Features on Demand

Nicolás Cardozo, Wolfgang De Meuter[*]
Software Languages Lab
Vrije Universiteit Brussel
ncardozo@vub.ac.be

Kim Mens, Sebastián González,
Pierre-Yves Orban
ICTEAM, Université catholique de Louvain
kim.mens@uclouvain.be, sgm@acm.org

ABSTRACT

This paper presents our vision of applications as feature clouds, providing software services that are composed dynamically from a set of available fine-grained features. Our feature cloud programming model realizing this vision, relies on context-oriented programming technology, enabling new or adapted features to be added to or removed from running applications on demand, according to contextual information, and taking into account feature dependencies. As a proof of concept, we implemented a prototype of an on-board car system running on a mobile device, using an instantiation of our feature clouds programming model for JavaScript.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.1.m [**Programming Techniques**]: Miscellaneous

General Terms

Design, Context-oriented programming, Dynamic software composition

Keywords

Context-oriented programming, Software-as-a-Service, Features, Dynamic software adaptation

1. INTRODUCTION

Computer and computing technology are rapidly evolving towards highly interconnected computing devices that can adapt promptly to contextual information about their execution environment, user preferences and available services. This evolution has been triggered by changes in hardware technology such as smartphones and cloud computing. We observe that software development technology is lagging behind to cope with these new technologies, causing current-day applications to miss important opportunities of delivering improved services on demand to its users.

Software technology has evolved from monolithic off-the-shelf products to offering software products that can be customized to match the specific needs of particular clients. Software products can vary in the amount and kind of services they offer, in the way in which they present such services to their users, or in the flexibility they offer to adapt their services to cope with changing situations. For example, with the advent of Software-as-a-Service (SaaS) [15], users can obtain temporary access to (parts of) products as needed. Using Context-Oriented Programming (COP) [3], the offered services or the behavior of those services can be adapted, even at run time, to respond to new situations arising in, or information coming from, the surrounding execution environment of the system.

Traditional programming technology is currently not yet well-equipped to support the development of applications in such highly dynamic settings as it faces three key challenges. Firstly, services offered by such software systems need to be modularized and composed at a *fine level of granularity*. Such support is required to enable the customization and evolution of specific system features for different users, devices or situations. Secondly, fine-grained features should be able to be *composed dynamically* into complete services. Dynamic composition is required since particular services, or parts of services, may be temporarily requested by users or may need to adapt to particular situations arising in the execution context of the application. Thirdly, since such fine-grained dynamic composition may give rise to subtle interaction problems as services are composed, appropriate solutions are needed to ensure that the requested features can be effectively composed into coherent services without inconsistencies between the features.

To provide such support for the dynamic composition of software services as sets of fine-grained features that can be activated and deactivated dynamically, we put forward the feature clouds programming model. Our model is based on COP which adequately supports the ability to dynamically modify a system to its run-time environment. As such, it provides a good basis for the next generation of SaaS systems offering *features on demand*. In particular, COP enables dynamic adaptivity of the system's behavior to its surrounding execution environment, which could be used for the dynamic composition of features according to particular situations. Thus, our model could foster further the advancement and

[*]Cite as Cardozo, N. Mens, K. González, S. Orban, P-Y. and De Meuter, W. Features on Demand.

flexibility of such SaaS applications. For example, it could be used to compose the most appropriate services required by a user according to its type of connectivity (no connectivity, WiFi, 4G, LAN, . . .), availability of additional services, particular platform or device used, current location, or the battery level or remaining memory of the device. The notion of *context* from COP is used as a basis to modularize applications into many different fine-grained features, each of which can be (de)composed dynamically, in a consistent way, to achieve the most appropriate system behavior.

2. FEATURE CLOUDS

To achieve our vision of building software applications as clouds of features, offering different services that can be added to the application on demand or depending on context changes, we advocate the usage of Context-Oriented Programming (COP). This technology satisfies the three main properties we believe to be essential to achieve service composition as dynamically evolving clouds of features, namely: (1) *fine-grained feature definition*, (2) support for *dynamic behavior adaptation*, and (3) *managing feature interaction*. Section 2.1 explains the main characteristics of COP used to provide support for each of these properties. Section 2.2 then discusses how these properties are used to conceive feature clouds.

2.1 Context-Oriented Programming

Context-oriented programming is a programming paradigm conceived to enable dynamic behavior adaption of software systems. It allows programmers to define fine-grained pieces of behavior specific to particular contexts and to activate or deactivate them according to specific situations in the surrounding execution environment, sensed through, for example, a sensor network. In what follows, we refer to these fine-grained pieces of behavior as *behavioral adaptations*, and the situations of the surrounding execution environment in which they are applicable as *contexts*.

Dynamic adaptation to the surrounding environment is achieved in COP via three main stages: definition, selection and composition of contexts and behavioral adaptations. To explain each of these stages we use the Context Traits language [6], a COP extension of JavaScript. The concepts introduced here are illustrated on our running example of an on-board car system, which will be presented in full detail in Section 3. Please keep in mind that most of the concepts presented here are not specific to the Context Traits language but apply to most other COP languages [13] as well.

