MoCoA: Customisable Middleware for Context-aware Mobile Applications

Aline Senart, Raymond Cunningham, Mélanie Bouroche, Neil O'Connor, Vinny Reynolds and Vinny Cahill

Distributed Systems Group, Department of Computer Science,
Trinity College Dublin, Ireland,
first.last@cs.tcd.ie

Abstract. Many programming models have been proposed to facilitate the development of context-aware applications. However, previous work does not offer support for building customised systems and has largely been targeted at a single application domain. In this paper, we describe MoCoA, a flexible middleware framework that permits the rapid development of context-aware applications and supports deployment scenarios ranging from augmented artefacts to city-wide smart-space applications. Crucially, MoCoA supports a small set of programming abstractions that are suitable for building a wide range of context-aware applications for deployment in a fixed or (ad hoc) mobile environment. For each of these abstractions, MoCoA provides a set of implementations via a library of components. We present three applications of the MoCoA framework that demonstrate both the use of the programming abstractions and the flexibility of the framework.

1 Introduction

Enabled by recent and expected developments in new sensor technologies, by wireless networking, and by miniaturisation of computational devices, context-aware mobile computing has become a reality. We can now envision a class of applications involving large collections of collaborating mobile devices that operate by sensing their surrounding environment and adapting their behaviour accordingly without human intervention [1]. These applications will typically address areas such as environmental monitoring, independent living, intelligent transportation systems, mobile robotics, and city-wide smart spaces.

Even though many programming models, supported by associated middleware, have been proposed to facilitate the development of context-aware applications, to date, none offers the flexibility required by the application space. They

are typically only appropriate for a single application domain or deployment scenario, e.g., smart spaces [2, 3], tracking of valuable goods [4, 5] or robotics [6, 7]. Moreover, they fail to handle some or all of the non-functional requirements expected of emerging context-aware applications such as timeliness, reliability, mobility and scalability.

The main contribution of this paper is the identification of a small set of abstractions that can be used to support a wide range of context-aware (mobile) applications. These abstractions have been implemented in MoCoA, a flexible middleware framework for building context-aware applications for deployment in fixed or (ad hoc) mobile environments. We describe how context-aware applications can be modeled with these abstractions and describe their implementation in a library of commonly-required and optional components that permits the rapid development of applications. Examples of components include specific sensors and actuators and high-level services such as context acquisition, event-based communication, and intelligent reasoning. These components can be assembled and specialised by developers to form customised middleware that fits application requirements exactly. The use of replaceable components and well-defined interfaces improves code reuse and allows the MoCoA framework to address multiple combinations of non-functional requirements, and therefore to be suitable for a wide range of applications.

To demonstrate the use of the programming abstractions and the flexibility of MoCoA, we have designed a number of representative context-aware applications that exhibit a variety of different non-functional requirements. Among these, we describe in this paper a sensor-augmented sofa with application in independent living, an autonomous mobile robot, and a simulated urban traffic control system. These experiments in assembling and customising components from our library serve to validate our approach.

The remainder of this paper is structured as follows. In Section 2, we describe three applications used in the paper to demonstrate the potential of MoCoA and introduce the key requirements of context-aware (mobile) applications. In Section 3, we present the programming abstractions and components supported by MoCoA. Then, in Section 4, we describe and discuss the use of these abstractions and of MoCoA in the application scenarios. Finally, Section 5 and Section 6 present related work and our conclusions respectively.

2 Application Requirements

Customisable middleware supporting common programming abstractions for the envisaged generation of context-aware applications is a key requirement for supporting a range of deployment scenarios. As discussed below, such middleware need to support non-functional requirements in order to cope with large-scale distributed applications, composed of heterogeneous and mobile devices interacting closely with a changing environment. This section presents examples of representative applications exhibiting non-functional requirements and derives requirements that context-aware applications have on middleware.

2.1 Application Scenarios

We introduce a few scenarios that are representative of expected classes of sensor-driven context-aware applications and indicate their main requirements.

The Sentient Sofa Falls present a serious health risk. Among older adults, falls are the leading cause of injury deaths and the most common cause of injuries and hospital admissions for trauma [8]. [9] argues that "technological devices such as alarm systems that are activated when patients try to get out of bed or move unassisted may be useful".

Motivated by these observations, we have designed the "sentient sofa" as a prototype of a bed for elderly or disabled people that raises an alarm if the occupant leaves the bed, for example, as a result of a fall. This scenario is representative of the *augmented artefact* class of context-aware application, where sensor-enabled systems are embedded in everyday environments, and addresses one of the most important application domains for this technology in the future, independent living.

The sentient sofa can identify the person lying in the bed, monitor her/his movements and autonomously decide to alert carers in case of unexpected behaviour. To achieve this, the sofa is equipped with a fixed number of load sensors enabling the mass currently on it to be measured. This assisted living application is relatively small scale and doesn't require mobility support.

