Model Inference Approach for Detecting Feature Interactions in Integrated Systems

Muzammil SHAHBAZ [a,1], Benoît PARREAUX [b] and Francis KLAY [b]
[a] *France Telecom R&D Meylan, France*
[b] *France Telecom R&D Lannion, France*

Abstract.
Many of the formal techniques are orchestrated for interaction detection in a complex integrated solution of hardware and software components. However, the applicability of these techniques is in question, keeping in view time-to-market delivery and scarcity of available resources. The situation is even more intractable when little or no knowledge of components is provided.

We introduce a novel approach of using model inference methods in the domain. We advocate that these methods can be used in detecting feature interactions among components by putting the integrated system under a systematic testing effort and extracting only "context-relevant" models. Our technique allows us to detect those interactions in the system which are normally hidden while testing the components in isolation. We apply our approach to an active problem we are facing in furnishing mobile phone services.

Keywords. Feature Interaction, Model Inference, Mobile Services Framework, Mobile Phone Integration

1. Introduction

1.1. Context

The feature interaction problem [8,19,7] has been widely studied since the last decade. Basically, the problem is that modern software architectures cannot be monolithic and static, rather they need to be modular and flexible for maintainability or economic reasons such as time-to-market delivery. In order to support such a structure, the notion of feature is defined. A feature is a software piece that adds some functionality in a system. In general, a system includes a basic structure S, some features F_1, F_2,..., and is defined as $S \times F_1 \times F_2 \times \ldots$, where $\times$ is a composition operator. Ideally the behavior of each feature should be independent from others but unfortunately it is never the case, since very often cooperative behaviors are required. This means that there are side effects between features which are called interactions. Some of them are desirable while others may lead to unexpected or unrequired system behaviors.

The handling of interaction problems is the ability to detect, understand, classify and manage feature interactions in the system. Actually software interactions arise more

[1] Corresponding Author: muhammad.muzammilshahbaz@orange-ftgroup.com

in fields like web services, plugin software architectures, home appliance automation, mobility, etc. There are mainly two approaches to solve this problem: either on-line or off-line. With on-line techniques [22,24,14,4,6], the interactions are handled at system runtime, which is well suited for quick time-to-market delivery and opens a multi-vendor environment. The main disadvantage of this approach is the processing overhead and the impossibility of getting finer interaction resolution. On the contrary, off-line techniques are often a combination of software engineering methodologies [9,26,25] and formal methods [11,17,1,23,12,13]. The software engineering methodologies define a basic structure with a feature integration process. The formal methods goal is to reach to in-depth understanding of interactions by performing logical analysis on system specifications. The advantage of these techniques is to achieve finer interaction treatment. But in order to efficiently applying these techniques, the problem space they are dealing with must be reduced [20].

In this paper, we present a new automated approach to handle the feature interaction problem, applying it to what we think is an original case study, i.e., mobile phones. There are many features found in today's mobile phones that offer many more capabilities than just simple functions such as voice calls or text messaging. The main problem is that for economic reasons in this context, this critical system needs to be open. For example, a phone manufacturer provides a phone device with basic features, but a phone distributor or a network operator would like to customize it with more features taking advantage of its network services architecture. Finally, a third party, e.g., a game provider, would also like to embed some additional features. In this respect, the phone is an integrated system of multiple components providing required features, plus many others for the purpose of reusability. Different systems may contain similar functionalities but their implementations and logics vary. Furthermore, the behaviors of components in the same system are not independent. They are often assumed to communicate with each other and may invoke required/unrequired functions of each other when accessing critical resources such as private data (e.g., address book) or communication functions (e.g., SMS). The real goal is that once the system is integrated, the integrator wants to know the possible interactions could occur between components.