Context definition.

In COP, contexts are defined as first-class entities of the system. Contexts reify situations, extracted from raw information about the surrounding execution environment, that are semantically meaningful for the system being build. For example, in our on-board car system, to reify a situation in which the car is moving and an SMS must be answered, we would define an Easy Answer context. Snippet 1 shows how this can be defined in Context Traits.

```
var EasyAnswer  =  new cop.Context({
  name: 'easyanswer',
  description: 'Quick answer to SMS'});
```

Snippet 1: Context definition.

Behavioral adaptation definition.

In order to adapt the behavior of a running system, COP languages allow to associate behavioral adaptations with each context. Behavioral adaptations modify the behavior originally defined in the system either by overriding or extending it. For example, for the Easy Answer context we may want to override the default SMS sending behavior by one that allows a driver to answer quickly just by selecting one answer from a short list of pre-encoded messages. Such behavioral adaptations are introduced to (resp. withdrawn from) the system through dynamic composition whenever their associated context becomes active (resp. inactive).

In most COP languages, behavioral adaptations are defined as regular methods of the system. However, these method definitions are not necessarily linked with the regular modules of the system (e.g., an object or class). Rather, these methods are associated to a particular context. Snippet 2 shows two behavioral adaptations associated with a UK Speed Gauge context, as defined in the Context Traits language.

```
1  RoyalSystem = Trait({
2    var CONV_RATIO = 0.621371192;
3    getSpeed: function(msg) {
4      _val = this.proceed();
5      Math.round _val * CONV_RATIO; }
6
7    getHtml: function() {
8      display.setGaugeDisplay(this.proceed().
           replace("km/h", "mph")); }
9    });
10
11 UKSpeedGauge.adapt(SpeedGauge, RoyalSystem);
```

Snippet 2: Behavioral adaptation definition and association with context.

The UK Speed Gauge context reifies the behavior of how to display the car speed when driving in the UK. The behavior associated with this particular context is therefore to display the car speed according to the UK measure system (i.e., the royal system). The behavioral adaptations of Snippet 2 define precisely this behavior. The first behavioral adaptation (`getSpeed`) takes the default speed value obtained from the speed sensor of the car, and converts it into its equivalent in the royal system. The second behavioral adaptation (`getHtml`) adapts the default display of the user interface to ensure that the unit of velocity shown to the user corresponds to the value of the displayed speed —that is, `mph` instead of `km/h`. Finally, the **`adapt`** construct on Line 11 associates these two behavioral adaptations with the UK Speed Gauge context as adaptations of the default Speed Gauge behavior.

Two things are worth noting about this definition of behavioral adaptations. First, in the Context Traits language, behavioral adaptations are defined as *traits*. Traits [4] are a mechanism to achieve fine-grained modularization and reuse in software systems. Traits define groups of behavior (i.e., methods) that are as small and cohesive as possible. Hence, traits represent an appropriate abstraction to modularize behavioral adaptations associated to particular contexts.

Secondly, the behavioral adaptations can access the behavior provided by other behavioral adaptations or the default behavior of the system by means of the `proceed()` construct as is the case for `getSpeed` (Line 4) and `getHtml` (Line 8). Proceed is a reuse directive similar to super calls in object-oriented languages. By using proceed the system calls the instantiation of the current behavioral adaptation on a context with less priority according to context composition. In the example of the getSpeed behavioral adaptation associated with the UK Speed Gauge context, proceed is used to get the speed value as defined by the default getSpeed method defined in the base system (in metric system units).

Context selection.

Contexts should be activated (resp. deactivated) dynamically whenever the situations of the surrounding execution environment they reify are sensed present (resp. not present). The process of selecting the contexts appropriate to the current situation of the surrounding execution environment of the system is managed by a so-called context discovery module. This module gathers raw information about the system's environment through a sensor network, and upon processing it selects the relevant contexts for the situation at hand. For example, an acceleration above 1000rpm obtained from the acceleration sensor of the car could be interpreted as "the car is moving". In such situation the Easy Answer context is selected (activated); in any other situation it should be deselected (deactivated). Snippet 3 shows how this activation and deactivation policy of the Easy Answer context would be defined in the Context Traits language.

```
acc_sensor.EventListener('accel_reading',
  function(info) {
      if(info.reading >= 1000)
          EasyAnswer.activate();
      else EasyAnswer.deactivate(); })
```

Snippet 3: Gathering and activation of contexts.

Context composition.

The traits mechanism provided by Context Traits for defining behavioral adaptations is also used for composition purposes. The composition mechanism of Context Traits requires two elements: a collection of traits and a composition policy. Each trait is defined by a set of methods it *provides*, and a set of methods that it *requires*.

