Dynamic Detection of COTS Component Incompatibility

Leonardo Mariani and Mauro Pezzè, *University of Milan Bicocca*

Developing component-based systems introduces new compatibility problems, which an analysis technique can reveal by automatically deriving and dynamically checking behavioral models.

The development of COTS-based systems shifts the focus of testing and verification from single components to component integration. Independent teams and organizations develop COTS components without referring to specific systems or interaction patterns. Developing systems that reuse COTS components (even high-quality ones) therefore presents new compatibility problems. David Garlan, Robert Allen, and John Ockerbloom reported that in their experience, integrating four COTS components took 10 person-years (rather than the one planned person-year), mainly because of integration problems.[1] According to Barry Boehm and Chris Abts, three of the four main problems with reusing COTS products are absence of control over their functionality, absence of control over their evolution, and lack of design for interoperability.[2]

Traditional integration testing techniques incrementally validate the integration of the modules that compose the system being tested. Test designers can derive test cases from both the source code and the specifications.[3,4] Unfortunately, COTS components are often distributed without the source code and with incomplete specifications. Moreover, developers can use them in ways that the original design didn't predict. So, testers must test them in the contexts in which they're used.

Our proposed technique, called *behavior capture and test*, detects COTS component incompatibilities by dynamically analyzing component behavior. BCT incrementally builds behavioral models of components and compares them with the behavior the components display when reused in new contexts. This lets us identify incompatibilities, unexpected interactions, untested behaviors, and dangerous side effects.

How BCT works

BCT builds two kinds of models for each service the components offer. I/O models are Boolean expressions describing the relations between the values that components exchange. Interaction models are finite-state automata (FSA) representing the sequences of interactions triggered by invoking the services. The models are built automatically and describe the components' behavior in different contexts. For example, we can build behavioral models during testing (unit, integration, and system testing) or while using the components in the field under different operative conditions.

When components are reused in new contexts—for example, for building new systems or updating obsolete components—we auto-

