Behavior Model Based Component Search: An Initial Assessment

Carlo Ghezzi and Andrea Mocci
Politecnico di Milano
DeepSE Group – Dipartimento di Elettronica e Informazione
P.za Leonardo da Vinci, 32 – 20133, Milano (Italy)
{ghezzi, mocci}@elet.polimi.it

ABSTRACT

We focus on the problem of searching components based on semantic queries on their provided interface. Although semantics-based search has long been advocated as a key enabler in the context of component-based software development and, more recently, service-oriented computing, no practical and scalable approach has been proposed yet. This paper presents a promising model-based search technique for interface behaviors based on operational specifications, called behavioral equivalence models (BEMs). Semantic queries are expressed equationally, following an algebraic specification style. The search engine tries to match specifications against queries. This can be done quite efficiently by encoding BEMs into relational models and queries into relational logic formulae, whose satisfiability is checked with the SAT-based constraint solver KodKod. We can report on an initial very promising assessment of the proposed technique, which has been applied to searching components in Java libraries providing container functionalities.

Categories and Subject Descriptors

H.3 [**Information Systems**]: Information Storage and Retrieval

General Terms

Documentation

Keywords

Behavior Models, Component Retrieval

1. INTRODUCTION AND MOTIVATIONS

The problem of effective artifact reuse has long been advocated as a crucial one from the software engineering community. It is viewed as a key factor to advance software to a mature enginering discipline. Reuse can in fact reduce the costs of software development and can improve the standardization of solutions and their overall dependability.

Software reuse has recently taken a new flavor in the context of *service-oriented computing (SOC)* [6]. Services may in fact be viewed as components that independent owners implement and run on demand, on behalf of possible clients. Their interface is exposed publicly for possible direct use and components are made accessible via some standardized protocols. This enables a new application development paradigm, which consists of integrating existing services to build complex service-based applications. Web services represent the perhaps best known practical instance of SOC.

Component reuse can be supported effectively by providing facilities for *component search*, that is, some technique to identify which components implement a specific functionality within the available ones. The most important critical aspect of component search is related to identifying component features and being able to query the component library, obtaining significant results. Many different techniques have been proposed to resolve this issue. For space reason, we cannot discuss and cite all the state of the art; for example, many approaches [8, 10] have addressed this problem by considering some kind of components behavior description, for example by using formal specifications as a representation of components exposed functionalities. However, most of these methods require complex reasoning involving specifications, for example by means of theorem proving. Conversely, recent research about the same problem casted within the context of SOC – that is, *service discovery* [2, 3] – mainly use keyword-based and ontology-based reasoning on signatures.

In this paper we explicitly address the problem of component search – even if the approach could be easily applied as well to service discovery – by means of using formal specifications for both the component description and functionality querying. In particular, we exploit finite state-like models for component behaviors, playing the role of the search base for the component search approach. Queries are instead expressed with an algebraic specification style. Both the models and the queries are encoded in relational logic, and the relational solver KodKod [9] is used to check the satisfiability of queries. The inspiration for the approach we present here came from both the experience we had in specification recovery [4] and from a recent work about specification consistency checking [5]. In this paper, the theoretical framework we used for consistency checking is exploited for

SUITE '10, May 1 2010, Cape Town, South Africa

$$
\begin{aligned}
& \alpha = ArrayDeque;\ \Xi = \{Object, Integer\} \\
\Phi = \{ & ArrayDeque :\rightarrow ArrayDeque, \\
& push : ArrayDeque \times Object \rightarrow ArrayDeque, \\
& addFirst : ArrayDeque \times Object \rightarrow ArrayDeque, \\
& addLast : ArrayDeque \times Object \rightarrow ArrayDeque, \\
& pop : ArrayDeque \rightarrow ArrayDeque, \\
& peek : ArrayDeque \rightarrow Object \cup \{Exception\}, \\
& pollLast : ArrayDeque \rightarrow ArrayDeque, \\
& peekLast : ArrayDeque \rightarrow Object \cup \{Exception\}, \\
& pollFirst : ArrayDeque \rightarrow ArrayDeque, \\
& peekFirst : ArrayDeque \rightarrow Object \cup \{Exception\}, \\
& size : ArrayDeque \rightarrow Integer\}
\end{aligned}
$$