The idea for the dynamic composition of behavior is that, given a collection of traits, we take the union of all their provided and required methods. Note that there is no priority order in the union of traits, hence, composition conflicts may arise when different traits provide the same method signatures but with different implementations. When such conflicts arise they must be resolved explicitly by means of a composition policy. Different composition policies have been defined in COP languages to automatically compose adaptations according to the execution environment [13].

A composition policy can be seen as a function that, given a set of adaptations (i.e., traits) to compose, provides a resolution set. That is, it states which of the methods in the composition are used, with their ordering, avoiding conflicts. In Context Traits, the default resolution strategy is to use the activation age of contexts, but other composition policies can be defined as required. The context age composition policy gives precedence to behavioral adaptations associated with the contexts activated most recently. This means that in a situation where two contexts offer a behavioral adaptation for a same method, the selected behavioral adaptation will be the one associated with the context activated the latest. Policies are applied in a given precedence order, if a conflict cannot be resolved by a given policy, then the policy with the next precedence is applied until no conflicts remain or no other policies can be applied. When no policy is defined, the context age policy is used to resolve all composition conflicts.

The RoyalSystem trait defined in Snippet 2 declared the behavioral adaptations associated with the UK Speed Gauge context, which were designed to override the default behavior of the Speed Gauge context for calculating and displaying the speed in the on-board car system. This means that we are required to use a composition policy in which, whenever these two contexts are active, the behavioral adaptations provided by UK Speed Gauge are applied before those provided by the Speed Gauge context. In Context Traits, such a policy can be expressed explicitly as shown in code Snippet 4.

```
cop.manager.addObjectPolicy(SpeedGauge,
              [RoyalSystem],
 return function(UKSpeedGauge, SpeedGauge) {
   return Trait.override
            (UKSpeedGauge, SpeedGauge); });
```

Snippet 4: Composition policy definition.

Context dependency relations.

Aside from the composition problems tackled by composition policies, other composition errors can arise when composing contexts. Inconsistencies may arise due to assumptions made by contexts about the presence or not of behavioral adaptations provided by other contexts. Managing and making explicit such assumptions has become an increasingly important concern among COP languages. Existing solutions amount to defining *context dependency relations* [1, 8, 12] which describe the interaction of a context (de)activation with respect to the (de)activation of other contexts defined in the system. For example, an exclusion dependency relation between two contexts describes that both contexts cannot be active at the same time. Whenever a context is to be activated or deactivated, a context manager verifies its context dependency relations with other contexts. Only if no inconsistencies are found during verification, the desired (de)activation of the context is allowed to take place. In Section 2.2 we present the context dependency relations we incorporated in the Contexts Traits language to manage interaction between features as they are dynamically (de)activated.

2.2 The Feature Clouds Programming Model

In our vision of applications as feature clouds, the services of a software system are composed dynamically upon demand from users, and by adapting their features so that they are the most appropriate (i.e., offer optimized functionality) according to the current executing environment of the system. For this reason we position feature clouds at the meeting point of three research directions, namely the highly dynamic software systems offered by COP, the modularization and composition of software products as proposed by feature orientation, and the accessibility and scalability of software services offered by SaaS technology. We discuss here the conceptual and technical underpinnings of feature clouds.

Feature clouds break the assumption of regular feature-oriented systems, where services or software products are defined and composed up front in a structured way, e.g., using feature diagrams [9]. Instead, services are composed dynamically by combining their independent features or by request of users, possibly including features that depend on the selected ones.

We identify COP as central to the development of feature clouds, based on the facilities it offers for fine-grained features, dynamic adaptation and feature interaction, as mentioned in Section 2.1. In feature clouds we introduce the concept of *contexts as features* [11]. Taking advantage of similarity between the concepts of COP and FOP [2], behind the curtains of our feature clouds model, features are defined as contexts.

Fine-grained features.

The property of fine-grained features is obtained directly from the definition of contexts. As contexts determine the minimal set of functionality to define or adapt for a particular situation of the system, features are therefore equated with contexts. Features can be defined to provide only specialized pieces of behavior, instead of complete services. Software services are then defined by composing several of these fine-grained features. The advantage of gathering services as a composition of features, is that localized pieces of behavior can be modified without having to modify the complete service. Moreover, it allows services to be extended or customized by adding specialized features that reuse the behavior of previously defined ones, by means of `proceed`. Such a mechanism is normally not supported in traditional feature-oriented programming, where features are defined as closed components and feature customizations would duplicate the behavior of the original feature.

Dynamic adaptation.

The property of dynamic adaptation follows from the ability to adapt to the surrounding context of execution combined with the ability to reuse previously defined behavioral adaptations, as put forward by COP. Different features customizing the behavior of a service for a particular situation can be defined. Upon context activation (resp. deactivation), a feature will dynamically become part of (resp., be removed from) the service if and only if it is the most appropriate feature according to the information obtained about the surrounding execution environment. In the feature clouds model we propose a Feature Discoverer module which is in charge of acquiring information about the surrounding execution environment of the system, and to (de)activate the respective features according to the obtained information and the conditions specified for each feature, as shown in Snippet 3.

