Strengthening Component Architectures by Modeling Fine-grained Entities

Tomáš Bureš*†, Pavel Ježek*, Michal Malohlava*, Tomáš Poch*, Ondřej Šerý*
*Charles University in Prague, Faculty of Mathematics and Physics
Malostranské náměstí 25, 118 00 Prague 1, Czech Republic
†Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2, 182 07 Prague 8, Czech Republic
{bures,jezek,malohlava,poch,sery}@d3s.mff.cuni.cz

***Abstract*—Component-based software engineering (CBSE) defines components as basic software building blocks with strongly formalized behavior and interactions. The key benefits of structuring code into components include good analyzability of performance and behavioral correctness, simpler code generation, and high documentation value. However, a sufficiently detailed formalization including all relevant parts of application behavior often requires finer granularity than of a software component – a typical example is component's data exposed to other components that can circulate through the application, e.g., opened files, client sessions. In order to propagate all the mentioned benefits of CBSE to this level of granularity, we propose a conservative component model extension which allows to capture those concepts on the architecture level. Our main goal is to define a model allowing seamless integration in existing behavior specification formalisms and implementation in current component systems.**

***Keywords*-Component model, architecture evolution, dynamic reconfiguration.**

I. Introduction

A software component is often viewed as a black-box, which is essential for many desired features such as separation of concerns, reuse, support of product lines, etc. In the *black-box view*, all interactions of a component with its environment occur only through its well-defined interfaces and any assumptions of the component regarding its admissible environment are made explicit, e.g., using a formal description of the component behavior or by specifying method contracts. This allows verifying that communicating components obey their mutual contract (in terms of method calls ordering, parameter/return value ranges, etc.).

When considering the formal behavior description of a black-box component, the behavior model must be associated with some concept in the black-box view, which is either a particular interface or a component as a whole and refer to the behavior events observable at the architectural level—method calls. However, components often operate with more fine-grained concepts (e.g., instances of different objects passed as method arguments), which are not reflected at the architectural level.

As an example, consider the `FileManager` component from the architecture depicted in Fig. 1. The `FileManager` provides a concept of a file to its environment—namely the

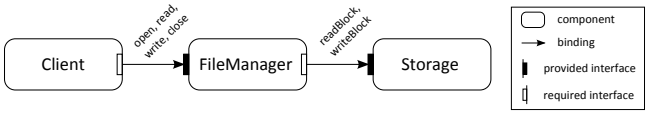

Figure 1. Solution A: An example architecture featuring a FileManager component

`Client` component. The files can be manipulated by calling the `open`, `read`, `write`, and `close` methods on the single provided interface of the `FileManager`.

When modeling admissible behavior on the `FileManager` interfaces, it is desirable to specify that a particular file has to be first opened, then it can be read from or written to a number of times, and finally it should be closed. Standard approaches to modeling behavior of components (e.g., [1]–[4]) consider only ordering of method calls on the component interfaces. Unfortunately, there are only restricted ways for relating method calls with the actual methods' parameters. Therefore, the resulting specification cannot express the required ordering of method calls on `FileManager` related to the different files. Without this dependency, one can only say that all the four methods can be called in any order, which is an over-approximation of the admissible behavior and not a particularly useful one. Significant aspect of the problem is that the objects (files) are not captured at the architectural level. Therefore, they can not be addressed by the methods working on that level. In the rest of the text, we will denote the objects important for the cooperation of components that are created dynamically during the runtime as *entities*.

Our goal is to provide means for capturing entities at the architectural level. In particular, we propose a conservative extension of typical component models without inherent support for modeling dynamism. The extension should include a minimal set of concepts necessary to capture entities with emphasis on (i) practical implementability in a component framework, and (ii) application of techniques well established in component systems. Apart from analysis of behavior compatibility among components and performance analysis, the entities can also increase documentation value of the architecture.

II. Capturing Dynamic Entities in Architecture

In this section, different approaches to modeling entities are considered. In particular, we use means of a typical component model to capture a static snapshot of a system involving entities—dynamic aspects are not modeled at this stage. The model examples serve as a basis for our proposal. Here, we first consider what existing concepts can represent entities. Later in Section III, we propose extensions of these concepts to capture the inherent dynamism of entities in a static architecture.

