SEIM: Static Extraction of Interaction Models

Leonardo Mariani
Dept. of Inform. Systems and Communication
University of Milano Bicocca
20126, Milan, Italy
mariani@disco.unimib.it

Mauro Pezzè*
Dept. of Inform. Systems and Communication
University of Milano Bicocca
20126, Milan, Italy
pezze@disco.unimib.it

Oliviero Riganelli
School of Science and Technology
University of Camerino
62032, Camerino, Italy
oliviero.riganelli@unicam.it

Mauro Santoro
Dept. of Inform. Systems and Communication
University of Milano Bicocca
20126, Milan, Italy
santoro@disco.unimib.it

ABSTRACT

The quality of systems that integrate Web services provided by independent organizations depends on the ways the systems interact with the services, i.e., on their interaction protocols, which are not always easy to deduce by inspecting the code. Accurate models of interaction protocols provide a comprehensive view of the interactions, and support manual and automatic analysis of corner cases that are often difficult to discover, and are responsible from many subtle failures.

Models of program behavior can be extracted either statically from the source code or dynamically from execution traces. Dynamic techniques cannot reveal behaviors that have not been executed, and thus dynamic models are partially useful to identify critical corner cases. Static techniques produce models of the whole execution space, and thus static models are more useful to find critical cases. Most static techniques to extract models of the code behavior focus on the usage protocol, i.e., how the services can be used, rather than the interaction protocol, i.e., how applications use services. While usage protocols can help software engineers understanding the service behavior, interaction protocols describe the subset of actual interactions between the system and the integrated services, and provide the information required to understand critical interactions.

In this paper, we present SEIM, a technique to extract models of the service interaction protocol. SEIM extracts models statically, and includes a novel refinement strategy to eliminate infeasible behaviors that reduce the usability of statically derived models. This paper describes the technique and provides some empirical data from the early experience of SEIM with an application that interacts with the eBay Web Services.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.2.11 [**Software Engineering**]: Software Architectures—*Service-oriented architecture (SOA)*

General Terms

Verification, Algorithms

Keywords

model extraction, interaction model, static analysis

1. INTRODUCTION

Service-based solutions are increasingly popular for developing loosely-coupled business-to-business service integrations in distributed cross-organization systems. Despite the many service-based technologies and platforms that support service integration, developing applications that involve non-trivial service interactions is difficult and error-prone, and it is hard to avoid all the incompatibilities between clients and services that may cause subtle failures [19]. Common interaction patterns are easy to design, implement and test, but understanding and identifying all corner cases, which can result in unexpected and sometimes erroneous interactions, can be extremely hard.

Accurate models of the interaction protocols, i.e., models of the interactions between applications and the integrated Web services, can facilitate both manual and automatic inspection and analysis, and can support the many existing model-based verification and validation techniques [12]. Interaction models can be extracted either statically from the source code or dynamically from a set of execution traces.

Dynamically inferred models generalize a set of observed behaviors, i.e., a set of interactions that have been executed and monitored. They provide a compact but incomplete representation of the system behaviors, since they miss behaviors unrelated with the observed traces. Thus, they are partially useful for the identification of both the untested and the rare spurious cases [15, 5, 14].

Statically extracted models represent the behavior of the system independently from the subset of observed traces,

*Mauro Pezzè is also professor at the University of Lugano, Faculty of Informatics, via Buffi, 13 6900 Lugano (Switzerland).

PESOS '10, May 1-2, 2010, Cape Town, South Africa

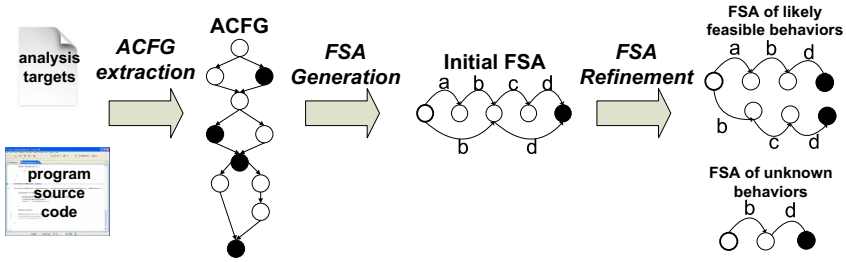

Figure 1: The main steps of the SEIM technique

and thus are more useful to identify rare corner cases and undesired interactions. However, static models techniques can include infeasible behaviors that can produce many false positives.