 0740-7459/07/$25.00

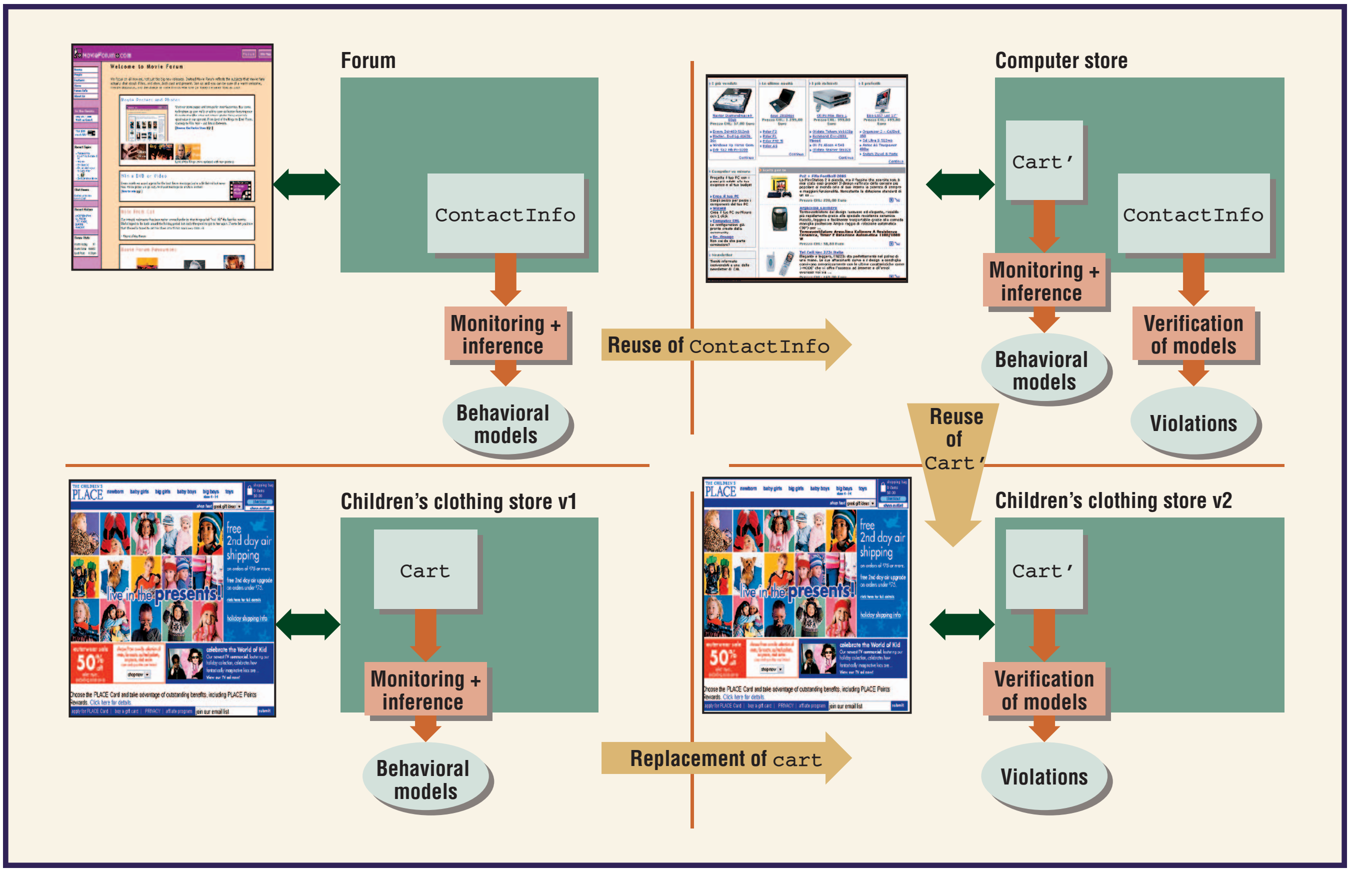

Figure 1. Automatic verification of COTS components.

matically compare their interactions in the new contexts with the behavioral models built into former executions to identify new behaviors. These behaviors might be legal interactions that haven't yet been tested well or erroneous interactions not previously revealed and fixed. System integrators can update test suites to identify and fix subtle integration faults before delivering the new system.

Dynamic analysis works directly on the executable code without requiring source code or specifications. So, the technique applies straightforwardly to COTS components, which are particularly difficult to address with classic test and analysis techniques. BCT can incrementally update behavioral models to automatically record new uses. So, our technique applies well to dynamically evolving component-based systems.

Dynamic analysis of COTS components

BCT has two phases. In the first phase, we monitor COTS components and distill behavior models from their executions. In the second phase, we upgrade and reuse components and dynamically verify the corresponding models. The test designers examine violations. If a violation is due to a legal behavior the component hasn't displayed before, the designers can refine the models; if it's due to an integration fault, they can fix the problem.

Figure 1 shows a typical example of COTS component reuse across different systems. The developers of a store that sells computers reuse a `ContactInfo` component that has previously been used in a forum application. Later, the developers of a store that sells children's clothing might decide to update their store's old `Cart` component with the computer store's new `Cart'` component, because the `Cart'` component supports registered users.

The computer store's developers can use the behavioral models inferred for the `ContactInfo` component in the context of the forum to verify `ContactInfo`'s behavior when used as part of the computer store. When updating the `Cart` component in the children's clothing store context, developers can use the behavioral models previously inferred for that component to verify the `Cart'` component's compatibility in this context.

As we mentioned earlier, component behav-

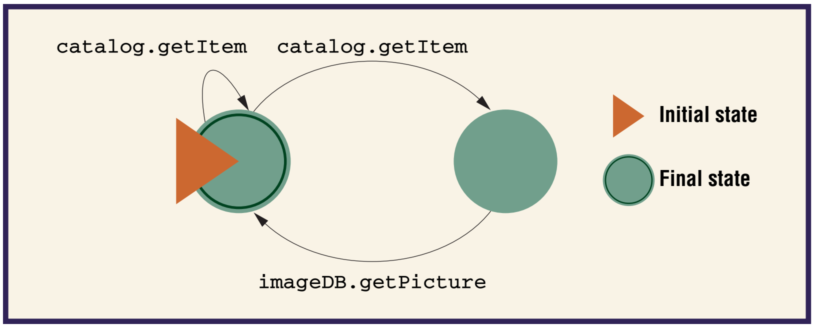

Figure 2. An interaction model inferred for the service `getCart()`, which the `Cart` component implements.

iors are expressed in the form of I/O and interaction model. Here is an example of an I/O model inferred for the service `void addItem (CartItem item)`, which the `Cart` component implements:

```
item.quantity>0
item.UnitCost <= item.totalCost
```

Figure 2 shows an example of an interaction model inferred for the service `getCart()`, which the `Cart` component also implements.

The interaction model specifies that each time the system invokes the `getCart()` service, the `Cart` component interacts zero or more times with a catalog component to retrieve details of each item (`catalog.getItem`). If an item is associated with a picture, the component then interacts with the `imageDB` component to get that picture (`imageDB.getPicture`).

