Runtime Monitoring of Functional Component Changes with Behavior Models [⋆]

Carlo Ghezzi, Andrea Mocci, and Mario Sangiorgio

Politecnico di Milano, Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32, 20133 Milano (MI), Italy
{ghezzi,mocci,sangiorgio}@elet.polimi.it

Abstract. We consider the problem of run-time discovery and continuous monitoring of new components that live in an open environment. We focus on extracting a formal model—which may not be available—by observing the behavior of the running component. We show how the model built at run time can be enriched through new observations (dynamic model update). We also use the inferred model to perform run-time verification. That is, we try to identify if any changes are made to the component that modify its original behavior, contradict the previous observations, and invalidate the inferred model.

1 Introduction and Motivations

Modern software systems increasingly live in an open world [6]. In the context of this paper, we assume this to mean that the components that can be used to compose new application may be dynamically discovered and they may change over time. New components may appear or disappear; existing components that were already available may change without notice. Indeed, in an open world context, software components can be developed by different stakeholders, for which there might be no control from the point of view of their clients. New applications may be developed in a way that they rely on third party components, often called *services*, that are composed to provide a specific new functionality[1]. In this setting, *models*, play the role of formal specifications and have a crucial importance. In fact, to be able to compose components in applications and make sure they achieve ascertainable goals, one needs to have a model of the components being used. Such model, in practice, mays not exist. For example, in the case where components are Web services, suitable notations (e.g., WSDL) exist to specify the syntax of service invocations, but no standard notation exists to specify the semantics (i.e., model the behavior) of the components. In this context, it becomes relevant to be able to infer a model for the component dynamically, at run time, by observing how the component behaves.

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

[1] Although the terms "component" and "service" can be (and should be) disinguished, in this paper the terms are used interchangeably

In addition to the previous problems, one must consider the fact that the component may change at run time in an unannounced manner. In other words, even if a model were initially provided together with the exposed service, it may become unfaithful and inconsistent because the implementation may change at run time. For this reason, in open-world context the role of models is twofold. It may be necessary to infer it initially and it becomes then necessary to use the (inferred) model at run time to verify if changes invalidate the assumptions we could make based on the initial observations.

In conclusion, in the case where the model is initially absent, we need techniques to infer a formal model (a formal specification) for the components we wish to combine. We then need to keep the (inferred) model to analyze the run-time behavior of the component and detect wether the new observed behaviors indicate that a change has occurred in the component which invalidates the model.

In this paper, we propose a technique for run-time monitoring of component changes that relies on the use of a particular class of formal models, *behavior models*. The proposed approach requires a *setup phase*, in which the component to be monitored must be in a sort of *trial phase* in which it can be safely tested to extract an initial specification. This phase integrates techniques to infer formal specifications [7] with a new kind of behavior model, the *protocol behavior model*. This model enables the main phase of the approach — a *run-time validation activity* —, which consists of monitoring the component behavior and detecting a particular class of component changes, which will be precisely described in the following sections. The approach is also able to distinguish likely new observations against component changes.

The paper is structured as follows. Section 2 presents the formalisms used in the approach, that is, the kind of behavior models that we can infer and synthesize. Section 3 describes how these models are constructed to enable the setup step of our technique, while Section 4 describes their use at runtime to detect component changes. A simplified running example is used to give a practical hint on how the approach works. Finally, Section 5 discusses related approaches and Section 6 illustrates final considerations and future work. Space limitations only made it possible to explain the innovative approach and to provide basic examples. Additional details and more complex examples are available online [2].