For the sake of completeness, we denote the model from Fig. 1 as *Solution A* (does not reflect entities at all).

A. Solution B: Entities as Separate Components

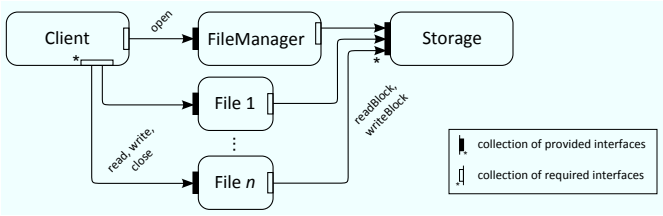

Figure 2. Solution B: Entities modeled as component instances

Let us consider a case where files are modeled as separate components. Such architecture is depicted in Fig. 2. In this case, the `FileManager` provides just the `open` method which results in instantiation of a new `File` component which then provides other methods related to individual files (`read`/`write`/`close`). When `close` is invoked, the particular `File` component instance is destroyed. In this case, individual files can be equipped with behavior description, performance information, and various quality attributes needed for different kinds of analysis. Apparently, since the application is evolving in time, the architecture in Fig. 2 is just a snapshot taken in a certain moment. For instance, if the figure was capturing the structure at the beginning of the computation, there would be no `File` component present.

B. Solution C: Entities as Separate Interfaces

Figure 3. Solution C: Entities modeled as interfaces

In some cases, using a component to model every opened file may be a too fine grained approach. The information kept for each file is not that large. In other situations, there might be also complex relations among individual entities and sub-components which should not be exposed to the rest of the architecture.

Figure 3 depicts a compromise solution where the files are modeled as separate interfaces in the architecture. The `FileManager` component provides an interface containing the `open` method. When it is invoked, a new interface representing the file is instantiated at the boundary of `FileManager` (the opened file itself will be represented by an internal component or object inside `FileManager`). In this case, the additional information can be associated with the interface representing the file. Notice that from the `Client` point of view the situation is the same as in Fig. 2. Similarly to the previous solution, the architecture evolves at runtime and the figure is just a snapshot. Although the number of components remains the same (in the black-box view of the `FileManager` component), interfaces and bindings are being created and destroyed at runtime.

III. Runtime vs. Design Architecture

We will use the concept of components and interfaces provided by legacy component models in two different roles. First, to model the structure of the application as usual—to capture the application in terms of functional blocks (components) and communication among them (interfaces). Second, we will use the concept to model entities and related communication.

First, let us distinguish *design architecture* from *runtime architecture*. The runtime architecture reflects the structure of the application (including entities) in a particular instant of time during the computation. Some of components and interfaces model entities, but since it is just a snapshot no dynamism is considered and there is no need to distinguish them from other components. The information in the runtime architecture is the same as in legacy component models – static components featuring interfaces connected by static bindings and entities are formally indistinguishable from other concepts. On the other hand, the goal of the design architecture is to cover all configurations the application structure can reach in terms of number of existing entities, their location and established communication links. The design architecture can be also thought of as a template for runtime architectures.

When the dynamism of the file manager scenario is considered, certain parts of the application remain the same during the computation. Those parts are captured in the design architecture using components, interfaces, and bindings as usual. However, some components are capable to instantiate new interfaces in a collection of interfaces[1], which are bound together by new bindings (e.g., a new file was opened). Those interfaces and bindings can also disappear later (e.g., a file was closed). Nevertheless, it is always known in advance what components are able to instantiate new interfaces, what components are supposed to communicate with each other,

[1] *Collection of interfaces* is a group of interfaces of the same type. The number is varying during the computation. Such concept is already present in some component models [5], [6]

and what components represent dynamically created entities. What is not known in advance and what is being under permanent change during the computation is the number of interfaces, bindings, and components. Thus, instead of capturing the exact numbers, we propose to capture in the design architecture the information known in advance using a new concept of *proto-bindings* as described bellow:

Proto-binding defines a location in the architecture where the bindings can be established. The proto-binding has two end-points and serves as a template of regular bindings that can be established between the end-points at runtime. The proto-binding end-point can be either a single interface or a collection of interfaces.

The concept of proto-binding allows to enrich a design architecture by information about dynamically created bindings and interfaces. The proposed degree of dynamism is designed with entities on mind.

