株式会社 SRA先端技術研究所
SRA Key Technology Laboratory, Inc.

Technical Report
SRA-KTL-2002E-003
November 2001

Software Component Search based on Behavioral Specification

Akira Mori, Toshimi Sawada,
Kokichi Futatsugi, Akishi Seo,Masaaki Ihshiguro

Abstract ：

In this paper, we report on an ongoing project to develop search engines for software components hosted by object request brokers (ORBs). Behavioral specification based on hidden algebra is used to allow search by functionalities rather than syntactic features. An algebraic specification language system CafeOBJ is used to support automation such as signature matching, refinement verification, and model checking. The user may test components found by search, by requesting ORBs to invoke operations remotely. Following an overview of hidden algebra and related specification/ verification methodologies, we describe the prototype system that plays role of a trader between the user and the ORBs. As the system uses standard internet protocols and web based interface only, any type of software service over the internet will be the target component by the method described in the paper.

Keywords:

Software component search, object request brokers (ORBs),behavioral specification, hidden algebra, signature matching, model checking, refinement verification

Notes: .

Software Component Search based on Behavioral Specification

Akira Mori
National Institute of Advanced Industrial Science and Technology
a-mori@aist.go.jp

Toshimi Sawada
SRA Key Technology Lab, Inc.
sawada@sra.co.jp

Kokichi Futatsugi
Japan Advanced Institute of Science and Technology
kokichi@jaist.ac.jp

Akishi Seo
Nihon Unisys, Ltd.
Akishi.Seo@unisys.co.jp

Masaki Ishiguro
Mitsubishi Research Institute, Inc.
masa@mri.co.jp

ABSTRACT

In this paper, we report on an ongoing project to develop search engines for software components hosted by object request brokers (ORBs). Behavioral specification based on hidden algebra is used to allow search by functionalities rather than syntactic features. An algebraic specification language system CafeOBJ is used to support automation such as signature matching, refinement verification, and model checking. The user may test components found by search, by requesting ORBs to invoke operations remotely. Following an overview of hidden algebra and related specification/ verification methodologies, we describe the prototype system that plays role of a trader between the user and the ORBs. As the system uses standard internet protocols and web based interface only, any type of software service over the internet will be the target component by the method described in the paper.

Keywords

Software component search, object request brokers (ORBs), behavioral specification, hidden algebra, signature matching, model checking, refinement verification

Acknowledgements

The research described in this paper has been supported by a contract under the management of the Information Technology Promotion Agency (IPA), Japan.

INTRODUCTION

With rapid progress of internet technologies, software is more likely to be reused or borrowed piece by piece rather than be built from scratch as a whole system. How to find "right" components will be an important issue if technologies like object request brokers (ORBs) and application servers establish themselves as infrastructures for software production in the future. However, the search for software is much more complicated than ordinary keyword search because defining precise requirements for software is difficult.

Formal methods have been expected to be useful for this purpose, yet very little has been achieved. Traditional formal methods advocate data type specification and require one to finish complete specification before doing anything, while on the other hand, model-based approaches stress accessibility and lack enough automation support. It is a challenging task to resolve trade-offs between precision and flexibility so that the search is conducted incrementally even with partial or incomplete information according to the desired level of abstraction.

In the current project, behavioral specification based on hidden algebra [7] is used to describe functionalities of software components. The CafeOBJ [8] system has been developed as a language system for hidden algebra and in fact it is the only system that supports automation such as signature matching, invariant checking, refinement checking, and coinduction. The inference engine (called PigNose) built on top of CafeOBJ, the specification repository, and ORBs, and a trader that manages user interface constitute the search service.

From user's viewpoint, the overall procedure looks as follows:

1. The component providers prepare specification for the components and register them in the specification repository as well as to ORBs.
2. The user gives the requirement specification of the components in need.
3. The interface part of the specification is matched to the ones in specification repository (signature matching),
4. Check if the matching found in the previous step preserves the requirement specification in the target specification (refinement verification).
5. Check, if necessary, invariant properties of the candidate specifications (invariant checking)
6. Test the component by invoking operations through ORBs (remote invocation)

Although the system is still premature and the examples we have tried so far are simple and somewhat artificial,

the experience has been positive and the technological implication of the experimental use of the prototype system seems exciting.