Feature interaction.

Feature interactions are handled by extending the notion of context dependency relations and composition policies to the case of features. As previously mentioned, composition policies dictate the order in which behavioral adaptations are composed whenever multiple active features specialize the same behavior. At a higher level, desired and undesired interactions between features can also be expressed by means of dependency relations. Such relations are used to express assumptions that features make about the presence or absence of other features. For example, to group sets of features that go together, to ensure that other needed features are available, or to verify absence of conflicting features in a service configuration.

The original Context Traits implementation did not provide context dependency relations. Therefore, taking inspiration from other COP languages such as Subjective-C [5] that do provide support for context dependency relations, we extended the Context Traits language by introducing six types of dependency relations for feature clouds. These relations were defined according to the needs observed in the case study of the on-board car system (cf. Section 3), and are presented below using examples taken from that case. Other types of relations could also be defined as needed. For more information on these dependency relations we refer to [10].

Dependencies between features are expressed by annotating a feature definition with the type of relation and the name of the corresponding feature. An example of using context dependency relations is shown Line 3 of Snippet 5.

Combination The *combination* dependency defines a kind of *'virtual'* feature as an aggregation of more specialized features. Activating the aggregated feature triggers the activation of each of its constituent features. Conversely, when all the constituent features become active, the aggregated feature is considered active as well. For example, a `UK Driving` feature could be declared as a combination of several more primitive translation features such as the `UK Speed Gauge` which converts the units of the `Speed Gauge` from `km/h` to `mph`.

Requirement When a feature *requires* another feature, its activation is only allowed to take place whenever the required feature is already active. For example, an `SMS Outbox` feature can only be activated if a more basic `SMS` feature on which it relies is already active.

Inclusion When a feature is *included* by another feature, it is automatically activated as a consequence of activating that other feature. For example, a high-level `SMS Sending` feature that provides an easy-to-use interface to send SMS messages could include a more primitive `SMS Outbox` feature in order to be able to send messages.

Suggestion A feature may *suggest* other features to be installed that are related to a given feature. However, the activation of these related features is not enforced. For example, the `Easy Answer` feature, which has to do with safety, suggests the `Safe Driving` feature. This relation is not taken into account when activating or deactivating features, but is only used as an indication of relevant related features in the Feature Store, as will be exemplified in Figures 3 and 4.

Exclusion Incompatible features *exclude* each other. In such a case, activation of a feature is prevented if any of its excluded features (or features excluding it) are active. For example, the `Easy Answer` feature is mutually exclusive with an `Easy Contacts` feature because they both adapt the `SMS Sending` feature in different incompatible ways. Only one of them can be active at a time.

Subsumption A feature may *subsume* multiple other features. This dependency is similar to the *combination* dependency, but differs from it in that the activation of all subsumed features in the relation does not automatically imply activation of the subsuming feature. An example of such a subsumption relation is the `Speed RPM Gauge` feature which subsumes the `Speed Gauge` and `RPM Gauge` features. The `Speed RPM Gauge` feature displays a combined speed and RPM gauge rather than two separate gauges for the speed and RPM, as illustrated in Figure 1. Indeed, when displaying the combined Speed RPM Gauge (Figure 1a), it makes no sense to still display the individual RPM Gauge and Speed Gauge (Figure 1b) separately as well. The purpose of the subsumption relation is to replace a set of features by a feature that subsumes the behavior provided by all subsumed ones, since it may be desirable to keep the behavior provided by the subsumed features independent. Feature replacement only takes place when the subsuming feature is explicitly activated.

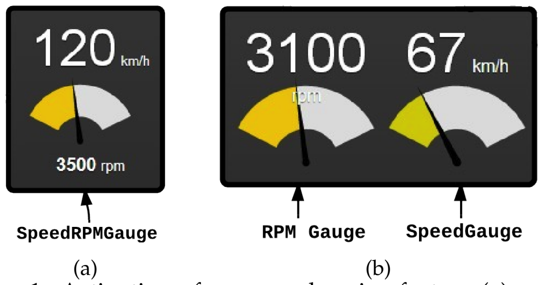


Figure 1: Activation of a comprehensive feature (a) and its two independent subsumed features (b).

3. AN ON-BOARD CAR SYSTEM

As a concrete illustration of our feature clouds research vision in which applications are composed from a set of features on demand, let us now take a closer look at the case of our on-board car system. This system can dynamically provide features such as a car dashboard, communication facilities, driving assistance, and satellite navigation.

Modern on-board car systems have become full-blown software systems that consist of a variety of different interacting features and that can access a large amount of context information coming from car sensors or external sources. Such systems should no longer ignore their surrounding environment by keeping their behavior and functionality fixed and rigid. On the contrary, they should be able to activate or deactivate certain features dynamically according to the surrounding context, and be flexible enough to allow the dynamic addition or removal of features —that is, when someone adds new car equipment or functionality to the system.

