Fine-grained Entities in Component Architectures

Tomáš Bureš[1,2] Pavel Ježek[1] Michal Malohlava[1] Tomáš Poch[1]
Ondřej Šerý[1]

[1] *Distributed Systems Research Group*
Charles University in Prague
Malostranské náměstí 25, 118 00 Prague 1, Czech Republic
{bures, jezek, malohlava, poch, sery}@dsrg.mff.cuni.cz

[2] *Institute of Computer Science*
Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2, 182 07 Prague 8, Czech Republic

Abstract

Component-based software engineering (CBSE) defines components as basic software building blocks with relatively 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 own data exposed to other components, e.g., opened files, client sessions. Moreover, we show these concepts should be captured at the architecture level in order to keep all the mentioned benefits of CBSE. In this respect, we propose a component model extension. 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

1 Introduction

Component based software engineering is a methodology, which allows building software of well defined blocks called components. A component explicitly defines its interaction points in the form of provided and required interfaces. The interaction among components is captured by so called component architecture which defines bindings (i.e., communication channels) between components required and provided interfaces. Such formalization brings important benefits in terms of documentation, analyzability, and code generation.

A component is often viewed as a black-box, which is essential for many desired features such as separation of concerns, 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. However, in existing approaches, this brings problems with granularity. In practice, components often provide concepts, which have smaller granularity than a single interface and substantially influence the component behavior.

As explained in the following text, these concepts have often a dynamic nature (i.e., they appear and disappear during runtime) and when not properly captured, they disallow accurate behavior specification. In the following text, we will call such concepts *entities* to emphasize their different granularity and the inherent dynamism.

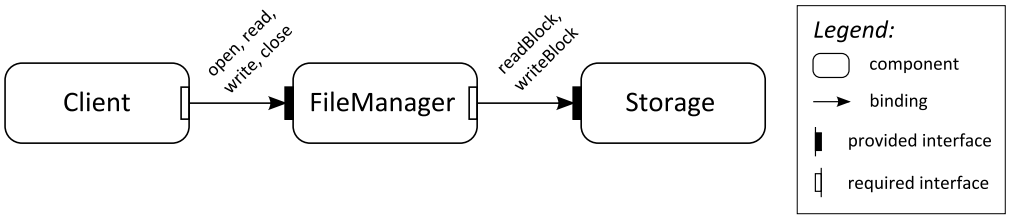

Fig. 1. An example architecture featuring a FileManager component

As an example, consider the FileManager component from the architecture depicted in Fig. 1. The FileManager provides a concept of a file (the file entity) to its environment (the Client component). The files can be manipulated by calling `open`, `read`, `write`, and `close` methods on the single provided interface of the FileManager. Inside the FileManager implementation, a file would probably be represented by a corresponding object or a structure, while in the provided methods an identification/encoding using some kind of an integer handle would be used.

When modeling admissible behavior on the FileManager interfaces, it is desirable to specify that a 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. However, standard approaches to modeling behavior of components (e.g., [18,11,3,1]) consider only method calls on the component interfaces ignoring the file entities (method parameters). The resulting specification cannot express ordering of method calls related to different files. Thus, one can only say that all the four methods can be called in any order, which is an over-specification of the admissible behavior and not a particularly useful one.

A straightforward solution to this problem is to extend the behavior modeling formalism by the notion of entities. In the example above, this would mean to reflect integer values representing the file handles in the behavior model. In some cases, this solution might be acceptable, however, it leads to a more complex specification of the interface behavior and brings problems with the encoding of the file handles. More important, enhancing the behavior modeling formalism by a concept not existing on the architectural level loosens the relation between the two. In contrast, we

believe that the entities should be explicitly represented at the architectural level. Secondary, this common representation should be understood and referred to by the behavior modeling formalisms.

1.1 Goals

The precision of behavior models and a consecutive correctness analysis would benefit from properly capturing behavior of dynamic entities. However, the benefits are not tied only to behavior modeling. In the context of the file entities example, the performance prediction analysis may estimate system performance based on the number of opened files. Documentation value of the architecture also increases by capturing the dynamic file entities, which the developers should be aware of anyway. Last but not least, capturing the entities in the architecture offers additional opportunities for code generation, e.g., the translation into handles and proxy generation might be done automatically as well as for example control of access rights to the entities. For all of this, capturing the entities uniformly at the architectural level is necessary.