The two kinds of models represent different aspects of a component's behavior. I/O models describe properties over the exchanged data, whereas interaction models capture properties of the interaction protocol that each component implements. This information is useful both when components are reused across different systems and when they're replaced with other components. Our running example considers both cases: `ContactInfo` is reused in the computer store after having been used in the forum, and the children's clothing store's `Cart` component is replaced with the `Cart′` component (previously used in the computer store). When we reuse a component, we can automatically reveal new interactions by checking at runtime its behavior in the new system with respect to the models computed for the component in previous systems. Such interactions might indicate both new legal behaviors and failures that arise with the new usage. When we upgrade an old component, we can compare the behavior models associated with the old and the new components to reveal incompatibilities. We can also monitor the behavior models associated with the new components to reveal new interactions.

Automatically deriving and checking models presents several issues:

- Component interactions involve not only simple data but also complex objects that must be suitably monitored.
- We must reduce the enormous number of I/O and interaction traces to models that generalize the observed behavior.
- The system must automatically identify behaviors incompatible with the models previously inferred.

Extracting and recording complex information

To identify information exchanged during computation, you must first monitor component interactions (usually given as references to complex objects) and then extract the information that the object attributes carry. You can monitor interactions in many ways. In our prototype, we implemented monitoring functionality with aspect-oriented frameworks (see http://aspectwerkz.codehaus.org), and we extracted the information encapsulated in the objects with a technique called *object flattening*. Object flattening recursively scans the object structure up to a given depth, similarly to serialization of objects to XML format.[5] It avoids looping in complex data structures by marking the fields it has already inspected. It accesses private fields using reflection (see http://java.sun.com/j2se/1.3/docs/guide/reflection), which is available in modern programming environments such as .NET and Java.

Aspect-oriented frameworks and reflection can also work in the absence of source code, so they're generally applicable to COTS and binary components. Additional techniques are available for monitoring and extracting information from components provided with source code.

Inferring behavior models

We generate I/O models with the Daikon inference engine.[6] Daikon accepts a trace file, which contains the data recorded by monitoring a given set of variables, and generates a set of predicates that hold over those variables for the given samples. Here's an example of an I/O model that BCT inferred using Daikon for a `ContactInfo` component:

```
contactInfo.address.streetName.
length > contactInfo.address.
city.length

contactInfo.firstName.length
>= 4
```

BCT records interactions between components by storing the beginning and termination of method executions. We record interaction sequences thread by thread, using the thread ID to distinguish interactions of different threads. Most modern languages (such as Java and .NET) and operating systems (such as Windows and Linux) provide thread IDs. We can therefore work with sequential traces only. Recording interaction traces produces huge amounts of data and increases accesses to the disk. So, recording all traces requires a large amount of storage and reduces the application's performance.[7] To overcome the negative effects of trace recording, we need a technique that consumes and eliminates traces incrementally. Traces represent positive samples—that is, samples that must be included in the inferred FSA. So, we need a technique that works with positive samples only.

Algorithms that generate FSA and require only positive samples are based on kTail.[7–9] These algorithms combine traces into a prefix tree automaton and then generalize the observed behaviors by merging states that share their k-future (that is, that are indistinguishable from the outgoing paths of length k). Because the algorithm can't determine a state's future until all traces have been processed, we can't apply kTail and its variants incrementally. Available incremental inference algorithms don't work with positive samples only; they rely on the availability of additional information.[10,11]

To satisfy our domain's requirements, we developed a new inference algorithm, called kBehavior, which works incrementally on a set of positive samples only. The algorithm exploits the considered traces' characteristics. Because traces correspond to component interactions, we expect several recurrent patterns. For example, when interacting with a cart, we might frequently observe subsequences that first get a cart and then get the items in that cart. The algorithm identifies subsequences of a new trace in the current FSA and connects the identified subsequences to include the new trace in the FSA.

Consider, for example, a new trace that gets a cart, gets the items in the cart, and then updates the quantity associated to an item. If the current FSA includes a subpath that gets a cart and its items but doesn't update the quantity, the algorithm will augment the FSA by adding a new edge that models the update-item operation. It thereby incrementally extends the current FSA with the new trace.

Figure 3 shows the kBehavior algorithm. For convenience, we've shown methods without parameters. The inference algorithm identifies methods through the complete signature.

Figure 4 illustrates the algorithm using examples taken from a bank account manager.

