Int J Softw Tools Technol Transfer (2008) 10:247–261
DOI 10.1007/s10009-007-0061-y

REGULAR CONTRIBUTION

Static verification of component composition in contextual composition frameworks

Mircea Trofin · John Murphy

Received: 4 December 2007 / Published online: 8 January 2008
© Springer-Verlag 2007

Abstract Contextual component frameworks, such as Enterprise JavaBeans (EJB), allow for components to specify boundary conditions for the runtime context. These conditions are satisfied at runtime by services of the underlying platform, thus ensuring that the context in which components run exhibits properties that allow them to operate correctly. Depending on how components call each other, it is possible that satisfying such conditions lead to problems such as reduced performance due to redundant service execution, or permanent errors (composition mismatches), due to incompatible boundary conditions. Currently, the semantics of these boundary conditions are expressed in natural language only, making it impossible to incorporate them into an automatic analysis tool. Furthermore, early understanding of how components call each other would be necessary, but it is currently difficult to achieve by means of a tool, as the method dispatch rules in a component system differ from the dispatch rules of the programming language(s) in which they were developed. We have developed a metamodel, $\mathbb{M}$, for describing boundary conditions, an analysis method, $\mathbb{A}$, and a static component-level call graph extraction method for EJB applications, CHA_{EJB}. $\mathbb{A}$ uses $\mathbb{M}$ models to analyze inter-component call graphs, and thus detect problems such as composition mismatches or redundancies, thus allowing for remedial action to take place. We present $\mathbb{M}$, $\mathbb{A}$ and CHA_{EJB} in this article, show that $\mathbb{A}$ produces correct results, and describe a prototype analysis tool implementing the three, which we used to validate our approach on two popular EJB applications.

Keywords Software components · Contextual composition · Static analysis · Enterprise Java

1 Introduction

Contextual composition frameworks allow components to specify, at deployment time, boundary conditions describing properties that the runtime context must meet. Based on these conditions, component platforms execute services when the control is passed from a component to another one, in order to ensure that the boundary conditions are satisfied.

Examples of such frameworks include Enterprise JavaBeans (EJB [1]), The COM+ model [2] and CORBA Component Model (CCM [3]). Typically, these frameworks are targeted at developing enterprise applications. For such applications, it is important to have a clear understanding of the behavior of an application, as early as possible, in order to detect possible errors, or to perform optimizations—especially since in the enterprise applications domain, any client request that cannot be satisfied can translate into loss of revenue.

Our work focuses on issues arising out of the very nature of contextual composition frameworks, issues that can lead to runtime errors or reduced performance (for example).

To best introduce the problems we are addressing, we use the example in Fig. 1. The server-side components A, B, C, and D are participating in a client interaction, as follows: the client starts by calling a method of A; this call is labeled as 1. While A's method is executing, it needs to call methods from B and C, as depicted (the call order is represented using UML notation, i.e. 1.1 is called from 1 and before 1.2, and all calls are synchronous)

The support of the Informatics Commercialisation initiative of Enterprise Ireland is gratefully acknowledged.

M. Trofin (✉) · J. Murphy
School of Computer Science and Informatics,
University College Dublin, Dublin, Ireland
e-mail: mtrofin@acm.org

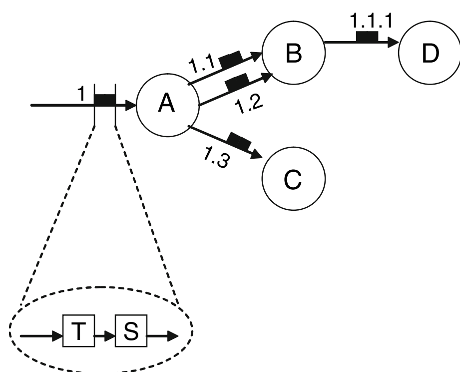

Fig. 1 Problem example

What is important to note is that, when the control is passed to a component method, there is some platform functionality executed just before the method code. This is represented by the *T* and *S* elements in the figure. We will refer to each such element as a "service". The role of some such services is to manage a particular aspect of the *runtime context* in which the component method is about to execute; as such, we refer to these services as *context management services (CMS)*. By context, we refer to (conceptually) a set of attributes associated with a thread of execution—by thread understanding a logical thread, not necessarily an operating system thread.

Each CMS controls one such attribute. The value of such an attribute influences the semantics of the execution of component methods. For example, in the above-mentioned frameworks, there are, typically, a security and a transactions (concurrency control) CMS. The security service would check that the user attempting to execute the component code belongs to some user group in some security realm, denying access otherwise. The transactions service ensures that, if the component method calls upon some datastore, that happens within some transactional isolation boundaries.

When a CMS executes, it takes the previous value of the context attribute it manages as input, and produces a new value as output. With this value, it creates a closure: the new value is associated with the thread for the entire call flow of the method in front of which the CMS executes. When the method returns, the CMS restores the previous context attribute value. Of course, CMSs in the call flow act similarly. An important observation is that, for the purpose of understanding how the context changes, the order of method calls *at the same level* does not matter, only their order (position) in the call stack. For example, in Fig. 1, the actual order in which method 1 calls the methods labeled 1.1, 1.2, and 1.3, does not matter, all that matters is that 1 calls them.

The functionality of a CMS is controlled by configuration parameters—the *boundary conditions (BC)*—that are associated with each component method. This association is either performed at component development time, or at deployment time - in any case, it does not change at runtime. For each different kind of CMS, there is a different set of valid boundary conditions. For example, the security service is controlled by an attribute indicating the security role to perform the check against (i.e., what role a user must have); in the transactions case, there is a set of possible attributes that one can chose from, each indicating how the transactional context in which the component method will execute relates to the one (if any) currently available.

It is possible that the execution of CMS (i.e., the enforcement of BCs) lead to runtime errors. A simple example is that of a security check that fails because the caller does not have the required security role. This is a desirable outcome; the problem is, however, when *regardless* of the role of an application client performing a call, the security check for a method *in a particular call sequence* is bound to fail. Referring again to Fig. 1, assume that A's method requires that a user be in the "admin" role, while one of B's methods require the user to be a "manager". If our security realm has no user satisfying both these roles, it is certain that either the security check in front of A's method would fail, or that in front of B's method.

In the case of transactions, in EJB, for example, let us assume that A's method uses the "required" flag, meaning that the method must execute in a transactional context, which is either already created, or, if not, one must be created by the transactions service. Let us also assume that C's method uses the "never" flag, meaning that it must execute outside a transactional context, and that, if such a context were present, it would constitute an error. It can easily be seen how, for the call scenario depicted in Fig. 1, the call to C would always fail.

These situations constitute *composition mismatches* or errors, and should be detected prior to runtime. In simple situations, applications built out of a small number of components with few methods and simple interactions, composition mismatches can be detected manually or prevented altogether, if the developers understand the "big picture" in which the components they develop will participate. The cases we are interested in are those in which the application is built out of a large number of components, maybe with some developed by thirdparties, where inter-component interactions are complex, and no developer had a clear overview of the entire application.

A second issue that can arise is that in which the execution of CMS is redundant. In our example, suppose that the security requirement of A's method is "admin", and so is D's method's requirement. In this case, there would be no need to perform security checks for D. The cost of redundancies, at least in the security case, has been documented [4]; in general, since CMS need to create closures, it is desirable to detect and remove redundancies.

In our previous work [5,6], we developed a boundary checks redundancy elimination mechanism, that employed an initial attempt at automatic analysis. This paper builds on our previous results, and its contributions are:

- A metamodel, $\mathbb{M}$, for modeling the semantics of CMS;
- An analysis method, $\mathbb{A}$, which uses $\mathbb{M}$ models on call graphs to detect properties associated with the enforcement of boundary conditions at various points in the graph; such properties include redundancy or composition mismatch;
- A discussion of the problem of static call extraction in component-based applications, and a solution for EJB applications;
- The validation of our methods, in the form of a prototype static analyzer for EJB, together with a Prolog rendering of $\mathbb{M}$ models for transactions and security. We used to extract and analyze call graphs in two popular applications.

2 Problem domain

In this section, we refine over some of the concepts already presented in our introductory section. We start by describing the component model on which we define the problem and construct the solution. The description consists of a definition of a "component", a description of how a component can be used, and a description of boundary conditions.