So far, static analysis techniques have focused mostly on the derivation of models that represent how components and services can be used (models of the component usage protocols) [1, 23], rather than how specific applications use components and services (models of the interaction protocols). An interesting contribution to the generation of interaction protocols is the technique defined by Wasylkowski et al. that statically learns how objects can be used from actual usages that occur in the application [24]. This technique discovers the interaction patterns that can occur within single methods with an intra-procedural analysis, but it does not support inter-procedural analysis, thus it cannot discover interactions that derive from the execution of multiple methods. This limitation can be relevant in the context of service-based applications because applications usually interact with Web Services by executing multiple calls spread among several methods.

In this paper we present SEIM, a novel static analysis technique that derives accurate models of the interactions between applications and the Web Services integrated in them, in the form of Finite State Automata (FSA) [21]. SEIM contributes to the state-of-the-art in two major ways: it proposes a model refinement technique to identify and eliminate many infeasible behaviors from inferred models, thus alleviating the problem of false positives that reduces the effectiveness of many static analysis approaches; it generates models that distinguish the likely feasible interactions from the interactions with an unknown level of feasibility, thus allowing engineers to distinguish the relevance of the produced information.

SEIM is specifically effective for service-based applications thank to the following aspects that facilitate the static extraction of the interaction models:

Independent states: client applications cannot share their state with Web Services, i.e., executing Web Service operations does not alter the state of the client application;

Easily identifiable services: in client applications, the interactions with Web Services are usually mediated by stubs that are easily automatically identifiable;

Easily distinguishable services: client applications usually instantiate a different type of stub for each used Web Service, thus, the stub type identifies the target Web Service.

SEIM derives a model of the interactions between a client application and the integrated Web Services by deriving an initial FSA that accepts a superset of the possible system behaviors, and then refining the initial FSA into two FSAs, according to the feasibility of the interactions. One of the refined FSA accepts only likely feasible interactions, while the other refined FSA accepts behaviors with an unknown level of feasibility. The refinement step eliminates interactions that are identified as infeasible.

This paper is organized as follow. Section 2 overviews the SEIM static analysis technique and identifies the three main phases that are described in the next sections. Section 3 describes the extraction of the Annotated CFG (ACFG), a control flow graph annotated with information useful to the generation of interaction models. Section 4 presents the generation of the initial FSA from the ACFG. Section 5 describes the refinement process that removes infeasible behaviors from the FSA and produces two refined FSAs. Section 6 frames the contribution of this paper in the context of the current research. Finally, Section 7 summarizes the results achieved so far and outlines our research agenda.

2. OVERVIEW OF SEIM

The SEIM technique produces an FSA model of the requests that the application under analysis produces when interacting with a set of Web Services. SEIM works in three main steps, as shown in Figure 1. At the time of writing, SEIM analyzes only synchronous flows of requests, and does not consider asynchronous and event-based interactions.

In the first step, SEIM derives an Annotated Control Flow Graph *(ACFG extraction)*. Software engineers indicate the *program to be analyzed* and the *analysis targets*, which restrict the set of service operations that can be included in the inferred model. This step produces an ACFG, that is an inter-procedural control flow graph of the program under analysis, where the nodes that represent service invocations are annotated with the name of the invoked operations. This model includes all the service invocations and the information about the possible execution flow, and it is the basis of our analysis.

In the second step, SEIM generates the initial FSA (*FSA generation*). It first reduces the ACFG by removing all the nodes and the transitions that are not relevant for the analysis. In particular, it preserves only the nodes and transitions that are related to either calls to service operations or to the application control flow. It then transforms the reduced ACFG into an equivalent FSA.

