Computing (2013) 95:191–221
DOI 10.1007/s00607-012-0214-z

Detecting component changes at run time with behavior models

Andrea Mocci · Mario Sangiorgio

Received: 6 January 2012 / Accepted: 21 August 2012 / Published online: 13 September 2012

Abstract Modern software systems are composed of several services which may be developed and maintained by third parties and thus they can change independently and without notice during the system's runtime execution. In such systems, changes may possibly be a threat to system functional correctness, and thus to its reliability. Hence, it is important to detect them as soon as they happen to enable proper reaction. Change detection can be done by monitoring system execution and comparing the observed execution traces against models of the services composing the application. Unfortunately, formal specifications for services are not usually provided and developers have to infer them. In this paper we propose a methodology which exactly addresses these issues by using software behavior models to monitor component execution and detect changes. In particular, we describe a technique to infer behavior model specifications with a dynamic black box approach, keep them up-to-date with run time observations and detect behavior changes. Finally, we present a case study to validate the effectiveness of the approach in component change detection for a component that implements a complex, real communication protocol.

Keywords Behavior models · Change detection · Monitoring · Dynamic analysis · Specification inference · Runtime validation

Mathematics Subject Classification 68N30

A. Mocci (✉)
Computer Science and Artificial Intelligence Laboratory,
Massachusetts Institute of Technology, Cambridge, MA 02139, USA
e-mail: am@csail.mit.edu

M. Sangiorgio
Dipartimento di Elettronica e Informazione, Politecnico di Milano,
Piazza Leonardo da Vinci, 32, 20133 Milano (MI), Italy
e-mail: sangiorgio@elet.polimi.it

1 Introduction and motivations

Modern software systems increasingly live in an open world [2,8]. 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 and made available by different stakeholders, who pursue their own objectives. In most cases, clients have no ways to control the development and evolution of external components. Still, 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. Although the terms "component" and "service" can be (and should be) distinguished, in this paper the terms are used interchangeably.

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. Unfortunately, such a model, in practice, may 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. We consider stateful components, hence the models we work with have to represent properly the state of the component and describe what happens when each operation is executed. We cannot consider each operation in isolation but their results depend on the current state of the component. Thus we model the behavior of components using finite state machines.

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. Changes may be caused both by the deployment of a new version of the service and by a misbehavior due to some internal failure. In principle the first class of change may be addressed by having the clients to check a service version identifier, but there is no way to detect misbehaviors without implementing a monitoring approach.

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 alive to analyze the run-time behavior of the component and detect whether 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*. We focus

on the description of the behavior of a single component rather of the overall system. From our perspective, a component is a library which enables the interaction of a system with a remote service. In this work we focused on services with independent client-server interactions. The proposed approach requires a *design 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. At this stage we infer two complementary models: Behavioral Equivalence Model (BEMS) [11] and Protocol Behavior Model (PBMS). With these two kinds of model we can describe both the protocol of interaction with an external service and the details of its behavior in a small scope. This pair of models enables the main phase of the approach presented in this paper—a *run-time validation activity*—which consists of monitoring the component behavior and detecting a particular class of component changes. With PBMS we can monitor the behavior of an external service. When a violation in the protocol of interaction is detected, we fall back to the information contained in BEMS to determine whether it is due to a previously unseen behavior or to a change in the component. Our approach can detect both changes happening while system is running and changes that take place between different invocations of the functionalities provided by an external service.

The paper is structured as follows. Section 2 illustrates a motivating example, an implementation of a storage service, and frames the contributions of the paper by intuitively describing the class of models we use, providing an overview of the monitoring approach and describing the kind of violations we support. Thus, in Sect. 3, we detail the formalisms we use, that is, the kind of behavior models we can synthesize. Section 4 describes how these models are constructed to enable the design phase of our technique, while Sect. 5 describes their use at runtime to detect component changes. A significant case study, considering an implementation of a complex protocol, is described in Sect. 6. Finally, Sect. 7 discusses related approaches and Sect. 8 illustrates final considerations and future work. Additional details and more complex examples are available online [29].

2 Motivating example and approach overview

This section introduces an illustrating example and outlines the proposed approach. To introduce the models and to explain our approach in the following section, we consider as a running example a simple component, called STORAGESERVICE, inspired by the ZIPOUTPUTSTREAM class of the *Java Development Kit* [23]. STORAGESERVICE allows the user to store data on a server organizing them in different entries. 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. It is not possible to create and store entries named with already used identifiers nor to write data in an invalid entry. Of course the close operation is allowed only after the component has been opened. When the client tries to execute a not allowed operation, the component throws a *StorageException* with a message specifying the precise nature of the exception. The Application Programming Interface API of the component is reported in Listing 1, described as a JAVA interface.

```
public interface StorageService{
 void putNextEntry(Entry e)
                throws IOException, StorageException;
 void write(String data)
                throws IOException, StorageException;
 void close()
                throws StorageException;
}
```

Listing 1: StorageService interface

The approach we propose considers *functional* behaviors of software components and uses *behavior models* as formal notation to describe such behaviors. A behavior model is a finite state machine that describes the behavior of a software component at a certain level of abstraction. In a behavior model, transitions represent modifier invocations, while states are labeled with a given abstraction of what an external client can observe by invoking observer methods. Behavior models differ on the different choices for the state abstractions and the way they represent modifier invocations on transitions. Examples of such behavior models are the finite state machines extracted by Adabu [6], Contractor [7] or other inference and synthesis approaches [4,11,17,32].

From a modeling point of view, we use two particular kinds of behavior models:

- a Bem is used to precisely represent a subset of the behaviors of the component, when considering a limited *scope*, consisting of a fixed set of parameters for methods and a bounded number of possible states;
- a Pbm is a synthesized generalization of the Bem, that is able to represent the protocol of interaction between the component and its clients for a generic trace, that is, if the trace has a normal or exceptional behavior.

The two models have a different perspective: the Bem is precise but within limited scope, while the Pbm is generalized but less precise. Intuitively, while the Bem is able to represent bounded container behaviors, and, for instance, it can distinguish between LIFO and FIFO behaviors, the Pbm only represents normal and exceptional sequences of invocations. Instead, while Pbm can describe the protocol aspect of the component's behavior for a generic sequence of operations, the Bem can represent only the subset of its possible behaviors, for a limited set of method parameters and for a bounded set of possible states.The differences in the considered scope and in the aspect of the execution of an operation represented in the models make them complimentary. Putting together the information contained in Bems and Pbms it is possible to develop methodology that are both general and detailed. In Sect. 3, we discuss in detail and formalize the two classes of models, Bems and Pbms.

The approach we propose has two phases: the *design phase* and the *runtime phase*. In the design phase, the Bem is synthesized through dynamic analysis, and the Pbm is abstracted from it. We extend our previous work that partially included Bem synthesis [11], and we assume that the component is available for testing. This assumption is shared with other approaches, such as [4], that proactively synthesize behavior models for deployed services; Sect. 4 discusses how these models are synthesized. The quality of the models produced with a black-box synthesis is related to the portion of the state

space explored during the BEM synthesis. The confidence and the behavioral coverage we can guarantee depends on the considered scope and it is comparable to the knowledge of the component that a developer can get when they have to deal with an unknown API. Developers base their experiments on the component documentation, on code snippets found in tutorials and possibly in their knowledge of the application domain. After the trials, developers have an idea of how the component works but they do not have any guarantee of having discovered all the behaviors and they have to check that nothing unexpected happens at run time. With our approach, we use the domain knowledge provided by developers or contained in the test cases of the component to automatically obtain specifications.

In the runtime phase, we keep models alive to update them with new observations and to detect a class of behavioral changes that correspond to inconsistencies with respect to what the PBM prescribes. Thus, our approach uses the synthesized PBM to monitor the component behavior during its execution. An observed violation in the PBM can be due to two different reasons:

- if the observations done in the design phase are still valid, the violation is a likely new observation of the possible behaviors of the component, and the PBM can be safely updated to include the new behavior;
- if some of the design-phase observations are invalidated, the component behavior is likely changed.

To distinguish between the two cases, our approach uses the inferred BEM, that synthesizes the observed behaviors in the design phase. Please note that for simplicity we assume that the interface of the component cannot change. In Sect. 5, we will discuss how the monitoring approach is realized and detail how this distinction is performed. Please note that in this work we address only change detection and not how to react to it, that is, how to react to an observed behavior that violates the PBM and the observations retrieved in the design phase. It is important also to state that our approach, during the monitoring phase, needs the possibility of querying the component to check if some behaviors hold (e.g., some traces execute normally or terminate with an exception). This can be easily addressed if the component admits the possibility to create new instances and execute arbitrary operations on them; otherwise, we require the service provider to expose a *shadow component* that reflects the same behavior of the analyzed component and whose state can be safely modified and inspected without interfering with the currently running application.