Existing approaches, however, do not provide means to cope with the dynamic entities. The goal of this paper is thus to propose a set of patterns for capturing dynamic entities in the architecture with emphasis on (i) practical implementability in a component framework, and (ii) use of the additional information in the various analysis tasks, and especially analysis of communication correctness among components.

2 Capturing Dynamic Entities in Architecture

In this section, different approaches to modeling dynamic entities are considered. For the sake of completeness, we denote the model from Fig. 1 as *Solution A*. However, as already mentioned in Sect. 1, this solution does not reflect entities. To capture an entity in the architecture, the entity must be either mapped to an existing abstraction of the component model or a new abstraction must be introduced. To keep the number of component model abstractions minimal we consider rather extending the existing ones to suit our needs than introducing new abstractions.

2.1 Solution B: Files as Separate Components

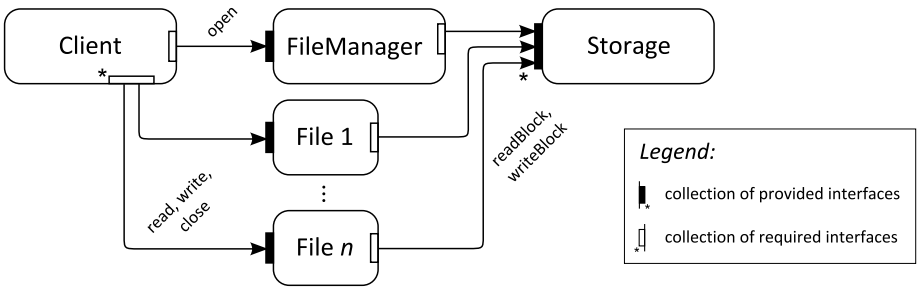

Fig. 2. Entities modeled as component instances

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 that 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.

2.2 Solution C: Files as Separate Interfaces

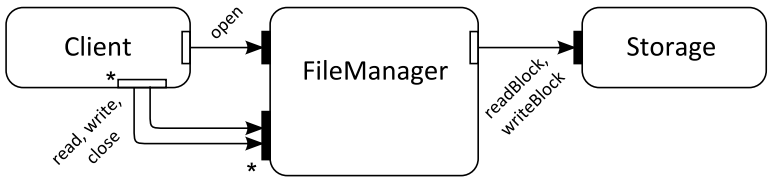

Fig. 3. Entities modeled as interfaces

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.

2.3 Requirements

All the discussed solutions have their pros and cons. When the information captured in Solution A is sufficient for a developer, it can be used. If it does not suffice, then Solution B and Solution C are preferred, depending on the required granularity. However, the main obstacle to their practical usage is insufficient support in current component models. From the solutions presented, it is apparent that the existing component abstractions are sufficient for capturing snapshots of the architecture in-

volving entities. To go further and describe also the way the entities evolve, certain kind of dynamism is required. To gain as much as possible from modeling entities, the component model must provide a consistent notion of dynamic entities from all points of view – describing evolution (creation of new entities, relations among them), describing behavior of individual components referencing individual entities, and also, the entities should be identifiable in the component model runtime to allow monitoring, profiling, etc. As the new entities (either modeled by interfaces or components) do not radically change the architecture and it is always known in advance where the new interfaces and components appear, the solution does not have to be too general and cover any architectural change. Although, some component models [12,26] do support quite general notion of dynamism, they consider only a part of the problem – e.g., evolution of architecture but not description of communication over bindings reflecting the evolution [26], the theoretical models of architecture evolution [7,29] are too complex to be easily captured by component systems implementation.

To solve this issue, we propose an extension of current component model concepts. The extension should satisfy the following requirements:

R1 To be strong enough to support Solution B and Solution C. In particular, the extension must support dynamic component instantiation, dynamic interface creation and dynamic bindings.

R3 To keep the architecture analyzable. The extension must be minimal in terms of number of new concepts, their complexity, and expressive power. The new concepts must be also consistent throughout the hierarchy of components to allow a compositional analysis.

R3 To keep the documentation role of the architecture. The architecture must capture all configurations achievable by the application structure at runtime. The information should be separated from implementation details.

R4 To preserve established concepts of component models.

3 Basic Concepts of the Proposal

In order to capture dynamic entities, a *design architecture* should be distinguished from a *runtime architecture*. The runtime architecture reflects the structure of the application in the particular instant of time during the computation, therefore there is no dynamism to be captured. The information in the runtime architecture is the same as in legacy component models – static components featuring interfaces connected by static bindings. On the other hand, the goal of the design architecture is to cover all configurations the application structure can reach during the computation. 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, 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 new concepts of *proto-bindings* and *proto-components* 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.