2 Behavioral Equivalence and Protocol Models

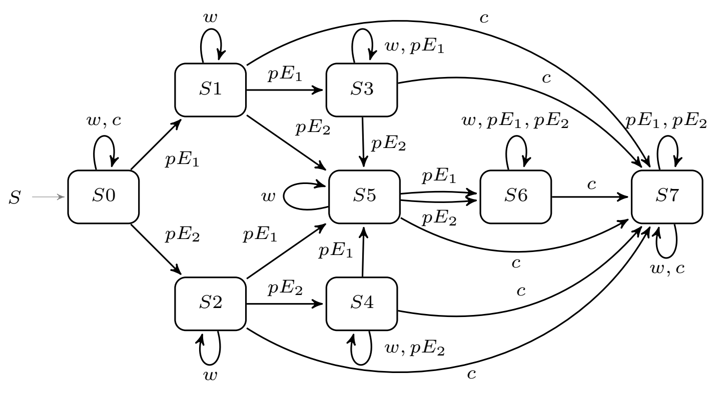
We consider software components as *black boxes*, that is, their internals cannot be inspected and they are accessible only through their API, which consists of operations that might modify or not the internal state. Thus, each operation can be a *modifier* or an *observer*, or it can play both roles. Operations may also have an exceptional result, which is considered as a special observable value. As a running example, we consider a simple component, called StorageService, inspired by the ZipOutputStream class of the *Java Development Kit* [1], which models a storage service where each stored entry is compressed. The component mixes container behaviors with a specific protocol of interaction. We consider the following operations: i) *putNextEntry*, which adds a new entry with a given name; ii) *write*, which adds data to the current entry; and iii) *close*, which disables any further interaction.

We now introduce the formal models used in our approach, which belong to the class of so-called *behavior models.* To accomplish the main task of the approach, that is, the runtime detection of component changes, we first need to define behavior models that they can "summarize" all the possible interactions with the software components, thus providing a description of the behaviors observed by its clients.

We start with *Behavioral equivalence models (*Bem [7]); i.e., finite state automata that provide a precise and detailed description of the behavior of a component in a limited *scope.* In a Bem, states represent *behaviorally equivalent* classes of component instances; that is, a set of instances that cannot be distinguished by any possible sequence of operations ending with an observer. Each state is labeled with observer return values and each transition models a specific modifier invocation with given actual parameters. The *scope* of the model defines the set of possible actual parameters used in the model (called *instance pool*), and the number of states we restrict to. Intuitively, these models define the component behaviors within a limited scope. Figure 1 represents a possible Bem for the StorageService component. We built it limiting the scope to two entries (e_1 and e_2) which are used as parameters for operation *putNextEntry.* Each transition represents a specific operation invocation. The table in Figure 1 describes observer return values; in this specific case, they are only exceptional results.

To describe every possible component interaction outside the Bem scope, we introduce a second kind of behavior model that generalizes the Bem through an abstraction: the *protocol behavior models (*Pbm*).* Pbms provide an abstracted, less precise but generalized description of the interaction protocol with the component as opposed to the precise description in a limited scope provided by Bems. The new model is still based on a finite state automaton, but now states encode whether the results of observers are normal or exceptional[2]. States also abstract the behavior of modifiers as *variant* or *invariant.* A modifier behavior is variant if there exists a possible invocation with specific actual parameters that brings the component in a different behavioral equivalence state. Otherwise, the modifier behavior is invariant. This abstraction is usually (but not always) associated with an exceptional result of the operation: it is the typical behavior of a removal operation on an empty container or a add operation on a full bounded container. Pbm transitions instead keep track only of the operation they represent, ignoring the values of the parameters. Thus they model the behavior of every possible modifier invocation; they synthesize the behavior of possibly infinitely-many behavior changes induced by the possible operation invocation.

[2] If observers have parameters, then the abstraction can be i) always (i.e., for every parameter) normal; ii) always exceptional; iii) any intermediate situation, that is, for some parameters the result is normal and for others is exceptional .



Legend:	*S:StorageService(), w:write(0), c:close()*
	 pE₁:putNextEntry(e1), pE₂: putNextEntry(e2)			