We can now proceed to analyze the behavior models used in our approach before discussing their synthesis and their use for behavioral monitoring.

3 Behavioral equivalence and protocol models

In service oriented architectures, developers compose several software components which unfortunately usually do not come with a complete formal specification of their behaviors. Developers knowledge of components have to rely on documentation expressed in natural language, when it exists, and on the provided APIS. In such an environment components internals and their source code cannot usually be inspected. Models of components behaviors can be inferred only considering them as *black boxes*

and relying on the information obtained from the invocation of the exposed operations. In this paper we focus on fine-grain components implemented as (Java) classes. Thus, each operation can be a *modifier* or an *observer*, or it can play both roles. As defined by Guttag and Liskov [14], observers are operations which can be used to inspect the state of an object while modifiers change it. An object can have both pure observers and modifiers, which respectively do not have side effects and which only change the state, and operations presenting both the behaviors. Operations may also have an exceptional result, which is considered as a special observable value.

From a formal point of view, we adopt a classic algebraic approach to component signatures [13]. We assume the reader to be familiar with basic algebraic concepts such as sorted set, signature, algebra and term [28]. In the hidden approach to algebraic specification, the type of the component to be specified is distinguished by the types used to specify method parameters and observer return values. Such types compose the so-called *visible data universe*, which is a tuple $\Delta = \langle V, \leq, \Psi, \mathcal{D} \rangle$, where V is a set of *sorts*, which represent types, $\leq$ is a subsorting relation, Ψ is the set of operations for these types, and $\mathcal{D}$ is a V, Ψ-sorted algebra which assigns semantics to these types. Common types used as visible data universe are integers and booleans. The subsorting relation $\leq$ is used to model subtyping and type extension; its use will be explained hereafter with a concrete example.

A visible data universe can be used to formally define the interface of a component by means of a *hidden signature*, which is defined as follows:

Definition 1 (*Hidden Signature* [13]) A hidden signature over a visible data universe Δ is a pair $H_\Delta = \langle h, \Sigma \rangle$ where $h \notin V$ is the *hidden sort* that defines the component to be specified, while Σ is a $\{h\} \cup hV^* \cup hV^*h$-sorted set of functional symbols which represents the component operations.

Every operation $\sigma \in \Sigma$ is classified as follows:

- $\sigma :\rightarrow h$ is the only constructor;
- $\sigma : hw \rightarrow h$, with $w \in V^*$ is a modifier symbol;
- $\sigma : hw \rightarrow v$, with $w \in V^*$ and $v \in V$ is an observer symbol.

We denote the set of modifier symbols as M and the set of observer symbols as O. We denote as $T^x_{H_\Delta}(Z)$ the set of terms up to length x using a set Z of $V \cup \{h\}$-sorted variables. Similarly, $M^x_{H_\Delta}(Z)$ is the set of modifier sequences, that is, terms composed by the constructor and a possibly empty sequence of modifier symbols, while $C^x_{H_\Delta}(Z)$ is the set of contexts, that is, modifier sequences ending with an observer. When x is omitted, we denote terms of any possible length.

For generality, every method is considered to have both the observer and modifier role; thus, two functional symbols, one belonging to M and one belonging to O are added to the hidden signature. The modifier role functional symbol keeps the same parameters as the method, while its range is obviously the hidden type h; it models the method behavior which possibly changes the internal state of the component. Instead, the observer role functional symbol models what the clients can observe from that execution, that is, its return value and the thrown exceptions. In the case of overloaded methods, we consider them as different symbols in the formal signature of the component.

Table 1 StorageService formal signature

$h = StorageService;$	
$V = \{Entry, String, void, void\_IOEx\_StEx, void\_StEx\}$	
Constructor	
$StorageService :\rightarrow StorageService$	
Modifiers	
$putNextEntry :$	$StorageService \times Entry \rightarrow StorageService$
$write :$	$StorageService \times String \rightarrow StorageService$
$close :$	$StorageService \rightarrow StorageService$
Observers	
$putNextEntry :$	$StorageService \times Entry \rightarrow void\_IOEx\_StEx$
$write :$	$StorageService \times String \rightarrow void\_IOEx\_StEx$
$close :$	$StorageService \rightarrow void\_StEx$

Consider the StorageService component whose interface is described in Listing 1. Table 1 shows the formal signature of the StorageService component obtained with the process described above.

As illustrated in the picture, the data abstraction to be defined is identified by the hidden sort $h = StorageService$, while the visible data universe is composed of three particular sorts: $void$, $void\_StEx$, $void\_IOEx\_StEx$. $void$ is a special type with no operations and interpreted as a singleton containing just the element $null$. $void\_StEx$ is a supersort of $void$ that contains objects of type $StorageException$, and $void\_IOEx\_StEx$ is another supersort which also contains objects of type $IOException$.

3.1 Behavioral equivalence models

Behavioral equivalence model [11] are finite state automata providing a precise and detailed description of the behavior of a component in a limited scope. They are based on two key ideas: *behavioral equivalence* [9,13] and the *small scope hypothesis* [16]. Two objects are in a behaviorally equivalent state if and only if it is not possible to distinguish them by looking at the result values obtained from any possible sequence of their operations ending with an observer. This definition does not take into account possible differences in the internal representation of the object states which are not observable and therefore are not know to an external observer.

With the small scope hypothesis we assume that the behavior of a component can be fully described *by example* by showing how it behaves in a limited but significant scope. Rather than modeling the behavior of a component with all the possible combinations of input parameters and methods interleaving, which would of course be infeasible, we only consider the scope able to show at least one example of each behavior. Usually the scope we have to explore to unveil all the possible behavior is quite small, given that it is selected properly. Of course, it is not possible to guarantee the presence of all the behaviors with a black box approach. While the notion of "behavior" might be left as intuitive, in our case we explicitly refer to the possible protocol behaviors that be expressed by the Pbm. We will discuss again this issue in Sect. 3.2, after we formalize Pbms.

From these two ideas we developed a characterization for BEMs and a methodology to infer them [11]. BEMs definition comes directly from behavioral equivalence: states represent *behaviorally equivalent* classes of component instances. Each state is labeled with observed 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 as the only generalization performed to build them is to group together object instances with states differing at most for something which is not observable. The bound on the number of states is required by the fact that, in principle, each operation could be called infinitely-many times producing a new state for each execution. Usually the operations which continue creating new states present some regularity, hence the model can provide an example of the behavior of the component by exploring a limited number of states.

Let us formalize the concept of scope for a BEM, to provide a precise description of the model. Let H_Δ be a hidden signature with visible data universe $\Delta = \langle V, \leq, \Psi, \mathcal{D}\rangle$. A set of sorts $\bar{V}$ is a set of *instance pool sorts* iff:

$$\bar{V} \cap V = \varnothing \wedge \forall \bar{v} \in \bar{V}, \exists v \in V : \bar{v} \leq v$$

A set of instance pools is a $\bar{V}$-order-sorted algebra $\bar{\mathcal{D}}$ such that each interpretation of $\bar{v} \in \bar{V}$ is *finite*. For a given visible data universe $\Delta = \langle V, \leq, \Psi, \mathcal{D}\rangle$, an *instance pool universe* is the set $\bar{\Delta} = \langle \bar{V}, \leq, \bar{\Psi}, \bar{\mathcal{D}}\rangle$ where $\bar{V}$ is a set of instance pool sorts, $\bar{\mathcal{D}}$ is the set of instance pools, and $\bar{\Psi}$ is the set of functional symbols whose constants is restricted to $\bar{D}$.

Thus, we are ready to bind a set of instance pools to methods, that is, we can define an instance pool configuration that defines the BEM scope.

Definition 2 (*Instance Pool Configuration*) Let H_Δ be a hidden signature, and $\bar{\Delta} = \langle \bar{V}, \leq, \bar{\Psi}, \bar{\mathcal{D}}\rangle$ be an instance pool universe.