Proto-component represents a template of a component which may appear in the runtime architecture at the certain place. The proto-component is connected to interfaces of other components.

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 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.

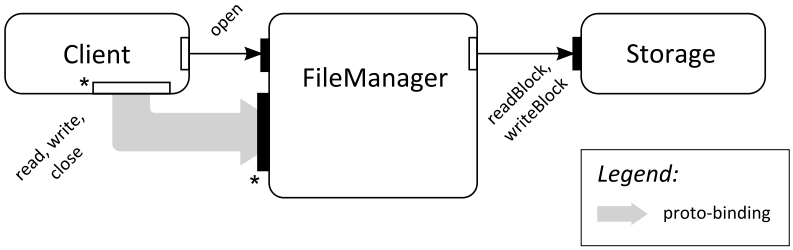

Fig. 4. FileManager example – design architecture

The runtime architecture in Fig. 5 captures the structure of Solution C at a certain moment at runtime. 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. 4). On the other hand, if the component is implemented by composition of other components, dynamic entities can be represented by separate components. In such case, proto-components are used (illustrated by an example in Sect. 4.4).

[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 [12,15]

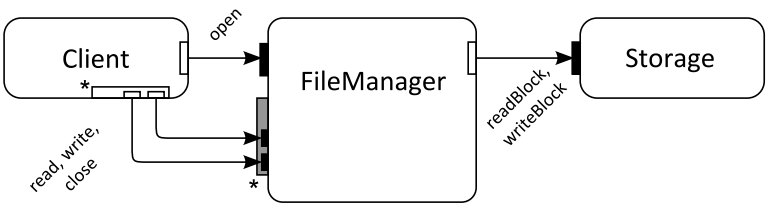

Fig. 5. FileManager example – runtime architecture

ss

4 Solution

In Sect. 3 the concept of proto-bindings was introduced. The goal of this section is to define mechanisms controlling creation of bindings templated by proto-bindings. The proposed mechanisms are specially designed to support the concept of entities as introduced in Sect. 2 and fulfill all the requirements R1 to R4 established in Sect. 2.3. Our proposal is structured into two parts:

(i) To meet the requirements R2 and R3, we propose that bindings templated by proto-bindings should not be allowed to be created or destroyed arbitrarily. Instead, these architectural changes should be triggered only by specific data passed among components. With respect to the overall goal, only data identifying entities have to be considered – in the following text we call these data the *entity references*.

(ii) To capture the relationship between the data flow of entity references and the supported architectural changes, we propose four basic reconfiguration actions used to annotate component interfaces – the reconfiguration actions, defined as part of the design architecture, constrain where, when, and which architectural changes can happen at runtime.

In order to show a simple example of a design architecture using the proposed reconfiguration actions, a short overview of all four reconfiguration actions and their properties follows (more details are provided later in Sect. 4.2):

To meet the requirements R2 and R4, 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.

Fig. 6 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. 2). 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. Client can then interact with the entity by calling the `use` method repeatedly 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.

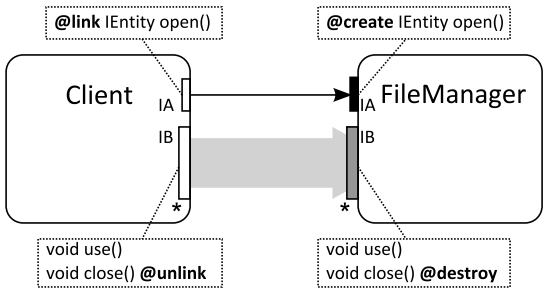

Fig. 6. Example – Client/FileManager

4.1 Entity References

All entity references representing entities have to originate from somewhere in the component architecture. More precisely, the first component publishing the entity reference in the architecture can be defined as an *owner* of the entity reference (and the entity behind it). We aim to support two types of entity reference owners: (i) a primitive component publishing an internal data structure or object via a handle or reference, (ii) a dynamic component publishing references to its own interfaces –

[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.

these are defined later in Sect. 4.4. The owner of an entity reference is known in the architecture only at the exact level of nesting, where the owner resides. Everywhere else in the architecture entity references fall into two groups at a certain level of nesting:

(i) *Type A*, no information about the entity reference is available to the component system, thus it can be only passed to other components, but cannot be used to establish any bindings,

(ii) *Type B*, the entity reference has been published via a reconfiguration action, the interface used to publish the entity reference on such a component is then defined as a *local source*; reconfiguration actions can be applied only to this type of entity references.