Our view of a component is similar to that in EJB. More specifically, it is code accessible via an interface which offers methods as points of access—much like a class implements an interface, in a Java context. The component is originally implemented in some programming language, however, the component source code cannot be assumed available; therefore, we assume that we work with compiled elements.

A component needs to be deployed on a platform in order to be used. When deployed, the component is associated with a name in a naming service. This name can be used by clients to contact the component. We refer to this name as the "global name". This is, essentially, a data-driven, runtime binding mechanism. Runtime, because it can happen when the component client is active, and data-driven, because the name the client uses can be a variable. Once deployed, the component performs no activity, unless a client invokes one of its methods.

The component client is either another component or an entity outside of the component platform boundary. In either case, we can assume that the client has some means of contacting the naming service. Additionally, client calls to a component are programmed as calls against an interface, following that, after binding, these calls be actually made towards an actual component method. It is possible, however, that the same interface be implemented by more than one component, meaning that the same client call might be dispatched to different method bodies, depending on the actual name used by the client to bind to a particular component.

We assume that component interfaces are of two types, externally accessible ("remote") and internally accessible ("local"), and that an application client can use only components with externally accessible interfaces, while components of the application can use both.

A component may specify via some means (in EJB, an XML document, the *deployment descriptor*) an association between local names and global names. This is a facility that decouples a component implementation from the actual names of other components: the component is written against the local names, following that at deployment time, the local names be linked to global ones. It needs to be made clear, however, that the list of local to global associations of a component does not completely designate the components that bindings will take place to: it is still possible for the component to lookup other components via global names, as well as, it is possible that elements listed be not used at all. Additionally, it is unknown which component method will bind to which other components. Still, the list of local-to-global associations can constitute a good indicative of what bindings *might* take place.

At this point, we can make the observation that statically deriving a call graph in a set of components is a non-trivial task, and that, additionally, language-specific static call graph derivation solutions (e.g., SOOT [7]), would be insufficient, since inter-component call dispatch follows non-language-specific rules. We need call graph information, however, since composition mismatches depend on the position of a method in a call stack, as it has been hinted in section 1. *This is the first problem we need to address*, namely, how to statically extract inter-component call graphs.

We have already discussed the typical chain of events that takes place when a component method is invoked—the fact that CMSs are called before the actual method body, and that the behavior of a CMS is fully controlled by boundary conditions. The actual mechanics through which these boundary conditions are associated to component methods do not matter—in EJB, for example, it is done via the deployment descriptor (and in future versions of the framework, via Java 5 annotations); what is important is that the CMS are solely responsible for managing the runtime context, and, as such, reasoning about problems such as composition mismatches or redundancies requires a formal description of the semantics of the CMS.

In turn, the semantics of a CMS are given by the controlling boundary conditions, however, currently, there is no formal description of BC semantics—they are simply provided in natural language in the framework specification—Table 1 summarizes the semantics of EJB boundary conditions. Note

Table 1 Semantics of boundary conditions in EJB

Transactions	
NotSupported	The method is called with no transactional context, regardless of whether there is one available
Required	The method is called in the transactional context of the caller; if no such context exists, one is created
Supports	The context of the caller, whether existent or not, is used
RequiresNew	A new transactional context is created, regardless of whether one was available from the caller or not
Mandatory	The context of the caller is used. If that is not present, the execution cannot continue
Never	The method is called without a transactional context. If the caller provides one, the execution cannot continue
Security	
Role	The caller of the method must provide a security context specifying a user which is in the role "Role". If that is the case, the method is called with this context; otherwise, the execution cannot continue

that the EJB specification [1] is more verbose, detailing what exceptions are thrown when the execution cannot continue, and how the creation of new contexts is handled—however, we are only interested in the transformation of the context, not how that is achieved.

If, additionally, we consider a case in which services and associated boundary conditions can be defined by application developers, then the problem of automatic reasoning becomes even more difficult. Essentially, we observe two possible "paths" for solving this problem:

- *Operate at the level of CMS.* At this level, we could apply existing formal methods for program semantics (e.g., JML [8]), however, some of these services (e.g., security) call upon remote servers, and therefore, the semantics derivable just from the service code might be too general to be of any use; additionally, the analysis would be heavily implementation- and language-dependent, making it hard to port to alternative implementations (different languages), although the semantics of the CMS/BC would be the same. This leads us to the next logical alternative:
- *Operate at the level of boundary conditions.* This is advantageous, since their semantics do not depend on the means these conditions are fulfilled, however, the lack of formalism at this level needs to be overcame. *This is the second problem we need to address.*

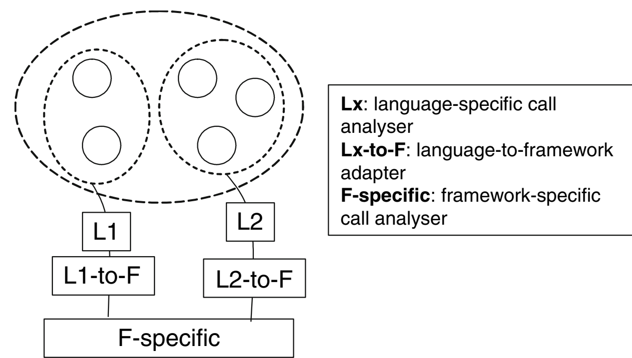

Fig. 2 Static call graph extraction in component systems

3 Statically deriving inter-component call graphs

We are interested in call graphs at a component level: which component methods might be called when an application client invokes a particular component method. As discussed, the order of calls is not relevant, however, the location in the call stack is. Therefore, the kind of call graphs we intend to obtain associate with a given method body a set of other method bodies that it *might* call; of course, the elements of this set have, in turn, a similar kind of set associated.

The central problem of statically deriving call graphs is deciding which method body will be selected at runtime, for a particular method invocation point in the program code (the dispatch problem).

To obtain an inter-component-level call graph, it is necessary to produce, first, a component-local method call graph. Therefore, a solution for deriving inter-component call graphs would need platform- or language-specific analyzers for each set of components developed for the same architecture. A framework-specific analyzer coordinates their activity—see Fig. 2. Language-to-framework adapters may need to be used where the framework-level understanding of the description of a method differs from the language-level one; again, for example, in CCM, the framework-level description of a method is IDL, which generally differs (slightly) from C++ or Java ones.

Neither of these modules are trivial to develop. In fact, language-specific analyzers alone are quite problematic. This can be further complicated by the fact that the analysis should be carried out on compiled code. As such, we will limit the discussion to the EJB framework and the Java language, and describe CHA_{EJB} an extension of class hierarchy analysis (CHA) [9] for EJB.

We start by discussing CHA for Java, in order to provide a frame for discussion for CHA_{EJB}.

3.1 Determining Java call graphs

Consider the code segment example in Fig. 3. The code for the method `aMethod` is written against the `java.util.Map`

```
public void aMethod(java.util.Map m){
  ...
  m.put(''aKey'',''aValue'');
  ...
}
```

Fig. 3 Plain Java dispatch problem example

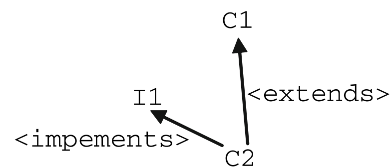

Fig. 4 Interface implementation providing method bodies through parent

interface. The application to which the code fragment belongs might provide more than one implementation for this interface. At runtime, the method call `put` in our example will need to be dispatched to an appropriate method body, however, which body that is depends on the actual type of the parameter `m`. We can, however, assert that the method body that will be executed belongs to an implementation of `java.util.Map`. Given the set of Java classes that form an application, it would be possible to find the subset that implements `java.util.Map`, and, as such, assume that any member of this subset is a potential target for our method call. However, it is possible to load classes at runtime, and their origin could be unknown at the time the static analysis is carried out—for example, a class could be a method parameter and be client-provided. As such, the set of implementers for our interface cannot be completely determined.

A similar discussion can be carried out for calls made towards an abstract class, as well as a regular class, with the same reservation about the completeness of the set of possible implementations (or subclasses, in this case) that can be known prior to runtime.

Further, if an application uses reflective calls, it would be required to carry out difficult data analysis in order to determine the method body that will be invoked by such call.

CHA determines call graphs without any form of data analysis, based solely on the set of classes (and their inheritance relationships) made available for analysis.

