HAL
open science

Model-Driven Design, Development, Execution and Management of Service-Based Applications

Diana Moreno, Jacky Estublier

To cite this version:

Diana Moreno, Jacky Estublier. Model-Driven Design, Development, Execution and Management of Service-Based Applications. SCC 2012 - International Conference on Service Computing, Jun 2012, Honolulu, HI, United States. pp.470-477, 10.1109/SCC.2012.64 . hal-00745510

HAL Id: hal-00745510

https://hal.science/hal-00745510v1

Submitted on 25 Oct 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Model-driven Design, Development, Execution and Management of Service-based Applications

Diana Moreno-Garcia, Jacky Estublier
Laboratoire d'Informatique de Grenoble
F-38041 Grenoble cedex 9, France
{*Diana-Guadalupe.Moreno-Garcia, Jacky.Estublier*}*@imag.fr*

***Abstract*—Service-based software applications, such as pervasive and ubiquitous ones, are increasingly embedded in our daily lives integrating smart communicating devices. Usually, changes in the execution context of these applications occur unpredictably over time, such as dynamic variations in the availability of the used services and devices, or of the user location and needs. This unpredictable variability in the execution contexts makes impossible to know at design-time the exact conditions under which these applications will be used and the services that will be most suited at a given time. Therefore, the architecture of such applications cannot be fully defined at design-time. These applications must be defined in abstract and flexible ways, allowing incremental composition and dynamic adaptation to their execution context at runtime. In this paper, we present a model-driven approach for designing, developing, executing and managing service-based applications. At design-time, an application is mainly defined by its requirements and goals. The application definition can be extended to add specific functional or non-functional concerns, such as dynamic adaptation, deployment or distribution. At development-time, the application can be automatically and incrementally composed, ensuring its consistency with respect to its definition. At runtime, the application execution is supported and controlled by our runtime environment.**

Keywords-service-based composition; desing and development engineering environments; execution platforms.

I. INTRODUCTION

Modern service-based software applications, such as pervasive and ubiquitous ones where services represent the functionalities provided by devices, present a number of characteristics and requirements that make their design, development, execution and management very difficult [1][2]. Indeed, these applications depend on services whose number, dynamic availability and behavior are not known before execution. The assumption that applications are built to run in well-defined contexts, with deterministic and constant behaviors, is no longer valid for modern applications. Thus, the architecture of these applications cannot be fully fixed nor predictable at design-time anymore.

Considering the variability in the execution contexts, modern applications must be defined in abstract and flexible ways, allowing their concrete incremental composition all along their lifecycle, and letting controlled opportunistic and dynamic behaviors at runtime.

We define **opportunism** as the capability of a service-based application to use services available at runtime. An opportunistic behavior may be required, for example, to improve efficiency, to support reuse and collaboration, to avoid conflicts, or simply because the application cannot instantiate the needed services (e.g. devices).

We define **dynamism** as the capability of a service-based application to manage services that can appear and disappear at runtime. A dynamic behavior may be required, for example, to manage services whose number, location and availability are variable, or whose deployment, instantiation and removal are controlled by third parties (e.g. administrators or other applications), or to consider new application requirements and needs. Managing dynamic services entails adapting applications dynamically, which implies architectural reconfigurations by adding, updating or removing components and connectors.

To address the complexity of designing, developing, executing and managing modern service-based applications, we propose: (1) **design and development environments** that allow, on the one hand, defining an application at a high-level of abstraction (via the set of properties it must have and satisfy) leaving room for flexible and incremental composition all along its lifecycle, and on the other hand, defining separately specific functional or non-functional concerns, such as dynamic adaptation, deployment or distribution, in order to complement the application definition; and (2) **a runtime platform** that supports the execution of (partial or complete) applications, managing their runtime compositions and ensuring the fulfillment of their definitions and of their associated properties.

Our proposed approach uses models at design, development and runtime [3]. We use development models to define the abstract architectures and goals of applications. We use runtime models to represent, in a high-level of abstraction, the current architectures of the running applications. The use of development models is extended to runtime, allowing to control the applications execution, and also to go on with design and development activities.

The remainder of this paper is organized as follows. Section II introduces our model-driven design and development environments, and details our metamodel for building service-based applications. Section III presents our model-driven runtime environment supporting and managing the execution of such applications. Section IV discusses related work, and finally section V concludes this paper presenting our major contributions.

II. DESIGN AND DEVELOPMENT ENVIRONMENTS

We propose a set of model-driven engineering environments, named CADSEs (Computer Aided Domain Specific Engineering environments) [4], whose goal is to help architects and developers performing software engineering activities in a specific domain. Thus, we have developed specialized CADSEs for the design and development of service-based applications taking into consideration the characteristics and requirements of modern applications such as pervasive and ubiquitous ones.