OVERVIEW OF HIDDEN ALGEBRA

This section presents a quick overview of hidden algebra. We will not go into technical details, for which the readers are referred to [7, 4].

Behavioral specification based on hidden algebra differs from traditional algebraic specification in its ability to define abstract state machines as well as abstract data types in the same algebraic formalism. State transitions are parameterized by data types and it makes easy to specify behaviors of persistent objects. However, hidden algebra does not force identification of state sets. Instead, they are defined as a result of **behavioral abstraction**, which means "two states are equivalent if and only if they cannot be observed different through experiments."

As an simple example, we give a behavioral specification of a flag object using CafeOBJ [4] notation below. A flag object is either up or down, and there are methods to put it up, to put it down, and to reverse its state.

```
mod* FLAG {
  *[ Flag ]*
  bops (up_) (dn_) (rev_) : Flag -> Flag
  bop up?_ : Flag -> Bool
  var F : Flag
  eq up? up F = true .
  eq up? dn F = false .
  eq up? rev F = not up? F .
}
```

With CafeOBJ specific keywords such as `mod*`, `*[ ... ]*` and `bop` aside, the meaning of the `FLAG` specification should be clear. However, some of the intended behaviors of the flag object, for example, a behavioral equation `rev rev F = F` cannot be deduced from the `FLAG` specification using ordinary equational reasoning. One will notice, however, that `rev rev F` and `F` cannot be distinguished by any means of method application and attribute observation, and also that there are only two distinguishable states, in which the flag is either up or down. Hidden algebra is formalized to capture this intuition.

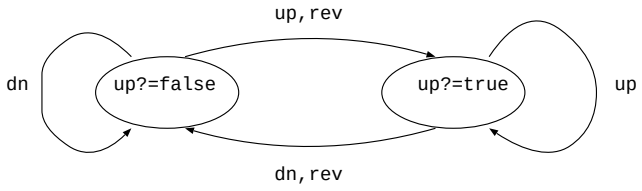

Figure 1: `FLAG` state machine

In `FLAG` example, hidden algebra allows one to specify the state machine in Figure 1 without saying that `Flag` consists of two elements. The number of the `Flag` elements is not determined a priori, but rather it is determined by the equivalence class on the "behaviors" defined by the equations in the `FLAG` specification.

Recent studies [3, 5, 13, 11] show that the proof systems for hidden algebra can be automated to some extent by symbolic calculation such as rewriting and resolution. Invariant checking for hidden algebra [11] is particularly promising, in which hidden algebra is regarded as a (infinite) state machine and well-known iterative calculation of fixpoints (also called closures) is applied. The study shows that verifying dynamic properties of software systems is much more manageable with behavioral specification than with data type specification.

CafeOBJ Keywords

Let us briefly explain the correspondence between hidden algebra concepts and CafeOBJ keywords so that the reader can interpret CafeOBJ code in hidden algebra.

The keyword `mod*` means that the module has a **loose** behavioral semantics in contrast with a **tight** (initial algebra) semantics specified by the keyword `mod!`. A pair of starred brackets `*[ ... ]*` is used for sort declaration of hidden sorts. Visible sorts are declared with `[ ... ]`. The keyword `bop` declares behavioral operations, i.e., attributes and methods. Ordinary equations are defined with keywords `eq, ceq` where `ceq` stands for conditional equations. Similarly, hidden equations are defined with keywords `beq, bceq`.

AUTOMATION FOR BEHAVIORAL SPECIFICATION

This section gives an overview of automated proof techniques for hidden algebra that are relevant to software component search.

Signature matching

Signature matching [14, 6] is a technique to construct signature morphisms from one specification to another without looking at the equation part of the specification. Let us give a definition of hidden signature morphisms.

Definition 1 *A **hidden signature morphism** $\varphi : \Sigma \to \Sigma'$ is a signature morphism*[1] *that preserves hidden sorts and behavioral operations. φ is also called a **view**.*

There are variations of signature matching as to which arguments are to be substituted. We put a strong restriction that only hidden sorts are substituted while visible sorts must be matched exactly. This may be justified on the ground that visible sorts model static data types in hidden algebra. One can use the name server to control name space if necessary.

[1] We do not give a definition of signatures and signature morphisms in this paper. These are found in introductory article for algebraic specification.