1.2. Summary of the Approach

Our problem domain deals with two kinds of scenarios. The components may be equipped with either extensible documentation listing all implemented features and inputs to the components, or with no technical/formal specifications at all. In both cases, finding interactions between components in an integrated system is a difficult task. In the former case, it is not wise and also may not be possible to exhaustively analyze the whole specification, since only a subset of functions is used. For that purpose, extracting only relevant abstractions from a complex technical corpus is a hard job. In the latter case, the absence of any formal specifications or little knowledge about the component is an obstacle in applying the majority of techniques. Therefore, we try to build our approach to address both issues, which is shortly described as follows.

For each component to be integrated in the system, we extract only its context-relevant model, i.e., the model that specifies only the required features, and then use it to detect interactions in the system with systematic testing techniques. To extract the models, we introduce the work in the machine inference domain [18,5,3,10] and apply

methods that can actively learn models out of black box machines with systematic testing effort. Contrary to other feature learning work [24], we are not learning models through general artificial learning techniques such as neural networks or rule based learning. Rather, we use active machine learning approach to devise testing strategy through which the model will be inferred incrementally.

Once the preliminary models of each component are extracted, we then integrate the system to detect interactions. We take advantage of the testing performed on each component during its inference and apply those tests to the integrated system. If the component or the system globally does not behave in the same manner as in isolation, then an interaction between two or more components is detected. After that, extracted models are refined iteratively so that they capture new behaviors, and then new tests are performed on the system. The procedure will end when the composite behavior of the system conforms to the inferred models of each component.

Roughly speaking, our approach considers components as black boxes and uses an incremental learning method to infer partial models. Thus, we are introducing machine inference work in the domain of feature interaction detection. At the end of the approach, we will have context-relevant models of each component representing the possible interactions in the system and projecting the up-to-date picture of the system behavior.

The rest of the paper is organized as follows. First, we introduce as intuitively as possible the general framework of our application, i.e., mobile phone system, in section 2. Later, we formally detail the inference technique of our method and its illustration on our example in section 3. Then the application of our method to an instance of the mobile phone system interaction problem is explained in section 4. Finally, section 5 concludes the paper.

2. Mobile Services Framework

Our service framework is to create customized systems by acquiring one general platform upon which components of-the-choice are integrated to build a required solution. Consider a basic platform that helps in customizing mobile phone systems by plugging available set of components from other parties. Such components usually conform to some standard means of interfacing with other components for the purpose of enhancing features in an integrated system. A typical example in mobile applications is a component that is developed under the J2ME environment [15] and provides an interfacing API adherent to some JSR specifications [16]. We describe an example of a system that is built up of such components in Figure 1. The system is a calling system that consists of four components, i.e., Call Screening CS, Address Book AB, Media Manager MM and Call Controller CC.

These components offer a variety of functionalities in their respective areas and options to interface with other components. For this purpose, each of them is provided with a very large set of inputs, in many of which we may have no interest. We are interested only in certain functionalities offered by each component to be used in our customized system. The exhaustive testing of the system for detecting possible component interactions with all combinations of inputs is not possible and also not required. Therefore, the goal of the integrator is to detect unknown interactions while keeping the scope of analysis relevant to the context.

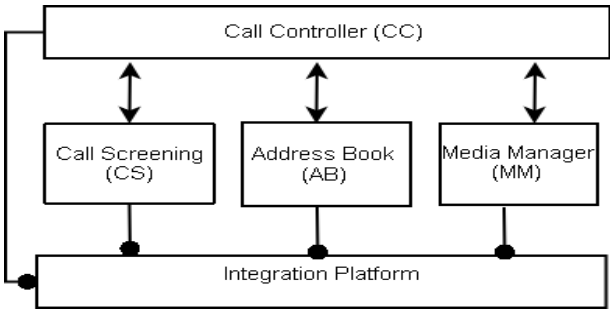


Figure 1. Calling System

In this example, our context of using components is for their prime tasks, which is described as follows with their context-relevant input (CRI) and output (CRO) details:

- **Call Screening** CS**:** keeps a blacklist of phone numbers from which calls are not accepted. The list is populated and managed by the user.
  CRI: An input $''no\langle\#\rangle''$ is given to check whether a number is blacklisted, where # is the caller's number.
  CRO: The component responds with $''OK''$ if the specified number is not found in the list, or $''KO''$ otherwise.
- **Address Book** AB**:** keeps contact records. The records are managed by the user.
  CRI: An input $''srch\langle query\rangle''$ is given to search a contact. A query can be either a name or a number.
  CRO: The component responds with the search result $''rslt\langle name, \#, profile\langle public, private\rangle\rangle''$ if the record is found, or *null* otherwise.
- **Media Manager** MM**:** manages media files for two main purposes, i) to ring the phone by playing a default tone when a call arrives ii) to play a specified media file.
  CRI: An input $''ring\langle\#\rangle''$ is given to invoke the default ring tone, where # is a number that is not concerned with the basic functionality. An input $''play\langle file\rangle''$ is given to play a specified media file.
  CRO: The component responds by giving command $''start\langle default.wav\rangle''$ to the internal phone media player for playing the default tone on the input $''ring\langle\#\rangle''$, or by giving $''start\langle file\rangle''$ on the input $''play\langle file\rangle''$.
- **Call Controller** CC**:** controls the call related operations by using other components.
  CRI: An input $''call\langle\#\rangle''$ is given to invoke the call delivery function of the component.
  CRO: The component responds by sending $''ring\langle\#\rangle''$ to invoke MM as a call incoming notification on receiving clearance of the caller's number from CS, or does nothing $''silent''$ otherwise.

3. Modeling the Components

3.1. Finite State Machine

The use of a finite state machine to model functional aspects of a component is well-known from the telecom and distributed systems communities. Therefore, we are dealing with systems that can be modeled by some finite state machine. Additionally, we are seeing a component as reactive, i.e., it receives an input from the environment and reacts by providing an output and possibly changes its state. This vision leads us to take a machine into account in which transitions can be labelled with inputs and outputs, also known as a Mealy machine. Another assumption is that the states of the machine are stable, i.e., a machine cannot continue without a stimulus from its environment. Following is the formal definition of our model we use in the approach.

Definition: A Finite State Machine M is a six-tuple $M = \{Q, I, O, \sigma, \lambda, q_0\}$, where Q, I and O are finite sets of states, input symbols and output symbols, respectively. $\sigma \quad Q \times I \rightarrow Q$ is the state transition function, $\lambda \quad Q \times I \rightarrow O$ is the output function, and q_0 is the initial state.

When M is in the current (source) state $q \in Q$ and receives $i \in I$, it moves to the target state specified by $\sigma(q, i)$ and produces an output given by $\lambda(q, i)$. In order to completely define our model, we require $dom(\sigma) = dom(\lambda) = Q \times I$. For this purpose, we add a loop-back transition on the state where the given input is invalid and add a symbol Ω as the output.

3.2. Model Inference Method

We are interested in methods that can infer a component with no detailed knowledge beforehand. This is due to the applicability of our approach to the components for which no formal specifications are available. An algorithm that can conjecture a model, defined in section 3.1, from a black box component is given in [21], adapted from [2]. In the following, we provide a succinct description of the algorithm.

The basic requirement for the inference algorithm is to construct an input set through which the algorithm performs testing on the component. The assumption to perform tests on the component is the access to its interfaces, i.e., an input interface from where an input can be sent and an output interface from where an output can be observed. Also, it is assumed that the component can be reset before each test.

The algorithm starts by testing the component with different combinations of input symbols and conjectures a model when a certain condition is satisfied. This condition is helpful in order to elucidate conflicts in the conjecture. The test cases are constructed automatically from an observation table T and the results of the test cases, i.e., the output strings of the component for the given input strings, are also recorded back into T as a partial mapping from I^* to O^*. The domain $dom(T)$ of T is the set of input strings from which the test cases are constructed.