If the owner component of an entity reference is known at a certain level of the architecture the entity reference is supposed to be of type B and the local source is defined to be the source interface of the owner. Entity references identifying the same entity (originating from the same interface of the owner) can be of both types A and B on different levels of nesting.

To simplify the description of architecture reconfiguration actions other criteria can be used to further divide the entity references into two disjoint groups:

(i) *Entering references*, i.e., entity references passed as input arguments of methods in provided interfaces, or as output arguments and return value of methods in required interfaces.

(ii) *Leaving references*, i.e., entity references returned as output arguments and return value of methods in provided interfaces, or input arguments of methods in required interfaces.

4.2 Basic Reconfiguration Actions

The goal of this section is to elaborate the four basic reconfiguration actions, that were introduced in the overview at the beginning of Sect. 4. To the general properties of reconfiguration actions, the following should be added:

(i) A reconfiguration action is always defined on an interface instance of a particular component and not generally on an interface type. As illustrated by examples in this section, a method of an interface type can have, and typically will have, associated different reconfiguration actions in different contexts (e.g., the difference in provided and required interface).

(ii) The actions can be associated not only with a direct argument of a method, but also with a method itself. Then by stating an action is associated with a method we mean the action is in fact associated with the *"this"* hidden argument of the method, i.e., it influences the reference on which the methods call occurs.

(iii) A reconfiguration action can be associated only with entity references that are of type B in the given context, i.e., the local source (interface) of the entity reference is known.

The section concludes with complete definitions of each of the four basic reconfiguration actions:

(a) *link*, creates new binding templated by a proto-binding leading from the target interface of the action. The other end of the proto-binding has to be a local source of the entity reference the *link* action is associated with. On composite components, the *link* also makes the target interface the local source of the entity reference for the immediate lower level of nesting inside the component with the target interface, i.e., the *target component*. Thus, the *link* action defines the entity reference to be of type B at the lower level of nesting. If the *link* action is omitted in the method specification, the entity reference would be only of type A in the lower level of nesting context, i.e., subcomponents would be only able to pass or store the reference, but would not be able to create a binding using the *link* action and call methods on the reference.

On *entering references*, the action executes when the entity reference is received – i.e., during invocation of an incoming call with input references and during return of an outgoing call with output references. Using the action on *leaving references* makes sense only in the context of a caller – then the action executes during the call invocation.

(b) *unlink*, undefines the target interface as the local source of the associated entity reference at the immediate lower level of nesting and removes the binding from the target interface to the local source at the level of nesting the target component resides.

On *entering references*, the action executes always at the end of a method call – i.e., when returning from an incoming call or when an outgoing call returns. The *unlink* action can be used in both caller and callee contexts on *leaving references* and also executes when a method call ends. The *unlink* action can be associated with method arguments, as well as methods themselves.

(c) *create*, on primitive components, it prepares the associated entity reference (directly representing the internal state of the component, e.g., an object instance) to be sharable, plus defines the component as the owner of the entity reference, implying the *create* action target interface becomes the local source of the entity reference.

On composite components (that, if not dynamically created as defined in Sect. 4.4, cannot own entity references), the *create* action creates a new internal binding templated by proto-binding leading to local source of the entity reference at the immediate lower level of nesting inside the target component. Again, the target interface becomes the local source of the entity reference at the level of nesting containing the target component.

Similarly, as the *link* action makes the associated entity reference type B in the inner context of the target component, the *create* actions makes the entity reference type B in the outer context of the target component, i.e., at the same level of nesting the target component resides (component environment). Without the *create* action the entity reference would be only of type A to the component environment.

The action is valid only on *leaving references* and always executes at the

time the entity reference is being passed – i.e., in caller context, during the method invocation and in callee context, during the method return.

(d) *destroy*, undefines the target interface as the local source of the associated entity reference and, on composite components, also removes the internal binding leading to the internal local source of the entity reference.

The action, being an inverse to the *create* action, can be used only on *entering references* and again executes as the reference is being received – i.e., in caller context, during method return and, in callee context, during method invocation. The *destroy* action can be associated with method arguments, as well as methods themselves.