To illustrate, we present a simplified home media center application that allows users browsing, selecting and reproducing audio and video files using different electronic devices inside the home such as televisions, speakers, screens, laptops and cellphones. This application is composed of a media manager service which interacts, on the one hand, with a number of media servers (such as home media servers or audio-and-video-on-demand Internet servers) containing audio and video files, and on the other hand, with a media player service controlling the reproduction of a selected media file.

New media server services can be dynamically available in the house. When a new media server service is available, it automatically joins the home media center application (i.e. it is connected to the media manager service) allowing accessing its contained media files.

A media player service interacts with audio and video services in order to reproduce media files. Audio and video services allow controlling the audio and video devices dynamically available in the house. Several types of devices can provide audio and video functionalities and several instances of a type of device can be present in the house. The home media center application must use not only the available devices, but also the best suited ones (e.g., to the user preferences or location) at a given point in time.

Finally, when a currently used service or device is removed or fails, the application should be dynamically adapted in order to continue providing its services using alternative services and devices.

Due to the unpredictable availability of its components, it is not possible to define concretely the architecture of this application at design-time. We propose then to define a service-based application in a high-level of abstraction, partly by intention (via a set of invariant properties that the application must satisfy, i.e. the application goal) and partly by extension (via a set of selected interconnected components, i.e. a concrete partial architecture) leaving room for incremental composition and adaptation all along the application life-cycle. Consequently, we propose a component metamodel that aims bringing to service-based architectures the benefits of component-based development, allowing the description, creation, reusability, evolution, composability and encapsulation of components providing and requiring services and resources.

A. *The component-service metamodel*

Component-service is the central concept of our metamodel shown in Fig. 1. A component-service provides resources (i.e. functional interfaces, typed data or events), requires resources, owns static and configurable properties, and is associated with constraints and preferences. Next, we detail the metamodel elements, presenting first the primitive component-services followed by the concepts and components allowing their composition.

1) Primitive components: A component-service can be either a specification or an implementation. A **specification** is an abstract definition of a component-service independent of any given implementation technology. It defines a contract that specifies the common and configurable properties, the provided and required resources, and the constraints and preferences that its implementations must respect. Common and configurable properties are specified as a tuple *<name, type, value>*. The values of the specified common properties are identical and immutable for all the implementations, while the values of the specified configurable properties, being customized by each implementation, are usually different allowing thus to distinguish them (e.g. during selection).

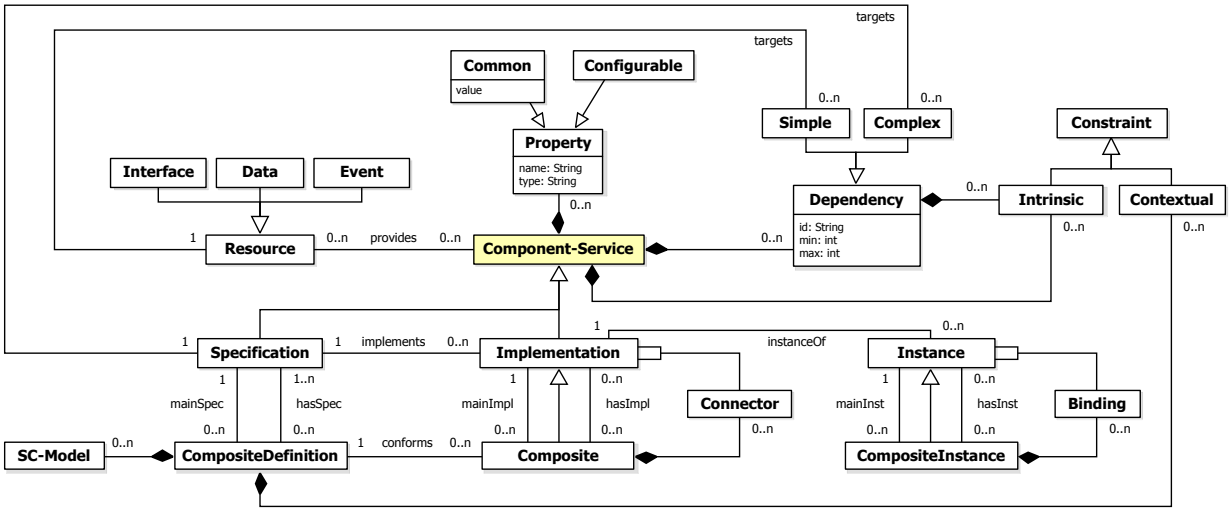

Figure 1. Component-service metamodel

Required resources can be defined via simple or complex dependencies. A simple dependency is defined towards a single resource. A complex dependency is defined towards a single specification that specifies a set of provided resources. A (simple or complex) dependency can indicate cardinality, selection constraints and preferences. Cardinality is defined by the tuple [min, max], where "min" can be either "0" meaning an optional dependency or "1" meaning a mandatory one, and "max" can be either "1" meaning a single dependency or "n" meaning a multiple one.