In the third step, SEIM prunes infeasible sequences of calls, and generates two FSAs that distinguish likely feasible

```
...
if (isInWatchList != true) {
   addToWatchList(itemToBuy);
}
ItemType biddenItem;
biddenItem = getItem(itemToBuy);
if (isInWatchList != true) {
   if(biddenItem.getSellingStatus().
        getMinimumToBid().getValue() <= maxBid) {
      placeOffer(itemToBuy, maxBid);
   } else {
      removeFromWatchList(itemToBuy);
   }
} else {
   if(!(biddenItem.getSellingStatus().
        getHighBidder().getUserID().equals(getUser
        ()))){
      placeOffer(itemToBuy, maxBid);
   } else {
      ...
   }
}
...
```

Figure 2: An excerpt of method `getItemByBid`.

sequences of calls from sequences of calls whose feasibility is not known (*FSA refinement*). One of the FSA produced by SEIM accepts only likely feasible sequences of calls, while the other FSA accepts sequences of calls whose feasibility is not known. SEIM marks a sequence of calls as a sequence with unknown feasibility when it cannot determine if there exists a concrete execution that corresponds to that sequence.

In the next sections, we detail the three main steps of the technique through a running example, a Java application that integrates eBay Web Services to provide searching and trading operations [8]. We present the analysis of two methods implemented by this application, and we show how SEIM can extract a precise representation of the interactions between the application and the eBay Web Services, related to the execution of these methods. The two methods that we analyze are `findBestExpiringItem` and `getItemByBid`. The method `findBestExpiringItem` searches an eBay auction and finds the product with the lowest cost among the ones that satisfy the following requirements: they match the description passed as parameter, have an actual cost below a given threshold, and are offered by sellers with positive feedbacks. The method `getItemByBid` adds the product passed as argument to the user watchlist, and makes an offer for the item to beat the best offer. Figure 2 shows an excerpt of the method `getItemByBid`. The total number of lines of codes analyzed by SEIM in the running example is about $1K$.

3. ACFG EXTRACTION

In the ACGF extraction step, SEIM produces an annotated control flow graph of the target system. The inputs of this step are the component to be analyzed[1] and the analysis targets that include both a list of *stub types* and a *usage protocol*.

SEIM interprets interactions with *stubs* as interactions with the Web Services, i.e., an invocation of a method implemented by a stub represents an invocation of a Web Service operation. The *usage protocol* is given as a FSA, where transitions are labeled with the names of methods implemented by the interface of the component under analysis.

In our running example, the stub types are `EBayAPIInterface` and `ShoppingInterface`, the interface methods that are analyzed are `findBestExpiringItem` and `getItemByBid`, and the usage protocol is specified by the FSA shown in Figure 3. Software engineers are welcome to specify other usages of interest. They may for example look for invocations executed within a loop or for the independent execution of the two methods under analysis.

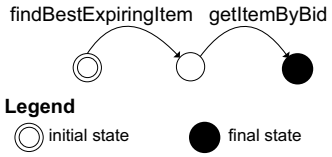

Figure 3: Usage protocol for the interface methods analyzed in the running example.

SEIM generates the ACFG by first generating the Inter-Procedural Control Flow Graph (ICFG) of each interface method, and then combining the ICFGs according to the usage protocol. The generation of the ACFG differs from the standard generation of an ICFG in the way the methods implemented by the stub objects are analyzed. When an invocation of a method implemented by a stub object is considered, SEIM does not generate the ICFG corresponding the method, but simply annotates the node that corresponds to the method invocation with the name of the invoked method. SEIM does does not add the ICFG of the method to the ICFG under construction either. Note that the construction of the ACFG requires the analysis of all the methods invoked from the interface methods. We implemented the ACFG extraction step for Java programs by extending the SOOT toolkit [10]. The ACFG extracted from the running example is too large to be shown in the paper. Figure 4 shows a reduced ACFG.

SEIM combines the ACGFs of the single methods by simply replacing the transitions in the usage protocol given as input with the ACFG of the methods that correspond to the label of the transitions. In particular, given a transition t from a state s_1 to a state s_2 labeled with an interface method m, and the ACFG $acfg(m)$ extracted from the analysis of the method m, SEIM removes the transition t, merges s_1 with the entry node of $acfg(m)$, and merges s_2 with the exit node of $acfg(m)$.