4.3 Examples of Basic Reconfiguration Actions

In this section, we present two more 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. 4.

The second example (Fig. 7) 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 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 a specific window provided by the Client via the `performWith` call.

The third example (Fig. 8) shows the behavior of reconfiguration actions in a context of nested components (if we no longer look at the Client from Fig. 6 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

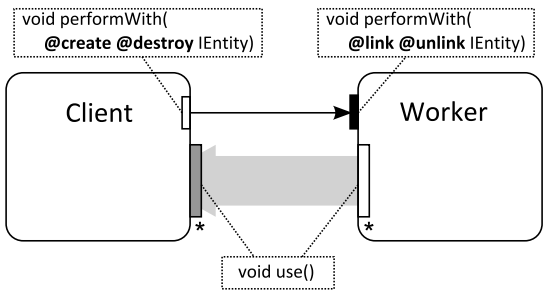

Fig. 7. Example – Callback

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.

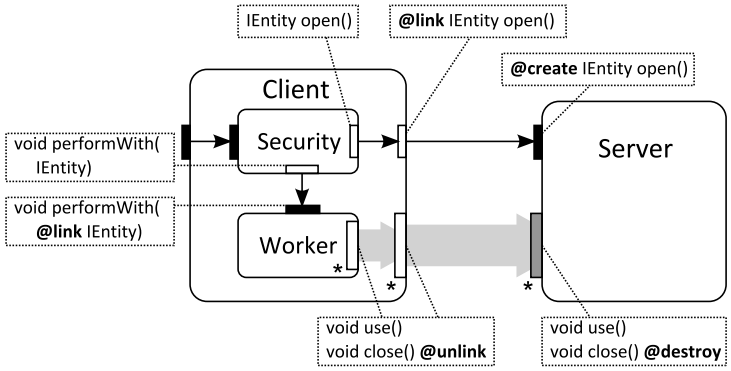

Fig. 8. Example – Passing a reference through a composite component frame

4.4 Reconfiguration Actions for Dynamic Components

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 from proto-components. Dynamic instantiation is however a much more complex task. The most challenging issues are the location of newly created components and the related need for their initialization. As a general evaluation of all possible options in dynamic component creation is beyond the scope of this paper, we present two reconfiguration actions – *new* and *delete* specifically targeting the scenario of encapsulated entities where the dynamic components need to be created at the callee side. An important difference from previously proposed basic reconfiguration actions is that the *new* and *delete* actions are defined at the level of the architecture and not at the level of component frames – i.e., the actions are added not by the component developers, but by the architecture designer. Another difference is they are never associated with a method argument or a reference but always with a whole method instead.

4.4.1 New

The *new* action can be most easily described on an example (see Fig. 9). The *new* action can be associated only with methods on unbound interfaces and only methods without input arguments are allowed. The action has also one proto-component (File in Fig. 9) associated with it – each time a method with *new* action is invoked a new component from that proto-component template is instantiated. All the method output arguments and return value with an associated *new* action then

correspond to provided interfaces on the proto-component, i.e., also on each of the dynamically instantiated components, and is used to pass references of these interfaces to the calling component (FMLogic in Fig. 9). An important feature of the *new* reconfiguration action is, that all provided interfaces of an instantiated component templated by a proto-component become local sources of the corresponding entity references returned from a *new* annotated method. From this point on all references to instantiated component interfaces are handled in the standard way as other entity references. As illustrated on Fig. 9, if FMLogic needs to call methods on IInit interface, the proper argument of `newFile` method needs to be associated with the *link* action. On the other, hand the IFile interface is directly passed as a return value of the `open` method call, so no reconfiguration actions are needed. Note that *link* action is added to the `newFile` method by the FMLogic component designer, but the *new* action is added later by the designer of the whole FileManager architecture.

A great advantage of the proposed *new* action is that it can be replaced by actual binding to another component that will provide entity references instead. This change is completely transparent and does not influence behavior or associated basic reconfiguration actions of the FMLogic component itself.

4.4.2 Delete

The *delete* action behavior is similar to the *destroy* action for interface bindings. At the end of the associated method call the dynamically instantiated component being called is destroyed – all bindings connected to it are destroyed as well. Such a component is then unreachable via standard method calls and can be stopped by the standard means of the component system.