We implemented such an on-board car system, to study if and how the COP paradigm could provide an appropriate mechanism to build dynamically adaptable context-aware systems as run-time compositions of interacting features [10]. Our prototype of such an adaptive feature-oriented on-board car system for the Android mobile platform can interface with a car through a generic OBD[1] Bluetooth adapter, as depicted in Figure 2. The on-board car system was tested in real life with a Nissan Primera car, an OBD Bluetooth adapter plugged into the car, and connected over Bluetooth with a Samsung Galaxy Nexus i9250 running Android 4.3.

The on-board car system initially provides a few *native features* such as an Internet feature providing internet access when the device is connected to the internet (through WiFi, 3G or similar), or an SMS feature providing short message

[1]With the publication of the European directive 98/69/EC, since 2002 it is mandatory for every car in Europe to be equipped with an OBD (On-Board Diagnostics) interface.

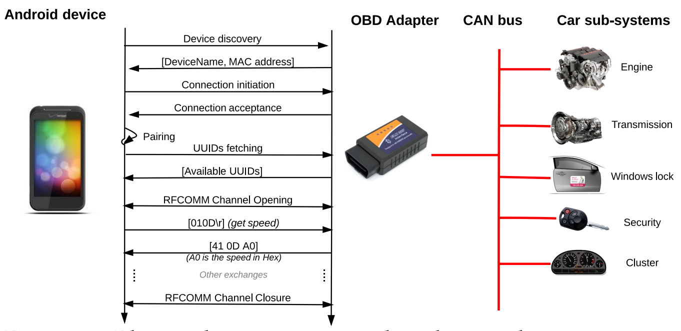


Figure 2: Bluetooth connection and exchanges between an Android device and an OBD Adapter.

service facilities. The system and its accompanying services are divided in three main modules.

3.1 Feature Store

In our feature clouds model, additional features can be added or removed at run time. A first way of obtaining such features is through a so-called *Feature Store*, where drivers can select what additional features they (no longer) desire for their car system. As illustrated by Figures 3 and 4, the Feature Store is akin to app stores as exist for the iOS, Android, or Windows Mobile platforms. It lists all the known, available or suggested features with their rating and price, and it allows drivers to select and (un)install the features they desire.

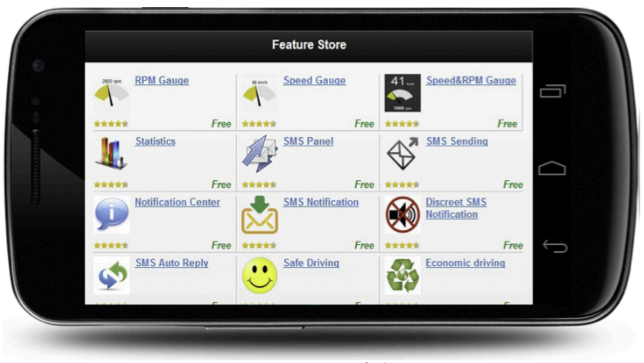
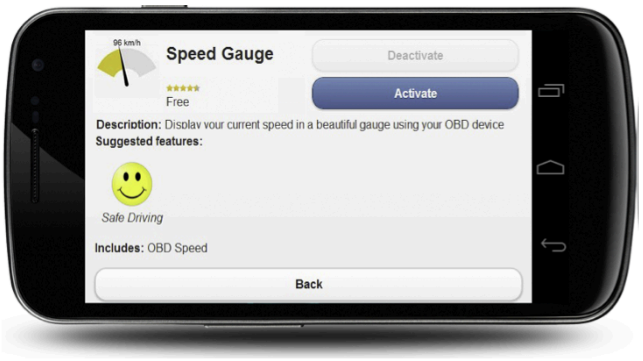


Figure 3: Screen capture of the Feature Store.



Description: Display your current speed in a beautiful gauge using your OBD device
Suggested features:
Safe Driving
Includes: OBD Speed

Figure 4: Screen capture of a detailed feature page.

This Feature Store contains a variety of dashboard features (e.g., a Speed Gauge, an RPM gauge, or Statistics on the driver's journeys), communication features (e.g., SMS Sending or SMS Notification), driving assistance features (e.g., Safe Driving or Economic Driving), and many more. Figure 5 shows a screenshot of an Android device used as a car dashboard, where several of these features (Speed Gauge, RPM Gauge, Statistics and SMSPanel) have been installed.