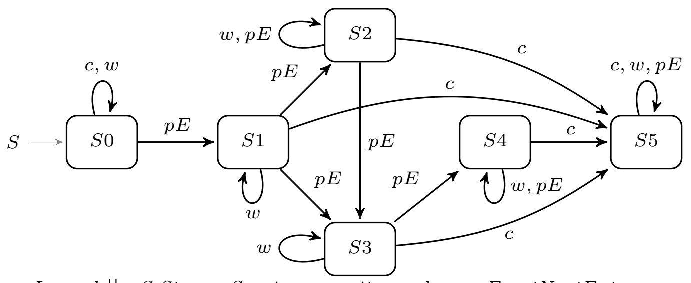
State	$close()$	$putNextEntry(e_1)$	$putNextEntry(e_2)$	$write(0)$
$S0$	$\rightsquigarrow ZipException_4$	—	—	$\rightsquigarrow ZipException_3$
$S1$	—	$\rightsquigarrow ZipException_1$	—	—
$S2$	—	—	$\rightsquigarrow ZipException_2$	—
$S3$	—	$\rightsquigarrow ZipException_1$	—	$\rightsquigarrow ZipException_3$
$S4$	—	—	$\rightsquigarrow ZipException_2$	$\rightsquigarrow ZipException_3$
$S5$	—	$\rightsquigarrow ZipException_1$	$\rightsquigarrow ZipException_2$	—
$S6$	—	$\rightsquigarrow ZipException_1$	$\rightsquigarrow ZipException_2$	$\rightsquigarrow ZipException_3$
$S7$	—	$\rightsquigarrow IOException_1$	$\rightsquigarrow IOException_1$	$\rightsquigarrow IOException_1$

$ZipException_1.getMessage()$ = *"duplicate entry: e1"*
$ZipException_2.getMessage()$ = *"duplicate entry: e2"*
$ZipException_3.getMessage()$ = *"no current ZIP entry"*
$ZipException_4.getMessage()$ = *"ZIP file must have at least one entry"*
$IOException_1.getMessage()$ = *"Stream Closed"*

Fig. 1. A BEM of the STORAGESERVICE component

In practice, they model the possibility that by performing an operation the set of operations enabled on the object may change. It is worth observing (but we do not show it here) that this abstraction may introduce nondeterminism in the automaton. Figure 2 represents the PBM derived by performing the abstraction described above to the BEM in Figure 1.

The main contribution of the proposed approach is the integration of PBMs and BEMs. Because the PBM is derived from the BEM through an abstraction process, its completeness and accuracy actually depends on the significance of the observations that produced the BEM during the setup phase. The setup phase is deeply rooted in the *small scope hypothesis*. In its original formulation [9], this hypothesis states that *most bugs have small counterexamples*, and that an exhaustive analysis of the component behavior within a limited scope is able to show most bugs. In our case, we cast it as follows: most of the significant behaviors of a component are present within a small scope. In our case, the term "significant behavior" refers to the abstracted version provided by a PBM. Thus, we expect that at setup time we can synthesize a likely complete PBM,



Legend: || *S:StorageService, w:write, c:close, pE:putNextEntry*

State	close	putNextEntry	write
		Observer Abstraction	
$S0$	$\rightsquigarrow$ *ZipException*	—	$\rightsquigarrow$ *ZipException*
$S1$	—	$[-, \rightsquigarrow$ *ZipException*$]$	—
$S2$	—	$[-, \rightsquigarrow$ *ZipException*$]$	$\rightsquigarrow$ *ZipException*
$S3$	—	$\rightsquigarrow$ *ZipException*	—
$S4$	—	$\rightsquigarrow$ *ZipException*	$\rightsquigarrow$ *ZipException*
$S5$	—	$\rightsquigarrow$ *IOException*	$\rightsquigarrow$ *IOException*
		Modifier Behavior Abstraction	
$S0$	*Invariant*	*Variant*	*Invariant*
$S1$	*Variant*	*Variant*	*Invariant*
$S2$	*Variant*	*Invariant*	*Invariant*
$S3$	*Variant*	*Variant*	*Invariant*
$S4$	*Variant*	*Invariant*	*Invariant*
$S5$	*Invariant*	*Invariant*	*Invariant*

The notation: $[-, \rightsquigarrow$ *ZipException*$]$ *means that for some parameter the method returns correctly (—), and for some other parameter throws ZipException*

Fig. 2. A Pbm of the StorageService component

which describes the protocol of all the possible interactions of clients with the component, while at runtime we can use the Pbm to find component changes.