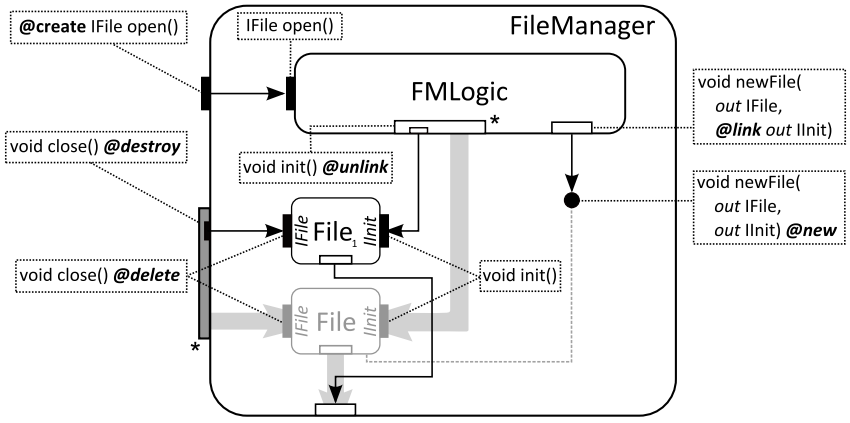

Fig. 9. Example – Component instantiation from a proto-component

5 Related Work

Since the proposed extension influences many aspects of component-based development, the related work ranges from formal verification methods through description of architecture dynamism in component models and its support at runtime to data modeling. To our best knowledge, there is no approach in the domain of

component-based systems that would tackle the problem of modeling dynamic entities in a comprehensive fashion. However, there are works that solve some of the aspects, though, often with different motivation.

5.1 Component models with support for data modeling

Although, the concept of data and data-flow modeling is well known and accepted in different domains of software engineering, contemporary component models often neglect this part of application design and do not provide any constructs of explicit representation. Nevertheless, there are attempts to fill the gap.

In [25], the authors provide a formalization of data passing and data circulation through a component-based application. The approach considers data as a part of a component. The data can be accessed from other components, updated, and transfered among components. The basic idea is to distinguish between control and data flow. Dedicated data connectors manage access to data through a global shared data space where data are created and shared. Although, the work tries to formalize the area of component-based systems which is not deeply explored, we see the main disadvantage in the shared data space where all data has to be stored and which can cause a substantial overhead at runtime. Comparing to our approach, we preserve the place where data are created.

Scade [24], Pin [22], PECOS [21], ProCom [28] propose a data-flow modeling based on connecting components with input and output pins which represent required and provided data. Such concept is typical for embedded systems where data are produced by various sensors, processed by control components, and then they serve as inputs for actuators. Such concept differs from our proposal where data are owned by a particular component and just a data reference circulates in the application.

Providing persistency in component-based applications can be seen as a data modeling approach [30], however this concept operates with the granularity of components. However, we claim that it is not enough to represent typical application data.

5.2 Behavior specification and verification

The need for associating properties directly with dynamic entities like a file is not new. In the context of general code analysis, it has already been applied using BLAST [8], SLAM [5]. The BLAST and SLAM model checkers allow to inspect state space of a program in C for assertion violations. Moreover, there is additional specification language that allows to associate an additional (shadow) state with a C structure and specify allowed sequences of function calls using the structure. Thus, it is possible to associate such shadow state with, e.g., the `FILE` structure in the C library and restrict the correct usage patterns to those starting with a call to `open` and ending with a call to `close`. Another way to associate a design information with an entity is using TraceMatches [9]. A TraceMatch is a negative specification, i.e., a set of traces (of method calls on a Java object) that are considered erroneous. Absence of such traces can be proved using either runtime checks (injected by AspectJ) and static analysis (by extension of the Soot framework).

These approaches can be used to analyze properties associated with dynamic entities. However, they expect availability of the source code and do not work compositionally, which is a basic requirement in the software component context.

There are behavior specification formalisms specific to component systems. However, majority does not consider dynamic entities at all (e.g., Interface automata [18], COIN [11], Wright [3], BP [1]). In principle, π-calculus [27] and some derived formalisms [16] have enough expressive power to describe this kind of dynamism, but the encoding is not trivial and generally unrelated with the implementation language.

5.3 Dynamic reconfiguration of architecture

Presently, dynamic architectures are quite well explored and several approaches to modeling them exist. The surveys of dynamic software architectures [10,14] present a categorization of different architecture evolution styles based on graphs [29,7], process algebras [3,26,17], UML profiles [23,4], or various logic [19].