To evaluate the quality of the inference our algorithm provides, we experimentally compared kBehavior and kTail-based algorithms. In particular, we considered kTail; Jonathan E. Cook and Alexander L. Wolf's algorithm, which adds an extra reduction step to kTail;[9] kInclusion, which merges two states if the k-future of the first is included in the kFuture of the second; and Steven P. Reiss and Manos Renieris' algorithm, which merges two states if they share at least one k-future.[7] Table 1 shows our comparison's results.

We considered three sets of traces. The first set included traces representing three basic cases:

- traces that share only the initial symbol,
- traces containing one pattern repeated zero or more times, and
- traces containing one pattern repeated one or more times.

These results obtained with this set are in the "Ad hoc traces" column. The second set included traces taken from Cook and Wolf's algorithm,[9] and the results are in the "Cook and Wolf" column. The third set included traces generated by executing components of Jedit (JeditBg and Sorting) and Object Flattener with integration test suites available for the components. (Jedit is an implementation of a general and extensible editor containing thousands of classes. See www.jedit.org for more information.) These results obtained with this set are in the "JeditBg," "Sorting," and "Object Flattener" columns, respectively.

All algorithms process basic traces, but kTail-based algorithms don't discriminate between sets of traces including at least one occurrence of the pattern and sets of traces also

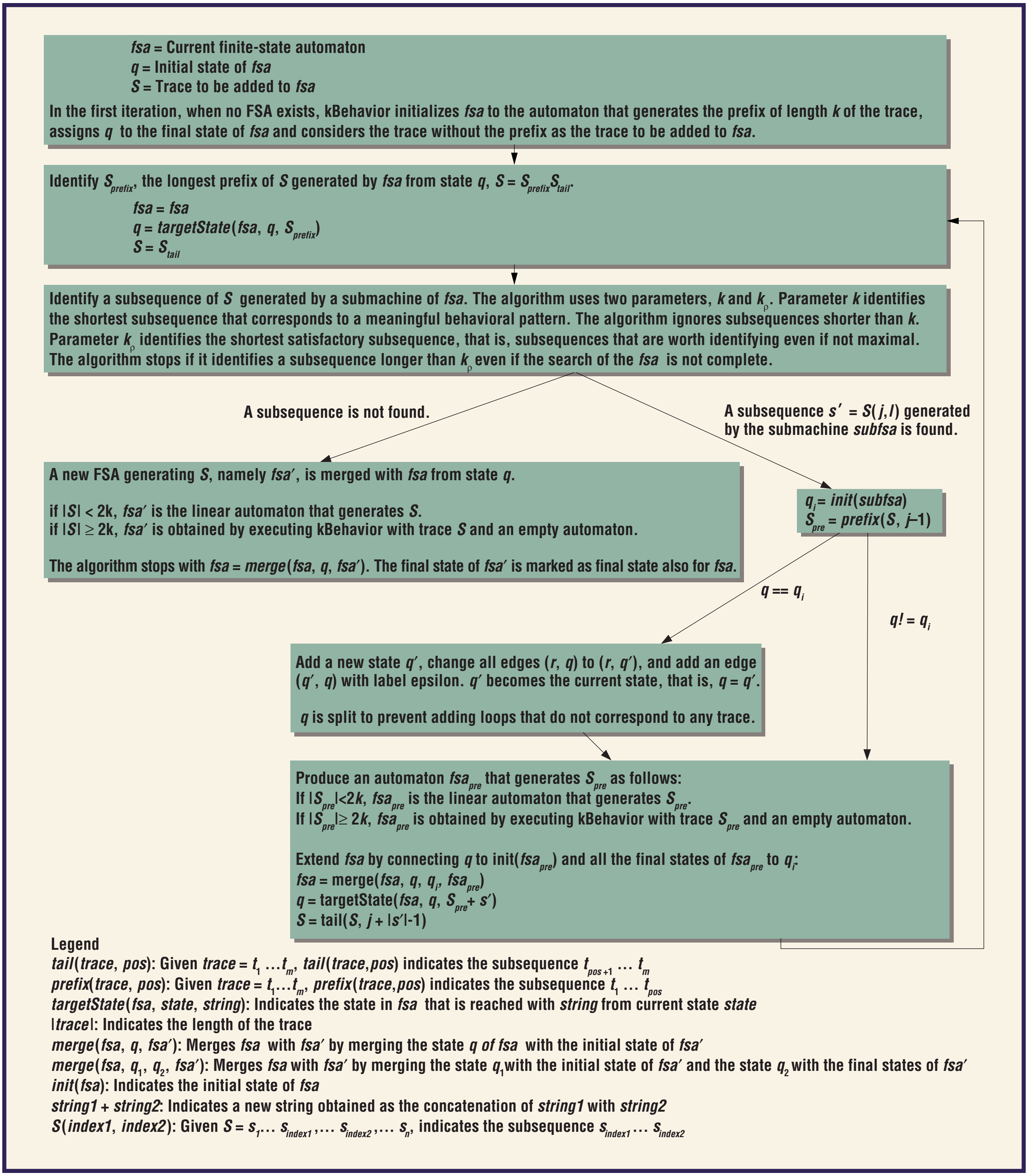

Figure 3. The kBehavior algorithm.