The two different models can be used together at run time. The behavior of a component is monitored and checked with respect to the Pbm. When violations are detected, a deeper analysis exploiting the more precise information contained in the Bem can be performed in order to discover whether the observation that is not described by the Pbm is a new kind of behavior that was not observed before, and thus requires a change of both the Bem and the Pbm to accommodate it, or instead it detects a component change that is inconsistent with the models. The Bem synthesizes the observations used to generate the Pbm, and thus it can be used to distinguish between likely changes of the analyzed component from new observations that instead just enrich the Pbm. In the following sections, we will discuss these aspects: the setup time construction of Bems and Pbms, and the runtime use of both models to detect likely component changes.

It should be noted that the Pbm is not a full specification of the component, thus it cannot be used to express complex functional behaviors, in particular the ones that are not expressible with a finite state machine, like complex container behaviors. Instead, the Pbm models the protocol that clients can use to interact with the component, that is, the legal sequences of operations. This limitation is also the enabling factor for runtime detection of changes: violations can be checked and detected relatively easily and the model can be promptly updated when needed. Instead, a full fledged specification that supports infinite state behaviors, like the ones of containers, is definitely harder to synthesize, check and update at runtime.

3 Setup Phase: Model Inference

As we illustrated previously, the approach we propose prescribes two phases. The setup phase is performed on the component in a trial stage. The other phase corresponds to runtime. In the former, the component is analyzed through dynamic analysis (a set of test cases) to infer a Bem for the component. A Pbm abstraction is generated next to generalize the observed behaviors. In the latter phase, the two models are used at run time to detect component changes. In this section, we describe the first phase, with particular focus on the generation of models, so that designers can get a formal description of a component whose behavior must be validated.

3.1 Generation of the Initial Behavioral Equivalence Model

To generate a Bem during the setup phase, we adapt the algorithm and the tool described in [7], which extracts Bems through dynamic analysis. The model is generated by incrementally and exhaustively exploring a finite subset of the component behavior, that is, by exploring a small scope. As illustrated previously, the scope is determined by a set of actual parameters for each component operation and a maximum number of states for the model. The exploration is performed by building a set of traces using the values in the instance pool and abstracting them to behavioral equivalence states. The exploration is incremental; that is, if a trace t is analyzed, then all its subtraces have been analyzed in the past. To build the Bem, the approach first uses observer return values: for a trace t and every possible observer o, with fixed actual parameters, we execute $t.o()$ and we build a state of the Bem labeled with observed return values. Unfortunately, such an abstraction does not always induce behavioral equivalence: for example, it could be that for some operation m, then two traces t_1 and t_2 such that for every observer the return values are equal, then there could be that $t_1.m()$ and $t_2.m()$ are not behaviorally equivalent. Thus, state abstraction is enriched with the information given by m as a discriminating operation. For space reasons, we cannot include the specific details of the algorithm, but the interest reader can refer to [7, 11]. This approach guarantees the discovery of all the behaviors presented in the class with the given scope. The way Bems are generated implies a strong correlation between the quality of the model and the completeness of the instance pools used to build it. The more the instances are significant, the higher the coverage of the actual behavior of the class is.

Given the importance of the objects used to perform the Bem generation phase, we want to exploit as much as possible all the knowledge available to analyze the class behavior with the most significant set of instances. The original

Spy tool relied entirely on instances provided by the user interested in obtaining the Bem of a component. While the assumption that the user is able to provide some significant instances is fair, it may be hard to achieve since it requires a lot of effort and a deep knowledge of the behaviors of the component. Fortunately, in practice the vast majority of the classes comes with a test suite containing exactly the operation calls with some significant instances as parameters.

The extraction of the significant instances is performed by collecting from the test suite all the objects passed as arguments in an operation call. Each value is then stored in an instance pool that collects all the values for each parameter. The values of the instances are stored directly in the instance pools, ready to be used in the exhaustive search. Instances collected from the test suite are very useful, but it happens that they may be redundant. To avoid the generation of models with a lot of states that do not unveil new behaviors, we should filter out the instances collected in order to keep a minimal subset able to exercise all the possible behaviors of the component without having to deal with a huge model. At this stage of the development, the tool is able to extract instances from a test suite but does not select the minimal subset of instances. This task is left to the user who has to find the best trade-off between the number of instances used for the analysis and the completeness of the Bem generated.