Constraints and preferences (expressed via our constraint language presented in section B) represent, respectively, the properties that a provider must have and those that are preferable. They are evaluated over the configurable properties of the providers. Using constraints and preferences allows reducing the number of providers that can be selected.

To illustrate the specification concept, consider a MediaManager specification (see Fig. 2), which defines a single provided resource via the interface MediaManagerIt. This interface describes the provided functionality, which is to show the list of current media files and to play a selected media file. The specification defines two complex dependencies: the first one, mandatory and multiple, is towards a MediaServer specification, the second one, mandatory and single, is towards a MediaPlayer specification. The specification defines two configuration properties, "*provider:String*" and "*freeware:Boolean*", to be configured by the MediaManager implementations.

A specification can be implemented by several implementations. An **implementation** implements a single specification with particular properties. An implementation has the characteristics and properties defined by its specification: the provided and required resources, the common properties and the configurable properties with particular values, the constraints and preferences. An implementation can provide and require additional resources, define common and configurable properties for its instances, and add selection constraints and preferences.

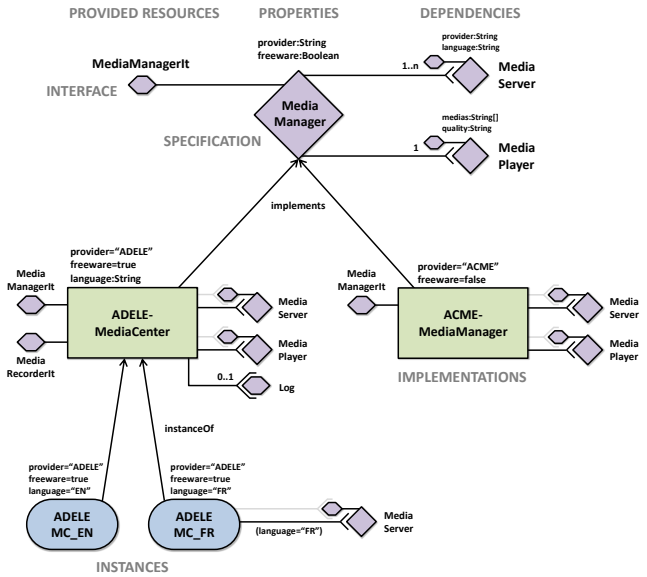

Figure 2. Primitive component-services example

To illustrate, consider two implementations of the MediaManager specification: ADELE-MediaCenter and ACME-MediaManager (see Fig. 2). Implementing the same specification, both implementations provide the interface MediaManagerIt, require the MediaServer and MediaPlayer specifications, have the same common properties (with the same values) and have the same configurable properties (with own values). ADELE-MediaCenter provides an additional resource, defined by the MediaRecorderIt interface, which allows recording media files in scheduled and automatic ways. In addition, ADELE-MediaCenter requires a Log service (defined via a simple dependency towards the Log interface), and defines the property "*language:String*" for configuring its instances.

An implementation can have several instances. At design-time, an **instance** is the declaration of a particular configuration of an implementation. Such an instance has all the characteristics and properties specified by its implementation, i.e. the provided and required resources, the common properties, the constraints, the preferences. An instance personalizes the configurable properties specified by its implementation. Configured properties allow, on the one hand, distinguishing instances from each other (during selection), and on the other hand, establishing the initial values for the instance creation at runtime. In addition, an instance can add selection constraints and preferences to the dependencies specified by its implementation (see Fig. 2).

2) Composite concepts: Primitive components can be assembled in order to compose an application or subsystem. The composition process is orthogonal to the design and development of primitive components. Our composition approach relies on two concepts: composite definition and composite implementation (hereinafter called simply composite).

A **composite definition** describes an application or a subsystem (i.e. composite) at a high-level of abstraction via the properties it must have and satisfy: its architectural characteristics (provided, required and contained resources), and its constraints and preferences (see Fig. 1). Thus, a composite definition specifies a structural composition in terms of specifications, as well as a semantic (or intentional) composition in terms of constraints and preferences. Similar to a reference architecture in software product line (or product family) approaches [5], a composite definition describes structural and semantic properties, including variation points, for a family of composites.