including empty traces. In other words, the FSAs generated from sets of traces including at least one pattern overgeneralize the sample, because they also generate empty traces. All algorithms process well the traces that Cook and Wolf proposed, but kBehavior generates the smallest FSA, and kBehavior processes traces extracted from the Jedit components well, while kTail-based algorithms generate automata that either overgeneralize or overrestrict

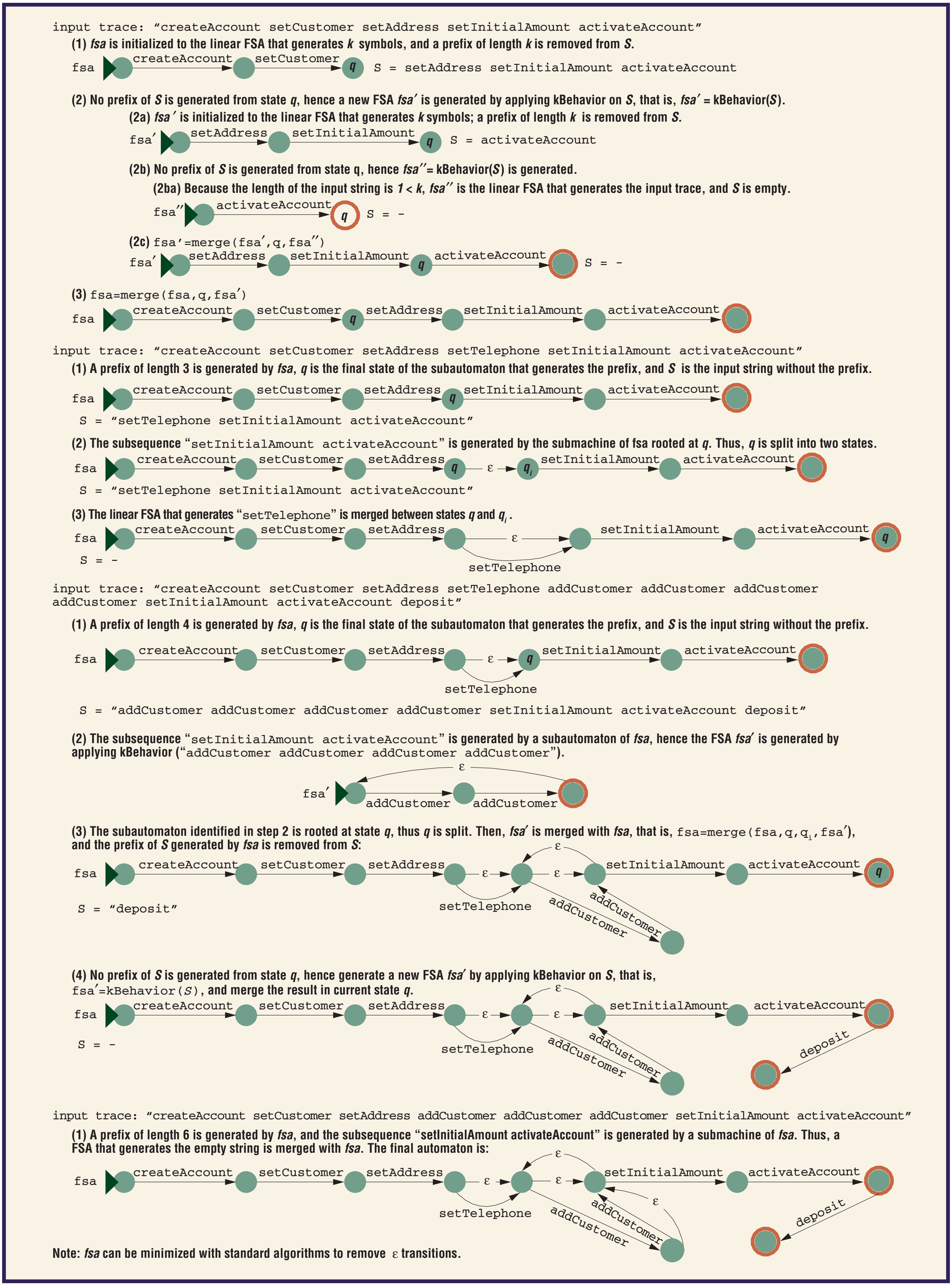

Figure 4. An example application of kBehavior with k = 2.