4. FSA GENERATION

In this step, SEIM transforms the ACFG produced in the former step into an FSA that represents the interactions between the component under analysis and the target services.

SEIM starts by eliminating data that are irrelevant for the analysis. The only data in the ACFG that are relevant for identifying sequences of requests are invocation and control nodes. Invocation nodes represent the invocation of Web service operations and are annotated with the operation names. Control nodes represent non-sequential execution flows, and are the nodes with more than 1 incoming edge or more than

[1]In the rest of the paper we use the general term *component* because SEIM can analyze software units of different granularity, including entire programs

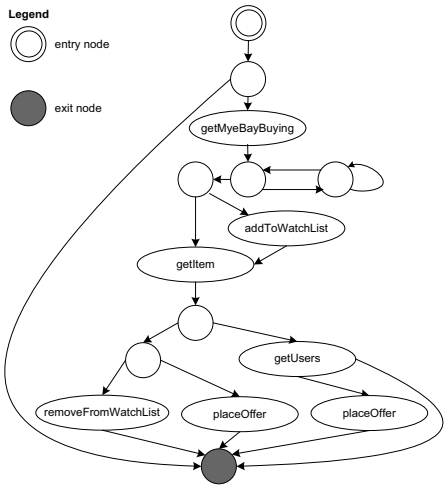

Figure 4: The reduced ACFG for the `getItemByBid` method.

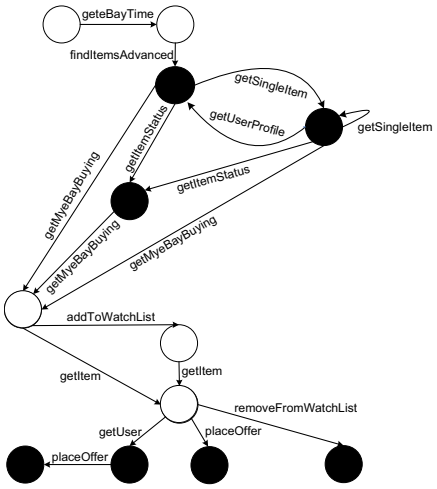

Figure 5: The interaction protocol produced for the running example.

1 outgoing edge. SEIM removes from the ACFG all nodes except invocation and control nodes. Figure 4 shows the reduced ACFG for the `getItemByBid` method. We do not show the reduced ACFG for the composition of the two methods considered as running example for space constraints.

After eliminating irrelevant nodes, SEIM transforms the ACFG into an equivalent FSA. An ACFG is a Moore machine [17], i.e., an automaton whose outputs depend only on the state. We can produce an equivalent FSA, i.e., an FSA that accepts the same language accepted by the ACFG, in few trivial steps. We first add a new final state to the ACFG. We then connect all former final states of the ACFG to the newly added final state with transitions, and we label all the transitions with the label of the source node, if not empty, or with the empty label ϵ otherwise. We finally transform the FSA in a deterministic one. The obtained FSA represents the interactions between the component under analysis and the Web Services when the component under analysis is executed according to the usage protocol passed as parameter to SEIM. Figure 5 shows the FSA obtained for the running example.

5. FSA REFINEMENT

The FSA produced by the second step over-approximates the behavior of the target component: it models the possible interactions with the specified Web services along with many infeasible ones. In the third step, SEIM ranks sequences of operations as infeasible, likely feasible, and with unknown feasibility, eliminates the infeasible sequences, and split the FSA into two FSAs that distinguish likely feasible sequences from sequences with unknown feasibility.

In a nutshell, the process works as follow. SEIM generates a finite set of sequences from the current FSA, passes these sequences to a *concrete execution discovery engine* that provides information about the feasibility of the sequences, and refines the FSA according to the collected information. A concrete execution discovery engine is any technique that can identify concrete executions that satisfy certain properties. In our experiments, we considered the property that consists of covering specific sets of statements an exact number of times and in a given order. The readers may have already noticed that this criterion is stronger than statement coverage, but weaker than path coverage, because it ignores the coverage of sub-paths that do not include calls to Web services. The effectiveness of the concrete execution discovery engine is extremely important for the effectiveness of SEIM.

