Configurator-as-a-Service: Tool Support for Deriving Software Architectures at Runtime

Varvana Myllärniemi, Mikko Ylikangas, Mikko Raatikainen, Jari Pääkkö, Tomi Männistö
Aalto University, Finland
firstname.lastname@aalto.fi

Timo Aaltonen
Nokia Research Center
Finland
timo.ta.aaltonen@nokia.com

ABSTRACT
Variability in software architectures, and especially dynamic variability in software architectures, calls for tool support. The complexity involved in the variability means that tools should be able to efficiently derive architectures at runtime. Our contribution is to offer concepts and an expository instantiation of Configurator-as-a-Service (CaaS). CaaS provides integrability, separation of derivation concerns, and automation. The approach is validated with a case of software devices, where proximity-based, distributed service compositions of mobile devices are derived at runtime.

Categories and Subject Descriptors
D.2.11 [Software Architectures]: Service-oriented architecture (SOA)

General Terms
Design

Keywords
Software architecture, dynamic variability, software as a service

1. INTRODUCTION
Software variability is the ability of software to be efficiently extended, changed, customized, or configured for use in a particular context [22]. Typically, variability in software is a consequence of reuse, for example, in software product lines. A software product line (also termed a software product family) is a set of products that share a common, managed set of features [4], a common architecture and a set of reusable components [3]. Thus, variability in a software architecture, features, and components is an important phenomenon in software product lines.

Managing variability can become complex, since the number of potential variants grows exponentially when new variability is introduced. Therefore, variability in software product line architectures is often described by a means of models. Individual architecture variants can then be derived by resolving variability in such models. Efficient tool support eases the complexity involved in both architecture modelling and derivation.

Not all variability is static. Certain situations require that architecture variants need to be derived or adapted at runtime. For example, a single mobile phone may need to adapt its services to context changes. As another example, web-based service compositions may need to be derived dynamically to match user preferences. Further, services provided in a smart space may need to react to newly identified devices. Dynamic derivation and adaptation cannot rely on human involvement and intelligence, but the rules of dynamic derivation are often captured into an architecture model of variability. A tool then utilizes the model in deriving an architecture variant at runtime. The derivation of an architecture variant should happen quickly enough, even for large and complex models.

Our contribution is to present the concepts and implementation of Configurator-as-a-Service (CaaS), a tool for deriving software architecture variants at runtime. In our contribution, variability in software architectures is modeled with the Kumbang language [2], which has been defined rigorous semantics. A Kumbang architecture consists of two viewpoints: a feature viewpoint, and a component-connector viewpoint; such viewpoints can be applied to several kinds of dynamic architectures. Given a Kumbang model, CaaS finds a consistent and complete architecture variant. CaaS encapsulates the computational complexity involved in the variability by utilizing an efficient inference engine smodels [21] that operates with the stable model semantics. CaaS is implemented and deployed as a stateless web service, which accepts input messages in the form of XML.

There are several benefits of CaaS. CaaS is integrable with any runtime derivation framework, since it is provided as a stateless service with a well-defined interface. CaaS separates several concerns related to runtime architecture derivation, e.g., computationally complex yet generic functionality from that more specific to the particular dynamic environment. CaaS enables automatic and autonomous computation, since there is no human involvement necessary to utilize CaaS; all needed derivation knowledge is captured into a model, while requirements for the derived architecture variants are given as input.

To follow the guidelines of the design science methodology [9], the suitability of our construct is evaluated with a
real and running case of social devices. In more detail, the case involves dynamic derivation of proximity-based service architectures, where services are provided by different mobile devices, and are constrained by the status of the devices as well as by the preconditions related to the services.

The rest of the paper is organized as follows. Section 2 discusses the previous research on top of which our construct is built. Section 3 describes the Kumbang ontology and language, which are used for capturing architectural variability. Section 4 presents the CaaS tool. Section 5 discusses the validation with the case, while Section 6 discusses the benefits and limitations of our approach. Finally, Section 7 concludes.

2. BACKGROUND

Our contribution utilizes many general concepts related to software product lines and software product line architectures [3,4]. For example, the distinction between domain engineering and application engineering is also present in our work: a product variant is derived from software product line models and assets. As another example, our contribution utilizes feature modelling [13], which has been heavily studied in the software product line community as a means of modelling variability and deriving variants.