Using our composition and constraint language (presented in section B), a definition for home media center applications can be the following:

```
CompositeDefinition HomeMediaCenter {
   Provides MediaManager;
   Select MediaServer
              (language="FR" or language="EN");
   Prefer MediaServer (language="FR");
   Select MediaPlayer (quality="HD");
   Select MediaRecorder
              (codec="g711" and codec="x264");
   Requires Optional Log (version="1.2");
}
```

This model, named HomeMediaCenter, specifies the architectural characteristics for home media center applications: MediaManager is the provided specification and Log is an optional composite dependency. Providing the MediaManager specification implies that the MediaManager dependencies (i.e. to MediaServer and MediaPlayer) are composite dependencies too. In addition, the model specifies constraints and preferences for selecting MediaServer, MediaPlayer, MediaRecorder and Log providers.

Several different composites, can be derived (created) or associated to a single composite definition, assuring then the fulfillment of the specified properties and constraints.

A **composite** represents an application or subsystem via an assembly of implementations. A composite is itself an implementation (see Fig. 1), and as such, it provides a single specification, inheriting then its provided and required resources, its common and configurable properties, and its constraints and preferences. Being an implementation, a composite can provide and require additional resources, define common and configurable properties for its instances, and add constraints and preferences influencing the selection of its contained and used implementations.

A composite contains a set of interconnected implementations. Before runtime, these implementations and their interconnections represent (partially or totally) the static structure of the application or subsystem to be executed. These implementations can be either primitive or composite, allowing thus the hierarchical composition of composites. Among the contained implementations, there is a **main implementation**, which provides at least all the resources provided by the implemented specification of the composite.

To illustrate the composite concept, consider that the ADELE-MediaCenter implementation (presented in Fig. 2) is a composite one, representing a home media center application. Its external structure remains as described earlier (see Fig. 3). Its internal structure is (currently) composed only of the ADELE-MediaManager implementation, which is its main implementation.

A composite is **incomplete** if any of its contained implementations has unresolved dependencies (like the ADELE-MediaManager in the ADELE-MediaCenter). An incomplete composite can be gradually refined, before or during runtime, by resolving such unresolved dependencies. The principle of such a refinement process, also called incremental composition, is explained later on section C.

Composite dependencies correspond to dependencies of the implementations contained inside the composite. In other words, composite dependencies are promoted dependencies of implementations contained in the composite, which will be resolved by implementations outside the composite. Consider for example the dependency of the ADELE-MediaCenter composite to the MediaServer specification (see Fig. 3). The dependency of its contained implementation ADELE-MediaManager to the MediaServer specification is automatically promoted and resolved outside the composite. This resolution results in a connection between the ADELE-MediaManager and a MediaServer implementation (outside the composite), and another one between the ADELE-MediaCenter and the MediaServer implementation.

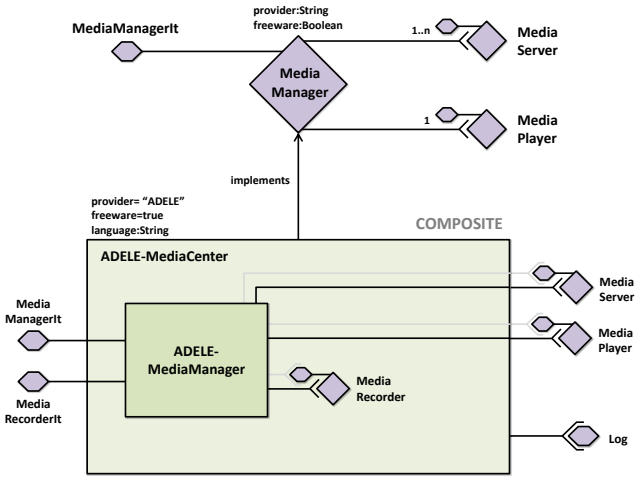

Figure 3. Composite component-service example

The ADELE-MediaCenter composite, associated to the HomeMediaCenter definition presented before, satisfies all the specified properties and constraints.

Being an implementation, a composite can have several instances. At design-time, a **composite instance** corresponds to a particular configuration of a composite, which may also contain the configurations for its contained implementations.

Note that at design and development times we have two levels of composition: abstract (in terms of architectural and semantic properties and constraints) and concrete (in terms of interconnected implementations and instance declarations). Thus, an application can be specified, as usually, by a list of implementations and connections between them; but it can also be specified via an abstract composite definition, leaving then room for flexible, opportunistic and dynamic composition at runtime. In our approach, a composite together with its composite definition constitute a **composite model**.

B. Constraint expression language

We have defined a constraint language that, like the Object Constraint Language (OCL), allows specifying **constraint expressions** on model elements. Our language allows navigating models (through its linked elements) and evaluating constraints expressions over the elements properties. Like in OCL, constraints expressions (hereinafter called simply constraints) can be used to enforce model consistency. Unlike OCL, constraints can be associated with both instances and types. Thus, in our metamodel, constraints can be associated with specifications and implementations (as types), but also with implementations and instances (as instances of the specification and implementation types respectively).