To define the structure of the table, let S and E be two finite sets of finite strings from I^*, then $S \cup S \cdot I$ makes the rows of the table and columns are made by E. Initially, S contains an empty string ϵ and $E = I$, i.e., every input symbol makes one column in the table.

The test cases are constructed by concatenating $s \in S \cup S \cdot I$ and $e \in E$, as $s \cdot e$. The resultant output string of the test case is recorded as an entry in the table, as follows: If α is the output string of the test case $s \cdot e$, then $T(s, e) = \alpha'$, where α' is the suffix of α and $|\alpha'| = |e|$. Let $s, t \in S \cup S \cdot I$, then we define an equivalence relation $\equiv$ over $S \cup S \cdot I$ as follows: $s \equiv t$ iff $T(s, e) = T(t, e), \forall e \in E$, i.e., when the rows s and t are same. We denote by s the equivalence class of rows that also includes s. The algorithm stops testing when the table is found $closed$[2]. A table is $closed$ iff for each $t \in S \cdot I$, there exists $s \in S$, such that $s \equiv t$. In other words, the stoping condition for testing occurs when no new output is observed in the component for longer sequence of test cases. Whenever table is not closed, t is moved to S and $T(t \cdot i, e)$ is extended for all $i \in I, e \in E$.

Once the table is closed, a conjecture $M = \{Q, I, O, \sigma, \lambda, q_0\}$ is constructed as follows:

- $Q = \{s | s \in S\}$
- $q_0 = \epsilon$
- $\sigma(s, i) = s \cdot i, \forall s \in S, i \in I$
- $\lambda(s, i) = T(s, i), \forall i \in I$

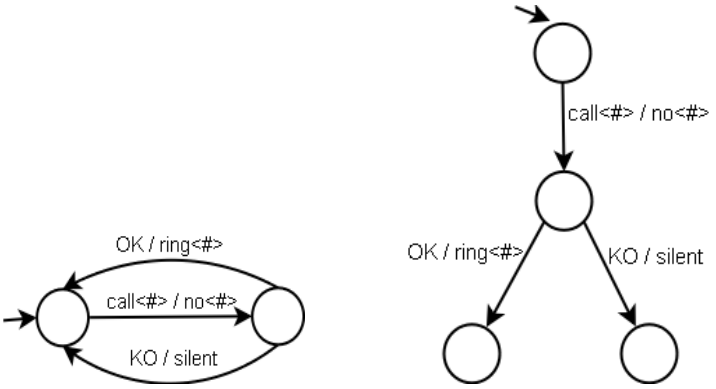
3.3. Inference of the Calling System

We have described a method in section 3.2 to infer an FSM model given in section 3.1 from a component. In this section, we illustrate how to infer the individual preliminary models of components in figure 1 using that method.

The starting point of the algorithm is to construct a set of abstract inputs for each component. This is an important step in our approach which aims to approximate or model only interesting aspects of a component. Therefore, we construct the set of inputs that are relevant to our context. It is the same for the outputs, i.e., we record only the relevant outputs in the observation table and brush aside all others. For example, we are interested to model a behavior of the component CC when a call arrives in the presence of CS. The basic relevant input for CC is $call\langle\#\rangle$. Also, CC is supposed to communicate with CS to block/unblock a particular call. Therefore, we also include OK and KO (the responses of CS) in its input set, which finally becomes $I_{CC} = \{call\langle\#\rangle, OK, KO\}$. The relevant outputs of CC are $no\langle\#\rangle$ when communicating to CS, $silent$ when the arriving call is blacklisted and $ring\langle\#\rangle$ to invoke MM when the call is acceptable.