Given the set of all available application classes *AllClasses*, and a type (i.e. class/interface) T, we define $Impl(T)$ as the set of classes $C \in AllClasses$ that directly implement/extend T. That is, if $c \in Impl(T)$ then the execution of the Java code $c.getSuperclass()$ returns T, or T is part of $c.getInterfaces()$.

In Java bytecode, a method invocation carries with it the information in the tuple *(T, methodName, paramTypeList)*, with T the compile time-determined type of the receiver, and $paramTypeList$ the list of compile-time determined types of the formal parameters of the method. In Fig. 3, T is java.util.Map; $paramTypeList$ is $(java.lang.Object, java.lang.Object)$ - because this is the only available signature that matches our call (refer to the Java Virtual Machine specification [10]). We are interested in obtaining the set of possible method bodies, $B(T, S)$, that the call could be dispatched to. These bodies would have the same signature S (i.e., $methodName$, $paramTypeList$ pair), what is left to determine is the set of classes T that declare this method (i.e., provide an implementation). We say a class c **declares** a method with signature S, and write $c :: S$, if the execution of the Java code *c.getDeclaredMethod(methodName,paramTypeList)* succeeds.

For a call (T, S), the possible method bodies belong to, $Impl^+(T)$, the transitive closure under the inheritance relationship of T. That is because "any" class having T as a parent could potentially contain the method body the call would be dispatched to. Additionally, if T is a class, then we want to consider it as well in our set of possible implementations. In general, we shall label with *AllImpl(T)* either $Impl^+(T)$, if T is an interface, or $Impl^+(T) \cup T$ if T is a class.

If $I1$ is an interface declaring a method with signature S (as defined above), it is possible that none of the elements of $Impl^+(I1)$ declare (implement) that method. Consider the situation in Fig. 4. Our interface, $I1$, is implemented by $C2$, which, in turn, extends $C1$. It is possible that our method (not depicted in the figure) be provided by $C1$ and not by $C2$, and $C2$ still be considered a legal implementation of $I1$.

In such a case, we need to extend the set of classes to search for a method body to include parent classes of the elements $c \in AllImpl(I1)$. For such a c, we would be interested only in the most specific implementation of the method with signature S, that is, the implementation provided by a parent class p of c, but not by any other parent of c that is a child of p. If we use the notation $a <: b$ to indicate that a is a subtype of b, we define $\sqcup_T^S = c$, with $T <: c$, $c :: S$ and $\nexists d$, with $T <: d$ and $d <: c$ and $d :: S$.

We can summarize that, given a method call specified by a pair (T, S) as above, the set of classes that provide method bodies that this call could be dispatched to is:[1]

$$Impls(T,S) = \begin{cases} \{c|c \in AllImpl(T), c :: S\}, \text{if } T \text{ class} \\ \{u|u = \sqcup_c^S, c \in AllImpl(T)\} \cup \\ \{c|c \in AllImpl(T), c :: S\}, \text{if } T \text{ interface} \end{cases}$$

Then, the set of method bodies corresponding to our call (T, S) is: $B(T, S) = \{m | c :: m, c \in Impls_T^S\}$, with m of the form (T_i, S_i), as before.

[1] The formula below includes classes that have all their methods overridden by subclasses. This is intentional, since it is still possible that such classes be instantiated.

Given a method (T_0, S_0), we define the set $C(T_0, S_0)$ as the set of method calls of the form (T_i, S_i) made from (T_0, S_0) (this set can be practically extracted with a bytecode analyzer, like ASM [11]), and the set $P(T_0, S_0) = \{(T_j, S_j)|(T_j, S_j) \in B(T_i, S_i), \forall(T_i, S_i) \in C(T_0, S_0)\}$. Based on the Java Virtual Machine Specification [10], it can be argued that $P(T_0, S_0)$ is guaranteed to contain *all* method bodies that *might* be called from (T_0, S_0), assuming that no reflective or native method calls are made, and that the application does not load new classes at runtime, unknown at the time of the analysis.

Definition 1 (*Call graph of a method*) Given a method m (through a pair (T, S) as before), we define the call graph of m a directed graph where nodes represent methods (and are labeled with (T, S) pairs). An arc from a node (T_1, S_1) to a node (T_2, S_2) exists, if $(T_2, S_2) \in P(T, S)$.

Based on the definitions and observations above, the resulting call graph is complete, in the sense that any node in the graph points to all the methods that it might call (under the assumptions listed above).

3.2 CHA_{EJB}: extracting EJB call graphs

In an EJB scenario, the definitions above are not readily applicable. The problem lies with the fact that components do not explicitly implement their interfaces via regular Java mechanisms (i.e., using the keyword "implements"). Instead, this relationship is expressed in the deployment descriptor using XML constructs. A second aspect specific to EJB applications is that components can give relative names to other components they might use, again via the deployment descriptor. In turn, this information can "hint" towards possible bindings taking place at runtime—this aspect, however, will be investigated as part of our future work.

To address the problem of interface—implementation relationship, we observe that it is sufficient to include the information contained by deployment descriptors when computing *AllImpl(T)*, where T is a component interface, namely, the relationship between a component's interface(s) and its implementation. Formally, we can refer to $AllImp_{EJB}l(T) = AllImpl(T) \cup \{C | C\ implements\ T\}$, where T is an EJB business interface and C is an EJB component implementation, and the relation "implements" between T and C exists if there is an entry in a deployment descriptor stating that.

We can now construct "component-aware" graphs using Definition 1, and replacing *AllImpl* with $AllImpl_{EJB}$ when computing any of the other sets (like P, etc.). The same claim can be made, namely, that the set of method bodies evaluated to possibly correspond to a call is complete, under the same set of assumptions (i.e. no native or reflective calls, and no runtime loading of new classes). Essentially,

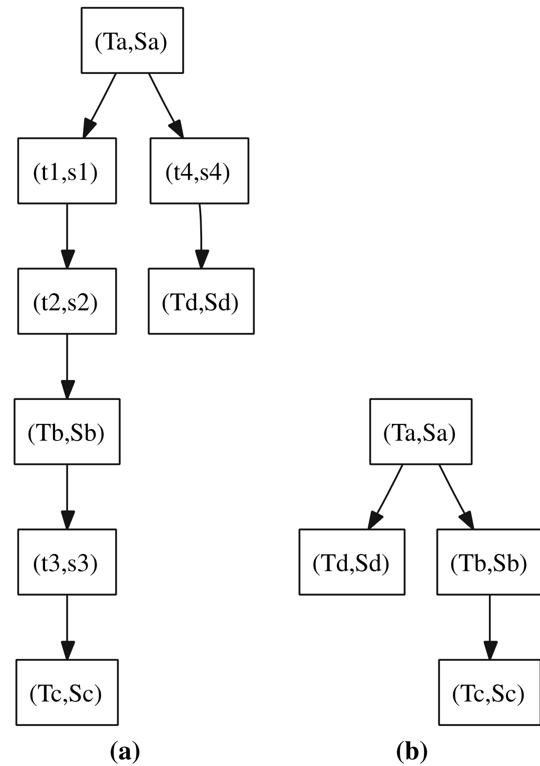

Fig. 5 Extracting component call graphs

CHA_{EJB} requires the inclusion in CHA of non-language-specific information.

So far we can obtain call graphs showing a mixture of component as well as plain Java calls. We define in what follows the component-level call graph of a method:

Definition 2 A CHA_{EJB} component-level call graph of a method is obtained from the "component-aware" call graph of that method as follows: non-component nodes are removed from the graph; from the remaining nodes, two such nodes, representing component methods (T_1, S_1) and (T_2, S_2) are reconnected if, in the original graph, there existed a path from (T_1, S_1) to (T_2, S_2) which did not contain another component method

Figure 5 shows an example, where nodes with capital letters (e.g. (Ta, Sa)) represent component methods. Figure 5a shows the initial call graph, while Fig. 5b depicts the resulting component call graph.

Note that details as to how EJB components are deployed (i.e., on different hardware servers) are not relevant to the call graph extraction, as long as all component classes, regardless where they would run, are considered.