Figure 1: The signature of ArrayDeque

component search. The paper is organized as follows. First, in Section 2, we briefly illustrate BEMs and algebraic specifications, together with their encoding in relational logic. Section 3 illustrates some example queries we actually support. Finally, Section 4 outlines conclusions and possible improvements of the proposed approach.

2. SPECIFYING COMPONENTS AND QUERIES

In this paper, we will focus on components with an internal state that behave as data abstractions. In these components, the internal state is hidden and the component interaction is available only through the exposed operations. As we already mentioned, our proposed semantics-based search technique for software components is based on two formal notations. The former is the specification language used to formally define component interfaces, which is based on *behavioral equivalence models (*BEM*)s*. The latter is the query language used to search for behaviors, which is based on *algebraic specifications*. Such specifications are particularly suitable for data containers; our initial assessment focuses on this kind of components and their behavior. We introduce BEMs and we briefly describe *algebraic specifications* below. Both specifications are defined over a *signature*, which describes the external interface available to the users.

A signature Π is a tuple $\langle \alpha, \Xi, \Phi \rangle$, where α is the data abstraction defined by the signature, Ξ is the set of external data abstractions used in the signature, and Φ is a set of functional symbols which describe the signature of the operations exposed by the data abstraction interface. Figure 1 shows the signature of the container *ArrayDeque*, which represents a subset of the homonymous container included in the Java container library, and implements a double-ended queue. Each functional symbol $\phi \in \Phi$ identifies an exposed operation. Each operation has a *domain* and a *range*. In this paper, we consider a particular class of signatures, the *linear signatures*. Within this class, operations can be classified as *constructors*, *modifiers* or *observers*. An operation is a *constructor* when the domain is empty and the range is α; there is exactly one constructor in the signature. A *modifier* is an operation where the domain is not empty and the range is α. Finally, an operation is an *observer* when the domain is not empty and the range is in Ξ. Exactly one type in the domain of observers and modifiers must be α; all the other types are the *parameters* of the operation. When a method plays both the role of an observer and a modifier, we consider those behaviors as two different functional symbols in Φ. Moreover, we model exceptions as special values of the range of observers.

2.1 Behavioral Equivalence Models

A BEM describes the behavior of a data abstraction within a precise and limited scope. The BEM is essentially a finite state machine where each transition is labeled with modifier invocations, each state is labeled with observer return values and the constructor determines the initial state. The model can be finite state because we constrain the specification in a limited scope. The scope of a BEM is defined by fixing a finite set of values for method parameters, that is, by defining the so-called *instance pools* to define every possible object belonging to external data abstractions in Ξ. The other scope limitation is related to the defined data abstraction α, which is determined by limiting the maximum number of states of the BEM. Figure 2 shows a BEM for the *ArrayDeque* data abstraction. The scope is defined by using a and b as two possible *Object* values and by limiting the number of states such that the BEM of Figure 2 models the behavior of the data abstraction only up to size 2. BEMs can be generated from a complete specification of the data abstraction, the *intensional behavior model* [4], which describes all the possible BEMs – and thus all the possible behaviors – with a grammar-like formalism. Alternatively, BEMs can also be automatically extracted with dynamic analysis [4].

Formally, a BEM over a signature Π is a tuple $\mathcal{B}_\Pi = \langle Q, I, \delta, q_0, \Psi \rangle$, composed of a set of states Q, an initial state q_0, an input set I, a transition function δ, and a set Ψ of state labelling functions representing observer return values. The input set of the BEM is the set of modifiers $I = \bar{\mathcal{M}}_\Pi$ instantiated within the BEM scope. The set Ψ is composed as follows: for every observer $\phi_o \in \Phi$, there is a $\Psi_{\phi_o} : Q \times \bar{\mathcal{O}}_\Pi \rightarrow \xi_{n_t}$, which is a state labelling function representing return values for the set of observers $\bar{\mathcal{O}}_\Pi$ instantiated within the BEM scope.