Coinduction

As we have mentioned earlier, behavioral equations such as `rev rev F = F` of `FLAG` cannot be proved by ordinary equational reasoning. Coinduction is a technique to prove equations of hidden sort.

A hidden equation can be regarded an invariant statement that the given pair of states always have the same attribute values. For example, insisting on behavioral satisfaction of `rev rev F = F` is the same as saying that `up? F = up? F'` is a relational invariant starting from `(rev rev F, F)`. This statement is trivially true if we admit double negation law of booleans. In general case, one has to perform safety model checking.

Model checking

In order to perform model checking on infinite-state systems, we need symbolic representation of sets of states. One may represent a set of states of hidden sort $\mathtt{h}$ by a predicate $\mathtt{P}(\mathtt{X}:\mathtt{h})$. For example, a predicate $\mathtt{P}(\mathtt{F}:\mathtt{Flag}) \triangleq \mathtt{up?}(\mathtt{F})$ represents the set of `FLAG` states in which the flag is up.

A nice thing about this predicate abstraction is that thanks to the syntactic restrictions of hidden algebra the set of previous states can be easily obtained through predicate transformer simply by replacing the base variable of the predicate with an expression composed of methods with appropriate existentially quantified variables. For example, $\mathtt{Q}(\mathtt{F}) \triangleq \mathtt{P}(\mathtt{rev}(\mathtt{F}))$ represents the set of states whose next states via `rev` operation will have the flag up. By taking disjunction of all possible transition (i.e., method application), we can obtain the set of (all) previous states of $\mathtt{P}(\mathtt{X}:\mathtt{h})$:

$$\mathtt{pre}(\mathtt{P}_{\mathtt{h}}(\mathtt{X}:\mathtt{h}))_{\mathtt{h}'}(\mathtt{X}';\mathtt{h}') \triangleq \bigvee_{\sigma:\mathtt{wh}'\to\mathtt{h}} (\exists \mathtt{V}:\mathtt{w})\ \mathtt{P}(\sigma(\mathtt{V},\mathtt{X}':\mathtt{h}'))_{\mathtt{h}}.$$

By abuse of notation, we let $\mathtt{V}:\mathtt{w}$ be a sequence of variable bindings $\mathtt{V}_1:\mathtt{w}_1,\ldots,\mathtt{V}_n:\mathtt{w}_n$ for a string of sort symbols $\mathtt{w}=\mathtt{w}_1\ldots\mathtt{w}_n$. To handle many-sorted cases more precisely, we shall consider a family (conjunction) of predicates $\bigwedge_{\mathtt{h}\in\mathtt{H}}\mathtt{P}_{\mathtt{h}}(\mathtt{X}:\mathtt{h})$ each of which ranges over a hidden sort $\mathtt{h}\in\mathtt{H}$. When considering predicate ranging over only one hidden sort $\mathtt{h}$, $\mathtt{P}_{\mathtt{h}'}$ $(\mathtt{h}'\neq\mathtt{h})$ may be regarded as `true`. For the matter of simplicity, we assume for the rest of the paper that the target system has only one hidden sort to model its state elements.

As a matter of well-known fact [15], `pre` is the only prerequisite for an abstract (safety) model checking assuming the classic fixpoint theorem. The backward safety model checking can be formulated as iterative calculation of the least fixpoints. Showing that the predicate P is safe starting from the set of initial states I is equivalent to showing that $I \not\subseteq \mathtt{pre}^*(\neg\mathtt{P})$ where $\mathtt{pre}^*(\mathtt{P}) \triangleq \mu\mathtt{Z}.\mathtt{P}\vee\mathtt{pre}(\mathtt{Z})$ is the least fixpoint of `pre` including P. $\mathtt{pre}^*(\mathtt{P})$ represents the set of states from which a state satisfying P may be reached. If one finds that $I \subseteq \mathtt{pre}^*(\neg\mathtt{P})$ during the iteration, a counterexample is found.