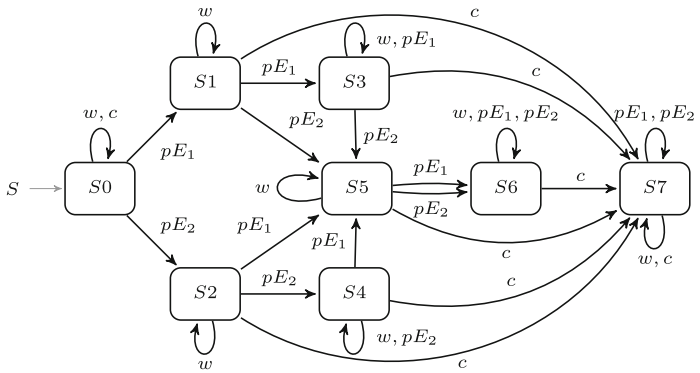
An *instance pool configuration*, or BEM *scope*, is a pair $\Pi = \langle \bar{\Delta}, \mathcal{IP}\rangle$, where $\mathcal{IP} : \Sigma \to \bar{V}^+$. The configuration is *consistent* iff:

$$\forall \sigma_{hw,s} \in \Sigma_{hw,s} |\ \ \mathcal{IP}(\sigma) \leq w \tag{1}$$

The consistency condition prescribes that instance pools configured for a given functional symbol are compatible with its signature, that is, they are subsorts of the sort specified in the signature for parameters. When we restrict a set of terms with parameters over a given instance pool, we denote them with a bar on top, and we explicit the scope Π as follows: $\bar{T}^x_{H_\Delta,\Pi}(Z)$ for generic terms, $\bar{M}^x_{H_\Delta,\Pi}(Z)$ for modifier sequences, $\bar{C}^x_{H_\Delta,\Pi}(Z)$ for contexts.

We are now ready to formally define a *behavioral equivalence model*:

Definition 3 (*Behavioral Equivalence Model*) Let H_Δ be a hidden signature over a visible data universe Δ, and Π an instance pool configuration. A *Behavioral Equivalence Model* (BEM) over H_Δ with scope Π is a tuple $\mathcal{B}_{H_\Delta,\Pi} = \langle Q, q_0, I, \delta, \Omega\rangle$, where:



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

State	$close()$	$putNextEntry(e_1)$	$putNextEntry(e_2)$	$write("x")$
$S0$	$\rightsquigarrow StorageEx_4$	—	—	$\rightsquigarrow StorageEx_3$
$S1$	—	$\rightsquigarrow StorageEx_1$	—	—
$S2$	—	—	$\rightsquigarrow StorageEx_2$	—
$S3$	—	$\rightsquigarrow StorageEx_1$	—	$\rightsquigarrow StorageEx_3$
$S4$	—	—	$\rightsquigarrow StorageEx_2$	$\rightsquigarrow StorageEx_3$
$S5$	—	$\rightsquigarrow StorageEx_1$	$\rightsquigarrow StorageEx_2$	—
$S6$	—	$\rightsquigarrow StorageEx_1$	$\rightsquigarrow StorageEx_2$	$\rightsquigarrow StorageEx_3$
$S7$	—	$\rightsquigarrow IOException_1$	$\rightsquigarrow IOException_1$	$\rightsquigarrow IOException_1$

$StorageEx_1.getMessage() =$ *"duplicate entry: e1"*
$StorageEx_2.getMessage() =$ *"duplicate entry: e2"*
$StorageEx_3.getMessage() =$ *"no current entry"*
$StorageEx_4.getMessage() =$ *"File must have at least one entry"*
$IOException_1.getMessage() =$ *"Stream Closed"*

Fig. 1 A BEM of the STORAGESERVICE component

- Q is the set of states of the BEM;
- $q_0 \in Q$ is the initial state, that represents the constructor invocation;
- $I = \bar{M}^1_{H_{\Delta,\Pi}}(\{z_h\})$ is the alphabet, which consists of modifier sequences of length 1, that is, modifier symbols with fixed parameters in the instance pool.
- $\delta : Q \times I \rightarrow Q$ is the transition function;
- $\Omega : Q \times \bar{C}^1_{H_{\Delta,\Pi}}(\{z_h\}) \rightarrow D$ is a state labeling function modeling observer return values; the function maps an observer with fixed parameters in the instance pool to an element in the data universe.

Please note that z_h is a variable needed to represent function applications where the component argument is not fixed; for this reason, it is usually omitted from terms. For example, the $putNextEntry(z, e_1)$ is a valid term in I, and by omitting the variable z, we will denote it as $putNextEntry(e_1)$. The values e_1 and e_2 are placeholders for a pair of entries initialized with different names. Since the actual content we write on the entires is not relevant, we decided to keep the scope small by writing only the *"x"* string. We choose that scope because it highlights what happens when the service has to deal with multiple entries.

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. The only instance for the *write* operation is *"x"*. Each transition represents a specific operation invocation. The table in Fig. 1 describes the labeling of each state reporting observer return values; in this specific case, they are only exceptional results.

The BEM describes precisely all the different situations the clients have to deal with while they are using the STORAGESERVICE. For instance in $S0$ the component has just been initialized and it is possible to create new entries but since no entry is available it is not possible to place data on the server. In state $S1$ the entry e_1 is ready to hold user data. $S3$ is reached by placing again the entry e_1. That operation closes the entry, making it impossible to write new data into it. The pair of states ($S2$,$S4$) is symmetrical to ($S1$,$S3$). The two pairs only differ for the name of the entries considered but they describe the same behavior. In states $S5$ and $S6$ both the entries have been created. In $S5$ the last entry inserted is still valid and it is possible to write data into it. That is not possible in state $S6$ in which both the entires have already been closed. Finally in state $S7$, reachable with the *close* operation, it is not possible to interact with the component.

At this point it should be clear how BEMS can show by example the behavior of the component. The main limitations of this kind of models is that they are strictly bound to the instance pools used to build them. Moreover BEMS tend to become quite big since the same behavior, obtained through different combinations of actual parameters, is described several times in "symmetrical" sequences of states.

3.2 Protocol behavior models

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 PBMS [12]. PBMS provide an abstracted, less precise but generalized description of the interaction protocol with the component rather than the precise description in a limited scope provided by BEMS. The two kind of models have a very tight relationship because they describe the behavior of the class at different abstraction levels. To address this fundamental difference, PBMS have a different semantics which is strictly related to the abstraction we want to perform on the information contained in BEMS. With PBMS we can describe the protocol of interaction with a component. We can also exploit PBMS generality to monitor the behavior of the pieces of software which are not under the direct control of the developers.

The new model is still based on a finite state automaton, but now states encode whether the operations are enabled or not. We consider as enabled the operations terminating normally. An operation is disabled if it terminates with an exception. Therefore, operations accepting input parameters may present three different result modes. An operation may terminate normally for all the possible values of the parameters, it may be always exceptional and it could even present a mixed behavior when its outcome depends on the input values. Since PBMS are an abstraction of BEMS the only possible source of mixed behavior are different parameter values. The way we define BEMS states and how we abstract them in PBMS makes it impossible for mixed behavior to rise from other sources such as non-observable portions of their state.

State abstraction also describes 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 an add operation on a full bounded container. Invariant operations do not change the state of the component and thus they are not influent in the protocol of the interaction with the class. For this reason such operations can be, to some extent, considered as disabled since their invocation is irrelevant for the protocol of interaction with the class. Combining the information coming from observer abstraction and from modifier behavior abstraction we can model effectively the protocol behavior of a software component. Modifier behavior abstraction refers to the effect of the operation on the concrete state of the component. It has not to be confused with PBMS transitions which describes changes in the behavior of the component. There may be operations which affects only the actual state without modifying the component behavior and vice-versa. For instance, the addition of an element to an already non-empty container does not changes its behavior while it does affect the actual state.

PBM transitions are labeled by the name 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. In practice, they model the possibility that by performing an operation the set of operations enabled on the object may change.

Formally, a PBM is defined as follows:

Definition 4 (*Protocol Behavior Model*) Let H_Δ be a hidden signature over a visible data universe Δ. A *Protocol Behavior Model (*PBMS*)* over H_Δ is a tuple $\mathcal{P}_{H_\Delta} = \langle Q, q_0, I, \delta, \Omega, B \rangle$, where:

- Q is the set of states of the PBM;
- $q_0 \in Q$ is the initial state, that represents the constructor invocation;
- $I = M$ is the alphabet;
- $\delta : Q \times I \to \wp(Q)$ is the transition function;
- $\Omega : Q \times O \to \wp(V)$ is the observer abstraction function;
- $B : Q \times M \to \{Variant, Invariant\}$ is the behavior abstraction function.

In our approach, we focus on deterministic components. By construction, BEMS are deterministic automata and they can precisely describe, leveraging the notion of behavioral equivalence, all the behavioral details of the components under analysis on a finite scope. On the opposite, PBMS model the interaction with components in every possible scope; for this reason, the abstraction used to build PBMS may introduce non-determinism. It can be introduced for two reasons:

- states in the PBM may represent a set of different behavioral equivalent states of the component;
- transitions may depend on the actual parameters that are abstracted away in PBMS.

In the former case, a non-deterministic transition appears in the PBM if the same method, with the same parameters, leads to different PBM states when applied to two

different BEM states that are abstracted to the same PBM state. In the latter case, non-determinism arises when invocations of a method with different parameters applied to the same BEM state lead to different PBM states. At the PBM level of abstraction, we cannot anymore describe precisely all the behavioral details of the component and thus BEMS presents non-determinism. BEMS generality—that is, the fact that every possible trace of the component can be interpreted in the PBM— compensates precision loss and it enables us to describe the behavior of the component in a broader scope than BEMS.

