer and

so not expressive enough to characterise at a proper level of abstraction the features of environment programming.

(ii) Improving modularity *in agent definition* — this is a main issue already recognised in the literature [7], [8], [9], where constructs such as *capabilities* have been proposed to this end. *Jason* still lacks of a construct to properly modularise and structure the set of plans defining an agent's behaviour —a recent proposal is described here [12].

(iii) Improving the integration with the OO layer — To represent data structures, *Jason* – as well as the majority of agent programming languages – adopts Prolog terms, which are very effective to support mechanisms such as unification, but quite weak – from an abstraction and expressiveness point of view – to deal with complex data structures. Main agent frameworks (not languages) in Agent-Oriented Software Engineering contexts – such as Jade [2] or JACK[11] – adopt object-oriented data models, typically exploiting the one of existing OO languages (such as Java). By integrating *Jason* with CArtAgO, we introduced a first support to work with an object-oriented data model, in particular to access and create objects that are exchanged as parameters in actions/percepts. However, it is just a first integration level and some important aspects – such as the use of unification with object-oriented data structures – are still not tackled.

Finally, concerning the specific mobile application context, JaCa-Android is just a prototype and indeed needs further development for stressing more in depth the benefits provided by agent-oriented programming compared to mainstream non-agent platforms. Therefore, in future works we aim at improving JaCa-Android in order to tackle some other important features of modern nomadic applications such as the smart use of the battery and the efficient management of the computational workload of the device.

VI. Conclusion

To conclude, we believe that agent-oriented programming would provide a suitable level of abstraction for tackling the development of complex software applications, extending traditional programming paradigms such as the Object-Oriented to deal with aspects such as concurrency, reactiveness, asynchronous interaction managements, dynamism and so on. In this paper, in particular, we showed the advantages of applying such an approach to the development of smart nomadic applications on the Google Android platform, exploiting the JaCa integrated platform. However, we argue that in order to stress and investigate the full value of the agent-oriented approach to this end, further work is need to extend current agent languages and technologies – or to devise new ones – tackling main aspects that have not been considered so far, being not related to AI but to the principles of software development. This is the core of our current and future work.

References

[1] A. Battestini, C. Del Rosso, A. Flanagan, and M. Miettinen. Creating next generation applications and services for mobile devices: Challenges and opportunities. In *EEE 18th Int. Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC), 2007*, pages 1 –4, 3-7 2007.

[2] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[3] M. Berger, S. Rusitschka, D. Toropov, M. Watzke, and M. Schlichte. Porting distributed agent-middleware to small mobile devices. In *AAMAS Workshop on Ubiquitous Agents on Embedded, Wearable and Mobile Devices*.

[4] R. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications (vol. 1)*. Springer, 2005.

[5] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, Ltd, 2007.

[6] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications (vol. 2)*. Springer Berlin / Heidelberg, 2009.

[7] L. Braubach, A. Pokahr, and W. Lamersdorf. Extending the capability concept for flexible BDI agent modularization. In *Programming Multi-Agent Systems*, volume 3862 of *LNAI*, pages 139–155. Springer, 2005.

[8] M. Dastani, C. Mol, and B. Steunebrink. Modularity in agent programming languages: An illustration in extended 2APL. In *Proceedings of the 11th Pacific Rim Int. Conference on Multi-Agent Systems (PRIMA 2008)*, volume 5357 of *LNCS*, pages 139–152. Springer, 2008.

[9] K. Hindriks. Modules as policy-based intentions: Modular agent programming in GOAL. In *Programming Multi-Agent Systems*, volume 5357 of *LNCS*, pages 156–171. Springer, 2008.

[10] F. Koch, J.-J. C. Meyer, F. Dignum, and I. Rahwan. Programming deliberative agents for mobile services: The 3apl-m platform. In *PROMAS*, pages 222–235, 2005.

[11] B. König-Ries. Challenges in mobile application development. *it - Information Technology*, 51(2):69–71, 2009.

[12] N. Madden and B. Logan. Modularity and compositionality in Jason. In *Proceedings of Int. Workshop Programming Multi-Agent Systems (ProMAS 2009)*. 2009.

[13] C. Muldoon, G. M. P. O'Hare, R. W. Collier, and M. J. O'Grady. Agent factory micro edition: A framework for ambient applications. In *Int. Conference on Computational Science (3)*, pages 727–734, 2006.

[14] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17 (3), Dec. 2008.

[15] A. S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In *MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away*, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[16] A. Ricci, M. Piunti, L. D. Acay, R. Bordini, J. Hübner, and M. Dastani. Integrating artifact-based environments with heterogeneous agent-programming platforms. In *Proceedings of 7th International Conference on Agents and Multi Agents Systems (AAMAS08)*, 2008.

[17] A. Ricci, M. Piunti, M. Viroli, and A. Omicini. Environment programming in CArtAgO. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications, Vol. 2*, pages 259–288. Springer, 2009.

[18] A. Ricci, A. Santi, and M. Piunti. Action and perception in multi-agent programming languages: From exogenous to endogenous environments. In *Proceedings of the Int. Workshop on Programming Multi-Agent Systems (ProMAS'10)*, Toronto, Canada, 2010.

[19] A. Ricci, M. Viroli, and A. Omicini. The A&A programming model & technology for developing agent environments in MAS. In M. Dastani, A. El Fallah Seghrouchni, A. Ricci, and M. Winikoff, editors, *Programming Multi-Agent Systems*, volume 4908 of *LNAI*, pages 91–109. Springer Berlin / Heidelberg, 2007.

[20] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[11] http://www.agent-software.com