

**FIGURE 7. Abstract syntax of dynamic preference aspect language**

```
<dynamic preference> = “DynamicPreference” preference_name “{”  
    <body> “}”  
  
<body> = <when-clause>* [“default:” <preference> “;”]  
<when-clause> = “when” “(” condition “)” <preference> “;”  
<preference> = static_preference_name | dynamic_preference_name |  
    “dynamic” class_name
```

## 4.0 Summary

In this position paper we have proposed an aspect language that supports the expression of communication preferences even if they have a dynamic nature. Sometimes it is better that aspects remain run-time entities, i.e. dynamic aspects. We have described some examples that can benefit from dynamic aspects, based on the SMove case study. The SMove case clearly shows the practical need for such dynamic aspects.

We argue that a survival of aspects at run-time is in many cases a necessary precondition to ensure maximal flexibility and to allow an aspect to adapt itself based on execution-time information. A feasible approach to handle such dynamic aspects is to implement them using meta-objects.

## 5.0 References

- AOP97 F. Matthijs, W. Joosen, B. Vanhaute, B. Robben, P. Verbaeten. “Aspects should not die.” In *European Conference on Object-Oriented Programming, Workshop on Aspect-Oriented Programming*. 1997.
- Kiczales G. Kiczales. “Towards a New Model of Abstraction in Software Engineering.” In *Proceedings of the International Workshop on New Models in Software Architecture '92; Reflection and Meta-Level Architecture*. 1992.
- SMdesign P. Kenens, S. Michiels, E. Truyen, S. Van Baelen, W. Joosen, F. Matthijs, P. Verbaeten. “Communication System: System Design Document.” *Confidential Information*. 1998.
- SMapi S. Van Baelen, E. Truyen, P. Kenens, S. Michiels, W. Joosen, F. Matthijs, E. Steegmans, P. Verbaeten. “The Development of an Application Programming Interface for the SMove Project”. *Confidential Information*. 1998.

To allow this, we need a more powerful construct than the `when`-clause in Figure 4. By using the `dynamic` keyword followed by a class name, it becomes possible to encapsulate in this class, an entire preference selection algorithm instead of a simple condition. It can be used for instance to couple communication preferences with individual objects, instead of with classes only.

**FIGURE 5. A dynamic preference**

```
DynamicPreference Preference {
    default: dynamic PreferenceChooserClass;
}
```

An example of this construct is shown in Figure 5. For every invocation to a vehicle, the preference chooser will be asked to return the appropriate preference for that invocation, depending on the current circumstances. The returned preference depends on the invoked vehicle.

**FIGURE 6. Pseudo code for PreferenceChooserClass**

```
class PreferenceChooserClass implements PrefInterface{
    public Preference getPreference(Vehicle vehicle) {
        if( vehicle.load() instanceof DangerousLoad )
            return _deliveryPreference;
        else
            return _peakHourPreference;
    }
}
```

Figure 7 shows the abstract syntax of the dynamic preference aspect language that we used in the examples described above. This language extends the language we have proposed in section 3.2. It enables to differentiate between different run-time situations.

### 3.4 Aspect weaving

The first aspect program shown in Figure 2 illustrates a static communication preference aspect. The specified preferences are static for all vehicles (all vehicles are treated the same), at all times (preferences will not change at run-time). A specialized aspect weaver can weave the aspect in the application code at compile-time.

The dynamic communication aspect in section 3.3 can not always be woven at compile-time. When the choice of preference depends on information that is only available at run-time, this preference cannot be known at compile-time. An approach to handle such dynamic aspects is to weave them at run-time. This can be accomplished a.o. by a meta-object protocol [Kiczales]. In this case, remote invocations are reflected to a metalevel where communication preferences can be added based on compile-time, as well as run-time information.