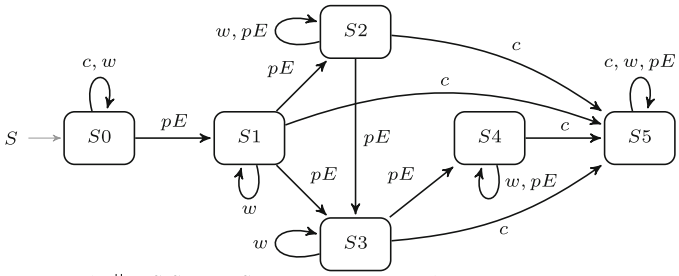
We now formalize the notion of *compatibility* of a component execution with a PBM. Let $t \in M_{H_\Delta}(\varnothing)$ be a modifier sequence. Its *anemic trace* $\alpha(t) \in M^*$ is the sequence of modifier symbols appearing in the same order as the trace. Moreover, let $\langle c, d \rangle$ be an *observed component execution*, composed of $c = t.o(p_1, \ldots, p_k) \in C_{H_\Delta}(\varnothing)$ as a modifier sequence t ending with an observer o, and $d \in D$ is a value in the observable data universe representing the result of the invocation of that term in the component. The observed component execution is *compatible* with a PBM $\mathcal{P}_{H_\Delta}$, written $\mathcal{P}_{H_\Delta} \models \langle c, d \rangle$ if it respects the following definition:

$$\mathcal{P}_{H_\Delta} \models \langle c, d \rangle \Leftrightarrow \exists v \in \Omega(\delta^*(q_0, \alpha(t)), o) \mid d \in D_v \tag{2}$$

The definition above is the foundation for the monitoring approach we propose. It states that an observed value d for a trace c is compatible with the PBM if in the state reached by considering the anemic sequence of modifier symbols of c, the observer abstraction function is compatible with d.

Figure 2 represents the PBM derived by performing the abstraction described above to the BEM in Fig. 1. In $S0$ STORAGESERVICE has just been initialized but no entry has been created yet. As soon as an entry is created which is always possible in $S0$, the component is in state $S1$ and it is ready to store some data. The subsequent invocation of the *putNextEntry* operation has a non-deterministic choice. If the operation is invoked with a different entry than the one previously inserted the component is in $S3$, a state in which it is possible to store some data but it is not possible to create other entries due to scope effects. Otherwise, if a duplicate entry is inserted, the component is in state $S2$ in which it cannot store data because the entry is not valid. State $S4$ is an artifact due to scope effects. In that state it is not possible to create new entries nor to store data. In state $S5$, which is reachable with the *close* operation, every interaction with the component is disabled.

The main contribution of the proposed approach is the integration of BEMS 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 design phase. The BEMS accuracy is deeply rooted in the *small scope hypothesis*. In its original formulation [16], 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; that is, with behavior, in our approach, we mean the property of a particular trace to execute normally or exceptionally as prescribed by



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

State	*close*	*putNextEntry*	*write*
	Observer Abstraction		
S0	$\rightsquigarrow StorageEx$	—	$\rightsquigarrow StorageEx$
S1	—	$[-, \rightsquigarrow StorageEx]$	—
S2	—	$[-, \rightsquigarrow StorageEx]$	$\rightsquigarrow StorageEx$
S3	—	$\rightsquigarrow StorageEx$	—
S4	—	$\rightsquigarrow StorageEx$	$\rightsquigarrow StorageEx$
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: $[-, \rightsquigarrow StorageEx]$ *means that for some parameter the method returns correctly (—), and for some other parameter throws StorageEx*

Fig. 2 A PBM of the STORAGESERVICE component

the PBM. Thus, we expect that at design time we can synthesize a likely complete PBM, which describes the protocol of all the possible interactions of clients with the component, while at runtime we can use the PBM to monitor compliance with the observed behaviors or possible mismatches that indicate component changes. Even though we are operating in an open world, we think that the small-scope hypothesis still holds. In an open world components may evolve or change, but the behaviors they provide, in terms of what BEMS can express, is still limited and thus we can explore it within a small-scope.

The two different models (BEMS and PBMS) 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 BEMS 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 BEMS. In the following sections, we will discuss these aspects: the design time construction of BEM and PBM, 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 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.

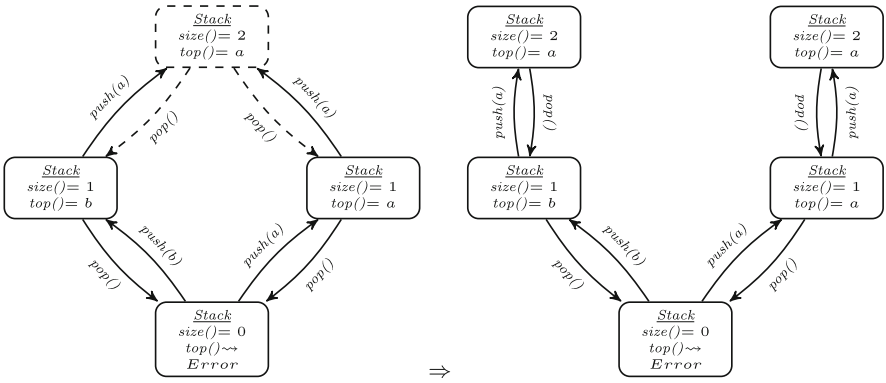
4 Design phase: model inference

As we illustrated previously, the approach we propose prescribes two phases. The design phase is performed on the component in a trial stage. The other phase is performed at runtime. In the former, the component is analyzed through dynamic analysis to infer a BEM for the component. We generate a set of execution traces to exhaustively explore the small scope defined for the model. An abstraction of the BEM, the PBM, is then generated to generalize the observed behaviors. In this section, we describe the design 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.

4.1 Generation of the initial behavioral equivalence model

To generate a BEM during the design phase, we adapt the algorithm and the tool described in [11], 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. 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. We cannot guarantee the purity of observers, therefore we have to perform independent executions each time we want to observe a return value.

Such an abstraction does not always induce behavioral equivalence: for example, it could be that for some operation m, there are two traces t_1 and t_2 such that for every observer the return values are equal, and $t_1.m()$ and $t_2.m()$ are not behaviorally equivalent. Operation m is called *behavioral discriminator*. Thus, state abstraction is enriched with the information given by m as a discriminating operation. An example of behavioral discriminator is the *pop()* operation of a stack. Suppose that *size()* and *top()* are the only two observers; thus, for all the stacks with the same size and the same element in the top of the stack the observer return values are the same, despite some of them might not be behaviorally equivalent. To discriminate part of this difference, it is possible to iteratively call the *pop()* operation and further calling observers. By calling the discriminating operation we can inspect the *hidden state* of the object and



(a) *pop()* behavioral discriminator discovery

(b) State abstraction enrichment with behavioral discriminator

Fig. 3 BEM behavioral discriminator discovery and resolution

thus we can take this information into account when we determine whether two states are behaviorally equivalent. Figure 3 shows an example of behavioral discriminator discovery and resolution as performed by the BEM inference algorithm. The example shows how, during the BEM construction for a stack, the *pop* behavioral discriminator is first discovered. Up to the invocation of the *push(a)* operations, the two stacks of size 2 are not distinguishable; they become distinguishable after the *pop()* invocation, that reveals part of the state that cannot be inspected with just the invocation of observers.

Further examples and uses of BEMs cannot be included for space reasons, but the interest reader can refer to [11,20]. 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. The original approach, as introduced in [11], 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 unit test suite containing exactly the operation calls with some significant instances as parameters.

To exploit the knowledge about the significant instances contained in the test cases, we build instance pools from the values of the parameters passed to the operations in the test suite. In this way, we can build automatically the instance pools containing relevant instances values. If the test suite is well written and tests all the possible behavior, the so obtained instance pools are able to unveil all the behavior of the component under analysis. This approach may have the drawback of generating instance pools containing several redundant entries. 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 generated Bem. In this context, a complete Bem is a model which contains an example of all the interaction patterns. We are not interested in the specific behavior with every possible parameter binding, which of course leads to a definition of completeness impossible to achieve.

Another important addition to the Bem extractor is the optional execution of a *finalizing operation*. When the component has some persistent state, or interacts with other components that have a persistent state, it might be useful to reset the persistent state in order to preserve deterministic behavior of the component to be analyzed from the client point of view. To extract a Bem as a model of a component, the user can specify a finalizing operation that is executed after each trace to reset the component's environment or to finalize component resources. We do not require the analyzed component to provide a finalizing operation. It is usually not provided by Service Oriented Architecture SOA application but it can be written by the developers who want to use our tool for the sake of model generation.