Table 1

Results of comparing kBehavior with kTail-based algorithms

	Ad hoc traces	Cook and Wolf	JeditBg	Sorting	Object Flattener
kTail	OK/OG*	OK (Suitable finite-state automata)	OK	OR (Overrestricted FSA)	>24 hours (Didn't terminate after 24 hours)
Cook and Wolf	OK/OG	OK	OK	OG (Overgeneralized FSA)	>24 hours
kInclusion	OK/OG	OK	OG	OG	>24 hours
Reiss and Renieris	OK/OG	OK	OG	OG	>24 hours
kBehavior	OK	OK smallest (Smallest suitable FSA)	OK	OK	OG

Suitable FSA for basic cases 1 and 2, but overgeneralized FSA for basic case 3

the sample. None of the kTail-based algorithms applied to the Object Flattener traces terminated after 24 hours of execution. kBehavior processed the input in minutes and produced a useful automaton that overgeneralizes the input sample. We obtained similar results while analyzing other components of large size, for example, the Aelfred XML parser within Jedit and PtPlot.[12]

The experimental evaluation indicates that in general, kBehavior performs better than kTail-based algorithms. Good performance depends on the characteristics of traces that represent typical component interactions and on kBehavior's incremental nature, which lets it quickly identify behavioral patterns and process many traces. However, kBehavior's performance depends on the order in which it processes traces. In particular, when the initial traces are short and consecutive traces are similar, kBehavior can perform worse than kTail-based algorithms. Fortunately, you can avoid these cases by suitably planning the executions of test cases. We've seen low, albeit acceptable, performance only with ad hoc-generated traces, but not in experiments with traces generated from components monitored so far.

Detecting behavior incompatibilities

We've previously presented a preliminary evaluation of BCT.[12] Here, we provide additional data we obtained from quantitative investigations of two freely available third-party systems. The first system is a small-size implementation (tens of classes) of the FreeSudoku game (see http://freesudoku.sourceforge.net). We used this system to study applying BCT to component replacement. The second system is Jedit, which we used to study applying BCT to components reused across systems.

In the first example, we replaced the component that creates and manages the board in FreeSudoku 0.9.6. This component plays a critical role in the architecture because it implements most of the application logic and will likely be replaced in future releases. We derived an initial set of test cases using the category partition method.[13] While testing the system, we monitored seven methods for I/O data and 11 methods for interaction data (we disabled monitoring of I/O data for highly executed methods to limit overhead). BCT produced 656 I/O predicates for three methods and none for the other four methods, which don't exchange relevant parameters with the rest of the system. The large number of predicates for three methods is due to the presence of conditions on all cells of the Sudoku board.

We then considered FreeSudoku 0.9.8, which uses a newer version of the component we were monitoring. We executed the new application with a test suite designed for FreeSudoku 0.9.8, and we dynamically checked the predicates we derived for FreeSudoku 0.9.6. We revealed violations of two interaction models. No I/O predicates were violated, indicating no changes in the values exchanged during the interaction with the new component. The two violations of interaction models resulted from a new logging facility that causes a new interaction and the refactoring of a method that reshuffles some method invocations. The limited number of violations shows that BCT can alert developers only to modified behaviors that can correspond to unexpected incompatibilities, limiting the number of false alarms.

In the second example, we focused on Jdiff, a Jedit plug-in for finding differences among documents. In this example, we considered the case of a system upgrade that doesn't affect the plug-in; so, two different systems were using the same component. We initially monitored the Jdiff plug-in within Jedit 4.1. We computed I/O predicates of six methods (all the methods that the main application can access) and interaction models of 68 methods (all methods that can trigger interactions with the main applications). BCT produced I/O predicates for two of the six methods. One method is associated with 52 predicates and another with 35,224 predicates. The large number of predicates depends on the use of the method as a generic dispatcher of messages from the application to the plug-in and on the large variety of exchanged messages.

We updated Jedit 4.1 to Jedit 4.2pre9 while leaving the plug-in unchanged, and we monitored the models on the new system. We revealed violations of both I/O and interaction models. I/O model violations indicate that the new version dispatches messages with content incompatible with the content used in version 4.1. These message incompatibilities correspond to new execution patterns and program faults. The model violations let you localize the incompatible object content that the system and the plug-in exchanged and the new code in the plug-in that the application has exercised.