**FIGURE 3. Abstract syntax of static preference aspect language**

```
<static preference> = "StaticPreference" preference_name "{"  
    <body> "}"  
  
<body> = <method-clause>* [{"default:" <attribute> ";"}]  
  
<method-clause> = method_name (";" method_name)* ":" <attribute> ";"  
  
<attribute> = "COST" | "SPEED" | "FAST_DELIVERY" | "PROVIDER"
```

**Shortcomings of this static approach.** Communication preferences can not always be statically defined. They can depend on external circumstances, like e.g. the time of the day or the current position of the vehicle. When communication at peak hour is more expensive than communication at off-hour, cheap communication at peak hour and fast communication at off-hour may be preferred. It is also possible that a user wants to change the communication preferences when the vehicle crosses a border.

### 3.3 Dynamic preferences

To express more sophisticated preferences which capture the problems described above we introduce dynamic preferences.

An example of an application with dynamic communication preferences is a vehicle control application that allows a transport company to monitor the current location of its vehicles. The aspect program in Figure 4 accomplishes the communication preferences for this application. Communication to vehicles for instance is handled by a `PeakHourPreference`: a `CostPreference` is preferred during peak hours, while communication will be based on a `FastDeliveryPreference` during off-hours.

**FIGURE 4. A dynamic preference**

```
StaticPreference CostPreference {  
    default: COST;  
}  
  
StaticPreference FastDeliveryPreference {  
    default: FAST_DELIVERY;  
}  
  
DynamicPreference PeakHourPreference {  
    when (peak hour) CostPreference;  
    when (off-hour) FastDeliveryPreference;  
}
```

Now suppose vehicles have different kinds of loads. Preferences could be based on the load a vehicle is carrying, e.g. a dangerous or an unimportant load. It is crucial that a truck with a dangerous load can be reached quickly under all circumstances, while this is less important in case of a regular truck. Communication to vehicles that carry a dangerous load should be handled by a `FastDeliveryPreference`.

our specific language to support complex dynamic preferences, we rely on an existing language for such preferences.

### 3.1 Attributes

The SMove architecture enables applications to suggest communication preferences (like speed and cost of communication) that influence the selection of a communication device. We introduce some attributes, which will be used in the aspect language, that enable us to express the communication aspect in a declarative manner. Some examples of attributes are:

- **COST**: communication with low cost is preferred
- **SPEED**: high performance communication is preferred
- **FAST\_DELIVERY**: low communication delay is preferred
- **PROVIDER**: a specific communication provider is preferred

### 3.2 Static preferences

Some applications have communication preferences that are statically defined (this means known at compile-time, and the same for all vehicles). An example is an application for remote control and immobilization of vehicles. Such an application can for instance be used by the police. A reasonable communication preference could be to communicate as cheaply as possible, but for immobilization of a vehicle (e.g. when it is reported as stolen), low latency communication<sup>1</sup> is preferred.

The aspect program in Figure 2 is an implementation of the static communication preference explained above. By mentioning the **COST** attribute as `default`, we state that a low cost communication device is preferred for all remote invocations, except for those methods that are explicitly specified in the aspect program. In this example, this is done by attaching the **FAST\_DELIVERY** attribute to the `immobilize` method. This means that immobilization messages are always sent by a low latency communication channel.

**FIGURE 2. A static preference**

```
StaticPreference PolicePreference {  
    ImmobilizationFeature.immobilize: FAST_DELIVERY;  
    default: COST;  
}
```

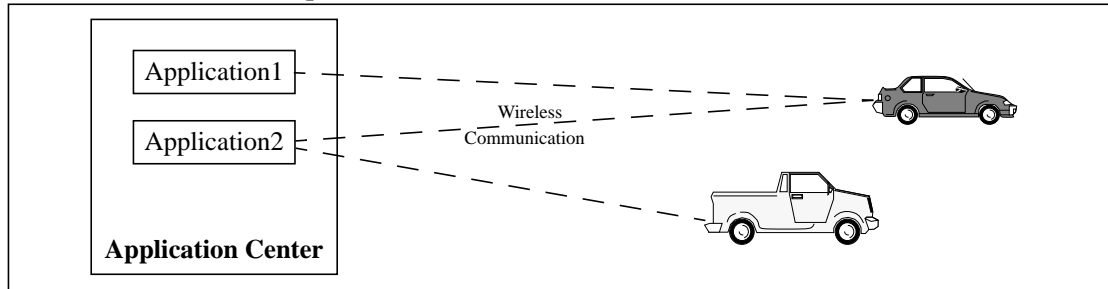
The abstract syntax in Figure 3 makes it possible to implement such a static communication strategy.