4 CMS/BC semantics and analysis

Let us start by considering the call graph in Fig. 6, where nodes depict component methods (e.g. `Command`

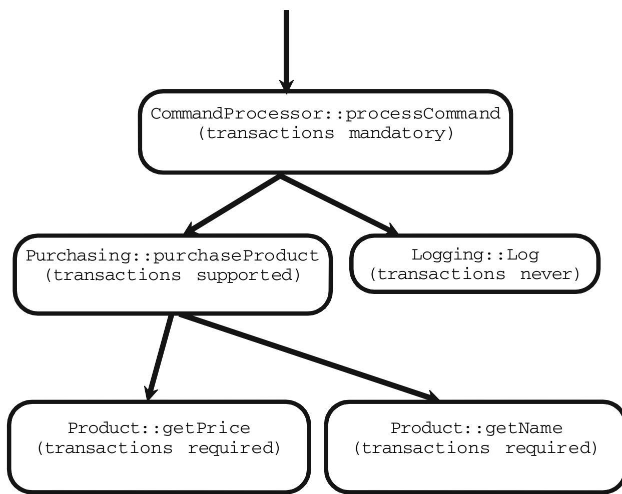

Fig. 6 Example Annotated Call Graph

`Processor:: processCommand`) together with their CMS boundary conditions (e.g. `(transactions required)`):

The external client calls `processCommand`. The value of the transaction context that the client provides cannot be predicted, so it can be assumed that it may be any valid value - including no value at all (the "null" value).

The transactional boundary condition (BC) of `processCommand` is `mandatory`. From the semantics of this BC (see table 1), we know that, if the client provides no transaction context, the execution cannot continue, however, if a context is provided, it is simply kept, and the execution continues (i.e. `processCommand` is called). We know, then, that if `processCommand` executes, it does so in some transaction context.

Next, let us pick the situation where `processCommand` calls `Log`. The latter's transactions BC is `never`. That means that, if a transaction context is provided, the execution cannot continue, however, if no transaction context is provided, it is left unmodified and the execution continues. Now, based on the reasoning above, we are guaranteed that, at this stage, there is, in fact, a transaction context available, meaning that the execution cannot continue, and `Log` is not going to be called. Since our initial assumption was that the transaction context the external client calls with can have *any* value (including no value), it means that, in *all* cases, in this particular call graph, `Log` will not be executed due to a composition error (BC mismatch).

On a separate path, `processCommand` calls `purchaseProduct`. The latter's transactions BC does not change the transactions context - therefore, the transactions context is still the value that it had in `processCommand` - therefore, a non-null value. Going further, `purchaseProduct` calls `getPrice`, with the transactions BC specified as `required`. That means that if a null transactions context is provided, a new one is created, however, if a non-null transactions context is provided, it is simply kept—a redundancy case. Since we are guaranteed, at this stage, that a non-null context is passed, the execution of the transactions CMS at this point is redundant for this call graph, regardless of the initial (external client-provided) value of the transactions context.

What becomes apparent from the analysis sketch above is that the focus of the analysis is what is known about the set of values the context lies on after the enforcement of a particular BC, given some knowledge about the set of values the context lied on before this enforcement; and what can be said about the execution of the CMS enforcing this BC, given such knowledge about the initial set of values (for example, that it is redundant).

The next section studies in more detail how to represent (model) such information about BC enforcement (the $\mathbb{M}$ meta-model), how to use such information to carry analysis with the aim of detecting properties of the execution of CMSs, such as redundancies or errors; and how to extract inter-component call graphs to be used for such analysis.

4.1 $\mathbb{M}$: Describing CMS/BC Semantics

Let us assume that boundary condition languages for some BC domain[2] d are described through a set, L_d, containing all valid expressions for this domain. For example:
$L_{transactions} = \{required, requires\ new, supported, never, not\ supported, mandatory\}$
$L_{security} = \{r | r\ is\ a\ role\ in\ a\ security\ realm\}$

We will next denote with D_d the set of all context values for some domain d. For example:
$D_{transactions} = \{x | x\ is\ a\ transaction\ id\} \cup E$
$D_{security} = \{x | x\ is\ a\ user\ in\ a\ security\ realm\}$

where the set E represents the set of "null" transaction contexts—that is, the absence of such context. This case needs to be considered, since the informal semantics of the transactions BC language mentions it.

Let us assume that CMSs behave like functions of the form $f_d : L_d \times D_d \to D_d$. The current examples of BCs in EJB suggest that the informal description of the semantics of BCs (or CMSs) is policy-like. This leads, conveniently, to a definition of f_d through partial functions, as follows:

$$f_d(e, x) = \begin{cases} \forall x \in Pre_1,\ f(e_1, x) = y,\ y \in Post_1 \\ \forall x \in Pre_2,\ f(e_2, x) = y,\ y \in Post_2 \\ \ldots \\ \forall x \in Pre_k,\ f(e_k, x) = y,\ y \in Post_l \end{cases}$$

[2] Domain, in the sense of "area of interest".

$$f_{transactions}(e,x) = \begin{cases} \forall x \in E, f_{transactions}(required, x) = y, y \in F \\ \forall x \in F, f_{transactions}(required, x) = x \\ \forall x \in D, f_{transactions}(supported, x) = x \\ \ldots \end{cases}$$

Fig. 7 A model for the transactions CMS

where $Pre_i \subseteq D_d$, $Post_i \subseteq D_d$, and $e_i \in L_d$.

The notations Pre_i and $Post_i$ were used to refer to the domain and co-domain, respectively, of the ith partial function.

For example, Fig. 7 depicts $f_{transactions}$ - incompletely, for reasons of space, a full depiction is given in section 4.3. The notation E was used to indicate the set of null transaction contexts (which has only one value), while the set F indicates the set of non-null transaction contexts.

By convention, in the case of an error, $f_d(e_i, x) \in \phi$.

Based on the discussion in the introduction of this section, the aim is to be able to assert the following:

1. given some *arbitrary* $S \subseteq D_d$, and some expression $e \in L_d$, what set does $f_d(e, x), \forall x \in S$, lie on; and
2. for the same S and e, what can be said about the execution of the CMS for the domain d, when it attempts to calculate $f(c, x), \forall x \in S$.

For example, in the example in Fig. 7, if $x \in F$, then we know that (1) $f_{transactions}(required, x) \in F$, and also that (2) the execution of the CMS calculating $f_{transactions}$ is redundant.

4.2 $\mathbb{M}$: Rules for Defining CMS semantics

Let us consider the following definitions:

Definition 3 (O) For some domain d, if $Pre \subseteq D_d$ and $e \in L_d$, we define $O : L_d \times \mathcal{P}(D_d) \rightarrow \mathcal{P}(D_d)$, with $O(e, Pre) \triangleq \{y | \forall x \in S, f_d(e, x) = y\}$.

In particular, in the partial function definitions used before, $Post_i = O(e_i, Pre_i)$.

Next, let us consider $TAGS$ a set of symbols that each describes a property of the execution of a CMS. This set can be chosen independent of the domain. For example, in both the cases of transactions and security, we are interested in redundancies and errors, so a set $TAGS = \{redundant, error\}$ can be chosen. The elements of the set do not carry any semantics for our analysis purposes, they are expected to mean something for the user of the analysis. However, it is expected that their meaning is mutually orthogonal (i.e. the set should not contain at the same time two symbols meaning `redundant` and `not redundant`, respectively). We can refer to an element of $TAGS$ as a *tag*.

$$f_d = \begin{cases} \forall x \in Pre_1, f_d(e_1, x) = (y, T_1), y \in Post_1 \\ \forall x \in Pre_2, f_d(e_2, x) = (y, T_2), y \in Post_2 \\ \ldots \\ \forall x \in Pre_k, f_d(e_k, x) = (y, T_k), y \in Post_k \end{cases}$$

where: $e_i \in L_d$, $Pre_i \in \mathcal{P}(D)$, $Post_i = O(e_i, Pre_i)$, and $T_i = T(e_i, Pre_i)$

Fig. 8 Defining a function through tagged expressions

We can now define:

Definition 4 (T) For some domain d, if $Pre \subseteq D_d$ and $e \in L_d$, $T : L_d \times \mathcal{P}(D_d) \rightarrow \mathcal{P}(TAGS)$, with $T(e, Pre) \triangleq \{Y | \forall x \in S$, the calculation by the CMS of $f_d(e, x)$ exhibits all properties in $Y\}$.

For example, in the case of transactions, $T(required, F) = \{redundant\}$ (based on Table 1).