Traditionally, the variability in software product lines has been bound at design time. However, the idea that some or potentially all variability in a software product line could be bound and rebound at runtime has been acknowledged; such a setting is called a dynamic software product line [10]. In many cases, approaches related to dynamic software product lines utilize features as a means to derive or adapt variants at runtime, e.g., [8,14]. However, there are also approaches that use variability in components as a driver for runtime adaptation, e.g., [11].

In service-oriented architecture (SOA) and service-oriented computing (SOC), distributed services should be discovered, composed and recomposed at runtime. However, different dynamic and adaptive aspects of service compositions are still research challenges [19].

In the domain of software architectures, the idea of components and connectors has been heavily utilized in many architecture description languages (ADL). For example, the Koala component model [24] utilizes components, interfaces, and bindings to capture software product population architectures. Within the domain of software architecture models, it has also been acknowledged that there may be a need for dynamic adaptation. For example, [15,23] use explicit models of components and connectors as runtime artifacts allowing architecture-based adaptation.

Finally, our contribution utilizes concepts from product configuration, which aims at satisfying different customer requirements through mass customization [20]. Instead of explicitly enumerating all products, a configurable product can be implemented as a configurator-as-a-service.

In the following, we discuss the Kumbang language, the concepts of which are implemented in the Configurator-as-a-Service (CaaS) tool. However, Kumbang is not the only possible conceptualization that a configurator could utilize but rather a proof of concept implementation. Further, the concepts and the design of CaaS (see Section 4) are not Kumbang-specific, but many different conceptualizations could be implemented as a configurator-as-a-service.

Kumbang is a modelling language and an ontology for modeling variability in software product line architectures from feature and component point of views. Kumbang is a synthesis of product configuration concepts [16], the Koala architecture modeling method [24], and feature modeling approaches [13]. In the following, we only briefly outline the basic capabilities of Kumbang, whereas a comprehensive description can be found in [2].

Kumbang differentiates between a configuration model, which corresponds to a software product line architecture, and a configuration, which represents one product individual derived from a configuration model. Variability in features and components is modelled explicitly in a configuration model, whereas in a configuration, all variability has been resolved. The elements in a configuration model are referred to as types, while the elements in a configuration are referred to as instances.

Kumbang models enable describing varying architecture from two viewpoints: from the feature and the component point of views.

The feature view is used for modeling feature types, which represent user visible functional characteristics of a system. Feature instances can contain other feature instances as their subfeatures. Such a varying composition structure is specified within a feature type using subfeature definitions, which specify the cardinality and the possible types of composed features. Further, feature types can inherit each other. Feature types can be characterized with attribute definitions, which represent name/value pairs. Finally, constraints can be used to specify more elaborate rules for the selection of different feature instances; in a very simple case, by specifying that a certain feature requires another feature. Hence,
the Kumbang feature modeling concepts synthesize many existing feature modeling methods, and can be used to capture typical variability constructs found in other feature modeling methods.

The component view specifies component types. A component type represents a distinguishable architectural element with explicitly defined interfaces. The approach is ignorant as to whether a component actually refers to, e.g., a runtime or design time element or a specific component technology, as far as the component adheres to this definition. Again, the term “component type” is used in the variability model similarly as in features. An interface type represents a set of operation signatures; these are attached to component types using interface definitions. An interface direction is either provided or required. Component instances can be composed with each other. The construct used for specifying component composition is a part definition that specifies the cardinality and possible types of composed components. Similarly to feature types, component types can inherit each other, be characterized by attribute definitions, and specify constraints that restrict how instances in the component view can be selected. Hence, the variants of a component view can differ in terms of the composition of components, the connections between the interfaces, and the attribute values defined.

In order to integrate the feature and the component views, implementation constraints can be used to specify relationships between them. In a very simple case, a specific feature may require a specific component. In general, the constraints can be bi-directional and impose many-to-many relations between views. Consequently, the implementation constraints can be as complex as can be specified with the Kumbang constraint language [1].

The semantics of Kumbang are rigorously described using natural language and a UML profile [2]. Kumbang has also been provided with formal semantics by implementing a translation from Kumbang to WCRL (Weight Constraint Rule Language) [21], a general-purpose knowledge representation language. This translation also enables the use of the smodels [21] inference engine in supporting tools.

4. THE CAAS TOOL

4.1 Overview of the concepts

The Configurator-as-a-Service (CaaS) is a member of a tool set called Kumbang Tools [1, 17]. CaaS is particularly targeted for runtime derivation. CaaS enables finding configurations at runtime by binding variability in Kumbang models. CaaS is model-independent and domain-independent in the sense that all logic and reasoning is based on the model given as input.