Sentient Traffic Lights Traffic congestion is an increasing problem in urban areas but is alleviated to some extent by urban traffic control (UTC) systems that control and coordinate traffic signal timings in order to reduce journey times and delays [10, 11]. Future UTC systems may be able to exploit a wide variety of fine-grained sensor data, e.g., GPS position data from individual vehicles, to inform attempts to optimise signal timings.

To explore these possibilities, we have designed a city-scale (simulated) UTC system in which traffic light controllers use sensor data from local vehicles to inform their decision making [12]. In this scenario, the traffic light controllers also exchange views of their environment describing the traffic conditions sensed locally through sensor data fusion. By incorporating these views into their reasoning, traffic light controllers can establish convergent views of the congestion level, which can in turn inform their local decisions as to which signal phase is currently most appropriate. Positive (or negative) feedback from the environment (e.g., a decrease or increase in the local congestion level) reinforces (or discourages) the selection of such phases and enables good strategies to be learned over time.

This scenario is representative of the class of *city-wide smart space* applications, that are typically large-scale and composed of many geographically dispersed entities. Traffic light controllers in this application are not mobile, but they need to communicate with a variable collection of anonymous mobile vehicles as well as a fixed collection of nearby junction controllers. A completely

decentralised learning technique is required so that consensus can be established on the optimal phases to choose at a junction.

Sentient Robots An autonomous mobile robot is a mechanical device that performs its tasks under real-time conditions, in a previously unknown environment, without human supervision. Potential applications include dull or dirty tasks like floor cleaning, transportation and surveillance, and tasks that are dangerous for humans, such as military missions [13] and emergency rescue [14].

To represent this class of application, we have designed simple sentient robots that navigate autonomously towards some destination, while avoiding obstacles and obeying traffic lights they encounter en route. In order to complete their mission, the sentient robots are equipped with various sensors, as illustrated in Figure 1. The outputs of these sensors are fused together in order to determine the robots' context with respect to obstacles and traffic lights and thereby infer what are the appropriate actions to take in order to avoid collisions. In this scenario, timely communication is required in order to ensure that the robots can react in time, for example, in response to a change of the traffic light colour.

Fig. 1. The sensor-augmented robot

This scenario is typical of *robotics* applications, where mobility and safety constraints are key requirements. As robots are mobile, they spontaneously communicate with anonymous neighbours that they discover in their vicinity, e.g., traffic lights or other robots. Furthermore, strong real-time constraints on communication, decision-making and reaction have typically to be guaranteed in order to ensure safety for this type of application.

2.2 Fundamental Requirements

From the scenarios studied previously, we can establish a list of requirements that middleware for (mobile) context-aware applications should support.

- **Spontaneous interaction:** Applications will be composed of a dynamically changing population of interacting devices. Hence, unanticipated interaction between nearby devices has to be supported, enabling a device to dynamically establish connections to other devices within its current vicinity.

- **Geographical dispersion**: Unlike current embedded systems, future context-aware applications will integrate components that are scattered over buildings, countries, and continents.
- **Mobility**: Parts of an application will possess the ability to move geographically (by virtue of being carried by a mobile device) or between hosts possibly on different networks, while remaining in continuous operation.
- **Large-scale**: Applications may be composed of millions of interacting hardware and software components and will be able to expand indefinitely.
- **Autonomy**: Typical context-aware applications will operate independently of human control and will be able of acting in a decentralised fashion according to their own knowledge.
- **Inference and learning:** Applications will typically have to cope with changing conditions during their lifetime. Not only must they be designed to reason about these changes according to their factual knowledge, but they must also gain knowledge through experience, e.g., by learning appropriate behaviour through trial-and-error interactions with their environment.
- **Time criticality**: These applications will interact with the physical environment within a given time interval, and will have to cope with its pace, regardless of adverse conditions due to scale and technology shortcomings.
- **Safety criticality**: As such applications are based on mobile devices embedded in everyday environments, they must obey strong safety requirements to interact with human users, whose well-being will frequently rely on them.

3 MoCoA: Middleware for Context-aware Applications

In this section, we describe the programming abstractions that are supported by MoCoA for the development of context-aware mobile applications. For each of the abstractions, we propose optional components that are available in MoCoA's library and that can be composed and customised to build the applications.

3.1 Sentient Objects, Sensors and Actuators

In the MoCoA framework, applications are structured using the ***sentient object*** abstraction [15, 16]. Sentient objects are mobile intelligent entities, that extract, interpret and use context information obtained from sensors, other sentient objects, and their computational infrastructure, to drive their behaviour. The granularity of a sentient object is not constrained: a robot, a component of a robot, or a traffic light controller are all potential examples of sentient objects.

In this model, a ***sensor*** is seen as an entity that produces software events in reaction to a real-world stimulus and is an abstraction of some physical device. To act upon their environment, sentient objects use ***actuators.*** An actuator is any component that consumes software events and reacts by attempting to change the state of the real-world via some physical device. To facilitate mobility as well as loose coupling between dynamically varying collections of anonymous mobile devices, a sentient object is both a producer and a consumer of asynchronous

events. As explained above and illustrated in Figure 2, a sentient object can also receive events from its infrastructure. Notice also that sentient objects may communicate directly by means of events or indirectly via the environment.