4.2 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. The generation of the Pbm directly from the Bem is a natural choice because of the first is an abstract view of the second model. Moreover, in our process we need both Bem and Bems. A black box inference algorithm for Bems would require to execute again a lot of traces juts to get the information we can already get abstracting the content of Bems.

Pbm inference algorithm consists of the following steps: (i) generalization of the Bem states through the Pbm abstraction function; (ii) mapping each Bem transition into a corresponding Pbm transition. 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.

5 Runtime phase: monitoring and change detection

Bems and Bems are kept alive at run time for two main purposes. First, it may happen that new behaviors manifest themselves while the system is running. In this case, the models must be updated and enriched so that they keep track of the newly discovered behaviors. However, it is also possible that the component undergoes modifications changing its visible behavior at run time. As mentioned, this is rather common in an open-world setting. For example, a Web service might undergo a change by the

service provider, which alters the behavior as seen by clients. Behavioral changes may also be caused by unforeseen faults. The method we describe here can automatically detect changes to the component that are inconsistent with the behaviors observed so far. This is achieved by monitoring and analyzing system execution to detect possible model violations that lead either to a model update or to the detection of a behavioral change. These aspects are discussed in detail hereafter.

The boundary between model inference carried on at design time and at run time is blurred. The goal of both the phases is to generate the best representation possible of the behavior of a software component but each one has its own peculiarities. At design time we build models by exploring exhaustively the defined scope. At run time we complete them by covering also the residual behavior with a different technique specifically designed to complete the model by adding only the new information. However, one should strive for inferring the most complete possible model at the set-up phase so that developers can leverage precise and reliable information about the behavior of the component when they build their applications.

5.1 Monitoring

A monitor is introduced into the running system to support 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, a monitoring process is associated with it to record the observed execution trace. The monitor analyzes the observed execution trace looking for violations with respect to the previously inferred protocol model. Violation detection is performed by comparing the actual behavior with the one encoded in the model. The system reports a violation when it detects an exceptional outcome for an operation that, according to the model, should always terminate normally or, conversely, when an operation that the model describes as exceptional does not throw an exception.

In order to keep overhead as low as possible, the violation detector relies, when possible, exclusively on the observed trace. This means that we do not inspect the actual state of the component and we detect violations only on the basis of the observed trace. State inspection would require a pro-active and very expensive monitoring step. Passive monitoring is adequate for the purpose of this work because it behaves accordingly to trace compatibility for PBM we gave in Definition 2.

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, container-like behavior are modeled by a nondeterministic PBM. However, since the components our approach supports are deterministic, nondeterminism can be resolved to discriminate between a potential or effective violation.

Before discussing how we implemented nondeterminism resolution, consider the two possible strategies to deal with nondeterminism resolution during monitoring:

- we can resolve nondeterminism only when a potential violation is observed;
- we can resolve nondeterminism each time a nondeterministic transition in the PBM is observed.

In general, none of the two approaches fits all the situations. The former is more suitable for components requiring long sequences of invocations for a client to accomplish a task. The latter works better when the interaction between the system and an instance of the component requires short traces. The two approaches are perfectly compatible and we might even switch from the one to the other as the interaction pattern changes. In that way we could ensure that the monitoring phase overhead is as little as possible for all the scenarios.

The former strategy keeps track of all the states compatible with the observed trace; thus, it does not require more information that what is already available to the monitoring system. In that way, it can deal with arbitrary long execution traces and the only overhead introduced by the presence of nondeterminism is the set of compatible states rather than the singles state in which the component actually is. If the set of compatible states presents a possible protocol violation, the strategy needs to check whether it is the actual state and then properly react. It is important to note that as we get more elements in the observed execution trace we can safely remove some states from the set of the compatible states as they reveal to be different from the actual state of the component. For simplicity, from now on, we implicitly consider the second strategy to be implemented, that is, the nondeterminism resolution is performed after each nondeterminism transition in the PBM.

We now discuss our technique for nondeterminism resolution for a given trace t. Nondeterminism resolution requires more information than what it is expressed in the PBM, thus there is need for 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, which does not rely 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 nondeterministic outgoing transitions, we must be able to uniquely select one of them. To do so, we need to determine which are the operations that make it possible to know which one has been taken. These operations, that we call (PBM) *state discriminators*, are the operations having different behaviors on the destination states. By executing the discriminators on the shadow component, we do not require state discriminators to be pure observers. In fact, every method which can have both a normal and an exceptional result mode is a suitable state discriminator. Nondeterminism can therefore be solved by invoking the state discriminators on the object under analysis. State characterization of BEMs guarantees that we can always find a discriminator. Each pair of distinct PBM states has at least an operation producing different results. With these additional operations it is possible to determine the compatible state among the different nondeterministic possibilities. Resolution of nondeterminism requires an entire execution trace to be executed and, in general, it may be expensive when there are a lot of operation performed on the external service. We do not claim this is the best solution in general, but it turned out to be effective in the case studies we analyzed.

It is important to remark that nondeterminism resolution is just an optimization and that we monitor separately each instance of the component. This make it possible to drastically reduce the length of the analyzed traces since we have to take into account

only the lifecycle of a single interaction with the component rather than the past execution of the whole system.

Our procedure to solve nondeterminism is an optimization to avoid to carry on nondeterministic choice when we do not actually need them. Even though BEMs may present nondeterminism they model the behavior of deterministic components. The procedure is related to the homing sequence for finite state machines [26], but in this case it is much simpler because we can rely on the information provided by state labeling. For every pair of PBM states, it is always possible to find a single operation that allows us to know in which one the monitored component is.

As an example, we may refer to the PBM of STORAGESERVICE reported in Fig. 2. Consider a simple trace of the form *StorageService().putNextEntry*(e_x). *putNextEntry*(e_y), where e_x, e_y are newly observed parameters, that is, they were not part of the instance pool used at design phase to infer the model. The invocation of the last *putNextEntry* in state $S1$ is nondeterministic. Looking at the characterizations of states $S2$ and $S3$ we can find two possible state discriminators: a further *putNextEntry* or a *write*. The component is in state $S2$ if the test executions of *write* or *putNextEntry* give us respectively an always exceptional behavior or normal results for some parameter values and exceptional results for others. Conversely the component is in state $S3$, if the results of the state discriminators are always normal for *write* or always exceptional for *putNextEntry*, respectively. State discriminators have to be invoked on a newly generated instance of the object, which must be initialized exactly in the same state of the actual component. This new instance behaves as a *shadow component*, which provides the same functionality of the actual component and whose state can be safely modified and inspected without interfering with the currently running application. Note that we do not require any particular support from the provider of the service. We just need to create a new instance of the service on which we can execute the operations required by our approach. To initialize the state of the shadow component, the monitored execution trace is replayed to bring it into the state we are interested in. After the state is initialized, the shadow component can be used to resolve nondeterminism by calling the state discriminator. The scope of the running component is usually different from the scope used to build the models during the design phase. In fact, at runtime, state discriminators must be called with parameter values coming from both the instance pool used during model inference, and the instances observed in the trace whose state must be discriminated. Using only the instances contained in the original pools may lead to wrong conclusions. For example, a behavior may arise when an instance is used twice with the component. In this case it is clear that we are not going to observe it if the instances used in the trace under analysis are different from the ones used at design phase.

In conclusion, the monitoring architecture requires: (i) instrumenting the application using the external black-box services to collect execution traces; (ii) enabling the possibility to call operations on a shadow component (i.e., a sandboxed instance of the service under scrutiny); (iii) enabling the possibility to replay execution traces in order to put the shadow component in a suitable 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. Albeit those constrains may seem very strict, they are met by a good set of SOA applications. Instrumentation have to be

performed client-side, and it is easy to wrap a library to enable the monitor capabilities we need. Invocation on a shadow component requires the possibility to interact with multiple instances, one of which will be used to replay execution traces. If the application supports independent instances we can apply our monitoring methodology and change detection approach.

5.2 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 it 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. The detection of a component change is important because this is going to surprise the clients with an unexpected behavior, inconsistent with previous observations. Our approach cannot detect changes that do not affect the protocol of interaction with the component. Albeit that would be interesting, the detection of that kind of change is out of the scope of this work.

A shadow component comes into play also in this case. In fact it is possible to discover whether the violations are due to the incompleteness of the PBM or to a change in the behavior by replaying on a shadow component 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 we can conclude that violations simply indicate behaviors never explored before. Otherwise the violation signals a change to the component that is unexpected and inconsistent with its previously observed behavior. This indicates that clients should plan suitable reactions to continue to satisfy their goal or reach some safe state.