In our language, constraints are strongly typed. Therefore, a constraint can refer to the property "*a*" of an element "*x*" only if either "*x*" (being a type) or the type of "*x*" has declared the property "*a*". In this way, the validity of constraints can be statically verified, enforcing the compatibility between elements.

Constraints can use LDAP-like expressions, navigation, or complex constructions (see [6] for more details). For example, the ADELE-MediaManager implementation associates the following constraint with its dependency to the MediaServer specification:

```
Select MediaServer (provider="ADELE");
```

This constraint indicates that ADELE-MediaManager requires a provider of the MediaServer specification whose "*provider*" property is "*ADELE*". This constraint is valid since the property "*provider*" is defined by the MediaServer specification. Another constraint associated with the ADELE-MediaManager implementation is:

```
Self.requires(name="MediaRecorder")..provides
        (audio="true" and video="true");
```

This constraint expresses, by navigation, that ADELE-MediaManager (specified via the *Self* operator) requires a provider of the MediaRecorder specification whose provided properties "*audio*" and "*video*" are both true.

The constraints directly associated with a component, as in the two previous examples, are called **intrinsic constraints**. Being associated with a component, intrinsic constraints must be satisfied by all composites using the component. Intrinsic constraints can be used to validate the compatibility between components, reducing thus the risk of errors at runtime.

A composite can be associated with a set of **contextual constraints** that must be satisfied by the implementations it contains and uses. The ADELE-MediaCenter composite, being associated with the HomeMediaCenter definition presented before, has the following contextual constraints:

```
Select MediaServer
        (language="FR" or language="EN");
Select MediaPlayer (quality="HD");
Select MediaRecorder
        (codec="g711" and codec="x264");
Requires Optional Log (version="1.2");
```

The implementations that a composite will contain must satisfy both the composite contextual constraints and the intrinsic constraints of the client contained implementations. For example, the ADELE-MediaCenter composite has the contextual constraint `(language="FR" or language="EN")` associated to its MediaServer dependency. The ADELE-MediaManager implementation, contained in that composite as its main implementation, has the intrinsic constraint `(provider="ADELE")` associated to its MediaServer dependency. These constraints will be aggregated in order to be evaluated when selecting a MediaServer implementation for that composite.

Contextual and intrinsic constraints can conflict. Constraint conflicts are checked, from design to runtime, during the composition process. Actually, we propose an automatic backtrack mechanism looking for alternative solutions when composition fails due to unsatisfied or conflicting constraints.

In a similar way, preferences can be associated with components (intrinsic preferences) and with composites (contextual preferences). Preferences are evaluated only if more than one provider satisfies the constraints. The provider fulfilling more preferences will be selected.

The ADELE-MediaCenter composite associates the following contextual preference with the MediaServer specification, indicating that the selection of a MediaServer provider having "*language=FR*" is preferable:

```
Prefer MediaServer (language="FR");
```

Constraints and preferences (intrinsic and contextual) are validated when defined, and evaluated when composing the associated composite and composite instance(s), enforcing then the selection of compatible and suited implementations and instances, respectively. Our composite composition system, presented next, performs these tasks.

C. *Composite composition system*

Our composite composition system is available all along the lifecycle of a component, from its design to its execution, allowing thus its incremental composition (including refinement and adaptation) at any time. This approach enables a flexible composition process in which some parts of an application are selected at design and/or development times, while others are left open for opportunistic and dynamic selection at runtime. The composition process implemented by our system relies on the equivalence group and resolution concepts.

1) Equivalence group: An equivalence group (group for short) is made of one representative object (also called group head) and a number of group members. A **group head** defines the common and the configurable properties for the **group members**. In this manner, the group members have the same common properties, and the same configurable properties with own values. The configurable properties defined by a group head, allow distinguishing the group members, for example, during the group resolution process.

A group type is defined as the tuple <headType, memberType>. In our component-service metamodel, two group types exist: <specification, implementation> and <implementation, instance>. Thus, considering the example of Fig. 2, the MediaManager group has the MediaManager specification as its head and the ADELE-MediaCenter and ACME-MediaManager implementations as members.

2) Resolution process: The composition process is based on the concept of group resolution. Resolving a group means selecting or creating one or several group members (according to the specified multiplicity) satisfying a given set of constraints and preferences. A group is instantiable if it is associated with a factory, allowing then the creation of group members during the resolution process.

Resolving a specification group means either selecting an implementation from a component repository or generating an implementation satisfying the set of specified constraints.

Actually, a specification group is instantiable if it is associated with a factory knowing how to generate implementations fulfilling the given set of constraints (proxy generation for example).