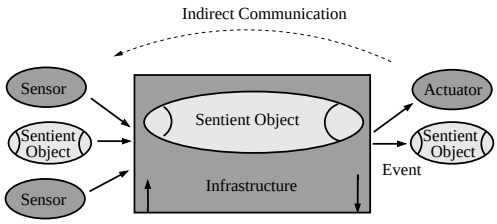

Fig. 2. An example of a sentient object

The behaviour of all sentient objects follows the same pattern. First, a sentient object may receive input from a variety of sources (sensors, sentient objects and infrastructure) that needs to be integrated before being used in determining the overall context of the sentient object. As an example, our robots are equipped with ultrasonic, orientation, and location sensors. The outputs of these sensors are fused together with input from nearby traffic lights in order to determine the robots' context with respect to obstacles, traffic lights, and their destination.

Context recognition determines the current context, based on fused data and past history. For example, that someone has left their bed is inferred from historical information that someone was in the bed and new sensor input indicating that they are no longer present.

Sentient objects are then expected to change their behaviour or act upon their environment based on some rules. This implies some form of intelligent reasoning captured in an inference component. Rules may be predefined or learned over time. In the robot scenario, the inference component reasons on a set of predefined rules, in order to take actions and/or to derive a higher-level context. For example, the detection of an obstacle results in a transition to the obstacle avoidance context, followed by a braking process. In the UTC scenario, rules are used to choose the appropriate signal phase given the current congestion level of the system and are learned based on feedback from the environment.

3.2 Events, Event Channels, and Proximities

While the basic concept of event-based communication is simple, there are a number of difficult issues to be tackled in large-scale and geographically dispersed applications in which large volumes of events are raised. In MoCoA, we support two abstractions to control event propagation: proximities and event channels.

Filtering ensure that events are only propagated to consumers that have expressed an interest in them [17]. Filters have typically been applied to the

subject or the content of an event. In addition to these filters, MoCoA also supports proximity-based filtering which is used to define geographical areas, ***proximities***, within which events are valid. Such proximity-based filtering confines propagation of events to a geographical area surrounding the producer since locality considerations suggest that the closer event consumers are located to a producer the more likely they are to be interested in the events that it produces.

Proximities not only serve as a unit of scoping large-scale systems into areas of localised interaction, but also provide the basis for time-bounded event delivery, even in wireless networks. ***Event channels*** are defined to specify constraints on propagation and delivery of asynchronous events within a proximity. An event channel is specific to an event type and allows a producer to define the real-time guarantees that have to be maintained within a given geographical area.

In the robot scenario, only the local geographical environment is required to be sensed, all other information outside the vicinity of the individual robots is not relevant and can be filtered out. Similarly, traffic lights disseminate their status to a nearby proximity over an event channel providing real-time delivery.

To support event-based communication with event channels and proximities, the MoCoA framework uses different instantiations of the STEAM event service [18]: STEAM for use in wireless ad hoc networks, and RT-STEAM for soft and hard real-time in ad hoc environments. In STEAM, proximities can be circular or hull-shaped. The set of filters available depends on the type of event parameters. For example, if the parameter is a location, the following filters can be used: `distanceIncreases`, `distanceDecreases`, `withinRange`, and `beyondRange`. Parameter types include location, time, integer, double and string.

Remote and Local Event Channels Given that sentient objects may communicate with collocated sensors and actuators, wireless communication is not always necessary. For instance, the ultrasound sensors mounted on the robots are a priori known and their location is fixed. In this case, when communication with identified sensors and actuators that are not discovered dynamically is required, STEAM uses local event channels. In contrast, communication via the network is required to support spontaneous interactions with anonymous producers of events that are discovered dynamically, e.g., remote event channels are used between robots and traffic lights encountered en route.

Event Channels with Timeliness Constraints To interact in the real-world, sentient objects may require temporal guarantees. To provide them, even where changes in the quality of communication are frequent, the proximities in which events are to be delivered are adapted to reflect changes in the underlying infrastructure under a model known as space-elastic adaptation [19].

In essence, when there is a change, like a disconnection or a significant decrease in message-delivery latency, that would prevent the temporal properties required by an event channel being achieved, the underlying infrastructure notifies the event producer by means of an adaptation event (an example of an event raised by the infrastructure) describing a revised proximity in which the

required guarantees can be met. The producer is then free to dynamically adapt its behaviour in order to respect the safety and timeliness requirements of the application. Even in an ad hoc network subject to partitions, this model provides well-defined guarantees to applications allowing stringent safety constraints to be enforced. For example, even if communication between one of our sentient robots and the traffic lights in its path is not possible, it is still possible to guarantee that no robot will ever pass through a red light. In this case, the traffic light adapts its behaviour when it realises that messages are not being delivered in a large enough proximity for the robots to stop safely [20].