By Kleene/Knaster/Tarski fixpoint theorem, one can calculate $\mathtt{pre}^*(\mathtt{P})$ by a simple iteration. The key observation is that the convergence of the iteration can be judged by predicated calculus by checking implication. To perform this, we have developed a theorem prover called **PigNose** on top of the CafeOBJ system for many-sorted first-order predicate logic with equality. It is based on resolution and paramodulation and shares many features with Otter [12] while Otter does not support many-sorted calculus. The features include the set of support strategy, hyper-resolution, demodulation and atomic formula rewriting. PigNose is seamlessly integrated into the CafeOBJ system in that CafeOBJ axioms are transparently converted to PigNose clauses for theorem proving while the user only sees the CafeOBJ syntax.

The model checking procedure below implements the iteration with slight simplification due to the fact that the set of possible transitions (i.e., method applications) is exactly the same in each state in hidden algebra. The procedure maintains an indexed family $\mathtt{F}^{\mathtt{i}}$ $(\mathtt{i}\geq 0)$ of sets of predicates and a natural number `N`. Let $\mathtt{F}(\mathtt{X}:\mathtt{h})$ be the disjunction of all predicated in $\mathtt{F}^{\mathtt{i}}$ $(\mathtt{i}\geq 0)$ and note again that we only consider the case with only one hidden sort.

Model checking procedure `MC(I,P)`
(I: initial state, P: safety predicate)

1. $\mathtt{F}^0 = \{\neg\mathtt{P}(\mathtt{X}:\mathtt{h})\}$; $\mathtt{F}^{\mathtt{i}} = \phi$ $(\mathtt{i}>0)$; $\mathtt{N}=1$
2. for each formula $\mathtt{Q}\in\mathtt{F}^{\mathtt{N}-1}$
   for each method $\sigma:\mathtt{wh}\to\mathtt{h}$, let
   $\mathtt{G}(\mathtt{X}:\mathtt{h})$ be $(\exists\mathtt{V}:\mathtt{w})\ \mathtt{Q}(\sigma(\mathtt{V},\mathtt{X}:\mathtt{h}))$ and do
   (a) if $(\forall\mathtt{X}:\mathtt{h})\ \mathtt{G}(\mathtt{X})\Rightarrow\mathtt{F}(\mathtt{X})$ is provable then skip
   (b) if $\mathtt{G}(\mathtt{I})$ is provable then exit with `failure`
   (c) if $\neg\mathtt{G}(\mathtt{I})$ is provable then add $\mathtt{G}$ to $\mathtt{F}^{\mathtt{N}}$ and skip
   (d) exit with `unknown`
3. if $\mathtt{F}^{\mathtt{N}}=\phi$ then exit with `success`
4. $\mathtt{N}=\mathtt{N}+1$; go to Step 2

The theorem prover is invoked when checking provability of the implication formulas. Since the task of first-order theorem proving is not decidable in general, it must be terminated when exceeding the predefined limit of computational resources such as the number of generated clauses and the time spent, in which case the result is reported as "unprovable".

By applying this model checking procedure to pairs of states and relations (rather than to states and predicates), one can perform coinduction. The initial pair of states will be the one for which the behavioral equivalence are in question, and the safety relation will be attribute equivalence (i.e., a set of pairs

with the same attribute values).

Refinement verification
Signature matching narrows the search by rejecting components that cannot meet the requirements at the syntax level. The next step is to check if the component has indeed the required functionalities. This is known as refinement verification. A purely model-theoretic definition is given below.

Definition 2 *A hidden signature morphism* $\varphi : (\Sigma, E) \to (\Sigma', E')$ *is a* ***refinement*** *iff for every* (Σ', E')*-algebra* M' *we have* $\varphi M' \models_\Sigma E$. *(*$\varphi M'$ *denotes* M' *viewed as a* Σ*-algebra.)*

It can also be shown that φ is a refinement iff all visible consequences of the source specification hold in the target specification [10].

Proposition 1 *A hidden signature morphisms* $\varphi : (\Sigma, E) \to (\Sigma', E')$ *is a refinement iff* $E' \models \varphi(c[e])$ *for each* $e \in E$ *and each visible* Σ*-context* c*, where if* e *is the equation* $(\forall X)\ t = t'$*, then* $c[e]$ *denotes the equation* $(\forall X)\ c[t] = c[t']$.

Therefore to check if a view $\varphi : (\Sigma, E) \to (\Sigma', E')$ is a refinement, one does the following:

- Apply φ to each equation e in E (denoted by $e * \varphi$) to obtain a translation e';
- If e is not behavioral, use ordinary equational reasoning to see if e' can be deduced from E';
- If e is behavioral, do coinduction for e' in (Σ', E') using methods only coming from (Σ, E) when calculating $\mathtt{pre}(\mathtt{p_n}(\mathtt{X}))$.