Several static and dynamic analysis techniques can play the role of concrete execution discovery technique, for instance concolic execution [22], symbolic execution [18], and random testing [20]. The current SEIM prototype implementation uses the JPF symbolic executor [18] as concrete execution discovery engine.

In the following we illustrate how SEIM generates a finite set of sequences from the current FSA, how SEIM obtains information about the feasibility of the sequences, how SEIM refines the FSA according to the collected information, and the result obtained for our running example.

Sequence Generation The concrete execution discovery engine (JPF) works on a finite set of behaviors (sequence of calls to Web Service operations) that cover a given set of FSA elements. SEIM is not bounded to a specific coverage criterion, but can use different criteria depending on the goals. In SEIM, the goal is to identify and eliminate as many infeasible behaviors as possible within reasonable execution boundaries. We thus decided to cover not only all the statements and the elementary control flows of the FSA (node and statement coverage), but also a sample of repeated subbehaviors (loop coverage). In this way, SEIM collects enough information to successively eliminate infinitely many infeasible behaviors from the FSA, when the unbound repetition of some loops in the FSA is infeasible. In the running example, we generated a set of behaviors that covers all loops of the FSA that traverse the same state no more than L times. Our early experience with SEIM indicates that with $L = 3$ SEIM can already identify and eliminate many infeasible behaviors efficiently. The choice of a proper value for L may depend on the application domain and the nature of the code. Tuning the choice of the coverage criterion and the parameter values is part of our ongoing research work.

For the running example, SEIM produces 252 call sequences (behaviors to be checked for feasibility) that cover all loops up to a depth of 3 ($L = 3$).

Feasibility Analysis To determine the feasibility of a given sequence of calls, SEIM attempts to generate a concrete execution that traverses the statements that call the Web Services the number of times and in the order indicated by the sequence. In the following, we indicate these statements as call statements for brevity. SEIM uses JPF to explore the set of behaviors extracted from the FSA according to the chosen coverage criterion. Here we illustrate the process by referring to a single sequence of statement calls. Generalizing the process to a finite set of sequences complicates the presentation but does not include complex technical steps. To check for the feasibility of a sequence of call statements, SEIM modifies the program under analysis by introducing a string variable that represents the sequence of call statements traversed by the current execution. SEIM initializes the string to an empty value, and incrementally adds to the string the call statements traversed while executing the program with JPF. JPF drives the symbolic exploration of the execution space by trying to produce a string that matches the target searched sequence. Thus, when the string variable added to the program does not match the prefix of the target sequence, JPF can immediately exclude the corresponding portion of the execution space. For instance, if SEIM looks for a sequence with calls to `m1`, `m2` and `m3`, JPF looks for executions that produce a value `"m1 m2 m3"` for the added string. When the value of the string is inconsistent with the searched value, for example `"m1 m3"`, JPF can exclude the corresponding portion of the execution space without further exploring it.

In summary, the overhead introduced by JPF is limited by three main factors: SEIM invokes JPF only once for the whole set of sequences to be analyzed, limits the exploration of the execution space by binding the number of loop executions to L, and prunes the portions of the space to be explored according to the given sequences as discussed above.

For each call sequence, the concrete discovery engine, in our case JPF, can produce three results: feasible, infeasible or unknown. SEIM classifies a call sequence as feasible, if JPF finds at least a concrete execution that matches the sequence. SEIM classifies a call sequence as infeasible, if JPF can show that there exists no concrete execution that matches the sequence. SEIM classifies a call sequence as unknown, if JPF does not reach a conclusion. Typically, this happens when JPF does not terminate because it cannot handle some symbolic expressions derived during the exploration of the execution space.