In MoCoA, the RT-STEAM component is used to support the definition of hard real-time event channels and space-elastic adaptation. RT-STEAM is implemented using RTAI [21] and relies on further components providing real-time resource reservation and routing in ad hoc networks, as well as time-bounded media access control [22, 23].

3.3 Readings and Facts

Sensor fusion is used to manage the uncertainty of data captured from the real-world and to derive higher-level context information. In MoCoA, the fusion process relies on a set of pipelines, each pipeline being composed of a combination of generic and sharable components (c.f. Figure 3). Input events in a pipeline are processed through different stages.

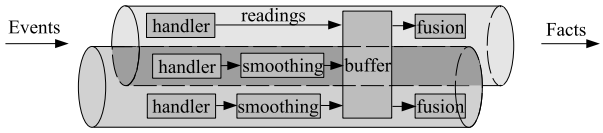

Fig. 3. Example of two pipelines

First, a pipeline extracts ***readings*** (i.e., raw values produced by a sensor, a sentient object, or the local infrastructure) from events. Then, a sequence of transformations is applied to the readings by the components present in the pipeline. The final result is a piece of higher-level information, in the form of one or more ***observed facts***.

A pipeline may be composed of different components: handler, smoothing, buffer, and fusion (see Table 1). MoCoA already provides a set of implementations of these components, e.g., fusion can perform a sum, an average function or might rely on a Bayesian network. However, we expect developers to implement their own (targeted to real-time usage or not), extending the existing library.

Handler	Extracts readings from events and propagates them in the pipeline.
Smoothing	Preprocesses readings before fusion.
Buffer	Stores readings delivered at different times, allowing synchronised fusion or smoothing.
Fusion	Derives high-level observed facts from readings.

Table 1. Types of pipeline components

3.4 Contexts

Context is defined as any information currently available in the environment that can be used to characterise the situation of an entity [24], such as its current location, the presence of other sentient objects in its vicinity or the state of its underlying infrastructure. In MoCoA, the contexts in which a sentient object can be during its lifetime are organised as a context graph, where only a subset of contexts can be transitioned to from the current context. Not only are contexts useful to structure complex applications, contexts are also the basic abstraction for Context-Based Reasoning [25] within a sentient object allowing the set of event channels, pipelines, facts, rules and actions that are relevant at any time to be filtered. When in a context, only one pipeline is active thereby constraining the set of event channels being used. Because the number of pipelines is restricted, only some facts may be asserted in this context. Subsequently, as depicted in Figure 4, only a subset of rules needs to be evaluated and the permitted actions are limited.

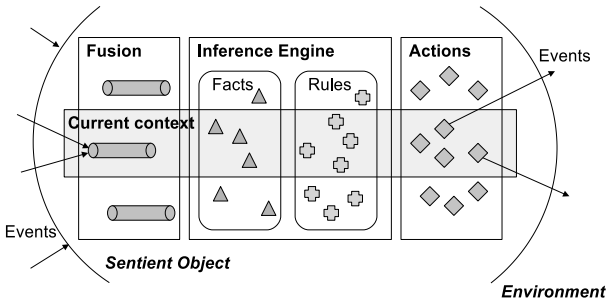

Fig. 4. The MoCoA architecture

3.5 Rules and Actions

The knowledge of a sentient object is structured into facts. To ultimately determine the appropriate behaviour of a sentient object in response to its environment, an inference engine is used to infer knowledge (i.e., to assert ***derived***

facts) from previously asserted facts and to select the actions to be taken. Table 2 summarises the different types of ***action*** that an object is able to perform. The action selection decision within a sentient object is based on ***rules.*** Rules may be predefined as applicable within specific contexts, or rules defining what actions to take in specific contexts may be learned.

Fact assertion	Action that asserts a fact.
Fact retraction	Action that retracts a fact.
Event production	Action that produces an event.
Execution of code	Action that carries out a portion of code.

Table 2. Types of actions

In MoCoA, a first order logic inference engine reasons about a set of facts and predefined rules in order to derive intelligent behaviour. The rules take the form of `condition/actions`. Conditions are expressed in terms of asserted facts and can include operators, e.g., `&&`, `||`, $\exists$ and $\forall$. The actions associated with a rule are executed when the condition evaluates to true. The inference engine that we have implemented uses forward-chaining. Rules are only evaluated when a fact relevant to their condition is asserted or retracted, i.e., by relevant changes to the set of facts. For example, the transition rule `(onCourse && !atWaypoint)` in the robot scenario is only evaluated if the facts `onCourse` or `atWaypoint` are changed in the knowledge base.

Since the set of actions available is typically fixed (e.g., constrained by the available actuators) rules may be learned by attempting actions in different situations. Depending on the feedback returned from the environment, this process attempts to learn the long term reward of choosing these actions again given the current context.