Let us reconsider the Solution C (Fig. 3), modeled using the proposed concepts. In Fig. 4, the design architecture of Solution C is depicted. While the interface for opening files of `FileManager` is connected to `Client` by a binding, interfaces for individual files and related bindings are not captured since their number varies during the computation. However, the corresponding collection interfaces are connected by the proto-binding to identify places where dynamically created bindings can be instantiated.

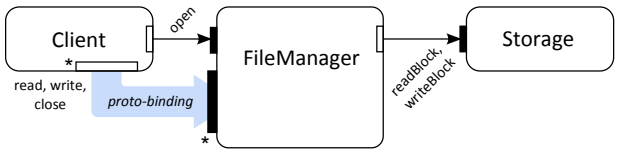

Figure 4. FileManager example – design architecture

The runtime architecture that captures the structure of Solution C at a certain moment at runtime is actually the Fig. 3. Currently, there are two files provided by `FileManager` to `Client`. Bindings between interfaces representing individual files are established with respect to the proto-binding from the design architecture.

The design architecture in Fig. 4 does not state whether the `FileManager` component is primitive or composite. If the component is primitive, the implementation of the `open` method just creates a new interface instance which is consequently bound to the interface of `Client` (illustrated by an example in the introduction of Sect. IV). On the other hand, if the component is implemented by composition of other components, dynamic entities can be represented by separate components.

IV. Solution

The goal of this section is to define mechanisms controlling creation of bindings templated by proto-bindings by means of four reconfiguration actions. In particular, the reconfiguration actions are defined as part of the design architecture and constrain where, when, and which architectural changes can happen at runtime.

To achieve correspondence with the architectural level, the reconfiguration actions should be defined in a way that they can be triggered by the events visible to the component system. For a typical component system, method calls are a common observable event, therefore we allow any architectural changes defined by reconfiguration actions to be triggered only as a reaction to a method call among components. All reconfiguration actions are associated with an *entity reference* argument or return value of an interface method and have another target interface or collection of interfaces specified (the target interface or collection are expected to be one end of a proto-binding). The proposed actions are:

(a) *link*, creating a new binding templated by proto-binding leading from the target interface and associating it with the entity reference,
(b) *unlink*, destroying the binding associated with the entity reference,
(c) *create*, publishing/associating the associated entity reference via/with the target interface,
(d) *delete*, removing the association of the entity reference with the target interface, making the entity reference unavailable.

If the target of a reconfiguration action is a collection of interfaces, the *link* and *create* actions will create a new interface instance in the collection and the entity reference is associated with it, the *unlink* and *destroy* actions will remove and destroy the interface instance. Further details on the entity references and reconfiguration actions considering the reference life-cycle are discussed in [7].

Fig. 5 presents a basic example illustrating behavior of reconfiguration actions[2]. The `Client` component is allocating multiple entities (i.e., opening multiple files) from the `FileManager` server component (this follows the motivation example presented in Sect. II). Each time the `Client` calls the `open` method of the `IA` interface it will receive a reference to a new entity (a newly opened file). The `open` call raises the following actions: (1) the *create* reconfiguration action on the return value of `FileManager` provided `open` method ensures a new interface is allocated in the collection of provided interfaces `IB` and it is associated with the returned entity reference, (2) the *link* reconfiguration action on the return value of `Client` required `open` method ensures a new interface is allocated in the collection of required interfaces `IB` and a new binding templated by the proto-binding is created. The `Client` component can then interact with the entity by calling the `use` method repeatedly

[2]In all figures the reconfiguration actions are marked by a `@` prefix to enhance readability, actions associated with an argument or return value are added before that argument or return value, actions associated with a method are added after the method declaration.

via the interface IB instance associated with the acquired entity reference. When the Client decides to stop the interaction, it will call the close method. The associated *unlink* reconfiguration action implies the close is the last call on this binding and will destroy the binding and remove the instance of the required interface at the end of the close call. Similarly, the *destroy* reconfiguration action ensures the instance of provided interface IB is removed from the FileManager collection.

Furthermore, the presented concept of dynamically created bindings over proto-bindings controlled by reconfiguration actions can be naturally extended to a concept of dynamically instantiated components. As a complement of the original quadruple, two more reconfiguration actions are proposed in [7].