3.2 Synthesis of the Protocol Behavior Model

Once the Bem is generated we can go further with the analysis and generate the corresponding Pbm. Generation is quite straightforward since the Bem already includes all the needed information about the outcome of each operation in each state of the model. Pbm inference algorithm consists of the following steps: i) generalization of the Bem states through the Pbm abstraction function; ii) introduction of each Bem transition in the Pbm. The generalization of the information contained in a Bem state is performed by applying to each state of the Bem the Pbm abstraction function we discussed earlier. Then for each transition of the Bem we add a transition to the Pbm starting from the representative of the equivalence class of the starting node in the Bem and ending in the representative of the destination node. Because parameters are ignored in the abstraction, the transformation is likely to produce a non-deterministic automaton: it may happen that, given a Pbm state, the invocation of the same operation with different values of the parameters produces different outcomes that turns into different destination states.

4 Runtime Phase: Monitoring and Change Detection

Bems and Pbms are used to perform runtime verification that the component, which may evolve independently, behaves accordingly to its models. To do so, we monitor the execution of an application and detect changes in the behavior of its components. Being able to detect changes is crucial when the application in which the component is embedded has to be self-adaptive and react to component misbehaviors in an appropriate way. More precisely, in this section we show that the data collected at runtime can be used on the one hand to enrich the model with previously unobserved behaviors and on the other hand to highlight behavioral differences unveiling changes in the component under analysis.

4.1 Monitoring

A monitor is introduced into the running system to allow the comparison of the actual behavior of the component under analysis and the ones encoded by the models. Each time an instance of the scrutinized class is created, it is associated to it a monitoring process in charge of recording the observed execution trace and analyzing it to discover violations with respect to the protocol described by the model. Violation detection is performed by comparing the actual behavior with the one encoded in the model. Therefore it has to gather enough information to determine the state in which the component is. The system reports a violation when it finds an exceptional outcome for an operation that, according to the model, should always terminate normally or, on the opposite, when an operation that the model describes as exceptional does not throw anything.

In order to maintain the lowest overhead possible on the system under analysis, the violation detector relies, when possible, exclusively on the observed trace. When the Pbm has only deterministic transitions this process is straightforward and violations can be detected directly from the execution trace. Unfortunately, almost all components with a complex behavior are non-deterministic so there is the need of a deeper inspection by executing operations that could provide more information and thus reveal the state in which the component is. The solution proposed in this paper is an enhanced monitoring phase, not relying exclusively on what it is observable from the current execution but also able to perform some more queries to the object under analysis. For any state having non-deterministic outgoing transitions, we can determine which are the operations that make it possible to know which one has been taken. These discriminators are the operations on the destination states having different behaviors. Nondeterminism can therefore be solved by invoking the discriminating operations on the object under analysis. With these additional operations it is easy to find the compatible state among the different nondeterministic possibilities. Discriminating operations have to be invoked on an instance of the object exactly in the same state of the actual component and should be tested with both the original instance pool and the instances observed in the trace for the operation. The original instance pool alone would not be so effective as it would make it impossible to find behaviors related to the parameter re-use.

Clearly it is not possible to call additional operations on the actual component under analysis. It would lead to interferences with the service provided by the system. Modifiers have undesirable side effects but also the invocation of a *pure* operation could introduce delays and reduce the quality of service. In order be able to carry on the analysis without disrupting the offered service, we need to assume that we can create a *clone* of the component behaving exactly in the same way the actual instance does. Moreover, the operations performed on such clone do not have to change the actual component environment. These assumptions reduce the number of components our methodology can deal with, but we are still able to monitor and analyze a vast class of commonly used elements.

Therefore the monitoring architecture requires: i) to instrument the application using the external services; ii) to have the possibility to call operations on a sandboxed instance of the service; iii) to be able to replay execution traces in order to put the sandboxed instance in a defined state. With such an infrastructure, the verification module can detect changes in the behavior of external services without interfering with the actual execution of the system.