Collaborative reinforcement learning (CRL), is a technique for building decentralised coordination models, that extends reinforcement learning [26] with feedback models to update an agent's (in this case, a sentient object's) policy and enables a consensus to emerge between individual policies [27]. Each sentient object uses its own policy to decide probabilistically on which action to take to attempt to achieve its goal. In dynamic distributed systems, sentient objects may also have a changing number of neighbouring sentient objects that can be used to help achieve the goal. For example, when a sentient object either cannot achieve the goal locally or when the estimated cost of achieving it locally is higher than the estimated cost of a neighbour solving it, an event is sent to its neighbour to delegate the action. Furthermore, each sentient object maintains a local view of its neighbours by exchange of advertisements.

3.6 Composition

Through an high-level API in Java, software architects can design sentient objects using the basic abstractions. To describe the services provided and required by a component, as well as the dependencies between components, we are defining an architecture description language using XML descriptors. As each component of the library exhibits its dependencies with a descriptor, MoCoA can select the appropriate components according to application requirements, customises them, and finally generates the executable code of the sentient object.

The code generation currently provides C++, but as it is decoupled from the object design process, different languages may be easily incorporated. Additionally, an XML descriptor of the sentient object, reusable in other applications, may be generated. To further assist the construction of sentient objects, we are currently extending MoCoA with a graphical interface.

4 Modeling the Application Scenarios

We have designed some scenarios representative of different classes of context-aware applications. As they expose different requirements, different components from the MoCoA library are employed. In this section, we describe how the abstractions outlined above are used to model these diverse applications and the components necessary to support each application.

4.1 The Sentient Sofa

Sentient Objects, Sensors and Actuators To monitor movements on the sofa, four load cell sensors fitted to the legs are used. Another sensor is employed to register new users with a keyboard. For this application, two sentient objects have been defined. As they are collocated and identified, local communication is used to support their interactions within a proximity that corresponds to the sofa. While the `sentient sofa` object produces events for the current mass on the sofa and its position, a `recogniser` sentient object uses this information to identify the user or to register him/her when necessary. Finally, to alert carers, an actuator producing an audio stream via a set of speakers is used.

Events, Event Channels and Proximities Sensor fusion consists of summing readings extracted from the events of the four load sensors in order to determine the total weight of the person (each reading corresponding to the weight applied to one of the legs of the sofa). The location is then determined by comparing each sensor reading with the average of the four readings: a person is said to be in one quarter if the load measured by the corresponding sensor is greater or equal to the average of the loads read on all sensors. Consequently, sum and average are used for the fusion of sensor information in this scenario.

Contexts The context graph of the `sentient sofa` is small, one context is identified as `Empty` when there is no one on the sofa and a composite context is `InUse` when there is someone on it. Fifteen distinct sub contexts of `InUse` can be transitioned to depending on the location and movements of the patient.

Rules and Actions The behaviour of the `sentient sofa` object is rule-based and consists of the publication of events of type `massChanged` when its context changes for consumption by the `recogniser` sentient object, that in turn raises an alarm in case of unexpected behaviour, or registers a new person if the new mass does not correspond to the weight of a previous user. An example transition rule is illustrated below, where `mass` and `average` are observed facts asserted by the fusion components. This rule states that if the load cell sensors on the bottom right and bottom left are measuring a mass greater than the average mass on the sofa, whilst load cell sensors on the top are measuring a mass less than the average, then a transition to the context `bottom` is triggered.

```
(mass.id == bottomleft && mass.value > average)
&& (mass.id == bottomright && mass.value > average)
&& (mass.id == topleft && mass.value < average)
&& (mass.id == topright && mass.value < average)/
transition(bottom)
```

The evaluation of this rule followed by the context switch takes approximately 91μs. Since the sofa and its sensors/actuators occupy a fixed location, components supporting mobility are not included in the application.

The sentient sofa runs on Debian GNU/Linux and stores user profiles in a relational database. It was first developed with a CORBA interface, including in total 1789 lines of code. The new implementation of the sofa generated with MoCoA represents for the two sentient objects around 900 lines of code and requires only 287 additional lines for the database access. Using MoCoA, even for a small sensor-augmented artefact, reduces the need for complex and error-prone programming.

4.2 Sentient Traffic Lights

Sentient Objects, Sensors and Actuators In this large-scale and highly decentralised application, which aims to minimise the average waiting time of all vehicles in the network, we have modeled traffic light controllers as sentient objects. Each controller manage a 4-way crossroad and obtain information about its local environment from simulated location sensors attached to nearby vehicles. Simulated actuators are responsible to change the light phases.

Events, Event Channels and Proximities A custom sensor fusion component is required to provide per-approach vehicle counts. To enable convergence towards a shared view of the congestion in the surrounding area, traffic light

controllers have to share their local view of the congestion at their junction with neighbouring controllers. Therefore, traffic light controllers advertise via events their view of the level of congestion associated with a particular approach from a neighbouring junction. As there is no real-time requirement, STEAM is used between traffic light controllers.

Contexts As a 4-way crossroad comprises 17 possible phases, sentient traffic light controllers can take 17 possible actions: stay in the current phase or switch to another phase. Each sentient object as part of its context has a representation of the congestion on each approach (i.e., high, medium or low). Sentient traffic light controllers are then composed of a 17 x $3^4 = 1377$ contexts corresponding to different signal phases and levels of congestion on the approaches to the junction.