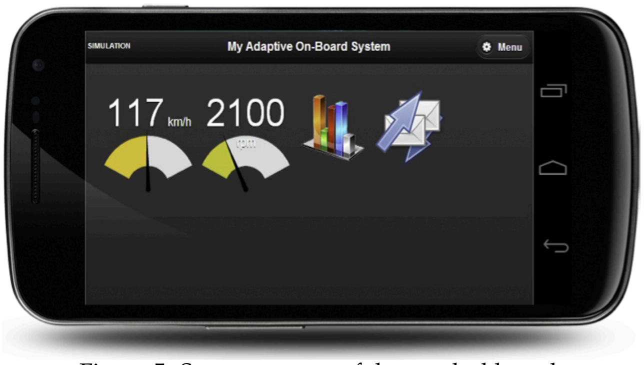


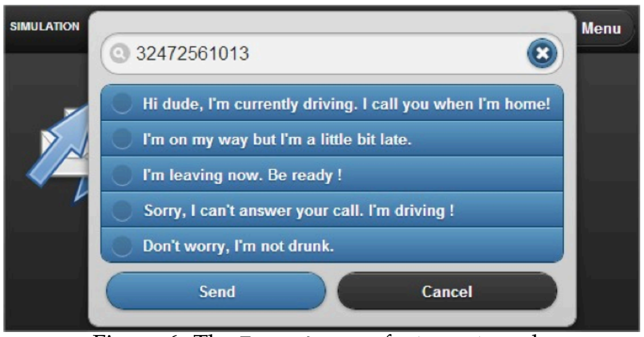
Figure 5: Screen capture of the car dashboard.

Currently, the Feature Store provides an exhaustive list all available features. In our vision however, ideally the Feature Store should present a list of available services (possibly filtered according to the state of the surrounding execution context), where each service is aware of the features it requires, and when selected gets composed automatically from the available fine-grained features, thus avoiding cluttering the Feature Store and avoiding the users from having to select each individual feature separately.

3.2 Feature Discoverer

The *Feature Discoverer* provides the ability to dynamically (de)activate features in the system. The role of the Feature Discoverer is to gather data coming from external sources (e.g., internet connectivity, or the device's battery status) and integrated car sensors. According to this available information, the Feature Discoverer dynamically activates or deactivates the features appropriate for the situation at hand.

Examples of features that could be activated through the Feature Discoverer are the Easy Answer and UK Driving features. UK Driving provides dedicated driving assistance for driving in the UK. This feature is activated automatically as soon as it is detected that the car is located in the United Kingdom, for example, using geolocation. The Easy Answer feature, if installed, would adapt the SMS Sending feature when the car is moving. Whereas the normal SMS Sending feature would allow the driver to type and send SMS messages, the Easy Answer feature would only allow to respond to an SMS with a single touch by selecting a response from a short list of pre-encoded responses like: "Sorry, I can't answer your call. I'm driving!". This is illustrated in Figure 6.



32472561013

Hi dude, I'm currently driving. I call you when I'm home!

I'm on my way but I'm a little bit late.

I'm leaving now. Be ready !

Sorry, I can't answer your call. I'm driving !

Don't worry, I'm not drunk.

Figure 6: The Easy Answer feature at work.

3.3 Feature Manager

The management of the (de)activation of features as a consequence of the (de)activation of other linked features is delegated to the *Feature Manager*. The Feature Manager contains a collection of all installed and active features and their dependencies. The Feature Manager implements different algorithms to activate and deactivate features based on those dependencies. Additionally, the Feature Manager is also in charge of applying the policy rules (explained in Section 2.1) whenever features are to be composed.

3.4 Working with Feature Clouds

Having explained how the on-board car system is divided in three main modules, let us now take a closer look at how the application itself can be designed as a cloud of features. As an example, we focus on the features related to the Speed Gauge service of the on-board car system. The feature cloud associated with this Speed Gauge service is shown in Figure 7, defining all fine-grained features that could compose it. Each of these features can be independently requested by users, yielding their composition while taking into account their dependencies with other features as defined by the feature dependency relations. Note that the structure of the feature cloud shown in Figure 7 is not modified when features are (de)activated. Feature (de)activation only makes the feature (and their associated features through context dependency relations) (un)available in the system. Structure of a feature cloud is modified when features are defined and published/removed in/from the Feature Store (or they are associated with a particular service in our vision).

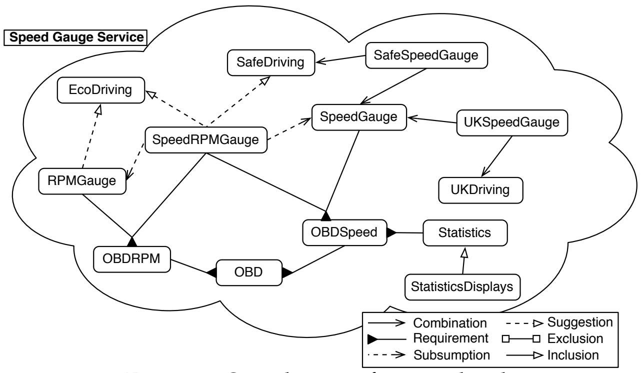


Figure 7: Speed gauge feature cloud.

Each of the features defined in this cloud defines or specializes one specific behavior, for example, the Safe Speed Gauge feature offers assistance while driving. Other features offer a dedicated behavior adapted to particular situations, as is the case for the UK Speed Gauge which extends the behavior of the Speed Gauge features where the former feature extends the behavior of the latter, as shown in Snippet 2.