This procedure is also implemented in CafeOBJ using the PigNose engine.

Overview of Prototype System
An architecture of the prototype system is shown in Figure 2. The system consists of four units: 1) a trader, 2) CafeOBJ system with PigNose engine, 3) a specification repository, and 4) ORBs. The system is distributed in the sense that each unit can be located on any host. The user accesses the service from the trader through a web browser. The trader make arrangements between the user and the other units, passing requests and results. CafeOBJ and PigNose do the most important "engine part" job of the search such as signature matching, coinduction and refinement verification, and invariant checking if necessary. The specification repository stores specifications for the components registered to the ORBs as well as the information concerning the correspondence between specification and interface/ implementation maintained by the ORBs. This includes mappings of module names, operations, and sorts (types); and more importantly the object references of corresponding components on the ORBs. The last information is essential for remote execution. The ORBs accommodate components keeping them ready for execution, maintain interface/implementation repository, and take care of remote execution.

In the prototype system, the trader is implemented in Java and the specification repository is designed using XML for future enhancement. JavaIDL is used for the ORB as mentioned earlier. Only standard communication protocols are used such as http, CGI, TCP/IP sockets, and Unix pipes.

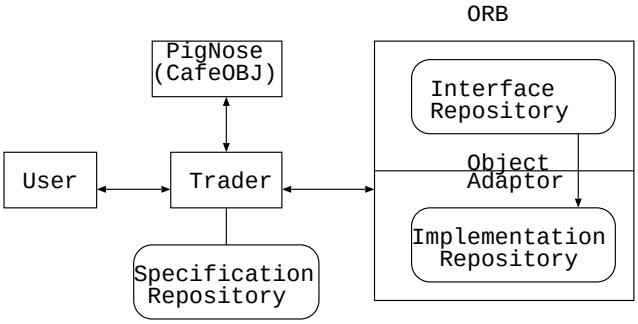

Figure 2: Search system overview

Use Scenario
A typical use scenario of the system looks as follows. This is also a rough specification for the prototype system.

The components registered to the ORBs must have corresponding specifications in the specification repository and the entries for the name server. The user issues a search request with a requirement specification through a web browser. Then the trader passes this information to PigNose. PigNose responds with a list of views if signature match is successful. The trader receives the result and displays it on the user's web browser. The user is presented with an options to check refinement for each view constructed by PigNose signature matching. If the user asks for refinement verification, the trader again passes this information to PigNose. PigNose this time responds with yes/no answer. The trader displays the results on the web browser.

The user can always ask for information about candidate specifications/components. This includes the information stored in the specification repository, code, names and object references. The user also can examine properties about the candidate specification by issuing CafeOBJ commands such as invariant checking.

If the user is satisfied, he or she can test the component by remote invocation through the ORB. The trader looks into the specification repository and interface repository to get the the object reference (via the name server), the number and types of the argument, and return value types for each method of the requested component. The trader prepares a GUI page on the user's web browser in which the user chooses the

method to invoke providing appropriate arguments. With this information, the trader prepares a request data for dynamic invocation interface (known as marshalling) and passes it to the ORB. The ORB delivers the request to the components implementation, receives the return value, and passes it to the trader. Trader again displays the result on the browser.

AN EXPERIMENT

In this section, we describe a simple experiments using the prototype system.

Container Class Search

This experiment is to find container components whose newly added element is always visible. A number (thirteen) of container specifications such as stack, list, one-capacity buffer, cell and queue are registered in the specification repository.

```
mod* CONTAINER(X :: TRIV) {
  *[ Container ]*
  op empty : -> Container
  bop store : Elt Container -> Container
  bop val : Container -> Elt
  var E : Elt   var C : Container
  eq val(store(E,C)) = E .
}
```

Figure 3: Requirement CONTAINER specification

We first give the requirement specification CONTAINER shown in Figure 3. Figure 4 shows the user interface for this, in which the user selects the file containing the requirement specification, provides the module name, and pressed the search request button.