In a similar way, resolving an implementation means either selecting or creating one or more instances satisfying the given set of constraints. Before runtime, an instance is a declaration corresponding to a particular implementation configuration. Like implementations, instance declarations are maintained in component development repositories. At runtime, an instance is a running entity corresponding to the execution of an implementation. Instantiation (i.e. the creation of a runtime instance) is performed using the corresponding implementation factory and the instantiation properties specified by an instance declaration. Usually, implementation groups (other than implementations of physical devices) are instantiable.

Resolving a (simple or complex) resource dependency consists in selecting a group head (i.e. a specification or implementation) providing the required resource, and then in resolving the group considering the constraints and preferences related to this group.

A group resolution is **complete** if: (1) the group head is resolved until the selected group members are not themselves group heads (e.g., the complete resolution of a specification results in one or more instances), and (2) the dependencies of the group head and of its selected members are transitively and completely resolved. A complete resolution may fail because it is not possible to resolve a group, for example, because no member satisfies the constraints or because intrinsic and contextual constraints conflict. Therefore, a complete resolution can be performed in backtrack mode, meaning that if a group resolution fails, the previous selection will be undone, and other resolutions will be performed. The backtrack resolution mode ensures that if a resolution solution exists, it will be found, but it may be very expensive for large component repositories.

The resolution (or composition) of a composite consists in resolving the unresolved dependencies of its contained implementations (starting from the main implementation). A successful resolution returns implementations that are added to the composite. In turn, resolving these implementations returns instance declarations, to be added to a composite instance declaration. Before runtime, a composite and an associated instance declaration represent the static structure of the application to be executed in terms of implementations to be deployed and instances to be created at runtime.

A composite is considered as complete (or completely resolved) if all its contained implementations have been completely resolved. Nevertheless, resolving completely a composite before runtime conflicts with the needs of using available services at runtime. Thus, we adopt a partial resolution mode leaving room for opportunistic resolution at runtime. The architect of a composite specifies then the implementations for which the resolution has to be performed before execution. At runtime, resolution is performed on-demand considering the services available on the execution platform. In this manner, composition is performed incremental and opportunistically.

As a result, our basis approach allows mixing two resolution modes:

- **static mode:** components are resolved before runtime. The implementations contained in the composite will be deployed and instantiated using their corresponding instance declarations, ensuring then a reliable composite execution.
- **opportunistic mode:** components are resolved on-demand at runtime. The services available on the execution platform can be potentially (re)used and integrated into the composite execution.

Other resolution modes (such as dynamic or remote) can be specified by independent models (referred as specific-concern models) complementing thus a composite definition with specific-concerns to be supported. For example, a **dynamic model** can specify the behavior that a composite must have when the availability of some components changes at runtime. Thus, a composite needing to react to the availability of components in the execution platform must be associated with a dynamic model describing the expected behavior (e.g. dynamic creation of bindings following the availability of an expected service, dynamic substitution of an unavailable service, and so on). A **deployment model** can complement a composite definition by specifying a set of addresses corresponding to component repositories to be used for the resolution of that composite.

The specific-concern models associated with a composite definition are controlled, at runtime, by concern-specific managers which are in charge of resolving and/or controlling the composite execution according to a specific-concern. This paper does not detail the specific-concern models that can be associated to composites nor the managers that interpret them at runtime.

D. Composite control properties

A composite contains a (partial or complete) set of implementations. In our approach, the result of the incremental composition process is, by default, a **white-box** composite. A white-box composite allows other components and composites to get access to its content. Our approach allows specifying if the content of a composite is not visible from the outside. Thus, in order to hide the content of a composite, we define a black-box property. A **black-box** composite is seen like a primitive implementation, meaning that the only way to interact with the composite is through its provided services or resources. The internal structure of such a composite (i.e. the implementations it contains) will not be visible from an external point of view.

Moreover, in order to control its runtime resolution, a composite can explicitly specify if its resolution space is **closed** (i.e. its resolution will be performed using only the specified component repositories). By default, the resolution space is **open** (i.e. the resolution is performed both opportunistically and using all the available repositories). The closed and black-box properties can be associated with composites or with composite definitions, like in the following example:

```
CompositeDefinition Closed BlackBox HomeMediaCenter {
   ...
}
```

III. RUNTIME ENVIRONMENT

Our model-driven runtime environment supports and controls the concurrent execution of various composites defined and built with our design and development environments. It implements the component-service metamodel presented in section A.

Before runtime, component-services are represented by artifacts that can be described, developed and packaged (as bundles). In order to **support the execution** of composites, our runtime environment supports the deployment, instantiation and activation of components and (partial or complete) composites. It promotes the opportunistic (re)use of the available running services. Hence, by default, the resolution of dependencies is performed on-demand, resulting in services, selected from the available running services or deployed and activated from the available component repositories.