The overall Speed Gauge service is composed and adapted dynamically from the activated features in its cloud. For example, the default Speed Gauge behavior is adapted whenever users location changes from Europe to the UK with the activation or deactivation of the UK Driving feature. Upon activation, the behavior of the service will seamlessly adapt from displaying the speed in the metric system to displaying it in the royal system. The speed is changed back to the metric system upon its deactivation.

Snippet 5 shows how the UK Speed Gauge feature is defined as a fine-grained feature that can adapt the behavior of

```
1 class window.UKSpeedGauge extends Feature {
2
3   @combination = ["speedgauge", "ukdriving"];
4
5   constructor: function() {
6     Feature.call(this, "ukspeedgauge"); }
7
8   RoyalSystems = Trait ({
9     //defined in Snippet 2
10  });
11  UKSpeedGauge.adapt(SpeedGauge, RoyalSystem);
12
13  activate: function() {
14    Feature.call(this);
15    SpeedGauge.init();
16    true; }
17
18  deactivate: function() {
19    Feature.call(this);
20    SpeedGauge.init();
21    true; }
```

Snippet 5: UK Gauge feature definition in Context Traits.

the default Speed Gauge feature dynamically. Line 3 shows how the UK Speed Gauge feature is defined as a combination of the Speed Gauge and UK Driving features, ensuring that the correct speed reading is displayed to users as their geolocation changes.

4. DISCUSSION AND FUTURE WORK

In this section we discuss some of the current limitations of our feature clouds programming model, as well as the extensions it would require, to turn it into industrial reality.

Granularity of features.

Traditional approaches usually adopt a component-based development process to realize service composition and adaptation, by defining course-grained components to be combined and integrated into full-fledged systems at run-time. More recent approaches have started looking into composing systems from more fine-grained features [7]. At this finer granularity level, features are described as sets of behavior rather than full components. Such fine-grained behavior may even cut across different system modules. Following this evolution, with the use of dynamic feature (de)activation, we have explored the usefulness of new SaaS models that dynamically assemble services from very fined-grained features [16].

In this paper, we proposed using traits as basic units of adaptivity. Since traits are groups of methods each defining a behavioral adaptation, we achieve fine-grained adaptivity at the level of methods. At the same time, traits provide a convenient composition mechanism to combine these methods in small cohesive units dealing with a related behavior, which can then be associated to particular execution contexts. The definition of features as contexts (and thus traits), allows us to provide specialized behavior to particular situations in the surrounding execution environment, and offers a modularization mechanism for the units of adaptation that is finer-grained than full components or classes, yet coarser-grained than independent individual methods.

Third party features.

The ability to dynamically compose features and to define behavioral adaptations of existing features offers interesting opportunities to vendors for providing third party features to be included in already existing software systems. The feature clouds programming model has the potential to realize this vision, by allowing vendors to deploy to and remove from the cloud their owned features. Such a service deployment process can be partially managed through the Feature Store. However, there is still a need for defining a common vocabulary, definitions and agreements on how and when to activate or deactivate these features and how they should interact with the already existing features in the feature cloud.

Features need to use a common ontology, provide a concrete usage API and dedicated specialization interface specifying which assumptions can be made about them and under what conditions their services are to be used (e.g., context activation conditions). Such specifications, together with the dependency relations between the features, could serve as a contract between the developer of the original system and the service provider.

Composition and run-time verification.

Since systems are composed from different features at run time, and especially if we take into account the possibility of having third party features, it is important for the different features to define their assumptions, expectations and dependencies with other features up front. To ensure that the composed system exhibits the expected behaviour and that no unexpected interactions arise at run-time, the contracts specified by the different features should be verified at run-time when the features are composed, and when features get activated or deactivated the feature dependency relations should be taken into account.

In previous work [1] we explored how and what formalisms could be used to verify the conformance of context dependency relations at run-time, and thus provide more guarantees on the consistent functioning of the system. The types of dependencies discussed in Section 2.2 could be extended or modified by taking inspiration from relevant dependencies and formalizations that have been proposed in the product-line or feature orientation communities. Furthermore, other run-time verification techniques should be explored to provide more guarantees on the correct behavior of the system after composition.

Security issues may also arise in a setting where services are composed dynamically from a cloud of features developed by different vendors. We therefore need techniques to verify and ensure security aspects at run time as features are composed, such as ensuring system integrity, protecting the system from harmful feature code or services anomalies, and protecting sensitive data. To protect the system from harmful features, for example, we could rely on a mechanism for the run-time inspection and verification of adapted code. Ideas from symbolic execution [14] could be used to inspect the code of features as they are composed at run time.

Feature clouds infrastructure.

Services provided by software systems not only require to adapt their behavior to particular situations of their surrounding context, but often need to adapt their state, data, or presentation layer as well. The development of our case study for feature clouds was not different. Sometimes a simple behavioral adaptation, like the UK Speed Gauge feature which provides a reading of the vehicle's speed using the royal measure system, requires some boilerplate code to ensure that the display is refreshed upon activation and deactivation of the feature, as shown in Snippet 5.