To accommodate for tags, we can extend the definition of f_d before to $f_d : L_d \times D_d \rightarrow D_d \times \mathcal{P}(TAGS)$. $\mathbb{M}$ is a set of rules for defining such f_d through partial functions, as shown in Fig. 8.

It can be observed that the essential elements of such a definition are the e_i, Pre_i, $Post_i$, and T_i, respectively. As such, a short-hand definition for such f_d is through a set of tuples $(e_i, Pre_i, Post_i, T_i)$—as the examples in Sect. 4.3 will further illustrate. These tuples must satisfy the following properties:

1. $\forall (e, x) \in L_d \times D_d$, there is one and only one partial function defined.
2. in any expression, T_i is the set of tags for f_i.
3. the set of tags T_i is complete, in the sense that if some $t' \in TAGS - T_i$, the CMS computing $f_d(e_i, x), \forall x \in D_i$, does not have the tag t' associated.
4. by convention, if for some $e' \in L$ and $Pre' \subseteq D_d$, $\forall x \in Pre'$, $f(e', x)$ leads to an error, then a partial function $f'_d : e' \times Pre' \rightarrow \phi \times T'$, for some T', must be present in the definition of f_d.

The next question is how to compute, from an $\mathbb{M}$ model, O and T sets for an arbitrary pair (e, S), with $e \in L_d$, $S \subseteq D_d$.

Theorem 1 *Given the $\mathbb{M}$ model of a domain d, some $e \in L_d$, and some set $S \in D_d$, $O(e, S)$ and $T(e, S)$ can be computed as follows:*

- *if there is some i for which $S \subseteq Pre_i$ and $e_i = e$, then $O(e, S) = Post_i$, and $T(e, S) = T_i$, from the construction of the $\mathbb{M}$ model. Because of condition 1 above, such an i is unique;*
- *otherwise, $O(e, S) = \bigcup_{all\ i} O(e, X_i)$, and $T(e, S) = \bigcap_{all\ i} T(e, X_i)$, where $X_i = S \cap Pre_i$, and $X_i \neq \phi$*

Proof The first part of the theorem is immediate, from the construction of a $\mathbb{M}$ model, and was included just for completeness. For the second part, for the O equation: $\forall x \in S$, from condition 1 in the definition, we know that we can find in the $\mathbb{M}$ model partial functions i where $e = e_i$ and $x \in Part_i$. For such i, $S \cap Pre_i \neq \phi$, since the intersection contains at least x. Then, from the model, $f(e, x) \in Post_i$, which is included in $O(e, S)$ (from the way $O(e, S)$ is calculated), so this means that $f_d(e, x) \in O(e, S'), \forall x \in S$.

For the T equation, similar to the first proof, $\forall x \in S$, we can find partial functions i so that $x \in Part_i$. From construction, $T(e, S) \subseteq T_i$. Since this holds for all $x \in S$, it follows that T thus calculated is the set of tags for the pair (e, S). □

Theorem 2 (Correctness of O and T under vague assumptions) *If $S, S' \subseteq D_d$ and $S \subseteq S'$, then for any $e \in L_d$, $O(e, S) \subseteq O(e, S')$ and $T(e, S') \subseteq T(e, S)$. We refer to the assumption that the context belongs to such a set S' as a vague assumption.*

Proof It follows from the expressions for O and T that the number of partial functions in a $\mathbb{M}$ model providing satisfactory Pre_i terms for S' is at least as large as that for S. That means more $Post_i$ terms in the expression for O and T. Then we have that $O(e, S') = O(e, S) \cup R$ and that $T(e, S') = T(e, S) \cap P$ (where R,P denote the additional $Post_i$ terms). Hence $O(e, S) \subseteq O(e, S')$ and $T(e, S') \subseteq T(e, S)$. □

This theorem guarantees that overestimating the set of values the context might lie on, yields correct results, in terms of finding the tags associated with the enforcement of the BC, as well as an overestimation of the set of the resulting context.

4.3 Illustration: EJB boundary checks

For illustration, we will describe the semantics of the security and transactions CMS as found in EJB.

4.3.1 Security

The $\mathbb{M}$ model for security is

$$f_{security} : \begin{cases} (r, u(r), u(r), \{redundant\}), \forall r \in L_{security} \ (1) \\ (r, u(r)^C, \phi, \{error\}), \forall r \in L_{security} \ (2) \end{cases}$$

where the notation $u(r)$ denotes the set of users in $D_{security}$ that have the role r.

It can be observed that the semantics are well-formed. The numbers in brackets (e.g. (1) or (2)) are simply used for labeling (e.g., $T_1 = \{redundant\}$)

The relationship between the different roles cannot be known in general. Let us consider the situation in Fig. 9,

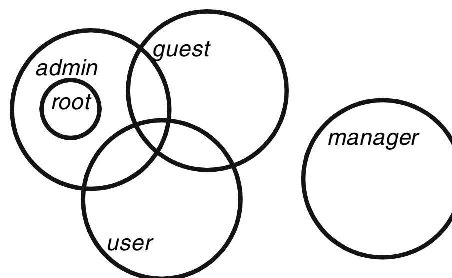

Fig. 9 Example of a security realm

where some roles share users and some not (i.e., there are users in multiple roles, while others have only one role).

We can see that, for example:

$O(admin, u(root)) = u(root)$; $T(admin, u(root)) = \{redundant\}$

This is expected, since all users in the `root` role are also in the `admin` role. Theorem 1 produces the same result: since $u(root) \subseteq u(admin)$, this example corresponds to the first part of the theorem, and $O(admin, u(root))$ is simply Pre_1. From expression 1 in the $\mathbb{M}$ model for security, Pre_1 is $u(root)$. Similarly, $T(admin, u(root)) = T_1 = \{redundant\}$.

Let us consider the following examples:

1. $O(admin, u(admin)) = u(admin)$;
   $T(admin, u(admin)) = \{redundant\}$
2. $O(admin, u(user)) = u(user) \cap u(admin)$;
   $T(admin, u(user)) = \phi$

In the first equation, it is known that the user belongs to $u(admin)$, therefore, as expected, enforcing the security constraint `admin` leaves the context unchanged and the security CMS is redundant. Let us analyze how these results are calculated. From Theorem 1, we know that, in order to calculate $O(admin, u(admin))$, we must calculate: $X_1 = u(admin) \cap u(admin)$ and $X_2 = u(admin) \cap u(admin)^C$. As such, $X_1 = u(admin)$ and $X_2 = \phi$. Then only the first partial function in the $\mathbb{M}$ model of security participates in the calculation of O and T, and $O(admin, u(admin)) = u(admin)$ and $T(admin, u(admin)) = T_1 = \{redundant\}$.

In the second equation, $X_1 = u(user) \cap u(admin)$ and $X_2 = u(user) \cap u(admin)^C$. Neither of these evaluates to ϕ. From the formula for O, we need now to determine: $O(admin, u(user) \cap u(admin))$ and $O(admin, u(user) \cap u(admin)^C)$. The former has the property that $u(user) \cap u(admin) \subseteq u(admin)$, which means that it evaluates simply to what the $\mathbb{M}$ model specifies: therefore, $u(user) \cap u(admin)$, which is $Post_1$ for $Pre_1 = u(user) \cap u(admin)$. Similarly, since $u(user) \cap u(admin)^C \subseteq u(admin)^C$, then $O(admin, u(user) \cap u(admin)^C) = \phi$ - again, this is known from the $\mathbb{M}$ model. The result is that $O(admin, u(user))$

$= u(user) \cup u(admin)$. This is expected: a successful check for a user being in the role `admin`, when the user is known to be a `user` means that the users that pass this check must be in both roles.

For $T(admin, u(user))$, since both X_1 and X_2 are not evaluating to ϕ, $T(admin, u(user)) = \{redundant\} \cap \{error\} = \phi$. Again, this is expected: checking for users in the role `admin` among users in the role `user` might succeed, but not always: hence, the CMS is not always redundant, nor always leading to an error, therefore, nothing can be said about it.