Executing a (partial or complete) composite on our running environment consists thus in deploying the composite and its main implementation, and creating their corresponding instances. The composite is incrementally composed (starting from its main instance), via the on-demand resolution of the components dependencies.

Our runtime environment provides mechanisms for **controlling the concurrent execution** of composites according to their associated properties (i.e. intrinsic and contextual constraints and preferences, visibility, sharing, open or closed resolution). Hence, it represents the current execution state of the supported composites via a **state model** conformable to our component-service metamodel. That runtime model is causally related to the underlying platform, allowing on the one hand having a representation of the current running components and composites (i.e. a descriptive model) and, on the other hand, managing and controlling their execution according to their associated properties (i.e. a prescriptive model). Then, the actions performed in that model (for example, adding a given implementation) are translated into the corresponding actions in the underlying platform (deploying and activating the corresponding implementation).

Therefore, our runtime environment ensures that the execution of a composite is conformable to its definition, and consistent with the composition performed before runtime. The information produced during its design and development times (i.e. the composite model) is known and managed at runtime. The resolution of its internal and external dependencies is then performed according to its definition, and using (by default) the same component repository used at development-time. Moreover, having the information produced during design and development times allows performing some design and development engineering activities at runtime.

Considering that additional models representing specific functional and non-functional concerns can be defined to complement a composite model, our runtime environment can be extended in order to support and control such concerns during the composite execution. Currently, dynamic adaptation and deployment are supported.

The basic functionality of our runtime environment, extended with specific-concern functionalities, allows supporting and controlling the concurrent execution of various composites ensuring, on the one hand, the satisfaction of their definition (i.e. of their goals) and of their general properties (i.e. visibility, sharing), and on the other hand, the management of specific-concerns (such as dynamic adaptation and deployment).

Our runtime environment, built on top of the OSGi framework [7], uses the iPOJO [8] and Rose [9] technologies, resulting thus in an extensible runtime environment handling the execution and composition of dynamic, distributed and heterogeneous component-services.

We have tested different scenarios and compared the performance of our core runtime environment [10] to those of iPOJO [8], FraSCAti [11] and Tuscany [12] platforms. We have measured the instantiation and method call rates, and the memory used in each case. The results have shown that our runtime environment is quite efficient. For instance, regarding the instantiation rate, our platform is 10% faster than iPOJO, even though the memory usage is 10% higher. Our current work aims at improving the robustness and performance of our runtime environment, and validating it by using different application scenarios requiring controlled opportunistic and dynamic behaviors. We intend to provide a runtime environment robust and efficient enough to be used as an open research platform.

IV. RELATED WORK

The development, execution and management of service-based applications have been addressed from different perspectives. Some works propose model-based approaches for their development. Others propose runtime platforms for supporting their execution. Recent works attempt to extend the use of models created during development to the runtime.

Using components to implement services has become relatively popular. Some service-oriented component models and execution platforms have been proposed, such as Service Component Architecture (SCA) [13], Declarative Services [7], Spring-DM [14] and iPOJO [8].

SCA proposes a service-oriented component and composition model. It supports a wide variety of technologies for implementing components. SCA allows describing composites as an assembly of interconnected components. Like in our approach, composites can be used as component implementations, allowing hierarchical construction of applications. Unlike our approach, composition is defined at development-time and wiring is fully performed when starting the application, prohibiting component selection and substitution at runtime. Some runtime platforms, such as FraSCAti [11] and Tuscany [12], implement and extend the SCA specification allowing the execution of SCA applications at runtime.

The OSGi specification [7] defines a service runtime platform supporting a minimal component model. OSGi automatically manages dynamic service deployment, including Java package dependency resolution. Nonetheless, service dependency management is left as a manual task for developers. Therefore, the Declarative Services specification,

inspired by the Service Binder model [15], proposes a service-oriented component model aiming at facilitating the creation of components on top of OSGi. It allows developers to describe components with its provided and required services. Descriptions are used by the Service Component Runtime (SCR) to automatically manage service dependencies at runtime. Nevertheless, unlike our approach, Declarative Services does not propose a composition model, leaving developers to code service compositions.

iPOJO is both a service-oriented component model and a runtime platform implemented on top of OSGi. Based on the concept of Plain Old Java Objects (POJO), iPOJO allows a straightforward development of components. POJOs are encapsulated in containers that manage dynamic service mechanisms (publishing, discovery, invocation) and other non-functional concerns (such as lifecycle or configuration) at runtime. iPOJO offers an extensibility mechanism based on the concept of handlers which are plugged into containers allowing managing other non-functional concerns. Indeed, our runtime environment, implemented on top of iPOJO, provides new functionalities, implemented via handlers, like on-demand dependencies resolution, intrinsic and contextual properties management, and so on.