Rules and Actions Instead of using predefined rules, we use the CRL inference engine component to learn the appropriate action to take within a particular context. With the incorporation of neighbour views into the knowledge base, the engine chooses a particular action to take, such as changing from one phase to another at a particular junction. This is achieved by sending an event to a simulated actuator responsible for changing the phases at this junction. Subsequently, a reward from the environment for taking this action is received, e.g., change in the velocities of vehicles on the various approaches to the junction.

Running the simulator described in [28] on Debian GNU/Linux 3.1 and using a map of Dublin with relevant junction information, we simulated a flow of vehicles in the city. Our initial results show that an individual sentient object at a junction can learn the different vehicle flow patterns across the junction and that the set of sentient objects in the system can optimise their collective behaviour.

4.3 Sentient Robots

Sentient Objects, Sensors and Actuators In this scenario, traffic lights and cars that navigate towards some destination are both implemented as sentient objects. In order to complete their mission, the vehicles are equipped with various sensors. Location and orientation information is received from a GPS receiver and digital compass respectively. Six ultrasonic sensors are also mounted on the vehicles to allow obstacle detection. Furthermore, traffic lights periodically produce events to inform approaching cars of their states. Once the cars have infered the actions to take in their current context, they send commands to the actuator, a software component that controls the gear and the wheels.

Events, Event Channels and Proximities As robots are mobile, they spontaneously communicate with anonymous traffic lights that they discover in their vicinity. To ensure that the robots can react in time to a message, e.g., by braking if the light is red, the traffic light must have timely communication within a

large enough coverage area. Traffic lights are guaranteed to receive adaptation events from their infrastructure in time when there is a significant change in the size of their proximity. As a consequence, they will always have time to adopt a fail-safe behaviour if communication with nearby robots is not possible, i.e., turn to green when they can not communicate with cars. This way, no car will ever pass through the traffic light when it is red [20]. Real-time wireless communication between remote and anonymous traffic lights and robots is provided by RT-STEAM. However, local event channels are used between the sensors mounted on a car and the car itself. Finally, as robots are mobile and only have interest in their local environment, proximity-based filtering filters out irrelevant events occurring in the environment.

Readings and Facts The autonomous vehicle uses four pipelines - `Command`, `ObstacleDetection`, `AcquireWayPoint` and `ObeyTrafficLight` - to produce observed facts. In the following, we only look in detail at the `ObstacleDetection` pipeline. The `ObstacleDetection` pipeline uses readings from the ultrasonic sensors to assert facts concerning obstacles near the vehicle. To fuse information and handle uncertainty, the Bayesian network defined in the MoCoA's library is used and, as a result, the observed fact `obstacleLeft/Right/StraightAhead` is asserted when an obstacle is detected with some associated probability. On average, a fact is asserted 100μs after a sensor reading is received by the Bayesian network. Training the network beforehand is necessary to obtain the probabilities of these hypotheses given available evidence from sensors.

Contexts The context hierarchy for the autonomous vehicle is too complex to examine in detail here (see Figure 5). Instead, we will look at the `AvoidObstacle` major context and some of the rules associated with that context. Notice that several contexts in the context graph can be active at the same time: the robots may be trying to avoid an obstacle while still obeying a relevant traffic light.

Rules and Actions To drive the behaviour of the autonomous robots, rules have been defined using the rule-based inference engine. For instance, the `Avoid-Obstacle` context is transitioned to when an obstacle is detected, i.e., each context which can transition to this context has a rule of the form:

```
obstacleRight||obstacleLeft||obstacleStraightAhead/
transition(AvoidObstacle).
```

In turn, the new context changes the behaviour of robots by changing the state of their actuator, e.g., `raise(leftCommand)`.

The rule below is an example of a behavioural rule defined in the `RelevantTL` context. Cars might have to stop if a traffic light is on their way and will be red by the time they arrive at it. The decision to start braking is made when the car reaches the braking distance from the traffic light. This rule expresses a condition on derived (e.g., `brakingDistance`) and observed (e.g., `trafficLight`) facts.

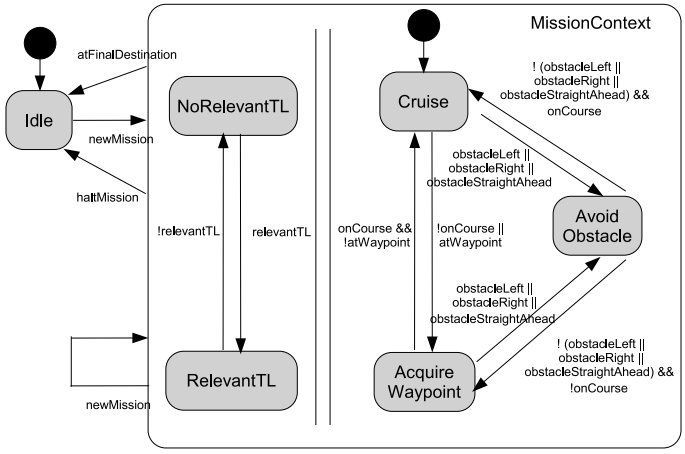

Fig. 5. Context graph of the sentient robots

`distanceTo` is a function that compares the current position of a robot with the position of a traffic light. If the condition is true, a stopCommand event is sent to the actuator that will apply the brakes.

```
distanceTo(trafficLight.position) == brakingDistance
&& trafficLight.colour == red/
raise(stopCommand)
```

This scenario represents 588 lines of high-level code and the complete application has been entirely generated by the MoCoA framework in 538ms.