Let us consider one more example based on Fig. 9, namely, checking for the role `manager`, when it is known that the context belongs to the set $u(user) \cup u(admin)$. For $O(manager, u(user) \cup u(admin))$, $X_1 = u(user) \cup u(admin) \cup u(manager)$ and $X_2 = u(user) \cup u(admin) \cup u(manager)^C$. Since the role `manager` shares no users with neither `admin` nor `user`, $X_1 = \phi$ and $X_2 \neq \phi$. This means that $O(manager, u(user) \cup u(admin)) = \phi$ - since just the second partial function in the $\mathbb{M}$ model of security participates. Similarly, $T(manager, u(user) \cup u(admin)) = \{error\}$ - since only the second partial function in $\mathbb{M}$ participates. This is expected: checking for the role `manager`, in this case, is bound to always lead to errors, since no user can possibly be in all three roles (`manager`, `admin`, and `user`).

4.3.2 Transactions

The $\mathbb{M}$ model for transactions is given by $f_{transactions}$ as follows:

$$f_{transactions}(e, x) : \begin{cases} (required, E, F, \phi)\ (1) \\ (required, F, F, \{redundant\})\ (2) \\ (supported, D, D, \{redundant\})\ (3) \\ (requires\ new, D, F, \phi)\ (4) \\ (never, F, \phi, \{error\})\ (5) \\ (never, E, E, \{redundant\})\ (6) \\ (not\ supported, E, E, \{redundant\})\ (7) \\ (not\ supported, F, \phi, \phi)\ (8) \\ (mandatory, F, F, \{redundant\})\ (9) \\ (mandatory, E, \phi, \{error\})\ (10) \end{cases}$$

Let us consider:

$O(mandatory, D_{transactions})$ and $T(mandatory, D_{transactions})$.

We start by calculating: $X_5 = D_{transactions} \cap E = E$ and $X_6 = D_{transactions} \cap F = F$. Neither evaluate to ϕ, and $O(mandatory, E) = \phi$ and $O(mandatory, F) = F$. Therefore, $O(mandatory, D_{transactions}) = F$ and $T(mandatory, D_{transactions}) = \phi$.

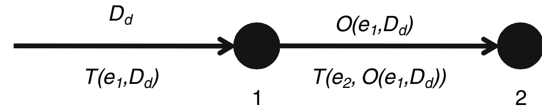

Fig. 10 Analysis method

4.4 $\mathbb{A}$: Analyzing call graphs

We can now describe our mechanism for analyzing call graphs. The goal of the analysis is associating tags to the execution of a particular CMS at every point in a call graph—in other words, to associate tags to nodes. The value of the context attribute associated with the top node of the graph is unknown, therefore, it lies somewhere on D_d. Refer to Fig. 10, which depicts a simple example, where two methods (the nodes) have constraints e1 and e2 associated (for the same domain d). We determine $O(e_1, D_d)$ and $T(e_1, D_d)$ for the top method in the graph. The T thus determined is the set of tags associated with the CMS executed in front of this method, while the O is used as parameter in the computation of O and T for the methods called from the current one. Hence, $O(e_1, D_d)$ is used to compute the tags for the CMS in front of method 2.

Definition 5 ($\mathbb{A}$) Given a call graph and any path in this call graph originating at the top node, and labeling the nodes in such path in order, starting with 0 for the top node, 1 for the node immediately connected to this node, and so forth, then the *tags associated with a call (node)* i are given by $T(e_i, O_i)$, with $O_i = O(e_{i-1}, O_{i-1})$, where e_i is the constraint associated for the domain d with the node (method) i, and $O_0 = D_d$.

Theorem 3 ($\mathbb{A}$ produces correct results) *Assuming that the tuples are properly defined, $\mathbb{A}$ guarantees that CMS executions are correctly (but perhaps incompletely) tagged, $\forall x$, x being the value of the context at the top node.*

Proof (Induction) Given a path as described in Definition 5, we observe that for $i = 0$ (i.e. for the first method), $T(e_0, O_0) = T(e_0, D_d)$ and that D_d is a vague assumption for the set of values of the context at this point, which means (Theorem 2) that the set of tags is correct.

For $i = 1$, $O_1 = O(e_0, O_0) = O(e_0, D_d)$. Since D_d was a vague assumption for the initial set of values of the context, from Theorem 2 it follows that O_1 is a vague assumption for the set of values the context has before method 1 is called. This means (based on the same theorem) that the set of tags for $i = 1$ is correct but maybe incomplete.

Then, for $i = k, k > 1$, if we assume O_{k-1} is a vague assumption, then $T(e_k, O_{k-1})$ is correct. Then, by induction, the theorem is proven. □

What does this mean? In general, it depends on the details of the analysis. For the cases considered throughout the paper,

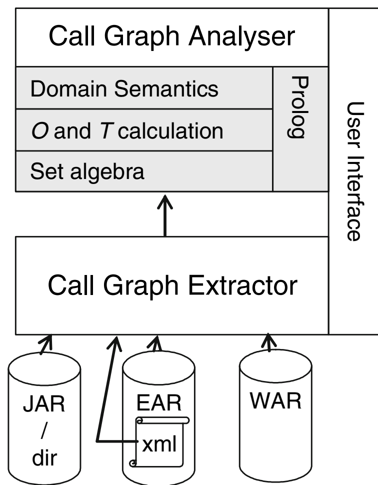

Fig. 11 EJB analyzer

it means two things are possible: (1) not all redundant CMS executions are detected, and (2) not all composition mismatches are detected.

In the first case, an effect is that redundancy elimination mechanisms leave the system in an optimized, but sub-optimal state. Since, however, the detection is correct, the transformation of the system to this state is guaranteed to be semantics-preserving.

In the second case, it simply means that, while the reported composition mismatches are correctly identified, it may be the case that additional mismatches be detected at runtime. The value here is that the analyst can confidently address the early-detected cases, and, while the need for runtime verification is not removed, the effort is reduced.

5 Validation

To validate $\mathbb{M}$, $\mathbb{A}$, and CHA_{EJB}, we need to show that they can be used to analyze real scenarios. We implemented a prototype call graph analysis tool for EJB applications. The prototype has three parts: an inter-component call graph extractor (implementing CHA_{EJB}), an analyzer (implementing $\mathbb{A}$), and a user interface.

The call graph extractor is given, via the user interface (UI), a list of locations covering all the classes of an application. These locations are either jar files, directories, enterprise archives ("EAR" files) containing EJB components and their XML deployment descriptors, or web archives (war files) containing servlets (Fig. 11).

The classes are loaded by a specialized class loader. This class loader maintains $Impl(T)$ relationships, and is "EJB-aware", meaning that it generates $Impl(T)$ for components as well. Class bytecode is visited using ASM [11]. The call graph of a method is generated using the information

```
security(Cond,In,Cond,Input,[redundant]).
security(Cond,Input,-Cond,f,[error]).

:-assert(disjoint(n,o)).
trans-attribute(required,Input,some,some,[redundant]).
trans-attribute(required,Input,none,n,[]).

trans-attribute(never,Input,some,f,[error]).
trans-attribute(never,Input,none,none,[redundant]).
```

Fig. 12 Example Prolog rendering of BC semantics

contained in the bytecode, as well as the $Impl(T)$ relationships provided by the specialized class loader. The method to start the analysis from is specified via the UI. It is left to the client of the analyzer to decide how top methods are selected. The full description of the call graph extractor is out of the scope of this article.

The call graph analyzer is defined around a Prolog interpreter. We decided to represent sets as set algebra expressions. Individual terms are Prolog atoms, and represent "basic" sets in the domain. How a domain is broken down into such basic sets is left open, however, the assumption is that they are neither disjoint, nor in any inclusion relationship with each other, unless explicitly specified.

A simple set algebra module is provided, which can construct and reduce expressions. It additionally provides the means for indicating whether two basic sets are disjoint or if one is included in the other. When an expression is constructed, it is always attempted to verify whether it reduces to ϕ (represented with the atom `f`). Equivalence to ϕ is the single relevant result, since we are interested in the value of intersection expressions as part of the calculation of O and T.

On top of the algebra module, we defined the computation of O and T as a single predicate, `o_and_t`. Given the name of a domain (as an atom), a constraint (an atom as well), and a set expression as constructed by our set algebra module, it returns the O as another set expression, and T as a list of atoms.

Tuples are defined as predicates which mimic their mathematical definition. The name of the predicate is the name of the domain. Figure 12 depicts the "translation" into Prolog of our original tuples, as defined in Sect. 4.3 (with n and o as defined in that section). Note that there is an additional element in the tuple, namely, the second parameter is used to provide the set of values the context is currently know to belong to (necessary to determine intersections with Pre_i sets).