To avoid such boilerplate code and in order to be able to adequately adapt the data and presentation concerns of software systems as well, a more holistic vision of the feature cloud model is required to truly realize it. To deal with the data and presentation concerns, inspiration can be taken from recent research in context-aware and adaptive information systems and user interfaces, but would still need to be integrated within the feature clouds model in order to provide a comprehensive programming model.

5. CONCLUSION

In this paper, we presented our vision of features on demand, where, as opposed to being monolithically structured blocks conceived for a particular purpose, user, or context of usage, we see software applications rather as clouds of features that are composed on the fly from sets of finer-grained features. Particular services are not pre-composed in a core system, but rather are dynamically composed from a set of available features or adaptations of existing features, as long as they are consistent with each other.

To achieve this goal, context-oriented programming is an appropriate and effective underlying technology, precisely because it supports the key properties needed for conceiving feature clouds: definition of fine-grained features describing new or adapted behavior, the ability to associate these features with particular execution contexts, declaration of dependencies between features, and dynamic composition of features depending on the execution context, declared dependencies and chosen composition policies.

To validate our claim, we implemented a feature-oriented programming extension on top of the Context Traits programming language extension for JavaScript. This model was successfully used to build an on-board car system, interfacing with and adapting to the execution context of a real car. The system provides a Feature Store offering a variety of features that can assist drivers, a Feature Discoverer which gathers relevant information about the surrounding environment and activates or deactivates features accordingly, and a Feature Manager which is aware of the different composition policies and interaction relations and which takes these into account whenever features are (de)activated.

Thanks to its intrinsic properties we thus believe COP to be an appropriate technological base which has the potential to realize the development of industrial software systems, which are built from a cloud of features by dynamically composing services from a set of features on demand.

6. ACKNOWLEDGMENTS

This work has been partially supported by the IWT funding agency through the ExaScience Life project. We thank the anonymous reviewers for their constructive comments on an earlier version of this paper.

References

[1] N. Cardozo. *Identification and Management of Inconsistencies in Dynamicaly Adaptive Software Systems*. PhD thesis. Université catholique de Louvain & Vrije Universiteit Brussel, 2013.

[2] N. Cardozo, S. Günther, T. D'Hondt, and K. Mens. *Feature-Oriented Programming and Context-Oriented Programming: Comparing Paradigm Characteristics by Example Implementations*. Intl. Conf. on Software Engineering Advances. IARIA, 2011.

[3] P. Costanza and R. Hirschfeld. *Language Constructs for Context-Oriented Programming: An Overview of ContextL*. Proceedings of the Dynamic Languages Symposium. ACM Press, 2005.

[4] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. *Traits: A Mechanism for Fine-Grained Reuse*. ACM Transactions on Programming Languages and Systems 28.2 (2006).

[5] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux. *Subjective-C: Bringing Context to Mobile Platform Programming*. Intl. Conf. on Software Language Engineering. Springer-Verlag, 2011.

[6] S. González, K. Mens, M. Colacioiu, and W. Cazzola. *Context Traits: dynamic behaviour adaptation through run-time trait recomposition*. Intl. Conf. on Aspect-Oriented Software Development (AOSD'13). ACM Press, 2013.

[7] S. Günther and S. Sunkle. *rbFeatures: Feature-Oriented Programming With Ruby*. Science of Computer Programming 77.3 (2012).

[8] T. Kamina, T. Aotani, and H. Masuhara. *EventCJ: A Context-Oriented Programming Language with Declarative Event-based Context Transition*. Intl. Conf. on Aspect-Oriented Software Development (AOSD'11). ACM Press, 2011.

[9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-21. Carnegie-Mellon University Software Engineering Institute, 1990.

[10] P.-Y. Orban. *Using Context-Oriented Programming for Building Adaptive Feature-Oriented Software for Car On-Board Systems*. MA thesis. Université catholique de Louvain, 2013.

[11] T. Poncelet and L. Vigneron. *The Phenomenal Gem: Putting Features as a Service on Rails*. MA thesis. Université catholique de Louvain, 2012.

[12] G. Salvaneschi, C. Ghezzi, and M. Pradella. *ContextErlang: Introducing Context-Oriented Programming in the Actor Model*. Intl. Conf. on Aspect-Oriented Software Development (AOSD'12). ACM Press, 2012.

[13] G. Salvaneschi, C. Ghezzi, and M. Pradella. *Context-Oriented Programming: A Software Engineering Perspective*. Journal of Systems and Software 85.8 (2012).

[14] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. *A Symbolic Execution Framework for JavaScript*. IEEE Symp. on Security and Privacy (SP). 2010.

[15] L. Tao. *Shifting Paradigms with the Application Service Provider Model*. Computer 34.10 (2001).

[16] E. Truyen, N. Cardozo, S. Walraven, J. Vallejos, E. Bainomugisha, S. Günther, T. D'Hondt, and W. Joosen. *Context-oriented Programming for Customizable SaaS Applications*. Symp. on Applied Computing. ACM, 2012.