4.4 Discussion

As space in this paper is limited, we cannot discuss in detail the different scenarios and the implementation of the components used, instead we prefer to highlight the flexibility and generality of our framework. Our experience and the above applications show that providing high-level abstractions for sensor-oriented computing allows to develop easily complex applications ranging from augmented artefacts to autonomous mobile robots to city-wide smart space applications. The developer needs only to identify facts, contexts and possibly rules, then MoCoA automatically selects the appropriate components and generates (most of the time) the complete code for the specified application.

5 Related Work

There have been numerous efforts at providing middleware to ease the development of context-aware applications. However, as discussed below, context-aware

frameworks are often limited to an application domain and fail to provide a generic solution for all types of context-aware applications.

Dey et al. [29] provide a toolkit which enables the integration of context data into applications, but do not provide mechanisms for systematically performing sensor fusion or reasoning about context in a mobile ad hoc environment, as our framework does. Other work provides mechanisms for reasoning about context [24] but still does not provide a well-defined programming model and does not address the challenges of mobility.

More recently, Solar [30] has been developed. This software infrastructure supports context collection, aggregation and dissemination, and provides a small composition language, allowing applications to construct a graph of operators in order to compute desired context from appropriate sources. Our middleware provides a more generic way of specifying the behaviour of context-aware applications using different reasoning and learning mechanisms and is not limited to infrastructure-based applications. Gaia [31] provides different components to support sensor fusion and reasoning about context. However, it fails to address the challenges of mobility and is primarily targeted at smart space applications.

Cooperative artefacts [32] are also restrained to one class of context-aware applications and additionally do not provide much support when reasoning about contexts. Context-based reasoning is not used to filter the relevant facts/rules/actions. Similarly, intelligent agents [33] select an action that is expected to bring them closer to a goal, given their built-in knowledge and the evidences that they perceive from the environment. Although they present powerful mechanisms to deal with uncertain knowledge, reason and learn, the agents do not exhibit a context-driven behaviour. As such, decision-making is less efficient and therefore seems suitable only for small-scale applications.

Another class of middleware for mobile context-aware applications based on tuple spaces exists. LIME [34] enables mobile coordination by abstracting communication into a tuple space that contains the data available on all connected devices. EgoSpaces [35], evolved from LIME, defines views that do not require distributed transactions to be maintained. The goals of these systems are in line with our objectives but their solutions do not allow decentralised optimisation through collaboration and are not suitable for applications with timeliness requirements.

6 Conclusion

The MoCoA middleware framework provides a systematic approach to the development of a wide range of context-aware applications in fixed or mobile ad hoc environments. Through its common programming abstractions, MoCoA describes a family of possible middleware platform addressing multiple combinations of non-functional requirements. The middleware framework builds on sentient objects, which support sensor fusion, context extraction, reasoning and event-based communication over fixed or wireless ad hoc networks with the possibility to include real-time support. Multiple off-the-shelf components offering

implementations of these abstractions are provided by the MoCoA library. In order to illustrate the generality of our approach, we have successfully instantiated MoCoA for three different types of sensor-based applications, which involve context-aware entities with different requirements in terms of safety and mobility. We are currently investigating the behaviour of context-aware application developers using MoCoA.