4.2 Response to Violations

During the monitoring phase it may happen that an observation on the actual execution conflicts with what it is described by the model. There are two possible causes for the violation observed: the model could be incomplete, and therefore needs to be updated, or the behavior of component has changed. The analysis phase has to be able to deal and react properly to both these situations.

It is possible to discover whether the violations is due to the incompleteness of the PBM or to a change in the behavior by replaying on the *clone* some significant executions encoded in the BEM. If all of them produce again the previously observed results, then the model needs to be completed and that violations simply indicate behaviors never explored before. Otherwise the violation signals a misbehavior of the component that should trigger a reaction aimed at bringing back the system in a safe state.

In order to keep the approach feasible, we cannot just test that everything described by the BEM is still valid. We should rather focus on the part of the model more closely related to the observed violation. The first step in the selection of the relevant execution traces is the identification of the BEM states corresponding to the state of the PBM in which the violation occurred. The initial part of the test cases can then be generated by looking for the shortest execution traces able to reach the selected BEM states. The traces obtained in that way have then to be completed with the operation that unveiled the violation. For any BEM state the operation has to be called with all the parameters present in the instance pool used to generate the model.

Model updates have to be performed when the monitoring tool discovers a violation but there is no evidence of behavioral change. Models are updated because the behavior of the observed execution does not contrast with what has been observed in the past. Model updates are first applied to the BEM and then to the corresponding PBM.

Updating the BEM means enriching the scope it covers with the trace unveiling the new behavior. Keeping all the information in a single BEM would make its dimension increase, so we decided to rely on a set of BEMs, each one describing a behavior of the component on the particular scope showing it. Doing that, we can easily keep track of all the relevant execution exposing the different behaviors. Although doing that we may miss some behavior due to the interaction of the behaviors described by different BEMs, this is not an issue: the model will describe them as soon as they appear at run time. From the set of BEMs it is easy to get the corresponding PBM. It is quite straightforward to adapt the inference algorithm described in section 3 to deal with the information contained on more than a behavioral model: the algorithm has to be applied to each BEM and the data gathered have to be added to the same PBM so that it contains information about all the observed behaviors regardless of the BEM it comes form. To produce correct abstractions for the new PBM, all the BEMs must have a coherent set of observers. To ensure that, we must update the scope for the observer roles in the already existing BEMs to have them take into account all the significant values of the parameters discovered at run time.

A violation requiring to update the models of STORAGESERVICE reported in figures 1 and 2 happens when we try to write an empty string of data when no entry is available. In such situation, the expected *ZipException* is not thrown because the *write* operation does not have to write anything and the component does not check for the availability of an entry. Therefore, we need to add a BEM containing the observed trace. Since the violating trace contained a previously unseen instance, we also have to update the existing BEM to have it consider the empty string as a parameter for the *write* operation. For space limitations the updated models and other examples are only available online at [2].

Change detection takes place when there is at least one test case behaving differently than what the PBM prescribes. Since the model encodes the behaviors observed in the past, any violation can be considered as an evidence of a change: at least in the case highlighted by the failing test, the same execution trace presented a different behavior than the one assumed by the model. The system has then to react to the behavioral changes detected. We identified two possible scenarios in order to be able to guarantee the maximal safety though trying to limit the number of service interruptions. The safer scenario presents a change that just turns one or more operation call with an exceptional outcome into invocations that terminates normally. Another possible and more critical situation affects more deeply the enabledness of the different operations and so requires a stricter reaction to ensure the safety of the system.

In the first case, the change has to be notified but it does not require to stop the execution of the application. The detected change is probably just an addition of new functionalities or interaction patterns that previously were not present or were disabled. However, for safety reason it is better to leave to the user the final decision about how to react to this kind of behavioral changes. More serious problems may arise form behavioral changes that turns the outcome of an operation from normal to exceptional. Such a change makes it impossible to *substitute* the new component to the one the system is expecting to deal with. At some point there may be an invocation to the operation that changes its behavior and it is going to always produce a failure due to the exception thrown. For this reason, when changes like this occur, the only safe solution is to stop the execution of the system requiring the intervention of a supervisor able to decide how to fix the problem.