Refinement SEIM uses the information about infeasible and unknown sequences to generate two FSAs representing these two classes of sequences. Here we illustrate the process of building the two FSAs. We first build two Prefix Tree Acceptors (PTA): `I-PTA` for infeasible sequences, and `U-PTA` for unknown ones, with the classic algorithm described by Bierman and Feldman in [3]. Paths in a PTA represent single executions with no loops, thus simply refining the FSA produced after the second step by removing the behaviors accepted by the `I-PTA` and the `U-PTA` can eliminate at most a finite set of infeasible behaviors. To generalize the finite set of executions and prune a possibly infinite set of executions, SEIM transforms the PTAs in FSAs by using the heuristic of the kTail inference algorithm proposed by Bierman and Feldman [3]. The kTail heuristics merges sets of likely equivalent states that are defined as pairs of states that accept the same behaviors up to length k. This heuristic produces two automata that include loops and other complex behaviors: `IA` from `I-PTA`, and `UA` from `U-PTA`, respectively. Since SEIM explores concrete executions by exploring loops up to length L, it makes sense to assign a similar value to the k parameter of the kTail heuristic. In the running example, we use $k = 4$.

SEIM uses `IA` and `UA` to refine the automaton `A` produced after the second step into two final automata `F-Aut` and `U-Aut`. The automaton `F-Aut` accepts the likely feasible behaviors of the program under analysis, and is defined as $\texttt{F-Aut} = (\texttt{A} \setminus \texttt{IA}) \setminus \texttt{UA}$. The automaton `U-Aut` accepts only the behaviors with unknown feasibility, and is defined as $\texttt{U-Aut} = \texttt{UA} \cap \texttt{A}$. In the formula above $\cap$ represents the intersection between automata, and $\texttt{A} \setminus \texttt{B}$ is defined as $\texttt{A} \cap \overline{\texttt{B}}$, where $\overline{\texttt{B}}$ is the complement operator [11]. The two automata provide a detailed representation of the behaviors of the component under analysis, together with important information about the feasibility of the behaviors.

Results from the Running Example In our running example, JPF terminates the analysis of all the 252 sequences, thus it does not find sequences with unknown feasibility. This result depends on the ability of JPF to handle all the statements in the component under analysis, but is not necessarily true for an arbitrary program. JPF classifies 198 sequences as infeasible, and 54 as feasible. The large amount of infeasible sequences, which represent behaviors that appear to be feasible from the ACFG, but are not feasible when analyzing the concrete execution conditions in the code, depends on restrictions in the combination of methods. As in many industrial cases, several behaviors that are feasible at the component level depend on the specific combination of the single units. The ACFG simply combines different behaviors according to the static structure of the code and thus cannot reveal the many infeasible combinations. On the contrary, SEIM refines the initial ACFG with an estimation of the feasibility of the call sequences, and can correctly identify many infeasible combinations. Thus, it produces a more accurate model.

For example, one of the execution flows of method `findBestExpiringItem` returns an empty string that prevents further requests to Web services in method `getItemByBid`. The ACFG contains many infeasible sequences that include a call to method `findBestExpiringItem`, which returns an empty string, followed by calls to method `getItemByBid`, while SEIM prunes all these infeasible sequences.

Figure 6 shows the refined model produced for the running example. The readers may notice that the FSA in Figure 5 seems to be easier to inspect by software engineers because of its regularity and correspondence to the code. However, the FSA before refinement shown in Figure 5 is less precise than the refined FSA in Figure 6, and thus the FSA before refinement is much less useful than the refined FSA for verification and validation. To confirm this intuition, we measure the quality of the model extracted by SEIM in the running example in terms of precision and recall before and after the refinement. We obtained the data with the QUARK framework [13] that can automatically generate traces from an inferred and a reference model, and compute precision and recall with respect to the FSA that exactly matches the behavior of the program. Table 1 summarizes the results.