The transactions domain is broken down in the two sets, as defined previously in Sect. 4.3. In the implementation, we used the atoms `some` for the set of non-null transaction ids, and `none` for the set of null transaction ids (F and respectively E).[3] The sets are represented by the Prolog atoms

[3] Since capital letters are interpreted by Prolog as variables, and since the "f" atom was already chosen for the ϕ symbol, we chose these more verbose labels for the E and F sets.

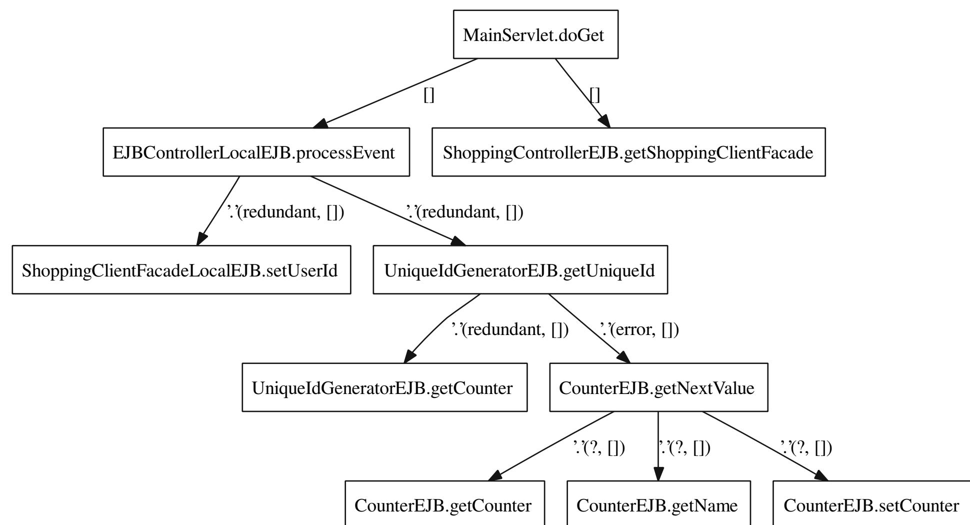

Fig. 13 Analysis of transactions boundary conditions in PetStore

with the same name, and, additionally, their disjointness is explicitly indicated, in order to ensure well-formedness. For security, any Prolog atom can be used to represent a security role, and the Prolog translation for the $\mathbb{M}$ model for security operates with roles generically (as did the $\mathbb{M}$ model).

It is the responsibility of the designer of such tuples to ensure that the atoms used to represent domains and constraints is consistent with the literal representation they have in the framework to which the application being analyzed belongs, or alternatively provide mappings. For example, in EJB, the transaction constraints start with capital letter, however, since that is interpreted by Prolog as variable names, we chose to lowercase them.

Currently, the output of the analyzer is graphviz files [12], where nodes represent methods, and arcs are labeled with the deduced tags.

For testing purposes, two EJB applications were used, PetStore 1.3.2 [13] and MedRec 1.0 [14].

PetStore is a reference EJB application that was chosen because it is meant to showcase "best practices" in EJB application design, and, thus, should constitute a good source of sample call scenarios. MedRec is a sample application developed as a demo for the WebLogic application server, and, like PetStore, it is meant to showcase best practice design patterns. Source code is freely available for both applications, and they are both relatively small in terms of number of components, making manual verification of the results of the analyzer possible.

We selected a servlet method as entry point for PetStore, and verified the output of the call graph generator against a manually produced output. For analysis, we introduced a number of errors manually, and checked the output for their presence. For exemplification, refer to Fig. 13.

The figure depicts a subset (for page space reasons) of the call graph that would result if a client called the `doGet` method of the `MainServlet` web component. The nodes represent component methods (the package names are removed for brevity). The arcs represent method calls. The label of the arcs indicates the set of tags, if any, produced by the analysis.

The analysis depicted in Fig. 13 is resumed to the domain of transactions. It can be observed how most transaction verifications are redundant - this is because most methods have the `required` transaction configuration—with the exception of the first component method, there is no need for the rest to ensure there is a transaction context available. The exception is the `getNextValue` method of the `CounterEJB` component, which we have modified the transaction constraint, for testing purposes, to `never`, and, as such, this call will always result in an error, in this call scenario. The calls stemming out of the call causing a composition mismatch are label led with `?`, indicating that the analysis could not be performed for them (since they would not be called).

In Fig. 14, the results of a similar analysis are presented, for MedRec (again, a subset is shown, for space considerations).

5.1 Experience with the analyzer and performance characterization

Developing the set of tuples for a particular CMS (or BCs) is a one-time effort, to be undertaken when such BCs are defined. This means that application developers need not be concerned at all with the definition of BC semantics, unless they intend to define new ones. Using the analyzer is, in our opinion, straight forward: all is needed is providing it with the set of archives or directories containing the classes of an application—listing the component EAR files would

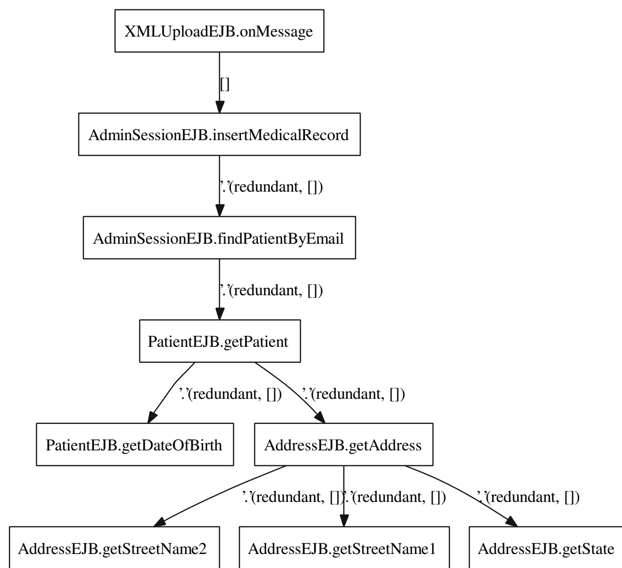

Fig. 14 Analysis of transactions boundary conditions in MedRec

Table 2 cost of analysis

class loading (211 classes, 5 depl. desc.)	10s
call graph generation - 599 methods	0.76s
analysis - 1 domain, 31 methods	0.07s

all tests were run on a Pentium M 1.7 GHz 1 GB RAM machine

normally be sufficient; and the name and list of parameter types of a method to start the analysis from. A particular strength of our approach is that no application instantiation is required—this, in the EJB case especially, saves considerable amounts of time.

Table 2 provides an orientative evaluation of the performance characteristics that can be expected from an implementation of our analysis method, as measured on the prototype. By far, the most expensive process is class loading and deployment descriptor parsing. Call graph generation for a servlet method (resulting in the reported 599 methods visited) and analysis—out of the 599, 31 were component methods—amount, together, to under 1 s. We envision our analysis solution as a development environment-integrated tool, and, as such, class loading would occur rarely and can be a background task, running in parallel with the developer's activity. Call graph analysis is then fast enough to be provided as a per-request operation.

The interpretation of the results of the analyzer, as well as the actions that might need to be undertaken, are out of the scope of this article. In the case of redundancies, we have described a self-optimizing platform design elsewhere [6].

6 Related work

The authors of [15] describe an extensible runtime verification framework. The framework requires the component application be deployed on an application server, where validation agents run testcases. By their nature, runtime testcases lead to incomplete results, and, furthermore, the degree of incompleteness/correctness is unknown. Or approach offers two benefits: it does not require the instantiation of the application, and the completeness and correctness of the results it produces is characterized.

Cadena [16] is an integrated environment for building and modeling CCM systems. Among other features, Cadena facilitates the specification and verification of correctness properties of models of CCM systems. The verification consists of model-checking using dSPIN [17], and, as such, the main issues being verified have to do with the distributed and concurrent characteristics of the application. As such, middleware (platform) services are modeled, however, only insofar as event and thread services.