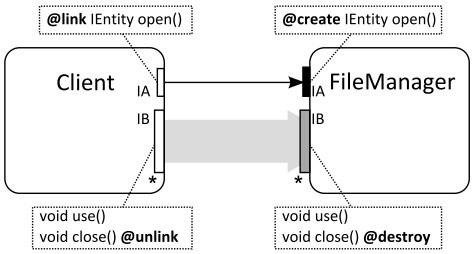

Figure 5. Example – Client/FileManager

A. Examples of Basic Reconfiguration Actions

In this section, we present other typical examples of using different reconfiguration actions at the level of design architecture to form a description of the allowed architectural evolution. The first example was presented in the introduction of Sect. IV.

The second example (Fig. 6) shows the importance of timing of architectural change denoted by each reconfiguration action. The example represents a common pattern of passing a callback reference – the entity reference is passed in opposite direction than in the first example. The binding to Client's entity (the callback) will be established at the beginning of the performWith method call, allowing the Worker to call back to Client during the call. At the end of the originating call the binding is destroyed again – so that the Worker is not allowed to call the Client out of the specific window provided by Client via the performWith call.

The third example (Fig. 7) shows the behavior of reconfiguration actions in a context of nested components (if we no longer look at the Client from Fig. 5 as on a black box) and also illustrates component ability to just pass the received entity references without a need to create a binding to the originating component. An important note is that making the Client a composite component does not change the reconfiguration actions on its frame, but its behavior is refined by the reconfiguration actions associated with its subcomponents. The Security component opens the entities on Server, but does not use them directly (only passes them to the Worker component). This way, no reconfiguration actions are needed on any of its methods. On the other hand, the Worker component needs a binding to interact with Server entities, so the *link* action is required on the method argument where it receives the entity reference – i.e., on the performWith method. Similarly to the first example of the whole Client frame, the *unlink* action is required on the close method. Worth noting is also the inherent mechanism of partial building of bindings from Worker component back to the Server component – the first half (between Client and Server) will be created at the end of the open call, but the second half (between Worker and Client frame) won't be created until the beginning of the performWith call. Should the Server component be also composite, the same partial building would occur inside it as the open method call returns up through Server hierarchy.

V. Related Work

Presently, dynamic architectures are quite well explored and several approaches of modeling them exist. The surveys of dynamic software architectures [8], [9] present a categorization of different architecture evolution styles based on graphs [10], [11], process algebras [3], [12], [13], UML profiles [14], [15], or various logic [16].

In contrast with our proposal, all described approaches applicable in the domain of component-based systems focus on capturing general architecture reconfigurations which manipulate with coarse-grained architecture artifacts like components, bindings, interfaces, and connectors. However, we focus on describing dynamicity of more fine-grained parts of component-based applications resulting from implementation demands.

Certain contemporary component models also provide a capability of modeling architecture evolution (ArchJava [17], ACME [18], Plastik [19]) or at least have some support of architecture modification at runtime (Fractal [20]). However, they do not provide smaller granularity of modeling than a component.

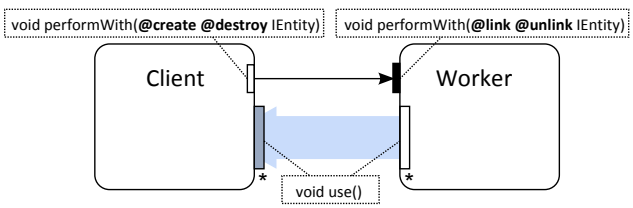

Figure 6. Example – Callback

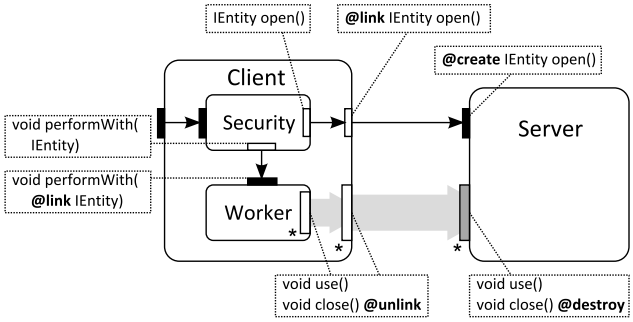

Figure 7. Example – Passing a reference through a composite component frame

VI. Conclusion

In this paper, we have presented a way of capturing dynamic entities (e.g., files, database handles, and session objects) in a component model. Our approach allows capturing the dynamism in the initial design architecture using the proposed concepts of *proto-bindings* and *proto-components*, which explicitly document possible future bindings and component instances. Further, we have introduced six reconfiguration actions, which describe at the design level when and how the runtime architecture evolves. This way, we lay down basis for more precise modeling and code generation, as we have also pointed out in this paper. The proposed approach works seamlessly both for flat and hierarchical component models. As for the future work, we plan to focus on adjusting our formalism for behavior modeling to reflect the proposed approach and also to focus more on a general way of component instantiation and initialization.