The overhead of checking more than 35,000 predicates is important and might suggest monitoring the whole set of models only during testing. Predicates might be redundant. Daikon offers an option to inhibit the generation of redundant predicates. During testing, we can identify the subset of relevant predicates, further reducing the number of predicates.

So, BCT enables us to do several things:

- Identify incompatibilities and faults as soon as they occur. In the Jedit example, incompatible messages are detected as soon as they are generated.
- Identify faulty program states prior to observing any failure. In the Jedit example, incompatible messages are detected before corrupting the state.
- Improve system debugging, thanks to detailed information about the exchanged data values and the executed interactions. In the FreeSudoku example, violations indicate modified behaviors, and in the Jedit example, violations indicate the source of failures.
- Save time and money that would be necessary to repair the incompatibilities if detected late.

Refining models

Not all violations are the result of undesirable behaviors. Developers often update components or add new ones to satisfy new requirements or correct faults, which can also lead to model violations. If the number of changes is limited, the violations can confirm the changes. We can eliminate such violations by updating the models once we've verified that they correspond to a new functionality or revealed fault. If the number of violations is high, the large amount of irrelevant information might make the technique less effective. Software engineers can reduce the number of violations by adding information about the upgraded components. We can then use this information to refine the models so they better represent the behavior we expect from the new component. In the examples we've evaluated so far, we noticed that most new legal behaviors violated only a few models, which we could easily identify and update, eliminating most irrelevant violations.

We identified seven types of upgrades that we can use to refine models in BCT. Table 2 summarizes the upgrades and the corresponding model refinements. When both the original and the upgraded components satisfy the same requirement, we identified five types of upgrades: create new dependencies, suppress existing dependencies, change the internal data model, refactor code, and fix faults. When the initial and the upgraded components satisfy different requirements, we identified two types of upgrades: small and large. Given the type of upgrade, the models that BCT produced can be properly and semiautomatically updated to reflect the rationale of the upgrade.

If the upgrade eliminates some dependencies (for instance, because part of the functionality is implemented internally instead of being delegated to external components), we can automatically update the interaction models by replacing calls to the eliminated dependency with epsilon transition (epsilon transitions allow moving from a source state to a target state without consuming any input symbol). Similarly, if the upgrade introduces new dependencies (for instance, because part of the internally implemented functionality is delegated to external components), kBehavior can

Table 2

A summary of model refinement strategies

Update	Interaction model	I/O model
Unchanged requirements		
Create new dependencies	Automatically extend the FSA with behaviors that use the new dependencies	Usual checking of I/O predicates
Suppress existing dependencies	Automatic update of the FSA	
Change internal data models	Usual checking of interaction models	
Refactor code	Incrementally extend or rebuild the model with new interactions	
Fix faults	Automatically extend interaction models with behaviors related to marked states	Checking of I/O predicates modified according to the fixed behavior
Changed requirements		
Minimal changes	See “Fix faults”	Checking of I/O predicates modified according to the new requirements
Major changes	See “Refactor code”	Checking of I/O predicates that the new requirements satisfy

incrementally extend the interaction model every time it detects an interaction sequence containing a call to the new dependency. In this way, the final interaction model will automatically include new interaction patterns. BCT can log interactions with added or suppressed dependencies for debugging purposes.

If the upgrade involves a new internal data model, the component inherits the previous interaction models without any modification. We don't expect that changes in the internal data model will produce new interaction patterns.

If the upgrade involves code refactoring, the new component usually violates interaction models. If changes affect only a minimal part of the code, test designers can inspect violations and update interaction models accordingly. If changes are pervasive, the frequency of violations soon becomes unacceptable, and we should derive new models. We can use existing interaction models to log the behavior differences between the original and the new component.

If a component is upgraded to fix a fault, test designers can mark the states of the interaction model related to the fault. kBehavior can add the new interaction patterns to the model by allowing the modification of the behaviors related to marked states. In this way, changes are automatically included in the interaction models.

I/O predicates are computed only for the interface methods, so they don't need to be updated when requirements don't change. Test designers can modify the I/O predicates that correspond to faulty behaviors that have been fixed.

Component-based development is an important approach to building flexible, reusable applications. Extensive component reuse creates new integration problems that traditional test and analysis techniques can't adequately address, and BCT can bring testers' attention to behaviors that could be a potential source of faults.

Other dynamic analysis techniques can complement the information that BCT provides. For example, Michael A. Copenhafer and Kevin J. Sullivan's technique[14] can discover assumptions about component performance and other component characteristics.