Figure 2 illustrates how CaaS can be utilized for runtime derivation. Since CaaS is a general-purpose tool, CaaS is operated by a runtime derivation engine (Figure 2) that is specific to the particular dynamic setting and environment. The runtime derivation engine takes care of the trigger for the runtime derivation, as well as the instantiation of components. CaaS is indifferent to the implementation and deployment of the runtime derivation engine. A runtime derivation engine can be deployed on a mobile device or on a separate server, or it can even be a service itself.

In contrast to the runtime derivation engine, CaaS encapsulates the functionality for finding a configuration. The reason for such an encapsulation is that checking models and configurations is computationally expensive. In the worst case, checking consistency requires at least an exponential
amount of time in relation to the size of the problem. Therefore, CaaS utilizes an efficient *smodels* inference engine [21], which is a general-purpose inference tool based on the stable model semantics of logic programs.

The runtime derivation engine operates CaaS through two temporally separate phases (Figure 2): model upload and runtime derivation.

In the model upload phase (Figure 2), the runtime derivation engine needs to provide CaaS with a Kumbang configuration model that describes the varying architectures from the feature and the component point of views. There are two options from where this configuration model can originate. An existing tool, Kumbang Modeler [1,17], can assist in creating the model. Since Kumbang Modeler is a GUI-based tool, it is applicable for situations in which models can be defined pre-runtime. If there is a need to create or modify configuration models at runtime, the runtime derivation engine can generate Kumbang models as needed. Section 5 discusses a case where configuration models were generated at runtime.

After receiving the configuration model (Figure 2), CaaS prepares the model to be used by the inference engine *smodels* [21]. CaaS translates the model to the Weight Constraint Rule Language (WCRL) [21], a general-purpose knowledge representation language. Thereafter, CaaS grounds the WCRL model with the *lpars* tool [21] to normal form constraints; grounding enables the model to be used with *smodels*. CaaS also checks whether the model is valid, that is, whether valid configurations can be derived from it.

The reason to have model upload separately from actual runtime derivation is twofold. Firstly, one configuration model can be used for deriving several configurations; thus uploaded configuration knowledge can be reused. Secondly, for large configuration models, the preparation of the configuration model, and the grounding in particular, have a certain performance penalty. Thus, the potentially time-consuming grounding can be done beforehand, and the actual runtime derivation can happen more efficiently.

The runtime derivation phase is illustrated in Figure 2. Identifying the need for runtime derivation, represented as a trigger in Figure 2, is the responsibility of the runtime derivation engine. CaaS is indifferent to whether the need arises from, e.g., a context change or a specific event. After detecting the trigger, the runtime derivation engine needs to identify the requirements for the desired configuration architecture variant. If the configuration model is valid and complete, CaaS finds a consistent and complete set of components that are instantiated and deployed. However, CaaS is not restricted to only this mode of operation, but configuration selections can be any partial configuration.

After CaaS is given a set of configuration selections as input, it returns a valid configuration, that is, a consistent and complete configuration (Figure 2). A consistent configuration is such that no rules of the configuration model are violated. A complete configuration is such that all the necessary selections have been made. A complete and a consistent configuration is calculated as a *stable model* by *smodels* [21]. With the stable model semantics, the configuration problem is expressed as a logic program, and stable models of the programs give solutions to such problems. Further, the configuration selections are represented as a compute statement that is used for calculating the stable models.

Finally, the runtime derivation engine instantiates and deploys the found configuration (Figure 2); exact details are out of the scope of the CaaS design.

In a typical usage scenario, the feature view and the component view have different roles. Typically, configuration selections are captured as a feature configuration; these features represent the requirements or goals of the desired runtime variant. Given features as configuration selections, CaaS finds a consistent and complete set of components that satisfies the features. Afterwards, components can be instantiated and deployed. However, CaaS is not restricted to only this mode of operation, but configuration selections can be any partial configuration.

### 4.2 The CaaS interface

This section briefly describes the main design principles of CaaS and the interface through which CaaS can be accessed. The CaaS interface is stateless in the sense that it does not distinguish between different clients, nor preserve any state between requests. Also for the ease of integration, CaaS operates on the basis of a single request – single response method over HTTP using XML as input, and XML and HTTP status codes with possible exception strings as output.
Two key points of interaction with CaaS are identified from (Figure 2): firstly, CaaS receives Kumbang configuration models; secondly, CaaS receives configuration selections and returns complete and consistent configurations. CaaS is designed to provide a single HTTP interface against which both the model upload and the configuration derivation are possible. Figure 3 illustrates the structure of the XML messages accepted by the CaaS interface.