The run of the inference method on CC takes two iterations shown in Table 1 and Table 2 respectively, while Figure 2(a) shows the conjecture from the closed table, i.e., Table 2. Since we are interested in modeling only a single behavior of the component, we eliminate loops and unfold conjecture as shown in Figure 2(b). Also, we do not show in the conjecture the transitions labelled with invalid inputs for the sake of simplicity. This can be seen in the observation table where the entries Ω against these inputs show their invalidity on the respective states. We keep the representation of test cases in the table and i/o behaviors on the conjecture symbolic. However, the run of test cases on actual components requires concretization of symbolic inputs, e.g., # must be replaced by some actual number in order to execute it on CC. If there is a behavioral difference between any two numbers, then the numbers can be represented as two separate symbolic inputs,

[2]For the reader who is familiar with the original algorithm [2], the other concept called *consistency* has been excluded in the optimized version of the algorithm (see [5]).



(a) Conjecture of CC from Table 2

(b) Conjecture of CC from Table 2 (unfolded)

Figure 2. Conjecture of CC from Table 2

i.e., $\#_1$ and $\#_2$, respectively, to maintain the deterministic property of the state machine. The criteria for selecting concrete values can be guided through certain domain specific policy, and hence not a part of the current approach.

Table 1. Not Closed Table for CC (First Iteration)

	$call\langle\#\rangle$	OK	KO
ϵ	$no\langle\#\rangle$	Ω	Ω
$call\langle\#\rangle$	Ω	$ring\langle\#\rangle$	$silent$
OK	Ω	Ω	Ω
KO	Ω	Ω	Ω

Table 2. Closed Table for CC (Second Iteration)

	$call\langle\#\rangle$	OK	KO
ϵ	$no\langle\#\rangle$	Ω	Ω
$call\langle\#\rangle$	Ω	$ring\langle\#\rangle$	$silent$
OK	Ω	Ω	Ω
KO	Ω	Ω	Ω
$call\langle\#\rangle, call\langle\#\rangle$	Ω	Ω	Ω
$call\langle\#\rangle, OK$	$no\langle\#\rangle$	Ω	Ω
$call\langle\#\rangle, KO$	$no\langle\#\rangle$	Ω	Ω

We construct the input set for CS as $I_{CC} = \{no\langle\#\rangle - cl, no\langle\#\rangle - bl\}$, i.e., the input when # is acceptable and the input when # is blacklisted, respectively. The relevant outputs are OK in the case when # is acceptable and KO, if blacklisted. The closed observation table for CS is given in Table 3 and the (folded/unfolded) conjecture is shown in Figure 3.

Table 3. Closed Table for CS

	$no\langle\#\rangle - bl$	$no\langle\#\rangle - cl$
ϵ	KO	OK
$no\langle\#\rangle - bl$	KO	OK
$no\langle\#\rangle - cl$	KO	OK

Similarly, the input set for AB is simply $I_{AB} = \{srch\langle query\rangle\}$ and the relevant output is a query result. For MM, the input set is $I_{MM} = \{ring\langle\#\rangle, play\langle file\rangle\}$ and the relevant outputs are ringing a default tone and playing a media file. The preliminary (unfolded) models of AB and MM are given in Figures 4 and 5 respectively.

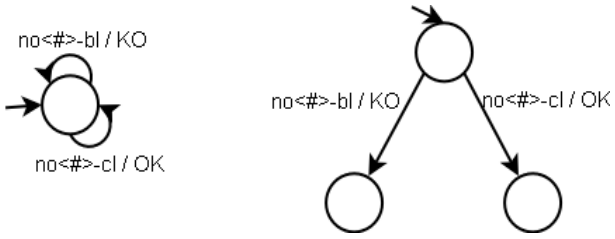


Figure 3. Conjecture of CS from Table 3 (left) and the unfolded version (right)

4. Detecting Interactions

We have explained in the previous section how the preliminary context-relevant models can be inferred from the individual components. In this section, we focus on the method of interaction detection between components after their integration into a system.

Our understanding of the concept of detecting interactions is due to the known problem when composing a system from individual components. The assumption that a component behaves in an integrated system the same as in isolation is not valid. This is because of the fact that components exchange data during the process which may lead to some unexpected behaviors. This exchange of data is actually an underlying interaction between components which we want to detect after the system is integrated. Therefore, we define the approach for detecting interaction as follows.