References

1. Addlesee, M., Curwen, R., Hodges, S., Newman, J., Steggles, P., Ward, A., Hopper, A.: Implementing a sentient computing system. IEEE Computer Magazine **34**(8) (2001)
2. Dearle, A., Kirby, G.N.C., Morrison, R., McCarthy, A., Mullen, K., Yang, Y., Connor, R.C.H., Welen, P., Wilson, A.: Architectural support for global smart spaces. In: 4th International Conference on Mobile Data Management. (2003)
3. Tapia, E.M., Intille, S.S., Larson, K.: Activity recognition in the home using simple and ubiquitous sensors. In: International Conference on Pervasive Computing. (2004)
4. Decker, C., Beigl, M., Krohn, A., Robinson, P., Kubach, U.: eSeal - a system for enhanced electronic assertion of authenticity and integrity. In: International Conference on Pervasive Computing. (2004)
5. De, P., Basu, K., Das, S.K.: An ubiquitous architectural framework and protocol for object tracking using rfid tags. In: International Conference on Mobile and Ubiquitous Systems: Networking and Services. (2004)
6. LaMarca, A., Koizumi, D., Lease, M., Sigurdsson, S., Borriello, G., Brunette, W., Sikorski, K., Fox, D.: Making sensor networks practical with robots. Technical Report IRS-TR-02-004, Intel Research and University of Washington (2004)
7. Sakamoto, D., Kanda, T., Ono, T., Kamashima, M., Imai, M., Ishiguro, H.: Cooperative embodied communication emerged by interactive humanoid robots. International Journal of Human-Computer Studies **62**(2) (2005)
8. for Injury Prevention, C.N.C., Control: Injury Fact Book (2001-2002)
9. Rubenstein, L.Z., Josephson, K.R., Robbins, A.S.: Falls in the nursing home. Annals of Internal Medicine **121**(442) (1994)
10. Sims, A.: The sydney coordinated adaptive traffic system. In: ASCE Engineering Foundations Conference on Research Priorities in Computer Control of Urban Traffic Systems. (1979)
11. Hunt, P., Robertson, R., Winton, R., Bretherton, R.: Scoot - a traffic responsive method of coordinating signals. Technical Report TRRL Report 1014, Road Research Laboratory (1981)
12. Cunningham, R., Dowling, J., Harrington, A., Reynolds, V., Meier, R., Cahill, V.: Self-optimisation in a next generation urban traffic control environment. ERCIM News: Special Edition on Emergent Computing (2006)
13. International Federation of Robotics. http://www.ifr.org/ (2006)
14. Hirose, S., Fukushima, E.F.: Development of mobile robots for rescue operations. Advanced Robotics **16**(6) (2002)
15. Ver?ssimo, P., Cahill, V., Casimiro, A., Cheverst, K., Friday, A., Kaiser, J.: Cortex: Towards supporting autonomous and cooperating sentient entities. In: European Wireless. (2002)

16. Biegel, G., Cahill, V.: A framework for developing mobile, context-aware applications. In: 2nd IEEE Conference on Pervasive Computing and Communications, Percom 2004. (2004)
17. Mühl, G., Fiege, L., Buchmann, A.: Filter similarities in content-based publish/subscribe systems. In: International Conference on Architecture of Computing Systems. (2002)
18. Meier, R., Cahill, V.: Exploiting proximity in event-based middleware for collaborative mobile applications. In: 4th IFIP International Conference on Distributed Applications and Interoperable Systems, Springer-Verlag (2003)
19. Hughes, B., Meier, R., Cunningham, R., Cahill, V.: Towards real-time middleware for vehicular ad hoc networks. In: 1st International Workshop on Vehicular Ad Hoc Networks. (2004)
20. Bouroche, M., Hughes, B., Cahill, V.: Building reliable mobile applications with space-elastic adaptation. In: International Workshop on Mobile Distributed Computing. (2006)
21. : Real Time Application Interface for Linux (2006) https://www.rtai.org/.
22. Hughes, B., Cahill, V.: Achieving real-time guarantees in mobile wireless ad hoc networks. In: Real-Time Systems Symposium. (2003)
23. Cunningham, R., Cahill, V.: Time bounded medium access control for ad hoc networks. In: Workshop on Principles of Mobile Computing. (2002)
24. Schmidt, A., Aidoo, K.A., Takaluoma, A., Tuomela, U., Laerhoven, K.V., de Velde., W.V.: Advanced interaction in context. Handheld and Ubiquitous Computing, Springer-Verlag (1999)
25. Gonzalez, A.J., Ahlers, R.: Context-based representation of intelligent behavior in training simulations. Transactions of the Society for Computer Simulation International **15**(4) (1999)
26. Sutton, R., Barto, A.: Reinforcement Learning. MIT Press (1998)
27. Dowling, J., Curran, E., Cunningham, R., Cahill, V.: Using feedback in collaborative reinforcement learning to adaptively optimise manet routing. IEEE Transactions on Systems, Man and Cybernetics **35**(3) (2005)
28. Reynolds, V., Cahill, V., Senart, A.: Requirements for an ubiquitous computing simulation and emulation environment. In: First International Conference on Integrated Internet Ad hoc and Sensor Networks. (2006)
29. Dey, A.K., Salber, D., Abowd, G.D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction **16**(2-4) (2001)
30. Chen, G., Kotz, D.: Solar: An open platform for context-aware mobile applications. In: International Conference on Pervasive Computing. (2002)
31. Roman, M., Campbell, R.H.: A middleware-based application framework for active space applications. In: International Middleware Conference. (2003)
32. Strohbach, M., Gellersen, H., Kortuem, G., Kray, C.: Cooperative artefacts: Assessing real world situations with embedded technology. In: International Conference on Ubiquitous Computing. (2004)
33. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 2nd edn. Prentice Hall (2003)
34. Murphy, A.L., Roman, G.C.: Lime: A coordination middleware supporting mobility of hosts and agents. ACM Transactions on Software Engineering and Methodology (To appear)
35. Julien, C., Roman, G.C.: Egospaces: Facilitating rapid development of context-aware mobile applications. IEEE Transactions on Software Engineering **32**(5) (2006)