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 [2], ACME [20], Plastik [6]) or at least have some support of architecture modification at runtime (Fractal [13]). However, they do not provide smaller granularity of modeling than a component.

6 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.

References

[1] J. Adamek and F. Plasil. Component composition errors and update atomicity: static analysis. Journal of Software Maintenance and Evolution, 17(5):363–377, 2005.

[2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In International Conference on Software Engineering, pages 187–196. ACM Press, 2002.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology, 6(3):213–249, 1997.

[4] D. Ayed and Y. Berbers. Uml profile for the design of a platform-independent context-aware applications. In MODDM '06: Proceedings of the 1st workshop on MOdel Driven Development for Middleware (MODDM '06), pages 1–5, New York, NY, USA, 2006. ACM.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. SIGOPS Oper. Syst. Rev., 40(4):73–85, 2006.

[6] T. Batista, A. Joolia, and G. Coulson. Managing dynamic reconfiguration in component-based systems. In EWSA 2005, pages 1–17. Springer, 2005.

[7] G. Berry and G. Boudol. The chemical abstract machine. In POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 81–94, New York, NY, USA, 1990. ACM.

[8] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with Blast. In M. Cerioli, editor, Proceedings of the Eighth International Conference on Fundamental Approaches to Software Engineering (FASE 2005, Edinburgh, April 2-10), LNCS 3442, pages 2–18. Springer-Verlag, Berlin, 2005.

[9] E. Bodden, L. J. Hendren, P. Lam, O. Lhoták, and N. A. Naeem. Collaborative runtime verification with tracematches. In O. Sokolsky and S. Tasiran, editors, RV, volume 4839 of Lecture Notes in Computer Science, pages 22–37. Springer, 2007.

[10] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, pages 28–33, New York, NY, USA, 2004. ACM.

[11] L. Brim, I. Cerna, P. Varekova, and B. Zimmerova. Component-interaction automata as a verification-oriented component-based system specification. SIGSOFT Softw. Eng. Notes, 31(2):4, 2006.

[12] 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.

[13] 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., 36(11-12), 2006.

[14] A. Bucchiarone. Dynamic Software Architectures for Global Computing Systems. PhD thesis, IMT Institute for Advanced Studies, Lucca, 2008.

[15] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In SERA. IEEE Computer Society, 2006.

[16] M. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. pages 18–32. 2007.

[17] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), pages 107–126, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[18] L. de Alfaro and T. A. Henzinger. Interface automata. SIGSOFT Softw. Eng. Notes, 26(5):109–120, 2001.

[19] M. Endler and J. Wei. Programming generic dynamic reconfigurations for distributed applications. pages 68–79, Mar 1992.

[20] D. Garlan, R. T. Monroe, and D. Wile. ACME: Architectural Description of Component-Based Systems. In Foundations of Component-Based Systems, pages 47–67. Cambridge University Press, New York, NY, 2000.

[21] T. Genssler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. Müller, and C. Stich. Components for embedded software: the PECOS approach. In Proceedings of the CASES'02, New York, NY, USA, 2002. ACM.

[22] S. Hissam, J. Ivers, D. Plakosh, and K. C. Wallnau. Pin Component Technology (V1.0) and Its C Interface. Technical report, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, USA, 2005.

[23] M. H. Kacem, A. H. Kacem, M. Jmaiel, and K. Drira. Describing dynamic software architectures using an extended uml model. In SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, pages 1245–1249, New York, NY, USA, 2006. ACM.

[24] O. Labbani, J.-L. Dekeyser, and P. Boulet. Mode-Automata based Methodology for Scade. In Hybrid Systems: Computation and Control (HSCC05), Zurich, Switzerland, 03 2005.

[25] K.-K. Lau and F. Taweel. Data encapsulation in software components. In H. Schmidt et al., editor, Proc. 10th Int. Symp. on Component-based Software Engineering, LNCS 4608, pages 1–16. Springer-Verlag, 2007.

[26] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, Proc. 5th European Software Engineering Conf. (ESEC 95), volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[27] R. Milner. Communicating and Mobile Systems: the π-calculus. Cambridge University Press, 1999.

[28] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic. A Component Model for Control-Intensive Distributed Embedded Systems. In CBSE, pages 310–317, 2008.

[29] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A graph based architectural (re)configuration language. SIGSOFT Softw. Eng. Notes, 26(5):21–32, 2001.

[30] OMG Corba Component Model Specification, http://www.omg.org/technology/documents/formal/components.htm.