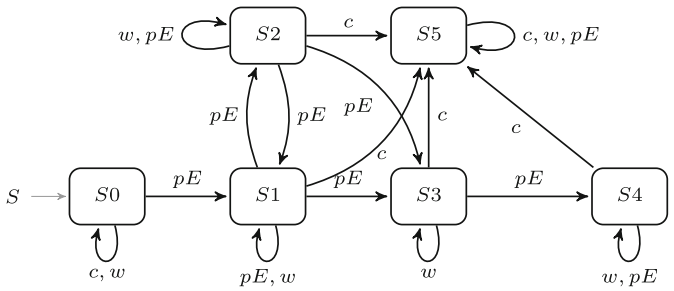
We work with deterministic components. When we cannot find any contradiction with the information contained in the BEM we can safely conclude that they did not undergo a change, at least in the behavior which is already described by our models. It may also happen that a component changes a previously unobserved behavior. In that case, we cannot present the violation as a change but we report it as a behavior we were not aware of and therefore developers should pay particular attention to it.

In order to keep the approach feasible, we cannot just test that all previously observed behaviors encoded by the BEM are 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 set of BEM states corresponding to the state of the PBM in which the violation occurred. The prefixes of the test case traces can then be generated by looking for the shortest BEM paths that reach the selected BEM states. The prefixes 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. If the result of execution of all traces on the shadow component coincide with the initially observed results we conclude that there is no evidence of behavioral change and therefore the model only needs to be updated.

5.2.1 Model updates

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 lead to an expensive update step. For example if the new behavior is caused by a certain value of a parameter that was not in the original instance pool, we would need to update the BEM by running on a shadow component a set of test cases that would complete the model with respect to the extended instance pool. Moreover this would lead to an increase of the BEM's size. We decided instead to place side by side the originally inferred BEM and a set of additional BEM fragments, each describing an additionally discovered behavior of the component. We need only BEM fragments because we are only interested in keeping track of the previously missing behavior. We then need to represent only an execution trace containing such behavior. We also have to report the states produced by the execution of each operation. That enriched execution trace is exactly what we call BEM fragment. Doing that, we can easily keep track of all the relevant executions 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 extend the inference algorithm described in Sect. 3 to build a PBM starting from the information contained in more than one BEM so that the resulting PBM 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.

As an example, a violation requiring to update the models of STORAGESERVICE reported in Figs. 1 and 2 occurs after the execution of the following trace: *StorageService().putNextEntry*(e_x)*.putNextEntry*(e_x)*.putNextEntry*(e_y)*. write(d)*. After the execution of the second *putNextEntry* the PBM prescribes the component to be in either state $S2$ or $S3$. By invoking *putNextEntry* as a state discriminator, we discover to be in $S2$, because for some entries (e_x) the operation is exceptional while for the entries e_1 and e_2, belonging to the scope used at inference time, the operation would terminate normally. The third *putNextEntry* is still nondeterministic and it is resolved in the same way. So according to the protocol described by the PBM, the component is in state $S2$ before the execution of the *write* operation. However, the component now predicts the write to be exceptional for every possible parameter, while the component executes normally, because the last created entry is valid. The violation unveiled by the trace above is actually due to scope effects. While the initial PBM correctly describes the component behavior when only two entries are used as parameters for *putNextEntry*, this scope is not sufficient to describe all the component behaviors. During execution, when the scope of the analysis is enriched with other different entries, such previously unexplored behaviors emerge. Such a violation due to scope effects is a typical case that requires only model update.



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

State	*close*	*putNextEntry*	*write*
	Observer Abstraction		
$S0$	$\rightsquigarrow StorageEx$	—	$\rightsquigarrow StorageEx$
$S1$	—	$[-, \rightsquigarrow StorageEx]$	—
$S2$	—	$[-, \rightsquigarrow StorageEx]$	$\rightsquigarrow StorageEx$
$S3$	—	$\rightsquigarrow StorageEx$	—
$S4$	—	$\rightsquigarrow StorageEx$	$\rightsquigarrow StorageEx$
$S5$	—	$\rightsquigarrow IO$	$\rightsquigarrow IO$

Fig. 4 An updated PBM of the STORAGESERVICE component

In this case, BEM state characterization is enriched with results obtained with entries e_x and e_y for the *putNextEntry* operation and considering d as another possible parameter for *write*. In addition to that we also generated another BEM describing the execution trace which unveiled the violation. Figure 4 shows the PBM obtained at the end of the model update process. Scope effects are still there in states $S3$ and $S4$, but the updated version of the model takes into account properly the possibility to store infinitely-many different entries with the loops through states $S1$ and $S2$.

The detection of another interesting behavior missing in the initial models occurs when we try to write an empty string of data when no entry is available. In this case, the expected *StorageException* 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 new BEM fragment describing 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 are only available online at [29].

5.2.2 Change detection

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 operations 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 Figs. 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 [29].

It is important to remark that this methodology is able to identify behavioral changes only when there is at least one failing test case in the ones that are synthesized by the Bem. In other words, our methodology identifies changes only if they manifest themselves as violations of previous observations synthesized by Bems. Since Bem describes the behavior of a component in a limited scope, and thus they do not contain information about every possible component execution, it is possible that what is an actual change is detected as a newly observed behavior. However, since the change is outside the knowledge inferred at design phase, this different interpretation is safe: it corresponds to the change of an initial behavior that has never been observed, and from a client's point of view, it can be safely considered as a new observation.

Our approach can also effectively detect removed behavior; since we do not consider changes in the component signature, what we mean is that a set of behaviors that previously executed normally now became exceptional. As an example of removed feature, consider a service having part of its operations always available while others are available only after user authentication. It may happen that the service policy changes and a new version of the server always requires the authentication to be performed first. We can consider the change as the removal of the service publicly available features. Clients may not be aware of the removal and thus they may try to use the component as the feature would still be available. At that point the client will get an exception as a response. In that case, the obtained result differ from the data contained

in the models and therefore we can provide the developers with the information about the behavioral change. Our tool does not explicitly reports it as a *feature removal*, but the provided counter-example should make it easy to understand what happened. The policy to stop the execution of the service when a change is detected avoids the building of inconsistent models. When a violation occurs, the application cannot rely anymore on the behavior provided by the external service. Therefore stopping its execution is not a big deal.

6 RabbitMQ example

We now consider RabbitMQ [25] as a case study to show the effectiveness of our approach with *real* software. RabbitMQ is an open source implementation of the Advanced Message Queuing Protocol (AMQP), a protocol for message oriented middleware. It consists of a server providing the messaging functionalities and several clients written in different programming languages. Developers can use the RabbitMQ functionalities in their applications by embedding the client as an external component. Available online documentation does not contain any formal description of how to interact with RabbitMQ. There are several examples of basic use cases and a comprehensive documentation of all the client's methods. That kind of documentation is good to get a grasp about how to have the client work, but does not give much information about what happens in special cases that arise when a particular sequence of operations is executed. Moreover, the documentation of each class of the Api presents methods in isolation, making it hard to understand how they interact and how component's state affects the outcome of an operation.

```
public interface Channel{
    void queueDeclare(String queueName)
            throws IOException;
    void queueDelete(String queueName)
            throws IOException;
    String basicGet(String queueName)
            throws IOException;
    void basicPublish(String queueName, String message)
            throws IOException;
    void close()
            throws IOException;
}
```

Listing 2: RabbitMQ channel interface

We considered the Java client and, more specifically, we modeled the behavior of the class representing the communication channel. It is the main class of the client and it is used to manage publish-subscribe queues as well as to send and receive messages. In this work we focused on channel's basic functionalities, selecting a set of methods which enabled the core capabilities of RabbitMQ. Since our approach focuses on models of single components in isolation, we ignored the most sophisticated features which would require us to model the interaction of multiple components. The Api we considered is reported in Listing 2. AMQP routes messages using named queues,

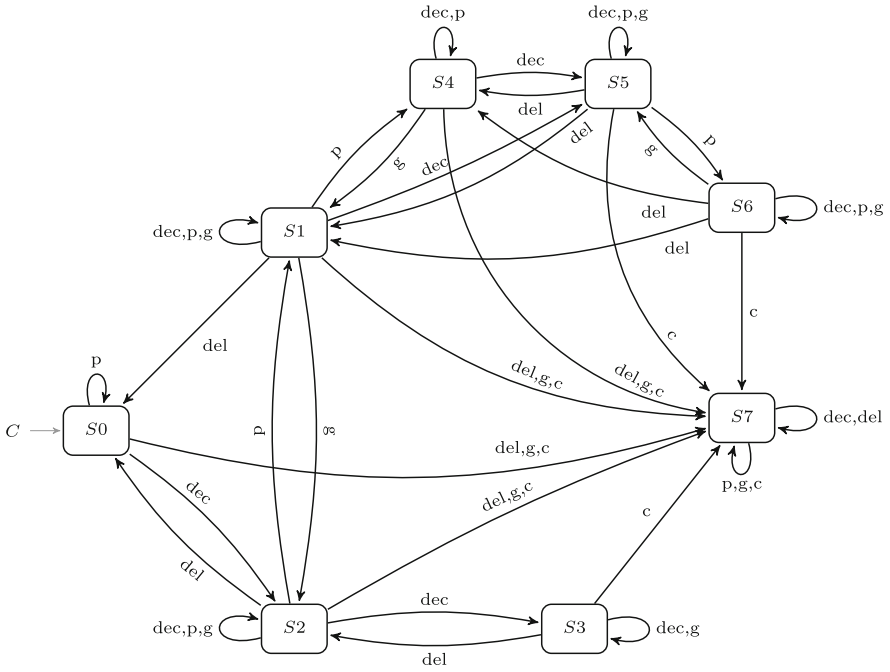
thus the Api have operations to declare and delete them. Queue names are, of course, also required by the operations to publish and retrieve data. When the client is not able to perform an operation, exceptions are thrown. Unfortunately, the client does not use only checked exceptions, that must be declared in the method signature. This may pose an issue in the understanding of the Api and may lead to errors which will manifest themselves only at run time. Our approach is particularly useful when there are unchecked (runtime) exceptions: Bems and Bems clearly show to the developers when that kind of exceptions are thrown. Moreover, with run time monitoring, we can help developers taking into account the unanticipated behavior as soon as it happens.