The sets of instantiated modifiers $\bar{\mathcal{M}}_\Pi$ and observers $\bar{\mathcal{O}}_\Pi$ are set of tuples, defined in function of the BEM scope. For example, in the case of the BEM of Figure 2, the set of instantiated modifiers contains the tuples $\langle push, a \rangle$ and $\langle push, b \rangle$, representing all the possible invocations of the *push* modifier. Similar tuples are included for the other modifiers. In this case, observers do not have parameters, so $\bar{\mathcal{O}}_\Pi$ contains one singleton tuple for each observer, e.g., the $\langle peek \rangle$ tuple. A complete formalization can be found in [5].

2.2 Algebraic Specifications

An algebraic specification for a data abstraction is defined over a given signature Π and adds semantic properties to it, that is, it specifies its behavior by means of a set E of (possibly conditional) algebraic axioms. Each axiom is a universally quantified formula expressing an equality among terms of the signature. For example, a classic axiom which describes a LIFO behavior for the pair of operations *push* and *pop* in the *ArrayDeque* container is the following: $\forall s \in ArrayDeque, e \in Object \mid pop(push(s, e)) = s$. Another typical example is the FIFO behavior, which can be expressed by the following conditional axiom involving the *push* and *pollFirst* modifiers: $\forall s \in ArrayDeque, e \in Object \mid pollFirst(push(s, e)) = \textbf{\textit{if}}\ (s = ArrayDeque())$

then s ***else*** $push(pollFirst(s), e)$. From a mathematical aspect, the concept of *algebra* assigns semantics to signatures and specifications. An algebra is defined as a set, the *carrier set* of the algebra, and a family of functions on that set. Our component retrieval approach is based on the possibility of interpreting BEMs as algebras, as discussed in [5]. In practice, to check if an axiom is consistent with the behavior described in BEM, we can interpret it as a property on the transition function δ of the BEM and the observer labelling function Ψ. For example, the LIFO axiom we presented before can be interpred as the following property on δ: $\forall s \in Q, e \in Object \mid \delta(\delta(s, \langle push, e\rangle), \langle pop\rangle) = s$[1] . Those axioms can be generalized as queries by adding an external level of quantification involving operations as follows: $\exists \mathcal{A} \in \Phi, \mathcal{B} \in \Phi \mid \forall s \in Q, e \in Object \mid \delta(\delta(s, \langle \mathcal{A}, e\rangle), \langle \mathcal{B}\rangle)$. This query can be read as searching if there exists a pair of operations which express the LIFO behavior typical of a Stack container. From now on, for simplicity, we will express queries as equivalent second-order quantified algebraic axioms. The query can be expressed as follows: $\exists \mathcal{A} \in \Phi, \mathcal{B} \in \Phi \mid \forall s \in S, e \in Object \mid \mathcal{B}(\mathcal{A}(s, e)) = s$. Moreover, we can also predicate about the signature of operations, for example in terms of their arity or the type of their parameters.

2.3 Relational Encoding of Bems and Queries

To perform queries, we encode the BEM to a relational model written in the subset of the Alloy language [7] supported by the KodKod solver. In the KodKod logic, a problem can be formulated as a set of *relation declarations*, a set of *relation bounds* and a *formula*, which contains relations quantified as variables. Relations in KodKod are not typed. They assume values that are drawn from the *universe* of atoms. With our encoding, the universe of atoms includes everything defined in the signature and in the BEM, that is, all the instance pools, the set of states, the functional symbols in Φ, etc. Since the BEM is a finite-state model, the relational encoding is straightforward: for example, ad-hoc relations model the transition function δ and the labeling functions in Ψ. For space reasons, we are not able to include the encoding details. However, we will show how an example of how the encoding can be used to perform queries.