Work in static analysis of role-based access control (security) in component applications [18] focuses on field access, using points-to analysis to determine which component fields are accessed by which component method(s). Reasoning about potential security problems is made on the assumption that, if some role q is given access to a method m_1 but not to a method m_2, then the level of access of m_1 is expected to be the same or less than that of m_2. Information about security realm configuration is not taken into account. Our approach, in contrast, is applicable to more than just the security aspect of an application (we discussed security and transactions in this article). Limiting the comparison to the domain of security, our analysis is more extensive, not being limited only to field access control, and the quality of the results can be improved, as the details of the security realm can be taken into account.

Other component verification techniques [19,20] focus on the validation of structural properties of components and are orthogonal to our efforts. Reference [19] focus on confinement checking: the verification of EJB framework-specific rules that specify that direct access to the Java object implementing a component should not be made directly, but only through the wrapper built around it by the component platform. The authors propose a "confinement discipline", a programming convention which can be verified statically at deployment time. Reference [20] focus on "bad store" errors. These errors result in data structures that violate framework rules—for example, an object cannot be stored into a data structure rooted at another object.

In [21], the authors describe a static method for detecting composition errors arising from component behavior. The component framework targeted is SOFA [22]. In SOFA, communication among components can be captured formally

using *behavior protocols*, component behavior descriptions based on regular languages. This formalism offers the basis for error detection.

ESC/Java [8] is a static analysis tool that attempts to find runtime errors in Java programs annotated using JML. In turn, JML annotations are Java-like. In effect, the analysis relies on the semantics of a closed set of languages (JML and Java), as opposed to the situation discussed here, where the set of languages used in the analysis is unknown.

Class hierarchy analysis [9] has been employed in a number of settings. In [23], for example, it is compared with two other static analysis algorithms in terms of their ability to aid the optimization of C++ programs. In [24], under the name of "data-driven simplification", CHA is used in a Modula-3 linker, mld, to direct optimizations.

SOOT [7], a Java optimization framework, can generate call graphs for entire Java applications [25]. In particular, the method described in [25] offers certain benefits over CHA, especially in terms of graph size. The applicability of call graph extraction methods, other than CHA, to the area of EJB (or similar) application analysis is subject of future work. In particular, the use of aspects specific to EJB, such as relative names specified in the deployment descriptor, hinting to more "probable" inter-component calls, is of interest.

Algebraic expression reducers have been previously built in academia, notably, REDUCE [26], or in the industry (e.g., Maple[4]). Any such tools could be used as a basis for defining boundary conditions semantics, as well as for performing the call graph analysis. We preferred building our own set algebra reducer, as the above solutions are unnecessarily large and generic for our purposes.

In general, the task of statically analyzing properties of a program arising out of composition has been that of a type checker, especially when the properties in question are forbidden runtime errors [27]. Our problem domain requires more than just runtime error detections, and we were able to address with one formalism an open number of properties, out of which we focused on runtime error and redundancy detection. The relationship between our work and type systems will be further investigated as part of our future work.

7 Conclusions and future work

We have presented an analysis method, $\mathbb{A}$, for contextual composition applications which can be used to detect properties associated with the services that enforce components' boundary conditions. The method uses $\mathbb{M}$ models of the semantics of such boundary conditions, and component-level call graphs extracted using the CHA_{EJB} method.

We proved that the analysis produces correct results. We have also shown that the correctness of the results is not affected by vague assumptions of the initial value of the context, however, the set of properties detectable may diminish. One effect of this is that redundancy removal [6] is guaranteed to be semantics-preserving.

A prototype implementation, targeted at EJB applications, has been used to validate the approach. The services/boundary conditions modeled were transactions and security. The properties targeted for analysis were redundancy and composition mismatch. We have validated our approach using the prototype on two popular EJB applications.

A number of issues remain to be solved, for example, whether a rigorous design strategy for designing $\mathbb{M}$ tuples can be specified, as well as methods for reducing the call graph produced through CHA_{EJB}. For example, some EJB containers apply rules for dependency injection—heuristics based on these rules would aid in pruning call graphs.

[4] http://www.maplesoft.com.

References

1. Sun Microsystems. Enterprise JavaBeans Specification (2003). http://java.sun.com/products/ejb/docs.html#specs
2. Don Box. Essential COM. Addison-Wesley (1998)
3. Object Management Group. Corba Component Model (2002). http://www.omg.org
4. Ammons, G., Choi, J.-D., Gupta, M., Swamy, N.: Finding and removing performance bottlenecks in large systems. In: Proceedings of ECOOP (2004)
5. Trofin, M., Murphy, J.: A self-optimizing container design for enterprise java beans applications. In: Proceedings of the Second International Workshop on Dynamic Analysis (WODA 2004), pp. 52–59 (2004)
6. Trofin, M., Murphy, J.: Removing Redundant Boundary Checks in Contextual Composition Frameworks. J. Obj. Technol pp. 63–82, July–August 2006
7. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: SOOT - a Java Optimization Framework. In: Proceedings of CASCON 1999, pp. 125–135 (1999)
8. Flanagan, C., Leino, K., Lillibridge, M., Nelson, C., Saxe, J., Stata, R.: Extended static checking for Java. In: Proceedings of Programming Language Design and Implementation (2002)
9. Dean, J., Grove, D., Chambers, C.: optimization of object-oriented programs using static class hierarchy analysis. Lecture Notes Comput. Sci. **952**, 77–101 (1995)
10. Lindholm, T., Yellin, F.: The Java^{TM} Virtual Machine Specification. Sun Microsystems, Inc. (1999)
11. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: a Code Manipulation Tool to implement adaptable systems. In: Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and Extensible Component Systems), November 2002
12. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Softw. Pract. Exp. **30**(11), 1203–1233 (2000)
13. Sun Microsystems. Java Pet Store Demo 1.3.2 (2003). http://java.sun.com/developer/releases/petstore/
14. BEA Systems, Inc. Avitek Medical Records 1.0 Architecture Guide (2003)

15. Grundy, J., Ding, G.: Automatic validation of deployed j2ee components using aspects. In ASE ’02: Proceedings of the 17th IEEE International Conference on Automated Software Engineering. Washington, DC, USA, p. 47. IEEE Computer Society (2002)
16. Hatcliff, J., Deng, X., Dwyer, M.B., Jung, G., Ranganath, V.P.: Cadena: An integrated development, analysis, and verification environment for component-based systems. In: International Conference on Software Engineering (ICSE), pp. 160–173 (2003)
17. Demartini, C., Iosif, R., Sisto, R.: dSPIN: a dynamic extension of SPIN. In SPIN, pp. 261–276 (1999)
18. Naumovich, G., Centonze, P.: Static analysis of role-based access control in J2EE applications. SIGSOFT Softw. Eng. Not. **29**(5), 1–10 (2004)
19. Clarke, D., Richmond, M., Noble, J.: Saving the World from Bad Beans: Deployment-Time Confinement Checking. In: OOPSLA ’03: Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications. New York, NY, USA, pp. 374–387. ACM Press, New York (2003)
20. Reimer, D., Schonberg, E., Srinivas, K., Srinivasan, H., Dolby, J., Kershenbaum, A., Koved, L.: Validating Structural Properties of Nested Objects. In: OOPSLA ’04: Companion to the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. New York, NY, USA, pp. 294–304. ACM Press, New York (2004)
21. Adamek, J., Plasil, F.: Component Composition Errors and Update Atomicity: Static Analysis. J. Softw. Mainten. Evolut. Res. Pract. **17**(5), 363–377 (2005)
22. Plasil, F.: Enhancing component specification by behavior description: the SOFA Experience. In: WISICT ’05: Proceedings of the 4th International Symposium on Information and Communication Technologies, pp. 185–190. Trinity College Dublin (2005)
23. Bacon, D.F., Sweeney, P.F.: Fast Static Analysis of C++ Virtual Function Calls. In: OOPSLA ’96: Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 324–341. ACM Press, New York (1996)
24. Fernandez, M.F.: Simple and effective link-time optimization of Modula-3 Programs. In: SIGPLAN Conference on Programming Language Design and Implementation, pp. 103–115 (1995)
25. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. In OOPSLA ’00: Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. New York, NY, USA, pp. 264–280. ACM Press, New York (2000)
26. Hearn, A.C.: REDUCE User’s and Contributed Packages Manual, Version 3.7. Knorad-Zuse-Zentrum Berlin, Germany, February 1999
27. Cardelli, L.: Handbook of Computer Science and Engineering, chapter Type Systems. CRC Press, New York (1996)