The state of the component under analysis is spread among the client itself and the server. This means that we have to control both the operations executed on the client and the internal state of the server in order to be able to model the RabbitMQ client. In fact, if the internal state of the server changes, the same sequence of operations may lead to different results depending on the active queues and on the messages they contain. Considering only the state of the RabbitMQ client without taking into account the fact that part of the state is held by the server may lead to different results for the same sequence of operations performed at different times. From the client perspective, this would result in an apparent non-deterministic behavior, while the whole system composed of client and server still behaves deterministically. In this work we face this issue by ensuring that each execution trace we consider starts from the same, clean, server state. To ensure this, we specify a special finalizing operation to be used during Bem generation. When the tool finishes to analyze an execution trace, it invokes the finalizing operation that cleans up the server state by deleting all the still active queues. Thus, all the execution traces behave as if they were performed on a just initialized instance of the RabbitMQ server. In that way we get rid of non-deterministic effects due to differences in server internal state.

6.1 Model inference

To infer initial models at design time we had to rely on instance pools provided by users. The technique we developed to extract instances from test cases is not so effective for RabbitMQ client because the class we want to model does not have related unit tests. Each channel functionality is tested in isolation and there are other test cases specifically designed to reproduce bugs. The high coupling between the internal state of the server and the outcome of the operations on client is a possible motivation for the choice of functional testing over unit testing. We then had to manually identify suitable values for the parameter instances. Luckily this was not a hard task because the considered operations accept any string value and there are not *magic values* triggering particular behaviors. We hence considered two different queue names shared by all the methods and a single message to be sent. We built the instance pool by looking at the code snippets contained in the documentation.

Figure 5 reports the Pbm obtained for RabbitMQ Java client. The model contains significant insights about the behavior of the component. For instance, the model clearly shows that *basicPublish* does not ever throw an exception. Its result is normal even when the channel is not ready to accept a message. This information is important



Legend: *C:Channel, dec:queueDeclare, del:queueDelete, p:basicPublish, g:basicGet, c:close*

State	*queueDeclare*	*queueDelete*	*basicGet*	*basicPublish*	*close*
	Observer Abstraction				
*S*0	—	$\rightsquigarrow IOEx$	$\rightsquigarrow IOEx$	—	—
*S*1	—	$[-, \rightsquigarrow IOEx]$	$[-, \rightsquigarrow IOEx]$	—	—
*S*2	—	$[-, \rightsquigarrow IOEx]$	$[-, \rightsquigarrow IOEx]$	—	—
*S*3	—	—	—	—	—
*S*4	—	$[-, \rightsquigarrow IOEx]$	$[-, \rightsquigarrow IOEx]$	—	—
*S*5	—	—	—	—	—
*S*6	—	—	—	—	—
*S*7	$\rightsquigarrow ClosedEx$	$\rightsquigarrow ClosedEx$	$\rightsquigarrow ClosedEx$	$\rightsquigarrow ClosedEx$	$\rightsquigarrow ClosedEx$
	Modifier Behavior Abstraction				
*S*0	*Variant*	*Variant*	*Variant*	*Invariant*	*Variant*
*S*1	*Variant*	*Variant*	*Variant*	*Variant*	*Variant*
*S*2	*Variant*	*Variant*	*Variant*	*Variant*	*Variant*
*S*3	*Invariant*	*Variant*	*Invariant*	*Variant*	*Variant*
*S*4	*Variant*	*Variant*	*Variant*	*Invariant*	*Variant*
*S*5	*Invariant*	*Variant*	*Variant*	*Variant*	*Variant*
*S*6	*Invariant*	*Variant*	*Variant*	*Variant*	*Variant*
*S*7	*Invariant*	*Invariant*	*Invariant*	*Invariant*	*Invariant*

Fig. 5 A PBM of the RABBITMQ component

to developers which cannot assume that the normal outcome of the operation means that other clients actually received the message. They have to use other strategies to acknowledge the reception of a message. Moreover the method declares to throw an *IOException*, which may be misleading. The inferred model presents scope effects in states *S*3, *S*5 and *S*6. In the STORAGESERVICE example we showed that they are not harmful because they only describe the behavior of the component when all the considered parameters are used. In the RABBITMQ example we can exploit the information they provide for all the observed parameters in order to find other interesting

insights about component's behavior that holds for all the possible input values. For example, given the invariance of the modifier behavior abstraction for *queueDeclare*, we can observe that the operation has an effect only the first time it is called with any given parameter. $S7$ represents the state in which all the functionalities of the component are disabled. Developers can put the component in that state by calling the *close* operation, but that is not the only way to get to that state. The channel is automatically closed every time an error occurs. Developers should be aware of this behavior in order to avoid problems at run time when an exception will be thrown. Other states represents the different combinations of behavior depending on queues availability and on the messages published on each queue. The model also highlights the dual operations of the component. When we declare a new queue we can go back to the initial state by deleting it. The same applies for the pair *basicPublish*, *basicGet*.

6.2 Run time monitoring

The model inferred at design time for the RabbitMQ client covers all its observable behavior. For this reason in this example we present only the change detection feature of our tool. A similar situation should be the preferred way to use our tool. Ideally developers should be able to get a full knowledge about the behavior of the component at design time and leave only the change detection task to be performed at run time.

To simulate a change that may actually take place in the real RabbitMQ server, we changed the interaction protocol with the component so that the *queueDeclare* operation does not allow multiple redeclaration of a queue. In our implementation the redeclaration of a queue causes an error which blocks further interactions with the component. A simple execution trace which unveils the change is *Channel().queueDeclare("q").queueDeclare("q"). basicPublish("q","m")*. Considering the original component, the execution trace terminates normally. In fact we publish a message on a properly declared queue; the redeclaration simply does not change the state of the server. Conversely, with the modified version of the component the outcome of the last operation is exceptional because of the error due to the queue redeclaration. That violation is then easily identified as a functional change when we replay an execution trace containing a queue redeclaration.

7 Related work

The models discussed and proposed in this paper describe the behavior of a software component by making explicit which operations are enabled in different states. This underlying idea has been introduced quite some time ago through the concept of Typestate in [30]. The most similar abstraction has been proposed in [7], which presents a technique called Contractor to build an enabledness model from declarative pre/postcondition-based specifications (contracts) through static analysis. The goal here is instead to validate software components seen as black boxes. From a formal point of view, Bems can be seen as an extension of enabledness models presented in [7]; however, the concept of enabledness, in our case, is deeply rooted on what the client can observe, that is, normal or exceptional behaviors. Instead, in

CONTRACTOR, the concept of enabledness is given in terms of satisfaction of operation preconditions, as typically expressed through contracts. Moreover, the models we propose also provide a more precise abstraction mechanism for operations with parameters.
TAUTOKO [5] generates finite state machine models starting from an existing test suite; in that approach, the abstraction is not based on operation enabledness. Our tool does not infer the model directly from a particular set of test execution traces, as TAUTOKO; it rather generates execution traces starting from instance pools.

Several works have been proposed to infer finite state models of software components through dynamic or static analysis [17,31]. For example, the work in [17] synthesizes a finite state machine representing legal sequences of operations from a set of legal traces. The resulting finite state machines model legal sequences of operations but without using any state abstraction; thus, there is no direct link to what the client can observe and what the state of the component is. Another approach that dynamically infers behavior model is ADABU [6]. Like our approach it infers model through dynamic analysis, but there are difference in states characterization. The valued returned from a class observers are abstracted depending on their data type. State characterization is obtained by combining the abstraction for all the observer of the analyzed class.