A request for uploading configuration models will contain a model identifier as well as the Kumbang model itself. A request for uploading a model contains only element model (Figure 3), where the content of element model is the Kumbang configuration model as a string.

A request for deriving a configuration will again contain a model identifier, but the Kumbang model itself is left empty (Figure 3). In addition, the request for derivation will contain configuration selections as element configuration. Configuration selections are listed separately for both the feature and the component views (elements feature and component in Figure 3).

An example of a derivation request is illustrated in Figure 4. The identifier of the pre-loaded configuration model is MyDynamicApp. The configuration selections consist of the feature MyAppFeatures with two subfeatures Status and Preferences; both the Status and the Preferences define values for several attributes. In the example, the status and the preferences are used to characterize the features of the component configuration that CaaS should derive.

4.3 CaaS internal functionality

Since CaaS is based on and is part of the previously implemented KumbangTools toolset, the implementation language is Java, except for the external systems lparse and smodels [21].

Figure 5 illustrates the handling of a request. If a configuration model is received, it is parsed into Kumbang language constructs, translated into WCRL (Weight Constraint Rule Language), submitted to an executable tool, lparse, for grounding, and finally saved to a disk drive for further use. Further, if any configuration selections are found from the request, a toolchain is invoked to find a complete and consistent configuration. Firstly, the configuration selections are instantiated as a new partial Kumbang configuration. The partial configuration is then translated into a compute statement, which is submitted to the smodels inference engine. If a complete and a consistent configuration is found, that is, a stable model is found, it is translated into a Kumbang configuration, exported to XML, and returned as a response.

5. VALIDATION

To validate the design presented in Section 4 and the soundness of the concepts presented in Section 3, we implemented Configurator-as-a-Service (CaaS) as a servlet running on top of the Apache Tomcat engine. At the time of writing, CaaS was deployed on our internal server and utilized for research purposes.

CaaS was further validated as a functioning part of a larger case of dynamic Social Devices (see Figure 6) [5]. The case is described in more detail in the following.

5.1 Case: Social Devices

Social Devices are characterized by computationally capable, smart devices that are aware of other social devices in their proximity. Social devices may perform interactive user-visible actions whenever the social devices are in each other’s proximity. The actions differ largely in nature but in general require interaction of different devices. A key characteristic of the actions is to provide human users some visible value intelligently but the social devices are also autonomous not requiring interaction from users. The action defines a script which is then orchestrated as user-visible behaviour.

Social devices resemble smart spaces to some degree: both utilize the proximity of devices to provide services. In contrast to smart spaces, the social devices are not tied to a particular location, and the actions always provide user-visible behaviour.

Running example. In the following, social devices are exemplified with a scenario of a social calendar reminder. Adam, Mia, Vincent and Jules are in a meeting. In the middle of the meeting, Adam’s mobile phone discovers that there is only 20 minutes before Adam has another appointment in his calendar. This triggers a collaborative interaction in which social devices discuss the upcoming event with
each other. Mia’s phone says: “Should we tell Adam that he should be in another meeting room soon?” Vincent’s phone replies: “Yes, we should!” Adam’s phone says: “Yes, I know that, I am about to set an alarm.”

Social devices near other social devices form a proximity group, meaning that devices are so close to each other that they can initiate a social interaction. When devices move, proximity groups are constantly changing; thus proximity groups are not tied to any particular place, like is the case with smart spaces. We consider proximity to be a transitive and a symmetric relationship between devices. In the current implementation, proximity is discovered by Bluetooth.

Social devices have different capabilities. For example, some devices have the capability to speak aloud by synthesizing text, or to produce audible sound or melody, while other devices have the capability to stream video or show pictures. The user of social devices can install and enable different capabilities to her liking, and different devices can implement capabilities in different ways. In addition, social devices have internal status and events that are related to the capabilities.

Running example. In the social calendar reminder scenario, the proximity group contains four devices: Adam, Mia, Vincent and Jules. All devices are able to speak aloud by synthesizing text; hence all have the capability TalkingDevice. Further, Adam also is a CalendarSource, since Adam is able to discover calendar events. A device may indicate its willingness to participate in a discussion, as well as indicate whether its environment is silent at a moment. Thus, all TalkingDevices have two Boolean-valued statuses isWillingToTalk and isSilence.