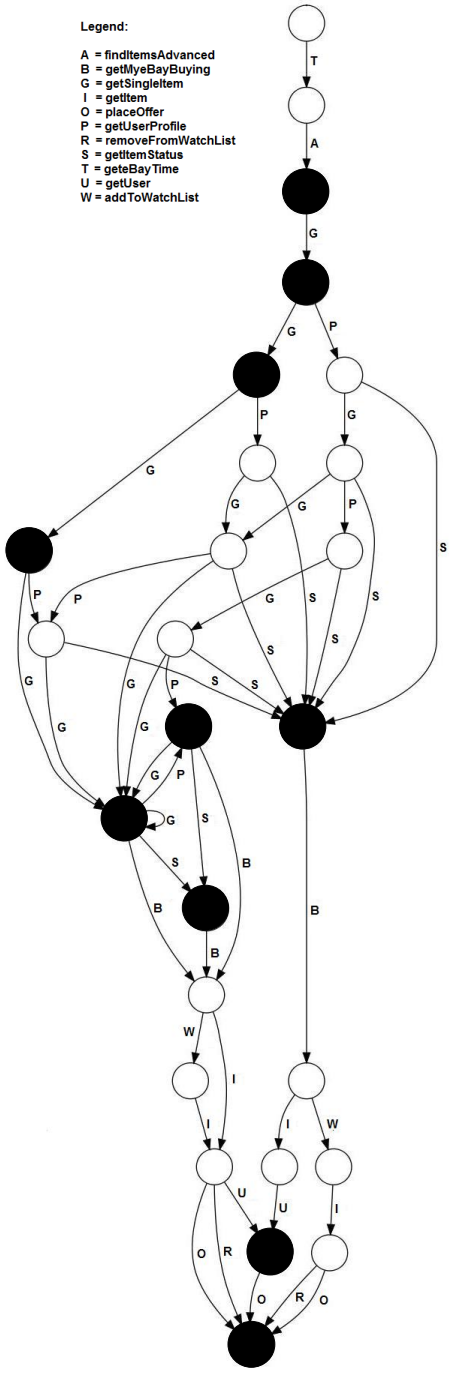

Figure 6: The refined FSA for the running example.

The precision and recall of the FSA built before the refinement step confirm that this FSA is a complete (recall = 1), but imprecise (precision = 0.32) representation of the possible interactions of the component under analysis. The precision and recall of the refined FSA confirm that the SEIM effectively removes most infeasible sequences (precision = 0.9) missing only few correct behaviors (recall = 0.92) The absence of an FSA with unknown behaviors in the case study indicates that the SEIM refinement step has been precise.

	precision	recall
FSA before refinement	0.32	1
FSA after refinement	0.9	0.92

Table 1: Precision and Recall.

6. RELATED WORK

SEIM is a static analysis technique that generates models for systems that integrate Web Services. There are several other techniques to generate interaction and behavioral models from programs. In this section, we discuss the main static, dynamic and hybrid analysis approaches, and we compare them with SEIM.

Static Analysis Most static analysis approaches to generate interaction models produce models that describe how services can be used rather than how applications use services. Good examples of recent work in this area are the proposals of Shoham et al. who derive models of the usage protocol for security analysis [23] and of Wasylkowski et al. who focus on bug localization [24].

These techniques deal with the problem of infeasible behaviors in the models by either limiting the scope of the analysis, for example by limiting the analysis to single methods, or by developing analysis techniques that tolerate big amounts of infeasible behaviors at the cost of an increasing number of false alarms.

SEIM infers models without limiting the scope of the analysis and with a small amount of infeasible behaviors by pairing model generation with a model refinement step. In the current prototype implementation of SEIM, we refine models with static analysis (relying on JPF), but the SEIM model refinement step can take advantage of many other techniques. We are currently investigating the use of dynamic analysis and random testing to better cope with limitations that may derive from complex source code. Moreover, SEIM augments the generated models with information that distinguishes sequences that are likely feasible from sequences whose feasibility is not known. This information may be precious for software engineers who can decide the focus of verification and validation activities based on the model.

Other work propose static analysis techniques to infers properties that are complementary to interaction protocols. For example, de Caso et al. generate models of operation contracts [6], while Bertolino et al. generate models of operation dependencies [2]. These aspects are important complement to the models of interaction protocols derived by SEIM. Our research agenda includes the study on how models produced by SEIM can be augmented with information about contracts and data dependencies.

Dynamic Analysis Several techniques have been proposed to derive different kinds of models from program traces. For example, Ernst et al. derive method contracts [9], Lorenzoli et al. and Lo et al. derive interaction protocols [15, 14], Dallmeier et al. derive object usage protocols [5]. These techniques produce compact models of the observed behaviors, and can generalize the observed behaviors to identify similar ones. However, these techniques cannot capture interactions that have not been observed. Thus, they do not help engineers in identifying corner cases and undesired interactions before they cause the failures. On the contrary, static analysis techniques, like SEIM, produce models that describe all possible interactions, and thus can more easily

identify corner cases and possible undesired interactions.