> The integrated system must exhibit the same behavior as prescribed by the inferred models of each component in the system, for all those test cases that are performed during their inferences. Failure of this indicates the underlying interaction(s) between components.

The collection of test cases performed during the inference of each component will be executed on the integrated system. The observed behaviors of the system as a result of these test cases will be compared to the expected ones, i.e., shown in the inferred models of the components. If there is any divergence found, we narrow down our focus to the i/o interfaces of the components which are involved in the test. If there is a component A whose output stimulates any other component B, the output will be treated as an interaction between A and B. This stimulus may not be seen in the preliminary inferred models of A and B as an output and an input respectively. Therefore, we update these models according to the new context by re-inferring them using the inference method. This can be done by recording the new observations in the observation tables of A and B, and then generating test cases until the tables are closed before making the new conjectures.

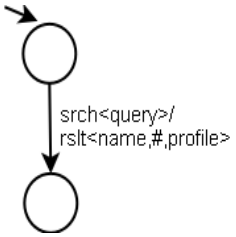


Figure 4. Conjecture of AB (unfolded)

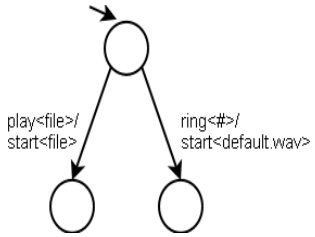


Figure 5. Conjecture of MM (unfolded)

Table 4. Closed Table for CS after fixing new observations

	$no\langle\#\rangle - bl$	$no\langle\#\rangle - cl$	$rslt\langle\ldots\rangle$	$null$
ϵ	$srch\langle query\rangle$	OK	Ω	Ω
$no\langle\#\rangle - bl$	Ω	Ω	OK	KO
$no\langle\#\rangle - cl$	$srch\langle query\rangle$	OK	Ω	Ω
$rslt\langle\ldots\rangle$	Ω	Ω	Ω	Ω
$null$	Ω	Ω	Ω	Ω
$no\langle\#\rangle - bl, no\langle\#\rangle - bl$	Ω	Ω	Ω	Ω
$no\langle\#\rangle - bl, no\langle\#\rangle - cl$	Ω	Ω	Ω	Ω
$no\langle\#\rangle - bl, rslt\langle\ldots\rangle$	$srch\langle query\rangle$	OK	Ω	Ω
$no\langle\#\rangle - bl, null$	$srch\langle query\rangle$	OK	Ω	Ω

We apply the procedure explained above to the system in Figure 1. The models of each component are already inferred, as described in section 3. Now, we apply test cases of each component on the system. Let the test case $call\langle\#\rangle$ has been executed on CC, where # is blacklisted. The expected behavior of CC is to remain $silent$ (as seen in its inferred model in Figure 2(b)) after receiving KO from CS. When the same test case is executed on the integrated system, it starts playing a media file. This divergence leads to investigate the involving components, i.e., CC and CS, according to the models.

It is found that CS emits OK which CC interprets as the number is not blacklisted and then sends $ring\langle\#\rangle$ to MM for call incoming notification. This means that CC is behaving as expected by its inferred model, whereas CS is diverging. It is observed that when receiving $no\langle\#\rangle$[3] from CC, CS emits an output $srch\langle query\rangle$, where $query$ is the number # and stimulates another component AB. The expected behavior of AB is to give out the result of the search query if the contact is found, or null otherwise. It turns out that # is found in the address book and hence AB responds with $rslt\langle name, \#, profile\rangle$, which changes the behavior of CS and generates OK instead of KO. This also discovers the underlying implementation of CS that if the number is found in AB then it should not be blocked. The new observation is recorded in the observation table of CS, shown in Table 3 and the new (unfolded) conjecture is shown in Figure 6.