Our experience with the early prototype opens new research issues. We're working on detailed models that integrate I/O and interaction in a unique framework (we've published our early results elsewhere[15]), extending BCT to deal with subsystem replacements, letting test designers drive analysis by specifying interesting properties of models, and using BCT as an enabling technology for self-healing solutions.

References

1. D. Garlan, R. Allen, and J. Ockerbloom, “Architectural Mismatch: Why Reuse is So Hard,” *IEEE Software*, vol. 12, no. 6, 1995, pp. 17–26.
2. B. Boehm and C. Abts, “COTS Integration: Plug and Pray?” *Computer*, vol. 32, no. 1, 1999, pp. 135–138.

3. B. Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, 1995.
4. H. Zhu, P.A.V. Hall, and J.H.R. May, “Software Unit Test Coverage and Adequacy,” *ACM Computing Surveys*, vol. 29, no. 4, 1997, pp. 366–427.
5. N. Abu-Ghazaleh, M.J. Lewis, and M. Govindaraju, “Differential Serialization for Optimized SOAP Performance,” *Proc. 13th IEEE Int’l Symp. High Performance Distributed Computing* (HPDC 04), IEEE CS Press, 2004, pp. 55–64.
6. M.D. Ernst et al., “Dynamically Discovering Likely Program Invariants to Support Program Evolution,” *IEEE Trans. Software Eng.*, vol. 27, no. 2, 2001, pp. 99–123.
7. S.P. Reiss and M. Renieris, “Encoding Program Executions,” *Proc. 23rd Int’l Conf. Software Eng.* (ICSE 01), IEEE CS Press, 2001, pp. 221–232.
8. A. Biermann and J. Feldman, “On the Synthesis of Finite State Machines from Samples of their Behavior,” *IEEE Trans. Computers*, vol. 21, no. 6, 1972, pp. 592–597.
9. J.E. Cook and A. Wolf, “Discovering Models of Software Processes from Event-Based Data,” *ACM Trans. Software Eng. and Methodology*, vol. 7, no. 3, 1998, pp. 215–249.
10. R. Parekh, C. Nichitiu, and V. Honavar, “A Polynomial Time Incremental Algorithm for Learning DFA,” *Proc. 4th Int’l Colloquium Grammatical Inference*, LNCS 1433, Springer, 1998, pp. 37–49.
11. S. Porat and J. Feldman, “Learning Automata from Ordered Examples,” *Machine Learning*, vol. 7, nos. 2–3, 1991, pp. 109–138.
12. L. Mariani and M. Pezzè, “Behavior Capture and Test: Automated Analysis of Component Integration,” *Proc. 10th IEEE Int’l Conf. Eng. Complex Computer Systems* (ICECCS 05), IEEE CS Press, 2005, pp. 292–301.
13. T.J. Ostrand and M.J. Balcer, “The Category-Partition Method for Specifying and Generating Functional Tests,” *Comm. ACM*, vol. 31, no. 6, 1988, pp. 676–686.
14. M.A. Copenhafer and K.J. Sullivan, “Exploration Harnesses: Tool-Supported Interactive Discovery of Commercial Component Properties,” *Proc. 14th IEEE Conf. Automated Software Eng.* (ASE 99), IEEE CS Press, 1999, pp. 7–14.
15. D. Lorenzoli, L. Mariani, and M. Pezzè, “Inferring State-Based Behavior Models,” *Proc. 4th Int’l Workshop Dynamic Systems Analysis* (WODA 06), ACM Press, 2006, pp. 25–32.

About the Authors

Leonardo Mariani is a researcher at the University of Milan Bicocca. His research interests include software engineering, in particular, software test and analysis. He received his PhD in computer science from the University of Milano Bicocca. He’s a member of the IEEE and ACM. Contact him at the Dept. of Informatics, Systems and Communication, via Bicocca degli Arcimboldi, 8, Bldg. U7, 4th Floor, Room 476, 20126 Milano, Italy; mariani@disco.unimib.it.

Mauro Pezzè is a software engineering professor at the University of Milan Bicocca and visiting professor at the University of Lugano. His research interests include software engineering, in particular, software test and analysis. He received his PhD in computer science from Politecnico di Milano. He’s member of the IEEE, where he served as executive chair of the Technical Committee on Complexity in Computing, and the ACM. Contact him at the Univ. of Lugano, Faculty of Informatics, Via Giuseppe Buffi, 13, Lugano, Switzerland; mauro.pezze@unisi.ch.