Recent works in dynamic adaptive systems and autonomic systems [16][17][18] advocate the use of runtime models (or state models) causally connected to the running systems. Runtime models provide abstract views of the running systems allowing their management (monitoring, analysis, adaptation). Thus, actions performed on a runtime model are transformed into the corresponding actions on the running system, and vice versa. In some works, the runtime models focus on reactivity to dynamic context changes [8] but they do not propose proactive (goal-oriented) actions. Other works focus on proactivity [16][17] but they usually do not focus on dynamic services. Our work claims that reactive and proactive actions should be both supported in order to allow dynamic adaptation and evolution of the running systems in controlled ways. Hence, our approach uses multiple runtime models simultaneously, including development models. Using development models at runtime allows us to support continuous design, blurring the line between development-time and runtime [19].

V. CONCLUSIONS

In this paper we have presented a model-driven approach to design, develop, execute and manage service-based applications, such as dynamic and ubiquitous ones. Our contribution lies in an abstract, flexible and automated composition process that promotes opportunistic and dynamic behaviors at runtime. We propose first a component metamodel that brings to service-based architectures the advantages of component-based development (such as description, composability and encapsulation).

Second, we propose design and development environments allowing developing components and composites defined at a high-level of abstraction. Our composition environment supports adding specific functional or non-functional concerns (to be defined in independent models) in order to complete the definition of an application.

Finally, we propose a runtime platform supporting and managing the concurrent execution of multiple dynamic applications, ensuring the satisfaction of their specified goals. Our runtime platform can be extended by concern-specific managers in order to support specific-concern management during execution. Our approach uses models from design to runtime. Extending the use of design and development models to runtime, allows us to control and ensure a consistent execution of the running applications, and also to perform some design and development activities if needed. This work is currently extended for the design and execution of autonomic applications.

REFERENCES

[1] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 3, no. 3, 1991.

[2] M. Satyanarayanan, "Pervasive computing: vision and challenges," *IEEE Personal Communications*, vol. 8, no. 4, pp. 10-17, 2001.

[3] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *Future of Software Engineering (FOSE)*, 2007, pp. 37-54.

[4] J. Estublier, G. Vega, P. Lalanda, and T. Leveque, "Domain Specific Engineering Environments," in *Asia-Pacific Software Engineering Conference (APSEC)*, 2008.

[5] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, vol. 0201703327, no. 4. Addison-Wesley, 2001.

[6] J. Estublier, I. A. Dieng, E. Simon, and G. Vega, "Flexible Composites and Automatic Component Selection for Service-Based Applications," in *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2009.

[7] The OSGi Alliance, "OSGi Service Platform Core Specification," 2011. [Online]. Available: http://www.osgi.org/.

[8] C. Escoffier, R. S. Hall, and P. Lalanda, "iPOJO: an Extensible Service-Oriented Component Framework," in *International Conference on Service Computing (SCC)*, 2007, pp. 474-481.

[9] J. Bardin, P. Lalanda, and C. Escoffier, "Towards an Automatic Integration of Heterogeneous Services and Devices," in *IEEE Asia-Pacific Services Computing Conference (APSCC)*, 2010, pp. 171-178.

[10] J. Estublier and G. Vega, "Managing Mutiple Applications in a Service Platform," in *Principles of Engineering Service Oriented Systems (PESOS)*, 2012.

[11] OW2 Consortium, "FraSCAti." [Online]. Available: http://wiki.ow2.org/frascati.

[12] Apache, "Tuscany." [Online]. Available: http://tuscany.apache.org/.

[13] M. Beisiegel, A. Miller, J. Marino, and L. Waterman, "SCA Service Component Architecture," *International Business*, 2007.

[14] A. M. Coyler, H. Hildebrand, C. Leau, and A. Piper, "Spring Dynamic Modules Reference Guide," 2009.

[15] H. Cervantes and R. S. Hall, "Automating service dependency management in a service-oriented component model," in *ICSE CBSE Workshop*, 2003.

[16] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, "Using Architecture Models for Runtime Adaptability," *IEEE Softw.*, vol. 23, no. 2, pp. 62-70, 2006.

[17] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes," *Computer*, vol. 42, no. 10, pp. 37-43, 2009.

[18] G. Blair, N. Bencomo, and R. B. France, "Models@run.time," *Computer*, vol. 42, no. 10, pp. 22-27, 2009.

[19] L. Baresi and C. Ghezzi, "The disappearing boundary between development-time and run-time," in *Proceedings of the FSE/SDP workshop on Future of Software Engineering Research (FoSER)*, 2010, pp. 17-21.