Social devices can participate in collaborative actions. Each action is divided into two parts. The action body defines how devices behave; the action execution is orchestrated among devices in a coordinated manner. The action precondition guards whether the action body can be executed. An action precondition can specify the number and the roles of devices, as well as their required capabilities. Typically, an action precondition states that devices must be in the same proximity group. Finally, action preconditions can specify constraints on the statuses of the devices.

Finally, actions are triggered by special status changes. For a certain trigger, there can be several alternative actions. However, a trigger cannot specify which action is executed, or which devices participate in an action, but this is decided at runtime.

Running example. In the social calendar reminder scenario, the discovery of an approaching calendar event can trigger two possible, alternative actions: Conversation, FakeCall, or none at all. The action Conversation has several preconditions: there must be three devices in each other’s proximity, these three devices must have the capability TalkingDevice; one device has to be the CalendarSource with a certain eventApproaching; and all devices must be willing to talk and have a silent environment (isWillingToTalk and isSilence).

5.2 Using CaaS for Social Devices

The configuration problem in social devices is to find an action and a set of devices from a proximity group, so that the action preconditions are fulfilled. The runtime derivation is triggered by a status change that can imply several alternative actions. The derivation and the execution of an action is needed to be done automatically without user involvement and without large latencies.

The architectural variability comes from several sources (see Table 1). New devices can be registered or new capabilities installed; the proximity groups change when devices move around; the status of the devices change with, e.g., context changes; a specific trigger can cause the execution of several alternative actions; devices may meet or not meet precondition criteria for an action; and devices may be in different roles within an action.

Firstly, social devices were captured in configuration models written in Kumbang. Table 1 illustrates how variability in the social devices and the calendar reminder scenario in

<table>
<thead>
<tr>
<th>Variation in Social Devices</th>
<th>Modelled in Kumbang as</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registered device</td>
<td>Component</td>
</tr>
<tr>
<td>Capability (TalkingDevice, CalendarSource)</td>
<td>Abstract component</td>
</tr>
<tr>
<td>Action (Conversation, FakeCall, none)</td>
<td>Component</td>
</tr>
<tr>
<td>Roles of devices in action (source, d1, d2)</td>
<td>Part specification</td>
</tr>
<tr>
<td>Device status (isWillingToTalk, isSilence)</td>
<td>Features and attributes</td>
</tr>
<tr>
<td>Trigger (eventApproaching)</td>
<td>Features and attributes</td>
</tr>
<tr>
<td>Which devices belong to the same proximity group</td>
<td>Features and attributes</td>
</tr>
</tbody>
</table>
particular was modelled. The feature view was used for describing all the statuses that affect the preconditions of an action. The component view was used for describing alternative actions, devices, and the rules in which devices participate in actions. Implementation constraints were used to map preconditions (feature view) to actions and devices (component view).

Secondly, social devices were implemented and integrated with CaaS, and actions were executed at runtime. Figure 6 illustrates the overall architecture of social devices. CaaS is utilized for deriving device and action configurations, whereas the Proximity-Based Controller (PBC) acts as a runtime derivation engine.

Step 0 in Figure 6 illustrates how the social devices architecture constructs the necessary configuration model for CaaS. Whenever new devices or new device capabilities are registered, or whenever new actions are uploaded, PBC generates a configuration model and uploads it to CaaS for further use.

Steps 1-6 in Figure 6 illustrate the steps needed for runtime derivation. Social devices constantly identify other devices in proximity (Step 1). Devices report changes in their status as well as their proximity info to PBC (Step 2). Whenever a device triggers an event, PBC attempts to derive an action and a device configuration (Step 3). Device status and proximity info are captured as configuration selections and sent to CaaS; CaaS returns a configuration that defines an action and the participating devices (Step 4). Finally, the action is executed and orchestrated among devices (Steps 5-6).

6. DISCUSSION

6.1 Benefits

Providing runtime derivation support as a service itself has many beneficial properties.

CaaS is easy to integrate with at runtime, since it is a stateless service. Further, well-defined interfaces that operate with XML input also ease integration. This means that CaaS can be used for several different purposes; the same CaaS instance can even support very different dynamic cases at once. The only requirement is that the concepts involved in the architecture can be expressed with the concepts provided by the Kumbang ontology (that is, as features or component-connectors).