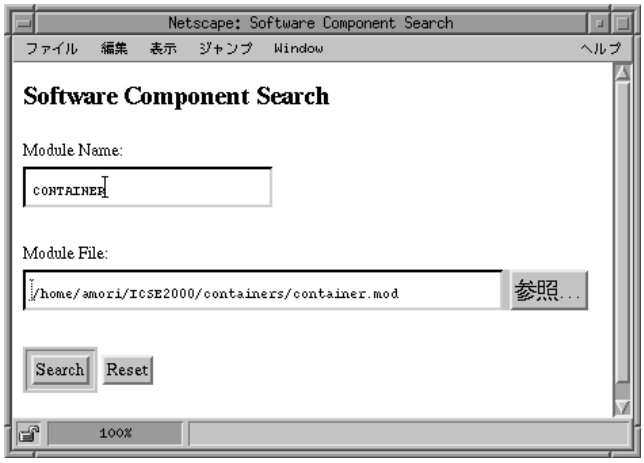

Figure 4: Search user interface

The result is shown in Figure 5. Six modules has passed the signature matching, but BUF and QUEUE 6 do not pass the requirement verification, which is not surprising since one cannot see the new element in a queue unless it has only one element, and a buffer may not have a “new” element if it was full.

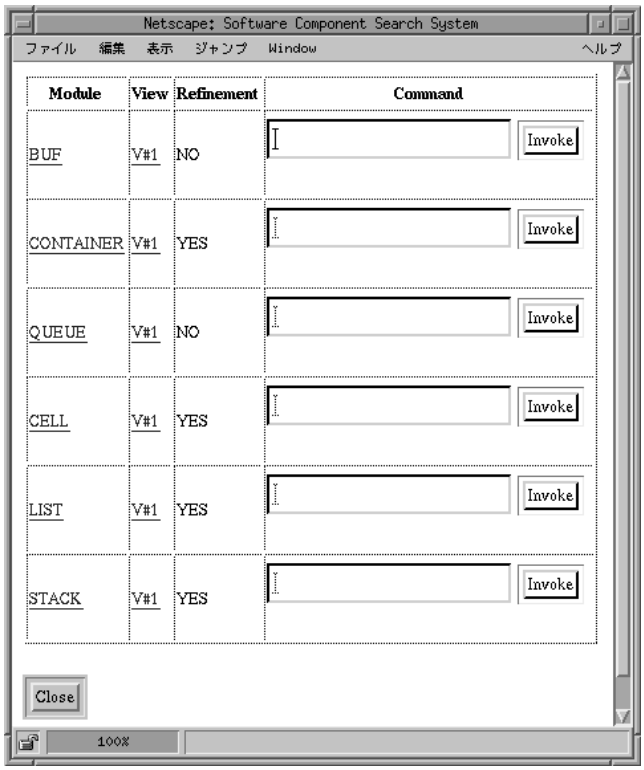

Figure 5: Search results

Note that the above process is completely automatic and no extra work is needed from the user. To see what information is exchanged between the trader and CafeOBJ (PigNose), we show in Figure 7 the CafeOBJ session of the signature matching and refinement verification between CONTAINER and QUEUE.

```
mod* QUEUE(X :: TRIV) {
  *[ Queue ]*
  op empty : -> Queue
  bop front : Queue -> Elt
  bop enq : Elt Queue -> Queue
  bop deq : Queue -> Queue
  vars D E : Elt   var Q : Queue
  beq deq(enq(E,empty)) = empty .
  beq deq(enq(E,enq(D,Q))) = enq(E,deq(enq(D,Q))) .
  eq front(enq(E,empty)) = E .
  eq front(enq(E,enq(D,Q))) = front(enq(D,Q)) .
}
```

Figure 6: QUEUE specification

```
CafeOBJ> sigmatch (CONTAINER) to (QUEUE)
(V#1)
CafeOBJ> show view V#1
view V#1 from CONTAINER(X) to QUEUE(X) {
  sort Elt -> Elt
  hsort Container -> Queue
  hsort ?Container -> ?Queue
  op (Container : -> SortId) -> (Queue : -
> SortId)
  op (Elt : -> SortId) -> (Elt : -> SortId)
  op (_=*=_ : Container Container -> Bool) ->
     (_=*=_ : _ HUniversal _ _ HUniversal _ -
> Bool)
  op (empty : -> Container) -> (empty : -> Queue)
  bop (val : Container -> Elt) -> (front : Queue -
> Elt)
  bop (store : Elt Container -> Container) ->
      (enq : Elt Queue -> Queue)
 }
CafeOBJ> check refinement V#1
no
  eq val(store(E,C)) = E
CafeOBJ>
```