A query is encoded as a KodKod formula, that is, an expression containing quantified variables. Second-order quantifications, used to search for specific operations, become quantifications bounded with the values of the set *operations*, that models the Φ set of the signature. Intuitively, the query we introduced previously, which searches for a LIFO behavior, can be described as follows. There exist two values $\mathcal{A}$ and $\mathcal{B}$ in the set *operations*, such that for every state s, and for every possible instantiation of $\mathcal{A}$ and $\mathcal{B}$, that is, for every $a \in \mathcal{A}.instantiations$ and $b \in \mathcal{B}.instantiations$, then $s.delta[a].delta[b] = s$. The *instantiations* relation models the binding between an operation and its instantiations, and it's the critical relation in the encoding that it's used to perform queries. We now proceed to present some preliminary results concerning some possible queries for behaviors obtained by a prototype tool we developed.

3. QUERYING BEMS

The encoding approach we presented in the previous section can be used to perform several kinds of queries for functionalities that can be expressed with algebraic axioms. We implemented our approach in a prototype tool; in this section, we present some basic queries and the results we obtained with our prototype tool [1].

As a case study, we used BEMs to model containers in the Java Collection Framework, such as *java.util.*{*Stack, ArrayList, ArrayDeque*}. One of the simplest queries we can perform is the one which searches for a LIFO behavior, that is, for a pair of modifiers $\mathcal{A}$ and $\mathcal{B}$, which can be used respectively to insert an element and remove the last inserted element, and an observer $\mathcal{O}$ which returns the last inserted element. We can express the query as follows:

$$\exists \mathcal{A} : \alpha \times Object \to \alpha, \mathcal{B} : \alpha \to \alpha, \mathcal{O} : \alpha \to Object \mid$$
$$\forall s \in \alpha, e \in Object \mid \mathcal{B}(\mathcal{A}(s, e)) = s \wedge \mathcal{O}(\mathcal{A}(s, e)) = e$$

Our tool retrieves several instances of operations which satisfy the specified algebraic equations. The simplest one is in the *java.util.Stack* class, where we can find the triplet $\langle \mathcal{A} = push, \mathcal{B} = pop, \mathcal{O} = peek\rangle$. More interesting result are obtained from *java.util.ArrayDeque* class. In fact, the deque can be used with a LIFO behavior considering either the head or the tail; thus, the query results contain two corresponding triplets, $\langle \mathcal{A} = addFirst, \mathcal{B} = pollFirst, \mathcal{O} = peekFirst\rangle$ and $\langle \mathcal{A} = addLast, \mathcal{B} = pollLast, \mathcal{O} = peekLast\rangle$.

KodKod formulae also support conditional expressions; for this reason, we are able to perform queries which correspond to conditional axioms, such as the ones that can be used to express the FIFO behavior of a queue. Suppose that we want to search for the behavior expressed by a method $\mathcal{A}$ which can be used to insert an element, a method $\mathcal{B}$ which removes the first element, a constructor $\mathcal{C}$ which produces the empty container, and an observer $\mathcal{O}$ which returns the first element. The query can be expressed as follows:

$$\exists \mathcal{A} : \alpha \times Object \to \alpha, \mathcal{B} : \alpha \to \alpha, \mathcal{C} :\to \alpha, \mathcal{O} : \alpha \to Object \mid$$
$$\forall s \in \alpha, e \in Object \mid$$
$$\mathcal{B}(\mathcal{A}(s, e)) = \textbf{\textit{if}}\ (s = \mathcal{C}())\ \textbf{\textit{then}}\ s\ \textbf{\textit{else}}\ \mathcal{A}(\mathcal{B}(s), e) \wedge$$
$$\mathcal{O}(\mathcal{A}(s, e)) = \textbf{\textit{if}}\ (s = \mathcal{C}())\ \textbf{\textit{then}}\ e\ \textbf{\textit{else}}\ \mathcal{O}(s)$$

Not surprisingly, our prototype tool can find several instances of $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{O}$ in the *ArrayDeque* class. Since a deque is, by definition, a double ended queue, we can use equivalently $\langle \mathcal{A} = addFirst, \mathcal{B} = pollLast, \mathcal{O} = peekLast\rangle$ or $\langle \mathcal{A} = addLast, \mathcal{B} = pollFirst, \mathcal{O} = peekFirst\rangle$.