Change detection can be demonstrated using again the models reported in Figure 1 and 2 to monitor the behavior of a StorageService. For a very simple example we can assume that the component stops working and changes its behavior to always throw an exception every time *putNextEntry* is invoked. In this scenario, any execution of *putNextEntry* now violates the Pbm. We are interested to check if the violation is specific to the trace observed or it is a component change; to check this, we derive the simple test case *StorageService().putNextEntry(*e_1*)* from the Bem. Since this test case violates the Bem, it highlights the change of the behavior of the component.
A more comprehensive evaluation of the effectiveness of the change detection methodology has been performed injecting faults into the component under analysis and is available online at [2].

It is important to remark that this methodology is able to identify changes only when there is at least one failing test case in the ones that can be derived from the Bem. Since the model does not contain information about every possible execution it is possible that an actual change is detected as a case in which there is the need to update the Bem and the Pbm because they does not contain anything about that particular case. However, since the updates to the model have to be reviewed by the designer of the system the procedure prescribed by our methodology can be considered effective.

5 Related Work

The protocol models discussed in this paper describe the behavior of a software component accordingly to whether its operations are enabled or not. The underlying idea has been introduced with the concept of Typestate in [13]. A similar abstraction has also been used in [5], which presents a technique to build an enabledness model from contracts and static analysis.
Tautoko [4] generates similar models starting from an existing test suite. Our tool does not infer the model directly from the test execution traces. It rather exploits the test suite to gather some domain knowledge to use with the Spy methodology.
Monitoring of both functional and non-functional properties of service-based systems are described in [3]. Our technique is based on Pbms and Bems, therefore we are able to model and monitor very precisely functional properties of a software component.
Invite [12] developed the idea of *runtime testing*, pointing out the requirements the running system has to satisfy in order to make it possible. In this work we also introduced a technique to encode test cases in Bems and to select the ones can highlight a behavioral change.
Tracer [10] builds runtime models from execution traces enabling richer and more detailed analysis at an higher abstraction level. In [8] models are used to

monitor system executions and to detect deviation from the desired behavior of consumer electronic products. Our approach combines these two aspects providing a methodology able to both detect violations and build models according to the information gathered at run time.

6 Conclusions and Future Work

Behavior models can be useful throughout all the lifecycle of a software component. Traditionally, such models are used at design time to support system designers in they choices. However, they can also play a significant role after the application is deployed by monitoring its execution and checking system properties. That is particularly useful in the context of systems in which verification must extend to run time, because of unexpected changes that may occur during operation.

This work focuses on the runtime aspects, extending the original scope of behavior models to running systems. The models and methodology proposed can maintain an updated representation of the behavior of the component considering observations made during the actual execution of a running system. Our approach is also able to detect and notify the system designer behavioral changes in the monitored components. Preliminary experiments show that our approach is effective and can deal with non-trivial components. Further research is going to enhance the models removing current limitations and thus making it possible to monitor an even broader class of software components.

References

1. Oracle, java se 6.0 doc., 2011. `http://download.oracle.com/javase/6/docs/index.html`.
2. Spy at runtime. http://home.dei.polimi.it/mocci/spy/runtime/, 2011.
3. L. Baresi and S. Guinea. Self-supervising bpel processes. *IEEE TSE*, 2011.
4. V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA '10*, Trento, Italy, 2010.
5. G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Automated abstractions for contract validation. *IEEE TSE*, 2010.
6. E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *ASE*, 2008.
7. C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *ICSE '09*, Vancouver, Canada, 2009.
8. J. Hooman and T. Hendriks. Model-based run-time error detection. In *Models@run.time '07*, Nashville, USA, 2007.
9. D. Jackson. *Software Abstractions:Logic,Language,and Analysis*. MIT Press, '06.
10. S. Maoz. Using model-based traces as runtime models. *Computer*, 2009.
11. A. Mocci. *Behavioral Modeling, Inference and Validation for Stateful Component Specifications*. Ph.D. thesis, Politecnico di Milano, Milano, Italy, 2010.
12. C. Murphy, G. Kaiser, I. Vo, and M. Chu. Quality assurance of software applications using the in vivo testing approach. In *ICST '09*, Denver, Colorado, 2009.
13. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, 1986.