The expected behavior of MM on receiving input $ring\langle\#\rangle$ is to invoke a default tone (as seen in the model in Figure 5), whereas the system behavior is noticed as playing a media file. This divergence finds an interaction between MM and AB as follows. MM searches the number (given in the ring command) in the address book. If the contact is found, it picks the contact profile as public or private, and plays the respective media file configured with the specific profile. Since the contact # is found in this case, MM plays the media file configured to its profile by sending command $start\langle profile.wav\rangle$ to media player. The new (unfolded) model of MM is shown in Figure 7.

5. Conclusion

We have presented a new approach for feature interaction detection in an integrated system of components using a machine inference method. We built our approach so that it

[3]The interpretation of this input in the inferred model of CS in Figure 3 is: $no\langle\#\rangle - bl$, since # is blacklisted in this example

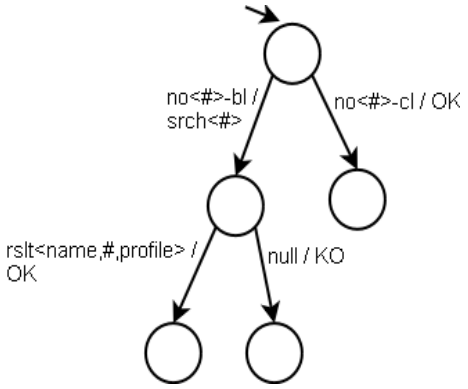


Figure 6. New Conjecture of CS (unfolded)

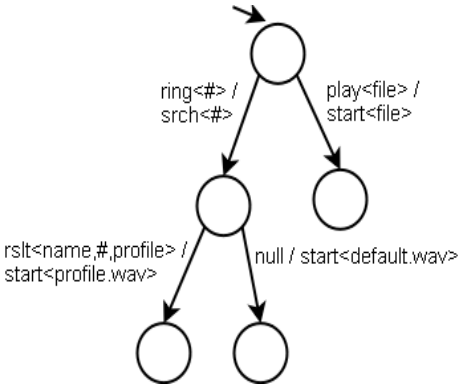


Figure 7. New Conjecture of MM (unfolded)

could be applied in both scenarios, i.e., when the components are provided with complex and huge formal specifications of the features but only few of them are required in the system, and when the components are just seen as black boxes (no specifications are provided). We showed how the use of machine inference methods can cater both scenarios by inferring only the context-relevant partial models of the components, and later how the interactions can be detected automatically by comparing it with the inferred models of the individual components. Our example lies in the framework of mobile phone system customization, in which telecom industries are facing problems for detecting feature interactions after the system is integrated.

There are several points under discussion in connection with the improvement of the overall approach. We are studying techniques to incorporate domain knowledge in the inference method that can guide more effective test cases. Also, the component integration requires some decision points about the way components are integrated. These decision points can help in improving test stopping criteria for testing. Regarding scalability, we are experimenting our technique on a kind of systems that typically consist of large number of components. Various questions seem to us interesting. For example, how efficient is our technique to compare a behavior of such system with the inferred partial models of the large-scale components? Secondly, how can the models can be enriched with parametric details of the components, so that they should not blow up in size? It seems also interesting to use constraints defined on these parameters. Thirdly, how the test cases of individual components can be combined as a basis for testing an integrated system?

We believe that the proposed method is complementary to those currently being studied. This new approach changes the way in which the problem is tackled, since the specification of the component in use is limited to what is necessary.

References

[1] Marc Aiguier, Karim Berkani, and Pascale Le Gall. Feature specification and static analysis for interaction resolution. In *Proceedings of the Formal Methods Symposium*, LNCS, pages 364–379, 2006.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2:87–106, 1987.

[3] Dana Angluin. Queries revisited. *Theor. Comput. Sci.*, 313(2):175–194, 2004.