Dynamic analysis techniques to build behavioral models are rooted in the more general methodologies to infer an automaton from a set of traces it has to accept. Techniques to infer finite state machine from a set of observation were first introduced in [1]. Further works on that topic are [10,24] which present incremental approaches. Incrementality is very close to our idea of extending the model as soon as we discover new behaviors.

Bertolino et al. [4] developed a technique to support web-service composition through behavior protocol models. They combine a data-type based model synthesis with a testing phase to assess the conformance of the inferred models with the actual service behavior. State-based models have also been used for testing purposes only. For instance in [19] they are used to test ajax web applications. That work combines static and dynamic analysis to generate behavior models. Test cases are then generated accordingly to the information contained in the model.

Rosu et al. [27] presented Monitoring Oriented Programming. They defined a methodology in which runtime monitoring becomes a basic design principle. Their framework is then able to generate the monitors and to integrate them with the actual system. With the analysis of the data collected by the monitors, system execution can be validated. Monitoring also enables reaction to the violation of certain user-defined properties.

Monitoring of both functional and non-functional properties of service-based systems are described in [3]. Our technique is based on BEMS and BEMS, therefore we are able to model and monitor very precisely functional properties of a software component. INVITE [22] proposed and 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 that can highlight a behavioral change.

Tracer [18] builds runtime models from execution traces enabling richer and more detailed analysis at a higher abstraction level. In [15] 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.

Diva [21] leverages the usage of models at runtime to support dynamic adaptation. The monitoring carried on by Diva focuses on the parameters describing the execution environment of the application while we are concerned about functional correctness. Both these approaches are required in the development of reliable and dynamically adapting systems. The Diva framework takes into account all the aspects of dynamic adaptation and it is based on a model driven engineering approach. In our work we target existing software that is not necessarily engineered with in a model driven fashion, therefore we had to introduce model inference methodologies and limit the application domain of our prototype to change detection.

8 Conclusions and future work

Behavior models can be useful throughout all the lifecycle of a software component. Like other software models, behavior models are traditionally used at design time to support system designers in their reasoning. However, they can also play a significant role after the application is deployed by monitoring its execution and checking system properties. This is particularly useful in the context of systems in which verification must extend to run time, because unexpected changes may occur dynamically.

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 concerning behavioral changes in the monitored components. Preliminary experiments focusing on Java classes 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.

In this work, we considered isolated deterministic components. It would be useful to extend our approach to model the interaction of multiple components. With interaction models we would be able to describe the complex behaviors which arise when there are several components sharing somehow their state. For example, in RabbitMQ, we may want to describe the interaction of multiple clients with the server as well as how the channel works together with a message listener. Another interesting research direction that is worth investigating is the re-definition of Bems to describe non-deterministic behaviors.

Albeit there are a lot of the components accessible through remote Apis which allow the instantiation of multiple instances which we can use as shadow components, we might develop techniques to relax the requirement of the availability of other instances. With that improvement we would be able to apply our approach even to model and

monitor Apis which allows the interaction with systems that only present a single instance shared by all the clients.

Other possible further directions concern reasoning at run time. Usually, runtime reasoning require a trade off between precision and the time required to perform the computation. Examples of these techniques are the ones providing exact solutions based on optimization algorithms and other, much faster, based for instance on heuristics or evolutionary techniques.

In this work, we relied on an exact reasoning technique which is more suitable for our application domain since we want to detect exact information about changes in the monitored components. Reasoning at the protocol level also provides relatively fast conformance checking against the behavior models. However, to monitor more complex properties in a possible extension of the current work, we may take into account also an a hybrid approach to try to get the benefits from both exact and approximate reasoning approaches.

Finally, we should also plan an empirical evaluation of the quality and the usefulness of the inferred models. We should provide our models to developers which actually have to use a software component. We then would evaluate whether developers can get a better understanding of the behavior of the component by looking at the models rather than by leveraging the knowledge they can get in other ways.

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

References

1. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 75(2):87–106
2. Baresi L, Ghezzi C (2010) The disappearing boundary between development-time and run-time. In: FoSER '10, New York, NY, USA
3. Baresi L, Guinea S (2011) Self-supervising bpel processes. IEEE Trans Softw Eng
4. Bertolino A, Inverardi P, Pelliccione P, Tivoli M (2009) Automatic synthesis of behavior protocols for composable web-services. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, ESEC/FSE '09. ACM, New York, pp 141–150
5. Dallmeier V, Knopp N, Mallon C, Hack S, Zeller A (2010) Generating test cases for specification mining. In: ISSTA '10: proceedings of the (2010) ACM SIGSOFT international symposium on software testing and analysis, Trento, Italy
6. Dallmeier V, Lindig C, Wasylkowski A, Zeller A (2006) Mining object behavior with adabu. In: Proceedings of the 2006 international workshop on dynamic systems analysis, WODA '06. ACM, New York, pp 17–24
7. de Caso G, Braberman V, Garbervetsky D, Uchitel S (2012) Automated abstractions for contract validation. IEEE Trans Softw Eng 38(1):141–162
8. Di Nitto E, Ghezzi C, Metzger A, Papazoglou M, Pohl K (2008) A journey to highly dynamic, self-adaptive service-based applications. ASE
9. Doong R, Frankl PG (1994) The ASTOOT approach to testing object-oriented programs. ACM Trans Softw Eng Methods 3(2):101–130
10. Dupont P (1996) Incremental regular inference. In: Proceedings of the third ICGI-96. Springer, Berlin, pp 222–237
11. Ghezzi C, Mocci A, Monga M (2009) Synthesizing intensional behavior models by graph transformation. In: Proceedings of the 31st international conference on software engineering, ICSE '09. IEEE Computer Society, Washington, pp 430–440

12. Ghezzi C, Mocci A, Sangiorgio M (2011) Runtime monitoring of functional component changes with behavior models. In: Models@run.time '11, Wellington, New Zealand
13. Goguen J, Malcolm G (2000) A hidden agenda. Theor Comput Sci 245(1):55–101
14. Guttag J, Liskov B (2001) Program development in Java: abstraction, specification and object-oriented design. Addison-Wesley, New York
15. Hooman J, Hendriks T (2007) Model-based run-time error detection. In: Models@run.time '07, Nashville, USA
16. Jackson D (2011) Software abstractions: logic, language, and analysis. MIT Press, Boston
17. Lorenzoli D, Mariani L, Pezzè M (2008) Automatic generation of software behavioral models. In: Proceedings of the 30th international conference on software engineering, ICSE '08. ACM, New York, pp 501–510
18. Maoz S (2009) Using model-based traces as runtime models. IEEE Computer
19. Marchetto A, Tonella P, Ricca F (2008) State-based testing of ajax web applications. In: 1st International conference on software testing, verification, and validation, pp 121–130
20. Mocci A (2010) Behavioral modeling, inference and validation for stateful component specifications. PhD thesis, Politecnico di Milano, Milano, Italy
21. Morin B, Barais O, Jezequel J-M, Fleurey F, Solberg A (2009) Models@ run.time to support dynamic adaptation. Computer
22. Murphy C, Kaiser G, Vo I, Chu M (2009) Quality assurance of software applications using the in vivo testing approach. In: Proceedings of the 2009 international conference on software testing verification and validation, ICST '09. IEEE Computer Society, Washington, pp 111–120
23. Oracle, java se 6.0 doc (2011) http://download.oracle.com/javase/6/docs/index.html
24. Parekh R, Nichitiu C, Honavar V (1998) A polynomial time incremental algorithm for learning dfa. In: Proceedings of the fifth ICGI-98
25. Rabbitmq website (2011) http://www.rabbitmq.com/
26. Rivest RL, Schapire RE (1989) Inference of finite automata using homing sequences. In: Proceedings of the twenty-first annual ACM symposium on theory of computing, STOC '89. ACM, New York, pp 411–420
27. Roşu G, Chen F (2012) Semantics and algorithms for parametric monitoring. Logical Methods Comput Sci 8(1):1–47, 2012. Short version presented at TACAS 2009
28. Sannella D, Tarlecki A (2010) Foundations of algebraic specification and formal software development. EATCS monographs on theoretical computer science. Springer, Berlin
29. Spy at runtime (2011) http://home.dei.polimi.it/sangiorgio/spy/index.xhtml
30. Strom RE, Yemini S (1986) Typestate: a programming language concept for enhancing software reliability. IEEE Trans Softw Eng 12:157–171
31. Whaley J, Martin MC, Lam MS (2002) Automatic extraction of object-oriented component interfaces. In: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, ISSTA '02. ACM, New York, pp 218–228
32. Xie T, Martin E, Yuan H (2006) Automatic extraction of abstract-object-state machines from unit-test executions. In: International conference on software engineering, research demos, pp 835–838, May 2006