Due to the relatively general nature of the Kumbang concepts (features, components), and easy integration, we believe that CaaS is suitable to be utilized in many different dynamic settings. Even in the case of social devices, when the dichotomy of features and components was not obvious, it was possible to use Kumbang modelling constructs to capture the necessary architecture and its variability.

Many approaches benefit from the possibility of having a configurator as a service itself. When several distributed services are composed into one, it is quite natural that also the composition support is a service. For example, self-adaptive and self-healing services may benefit from being able to invoke a configurator service that can be used for the reconfiguration of a service composition. Further, mobile devices that have to adapt to context changes are often relatively limited in processing power and battery life. Hence, it may be worthwhile to assign the computationally complex task, finding an architecture, to an external service. In general, CaaS is an enabler for many automatic and autonomous computing settings.

Further, CaaS can aid in dynamic decentralized service compositions [18]. In the social devices, a service composition and its execution is decentralized among different devices. In some case, also architectural knowledge can be decentralized and scattered among several deployment nodes [18]. In a decentralized setting, there may not be a single system, e.g., a runtime derivation engine, that is responsible for managing derivation. For this purpose, we are currently studying how architectural knowledge could be provided to CaaS in separate fragments: CaaS would then compose such fragments to a meaningful configuration model.

6.2 Limitations

The responsibility of CaaS in runtime derivation has been intentionally limited to configuration reasoning, that is, to finding a consistent and complete configuration. Thus, CaaS provides no support for detecting runtime triggers, nor support for instantiating any found configurations. However, there are also benefits of this scope: runtime triggers and instantiation are specific to a dynamic setting, which means that CaaS is more generic and applicable to many situations.

A limitation in the current design of CaaS is its relatively weak support for adaptation and reconfiguration issues. The current design as well as the case of social devices only considered derivation: each configuration is created from scratch, without considering an existing, operational configuration. Compared to derivation, adaptation has certain special requirements, like aiming for minimal changes in the configuration, keeping system state, and shutting and starting components gracefully. CaaS could support the adaptation of existing architectures in the following way. Instead of giving only configuration selections as derivation input (see Figure 2), the input should also include the current components in the existing, executed architecture. CaaS should then change only those components that are conflicting with the new set of configuration selections, and keep the rest.

Another limitation in the current implementation is that CaaS only finds one valid configuration. A valid configuration is one that satisfies all of the configuration selections and is consistent with the configuration model. If there are several valid configurations that satisfy the configuration selections, the smodels inference engine picks one of them based on its internal logic. Thus, the runtime derivation engine cannot affect which specific one of the many valid configurations is received. This is not an issue, if configuration selections unambiguously determine the rest of the configuration (e.g., a feature configuration determines components unambiguously). Further, this is also acceptable, if all valid configurations are as good as any other (like is the case in the social devices currently). However, in general, there may be a need to rank all possible valid configurations in some way, and select the most suitable of them based on, e.g., quality properties and user preferences. We are currently working on extending CaaS to utilize recommendation techniques to be able to prioritize and select the best or a good enough configuration among all possible valid configurations. Such recommendation-based derivation may require that the configuration model is augmented with information of quality properties such as the cost or the performance of components.
7. CONCLUSIONS

We introduced the concept and design of Configurator-as-a-Service (CaaS), which is a general-purpose and standalone tool for deriving architecture variants at runtime. CaaS aids runtime derivation by finding a suitable architecture variant, that is, a complete and consistent configuration. In contrast, identifying the trigger for derivation as well as instantiating the derived architecture variant is outside the scope of the CaaS design. For capturing runtime architectures, CaaS utilizes feature and component points of view, defined by the Kumbang language.

To demonstrate the contribution, CaaS was implemented and deployed. Further, the contribution was validated with a case of social devices, where architecture variants consisted of device and action configurations. Finally, the benefits and the limitations of the current approach were considered.

There are several future work items that we are currently studying.

Firstly, applying CaaS in various autonomous, dynamic settings is a promising research area. This may either mean self-managing device or service compositions, or even situations in which architectural knowledge itself is decentralized. For self-managing services, a better support for architectural adaptation instead of mere derivation is needed.

Secondly, there may be several valid configurations that CaaS can identify for one configuration request. We are currently working on utilizing recommendation techniques for ranking and prioritizing possible architecture variants, so that an informed selection among all valid configurations can be made.

8. REFERENCES