Acknowledgement

The work was partially supported by the grant SVV-2011-263312 and by the Grant Agency of the Czech Republic project P202/11/0312.

References

[1] L. de Alfaro and T. A. Henzinger, "Interface automata," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, 2001.

[2] L. Brim, I. Cerna, P. Varekova, and B. Zimmerova, "Component-interaction automata as a verification-oriented component-based system specification," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 2, p. 4, 2006.

[3] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Trans. on Softw. Eng. and Methodology*, vol. 6, no. 3, pp. 213–249, 1997.

[4] J. Adamek and F. Plasil, "Component composition errors and update atomicity: Static analysis," *Journal of Softw. Maintenance and Evolution*, vol. 17, no. 5, 2005.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "An open component model and its support in Java," in *Proceedings of CBSE'04*, 2004.

[6] T. Bures, P. Hnetynka, and F. Plasil, "SOFA 2.0: Balancing advanced features in a hierarchical component model," in *Proceedings of SERA'06*. IEEE Computer Society, 2006.

[7] T. Bures, P. Jezek, M. Malohlava, T. Poch, and O. Sery, "Fine-grained entities in component architectures," Dep. of Softw. Engineering, Charles University in Prague, Tech. Rep., June 2009. [Online]. Available: http://d3s.mff.cuni.cz/publications/2009-entities-report.pdf

[8] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A survey of self-management in dynamic software architecture specifications," in *Proceedings of WOSS '04*. New York, NY, USA: ACM, 2004, pp. 28–33.

[9] A. Bucchiarone, "Dynamic software architectures for global computing systems," Ph.D. dissertation, IMT Institute for Advanced Studies, Lucca, 2008.

[10] M. Wermelinger, A. Lopes, and J. L. Fiadeiro, "A graph based architectural (re)configuration language," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 21–32, 2001.

[11] G. Berry and G. Boudol, "The chemical abstract machine," in *Proceedings of POPL '90*. New York, NY, USA: ACM, 1990, pp. 81–94.

[12] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *Proceedings of ESEC'95*, vol. 989. Springer-Verlag, Berlin, 1995.

[13] C. Canal, E. Pimentel, and J. M. Troya, "Specification and refinement of dynamic software architectures," in *Proceedings of WICSA'99*. Deventer, The Netherlands, The Netherlands: Kluwer, B.V., 1999, pp. 107–126.

[14] M. H. Kacem, A. H. Kacem, M. Jmaiel, and K. Drira, "Describing dynamic software architectures using an extended uml model," in *Proceedings of SAC '06*. New York, NY, USA: ACM, 2006, pp. 1245–1249.

[15] D. Ayed and Y. Berbers, "UML profile for the design of a platform-independent context-aware applications," in *Proceedings of MODDM'06*. New York, NY, USA: ACM, 2006.

[16] M. Endler and J. Wei, "Programming generic dynamic reconfigurations for distributed applications," in *Configurable Distributed Systems, 1992., International Workshop on*, Mar 1992, pp. 68–79.

[17] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting software architecture to implementation," in *Proceedings of ICSE'02*. ACM Press, 2002, pp. 187–196.

[18] D. Garlan, R. T. Monroe, and D. Wile, "ACME: Architectural description of component-based systems," in *Foundations of Component-Based Systems*. New York, NY: Cambridge University Press, 2000, pp. 47–67.

[19] T. Batista, A. Joolia, and G. Coulson, "Managing dynamic reconfiguration in component-based systems," in *Proceedings of EWSA'05*. Springer, 2005, pp. 1–17.

[20] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The Fractal component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems," *Softw. Pract. Exper.*, vol. 36, no. 11-12, 2006.