Figure 7: CafeOBJ session for container class search

CONCLUSION AND FUTURE WORK

In this paper, we have reported on a project to develop a search engine for software components hosted by object request brokers. We advocate using behavioral specification based on hidden algebra for automation such as signature matching, refinement verification and model checking. Although the project is still in an early stage, we have gained positive experience through the use of the prototype system presented in the paper.

We have received criticism about the cost of formal specification and the scalability of the methods. As for the cost, we regard behavioral specification just as a sophisticated form of programming and believe that anyone who wishes to offer software in public should be able to write specifications. Moreover, behavioral specification does not require complete specification and one could just write a partial signature and a few equations that is absolutely critical for the functionality of the software. Although we do not have a firm answer for the scalability question, we like to stress that our goal is not to make a miracle but to offer tools that can do simple tasks automatically. We believe that the search engine for software components is a plausible target in this regard.

As for future plans, there are many things to be worked on. Among them we are currently looking at the following topics:

- An IDL compiler for CafeOBJ. This involves further examination of the CafeOBJ/IDL mapping.
- How to maintain module constructs such as parameterized modules for the component implementations on ORBs. This includes the study of remote execution mechanism for compound components.

REFERENCES

1. Cook, W., Hill, W., Canning, P.: Inheritance is not Subtyping. Proc. of *17th ACM Symp. on Principles of Programming Languages POPL'90* (1990) 125–135
2. Bartlett, K., Scantlebury, R., Wilkinson, P.: A Note on Reliable Full-duplex Transmission over Half-duplex Links. *Communication of the ACM* **12(5)** (1969) 260–261
3. Bidoit, M., Hennicker, R.: Observer Complete Definitions are Behaviourally Coherent. Research Report LSV-99-4 Lab. Specification and Verification, ENS de Cachan (1999)
4. Diaconescu, R., Futatsugi, K.: CafeOBJ *Report*. World Scientific (1998)
5. Diaconescu, R: Behavioural Coherence in Object-oriented Algebraic Specification. Technical Report IS-RR-98-0017F, Japan Advanced Institute of Science and Technology (1998)
6. Goguen, J., Meseguer, J., Luqi, Zhang, D., Berzins, V.: Software Component Search. *Journal of Systems Integration* **6** (1996) 93–134
7. Goguen, J., Malcolm, G.: A Hidden Agenda. To appear in *Theoretical Computer Science*, also available as Technical Report CS97-538, Computer Sci.& Eng. Dept., Univ. of Calif. at San Diego (1997)
8. Futatsugi, K., Nakagawa, A.: An Overview of CAFE Specification Environment: an algebraic approach for creating, verifying, and maintaining formal specification over networks, Proc. of *First IEEE Int'l. Conf. on Formal Engineering Methods* (1997)
9. Jacobs, B.: Inheritance and Cofree Constructions. Proc. of *European Conference on Object-Oriented Programming ECOOP'96 Lecture Notes in Computer Science* **1098** (1996) 210–231
10. Malcolm, G., Goguen, J.: Proving Correctness of Refinement and Implementation. Technical Monograph PRG-114, Programming Research Group, University of Oxford (1994)
11. Mori, A., Futatsugi, K.: Verifying Behavioural Specifications in CafeOBJ Environment. Proc. of *World Congress on Formal Methods FM'99, Lecture Notes in Computer Science* **1709** (1999) 1625–1643
12. McCune, W.: OTTER 3.0 Reference Manual and Guide. `http://www-unix.mcs.anl.gov/AR/otter/`
13. Rosu, G., Goguen, J.: Hidden Congruent Deduction. To appear in *Lecture Notes in Artificial Intelligence* (1999)
14. Zaremski, A.M., Wing, J.M.: Signature Matching: a Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **4(2)** (1995) 146–170
15. Cousot, P., Cousot, R.: Refining Model Checking by Abstract Interpretation. *Automated Software Engineering Journal* **6(1)** (1999) 69–95