Hybrid Analysis Hybrid analysis techniques combine static and dynamic information to analyze software programs. The rationale underlying these approaches is to leverage the weak aspects of a technique with the complementary one. Relevant representatives of hybrid techniques are solutions to automatically identify faults that cause crashes in Java applications like DSDCrasher [4], solutions to automatically identify violations of contracts in component-based software [16], and solutions to automatically identify behaviors of concurrent systems [7]. Hybrid solutions have not been investigated to discover interactions with Web Services yet. So far we paired SEIM with JPF to identify infeasible sequences of calls to Web services, but SEIM can be paired with other techniques, and we are currently investigating some promising dynamic analysis techniques as well.

7. CONCLUSIONS

The design and implementation of the interactions with Web Services can be tricky and error-prone. Applications that use Web services can generate undesired interactions for rare executions that are difficult to observe.

In this paper, we presented SEIM, a static analysis technique that can extract interaction models from a program. SEIM pairs a generation step with a refinement step to identify and eliminate many infeasible elements from the static model. Moreover, SEIM generates also information about the precision of the identified behaviors by distinguishing likely feasible from interactions whose feasibility is not known. Currently SEIM relies on JPF to identify infeasible sequences, but can be easily paired with other static as well as dynamic analysis techniques to obtain better results.

Our research agenda includes the use of dynamic analysis to improve model refinement and the use of the extracted models in verification, validation and conformance analysis of service-based applications.

8. REFERENCES

[1] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. *SIGPLAN Notice*, 40(1):98–109, 2005.

[2] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *proceedings of the ESEC/FSE*, 2009.

[3] A. Biermann and J. Feldman. On the synthesis of finite state machines from samples of their behavior. *IEEE Transactions on Computer*, 21:592–597, 1972.

[4] C. Csallner and Y. Smaragdakis. DSD-crasher: a hybrid analysis tool for bug finding. In *proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2006.

[5] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *proceedings of the International Workshop on Dynamic Analysis*. ACM, 2006.

[6] G. de Caso, V. A. Braberman, D. Garbervetsky, and S. Uchitel. Validation of contracts using enabledness preserving finite state abstractions. In *proceedings of the International Conference on Software Engineering*, 2009.

[7] L. Duarte, J. Kramer, and S. Uchitel. Towards faithful model extraction based on contexts. In *proceedings of the International Conference on Fundamental Approaches to Software Engineering*, number 4961 in LNCS, 2008.

[8] eBay Inc. ebay web services. http://developer.ebay.com/, visited in 2010.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[10] S. R. Group. Soot. http://www.sable.mcgill.ca/soot/, visited in 2010.

[11] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[12] R. Hull and J. Su. Tools for composite web services: a short overview. *SIGMOD Record*, 34(2), 2005.

[13] D. Lo and S. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *proceedings of the Working Conference on Reverse Engineering*, 2006.

[14] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *proceedings of the ESEC/FSE*, 2009.

[15] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *proceedings of the International Conference on Software Engineering*. ACM, 2008.

[16] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *proceedings of the ESEC/FSE*. ACM, 2003.

[17] E. F. Moore. *Sequential Machines: Selected Papers*. Reading. Addison Wesley, 1964.

[18] NASA. Java pathfinder. http://babelfish.arc.nasa.gov/trac/jpf, visited in 2010.

[19] H. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *proceedings of the International Conference on World Wide Web*. ACM, 2007.

[20] C. Pacheco, S. K. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *proceedings of the International Conference on Software Engineering*, 2007.

[21] M. Santoro. Detecting precise behavioral models. In *proceedings of the ESEC/FSE Doctoral Symposium*, 2009.

[22] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *proceedings of the International Conference on Computer Aided Verification*, LNCS. Springer, 2006.

[23] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *proceedings of the International Symposium on Software Testing and Analysis*, 2007.

[24] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *proceedings of the ESEC/FSE*, 2007.