[4] M. Arango, L. Bahler, P. Bates, M. Cochinwala, D. Cohrs, R. Fish, G. Gopal, N. Griffeth, G. E. Herman, T. Hickey, K. C. Lee, W. E. Leland, C. Lowery, V. Mak, J. Patterson, L. Ruston, M. Segal, R. C. Sekar, M. P. Vecchi, A. Weinrib, and S.-Y. Wuu. The touring machine system. *Commun. ACM*, 36(1):69–77, 1993.

[5] Jose L. Balcazar, Josep Diaz, and Ricard Gavalda. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72, 1997.

[6] R. Buhr, M. Amyot, D. Elammari, T. Quesnel, and S. Gray. Feature-interaction visualization and resolution in an agent environment.

[7] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comput. Networks*, 41(1):115–141, 2003.

[8] E. Cameron. A feature interaction benchmark for in and beyond, 1994.

[9] Jane Cameron, Kong Cheng, Sean Gallagher, Fuchun Joseph Lin, Peter Russo, and Daniel Sobirk. Next generation service creation: Process, methodology, and tool integration. In Kristofer Kimbler and Wiet Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 299–304. IOS Press, Amsterdam, Netherlands, September 1998.

[10] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, 7(3):215–249, 1998.

[11] Lydie du Bousquet and Olivier Gaudoin. Telephony feature validation against eventuality properties and interaction detection based on a statistical analysis of the time to service. In *FIW*, pages 78–95, 2005.

[12] Amy P. Felty and Kedar S. Namjoshi. Feature specification and automated conflict detection. In *Feature Interactions Workshop*. IOS Press, 2000.

[13] J. Paul Gibson. Towards a feature interaction algebra. In Kristofer Kimbler and Wiet Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 217–231. IOS Press, Amsterdam, Netherlands, September 1998.

[14] Seth Homayoon and Harmi Singh. Methods of addressing the interactions of intelligent network services with embedded switch services. *IEEE Communications Magazine*, pages 42–47, December 1988.

[15] Java 2 Platform, Micro Edition - J2ME. http://java.sun.com/javame/index.jsp.

[16] Java Specification Requests. http://jcp.org/en/jsr/all.

[17] Hélène Jouve, Pascale Le Gall, and Sophie Coudert. An automatic off-line feature interaction detection method by static analysis of specifications. In *Proceedings of the 8th International Conference on Feature Interactions in Telecommunications and Software Systems (FIW'05)*, pages 131–146. IOS Press, 2005.

[18] Michael J. Kearns and Umesh V. Vazirani. *An introduction to computational learning theory*. MIT Press, Cambridge, MA, USA, 1994.

[19] Dirk O. Keck and Paul J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Trans. Softw. Eng.*, 24(10):779–796, 1998.

[20] Kristofer Kimbler, Carla Capellmann, and Hugo Velthuijsen. Comprehensive approach to service interaction handling. *Comput. Netw. ISDN Syst.*, 30(15):1363–1387, 1998.

[21] Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of components guided by incremental state machine learning. In *TAIC PART*, pages 59–70. IEEE Computer Society, 2006.

[22] David Marples and Evan H. Magill. The use of rollback to prevent incorrect operation of features in Intelligent Network based systems. In Kristofer Kimbler and Wiet Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 115–134. IOS Press, Amsterdam, Netherlands, September 1998.

[23] Malte Plath and Mark D. Ryan. The feature construct for SMV: Semantics. In Muffy H. Calder and Evan H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 129–144, Amsterdam, Netherlands, May 2000. IOS Press.

[24] S. Tsang and E. Magill. Behaviour based run-time feature interaction detection and resolution approaches for intelligent networks, 1997.

[25] Greg Utas. A pattern language of feature interaction. In Kristofer Kimbler and Wiet Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 98–114. IOS Press, Amsterdam, Netherlands, September 1998.

[26] Pamela Zave and Michael Jackson. A component-based approach to telecommunication software. *IEEE Softw.*, 15(5):70–78, 1998.