---

1. Certain cheap communication types can have delays of several hours for long distance communication.

API (Application Programming Interface) that hides the specific underlying communication devices. They can only indicate high level communication preferences, like communication cost, communication speed, etc... This way, applications do not have to be rewritten whenever an underlying communication device changes or when a new one becomes available.

**FIGURE 1. The SMove platform**



## 2.2 Flexible communication

Communication with a vehicle happens by means of so called *features*. A feature is an object that encapsulates a service provided by a vehicle (e.g. a positionfeature that supports requests for, and control of position coordinates, a speedfeature for requesting and controlling the speed of a vehicle, etc.). Each vehicle can have a different set of supported features. These features can be added/removed per vehicle, at run-time. Applications can interact with vehicles by invoking operations on the feature-objects that are available.

Invocations on features are physically routed over one of many possible wireless communication channels. Each of these communication channels has its own specific characteristics. An important characteristic of the SMove platform is that the selection of a specific communication device can be influenced by application-specific preferences. Such preferences can even differ for each invocation of a vehicle.

Communication preferences are scattered over different levels of the system, what leads to a problem of encapsulation. At the application level, preferences are bounded to invocations and in the protocol stack the actual device selection takes place. This means that communication preferences cross-cut the system component boundaries. They are related to the essential semantics of the SMove platform and can not be clearly encapsulated in some kind of component. The aspect paradigm helps to separate communication preferences from the functional code!

## 3.0 An aspect language for communication preferences

The SMove platform can be viewed from many perspectives, showing us different kinds of aspects related to communication, distribution, performance, synchronization, logging, accounting, etc. We describe a way to handle communication preferences using the aop approach. We propose a specific aspect language that enables programmers to add static as well as dynamic communication preferences to remote invocations. Instead of extending

# An AOP Case with Static and Dynamic Aspects

**Peter Kenens, Sam Michiels, Frank Matthijs, Bert Robben, Eddy Truyen, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten**  
**{Peter.Kenens, Sam.Michiels}@cs.kuleuven.ac.be**  
**Dept. of Computer Science - K.U.Leuven**  
**Celestijnenlaan 200A B-3001 LEUVEN BELGIUM**

## 1.0 Introduction

Aspect-oriented-programming (aop) is a promising new approach where the description of a complex system/application is enhanced with various aspects, related to communication properties, distribution, synchronization, etc. All aspects can be described separately and are brought together by using a so-called weaver. Mostly, this is performed at compile-time, what makes that aspects disappear in the final software version. We argue that in some cases aspects should remain run-time entities in order to capture the dynamic properties of an application [AOP 97]. We believe there is a need for dynamic aspects, e.g. strongly related to objects, which are clearly run-time entities.

Our experiences in a project called SMove clearly show the practical need for such dynamic aspects. In this position paper we propose an aspect language that supports a communication aspect with a dynamic nature. We describe some examples that can benefit from dynamic aspects, based on the SMove case study ([SMdesign] [SMapi]). We hope that feedback on our approach of dynamic aspects can inspire the ongoing implementation.

This position paper is structured as follows: in section 2 we shortly introduce the SMove platform. Section 3 describes the proposed aspect language. We summarize in section 4.

## 2.0 The SMove platform

### 2.1 Architecture

The SMove architecture consists of Application Centers and vehicles (see Figure 1). The goal of the platform is to support the development of applications that can communicate with vehicles through a number of wireless communication channels, a.o. ERMES and GSM. It enables the use of a wide variety of applications, such as remote vehicle immobilization, traffic monitoring, localization, border control, toll collection, etc. that can operate under a wide range of conditions (e.g. in a parking garage,...) and with different costs. To develop these applications in a cost effective way, programmers can use the SMove