3.1 Advanced Queries

In the previous section, we searched for operations which matched an exact signature of typical container operations. However, our relational encoding can be used for more general queries, which can be used to search for specific behaviors obtained by fixing specific values for method parameters. Consider the following example. A class implementing a list usually have several modifiers in its interface. Usually its modifiers include some kind of a $add : \alpha \times Object \to \alpha$ operation which adds an element to the end of the list. Similarly, an observer $get : \alpha \times Int \to Object$ can be used to obtain the element in a specific position and a modifier $remove : \alpha \times Int \to \alpha$ removes the element in a specific position. If we query a BEM of a class implementing a list using the queries we provided previously we are not able to find any operation that behaves either as a FIFO or a LIFO observer or modifier. However, the list can be in fact used

[1]Particular attention must be given because of BEMs are partial, as explained in [5].

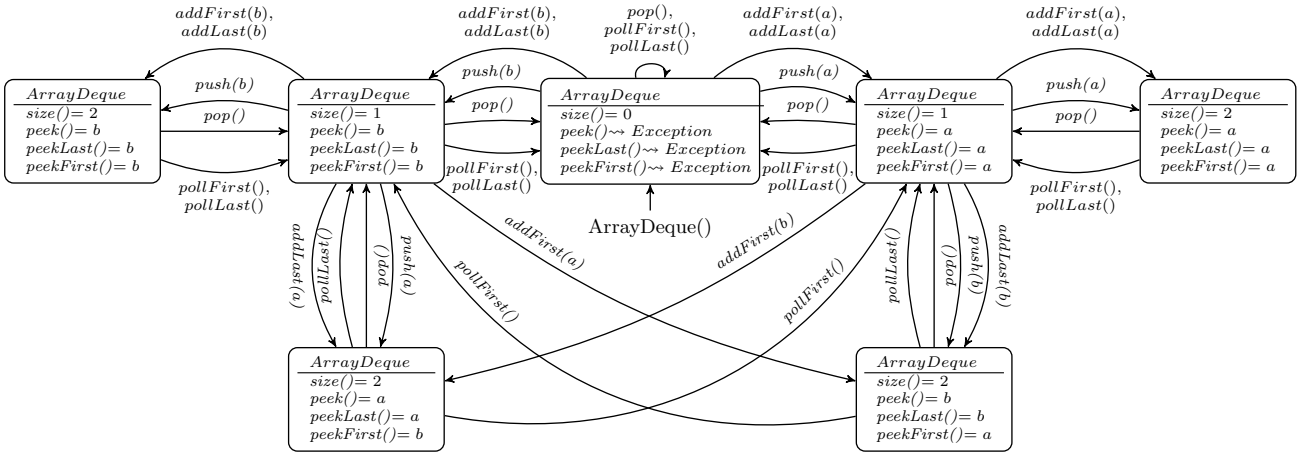

Figure 2: A Bem of ArrayDeque

as a FIFO or LIFO container if we consider some specific invocations of their operations with fixed parameter values. Suppose to search of a FIFO behavior in a class implementing a list with the interface we described before. The behavior of the observer returning the head of the queue can be obtained by calling *get(0)* on a non-empty list, while the behavior of the modifier removing the element on the head of the queue can be obtained by calling *remove(0)*. Intuitively, we could formulate the query as follows. The query predicates about the existence of a modifier $\mathcal{A} : \alpha \times Object \to \alpha$, a constructor $\mathcal{C} :\to \alpha$, an observer $\mathcal{O}$ with range $Object$ and with an arity greater than 1, and a modifier $\mathcal{B}$, with range α and arity greater than 1. Thus, for every state of the Bem and for every element $e \in E$ applied to $\mathcal{A}$, it exist a specific set of parameters $\bar{p_{\mathcal{O}}}$ for $\mathcal{O}$ and $\bar{p_{\mathcal{B}}}$ for $\mathcal{B}$ such that the container exhibits the FIFO behavior:

$$\exists \mathcal{A} : \alpha \times Object \to \alpha, \mathcal{C} :\to \alpha,$$
$$\mathcal{O} : range(\mathcal{O}) = Object \wedge arity(\mathcal{O}) > 1,$$
$$\mathcal{B} : range(\mathcal{B}) = \alpha \wedge arity(\mathcal{B}) > 1 \mid \forall s \in \alpha, e \in Object \mid$$
$$(\exists \bar{p_{\mathcal{O}}} \in params(\mathcal{O}) \mid \mathcal{O}(\mathcal{A}(s,e), \bar{p_{\mathcal{O}}}) = \textbf{\textit{if}}\ (s = \mathcal{C}())\ \textbf{\textit{then}}\ e$$
$$\textbf{\textit{else}}\ \mathcal{O}(s, \bar{p_{\mathcal{O}}})) \wedge (\exists \bar{p_{\mathcal{B}}} \in params(\mathcal{B}) \mid$$
$$\mathcal{B}(\mathcal{A}(s,e), \bar{p_{\mathcal{B}}}) = \textbf{\textit{if}}\ (s = \mathcal{C}())\ \textbf{\textit{then}}\ e\ \textbf{\textit{else}}\ \mathcal{A}(\mathcal{B}(s, \bar{p_{\mathcal{B}}}), e))$$

This query is correctly able to retrieve the *get(0)* and *remove(0)* methods as expected.

4. CONCLUSIONS

We presented a model-based technique to address the problem of searching component functionalities considering queries on their provided interface. The technique uses operational specifications, and in particular Bems, as behavioral descriptions and queries are expressed with an equational style, generalizing algebraic axioms by existentially quantifying the name of the operations. The search process is encoded as a relational problem, and in particular the Bems are encoded as relational models and the queries are encoded as relational formulae. The constraint solver KodKod is used to obtain search results. We illustrated a preliminary assessment where some basic and more advanced queries that could be performed to retrieve container functionalities from Bems describing the behavior of real Java library classes. A complete assessment of the required effort to produce Bems is part of the future work. However, we believe this is feasible because Bems can be automatically extracted with dynamic analysis. Future work also includes a better assessment of the querying capabilities of the approach, for example by applying it to components different than containers, and an evaluation of the required effort to produce queries, which can be reduced by wrapping the formalism in a more user friendly notation.

Acknowledgments

This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

5. REFERENCES

[1] Spy Search Website. http://home.dei.polimi.it/mocci/spy/search/.

[2] I. B. Arpinar, B. Aleman-Meza, R. Zhang, and A. Maduko. Ontology-driven web services composition platform. In *CEC '04*, Washington, DC, USA, 2004.

[3] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity search for web services. In *VLDB '04*, pages 372–383, 2004.

[4] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *ICSE '09*, Vancouver, Canada, 2009.

[5] C. Ghezzi, A. Mocci, and G. Salvaneschi. Automatic cross validation of multiple specifications: A case study. In *FASE 2010*, Paphos, Cyprus.

[6] M. N. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[7] D. Jackson. *Software Abstractions:Logic,Language,and Analysis*. MIT Press, '06.

[8] A. Podgurski and L. Pierce. Retrieving reusable software by sampling behavior. *ACM Trans. Softw. Eng. Methodol.*, 2(3):286–303, 1993.

[9] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, pages 632–647, 2007.

[10] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.*, 6(4):333–369, 1997.