Available online at www.sciencedirect.com
SCIENCE DIRECT
ELSEVIER
Electronic Notes in Theoretical Computer Science 116 (2005) 17–30
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Technique for Verifying Component-Based Software [3]

Leonardo Mariani [1] Mauro Pezzè [2]

DISCO
Università degli Studi di Milano Bicocca
via Bicocca degli Arcimboldi, 8
20126 Milano, Italy

Abstract

Component-based software systems raise new problems for the testing community: the reuse of components suggests the possibility of reducing testing costs by reusing information about the quality of the software components. This paper addresses the problem of testing evolving software systems, i.e., systems obtained by modifying and/or substituting some of their components. The paper proposes a technique to automatically identify behavioral differences between different versions of the system, to deduce possible problems from inconsistent behaviors. The approach is based on the automatic distilling of invariants from in-field executions. The computed invariants are used to monitor the behavior of new components, and to reveal unexpected interactions. The event generated while monitoring system executions are presented to software engineers who can infer possible problems of the new versions.

Keywords: Component-based software, testing, reuse, invariants

1 Introduction

Component-based system (CBS) technology facilitates flexibility: systems can be adapted, modified, and updated by adding, removing, and changing their

[1] Email: mariani@disco.unimib.it
[2] Email: pezze@disco.unimib.it
[3] This work has been partially supported by the Italian Ministry of University and Research within the COFIN 2001 project "Quack: a platform for the quality of new generation integrated embedded systems"

doi:10.1016/j.entcs.2004.02.089

components. Adding or modifying components require testing the components added or modified as well as all components that depend from them. Modifying components can introduce dangerous side effects that are difficult to predict, thus, in principle, we need to retest the whole system. Retesting the whole system may be extremely expensive and sometimes not even possible. Testing effort can be reduced by selecting a subset of tests using either regression testing techniques (see for instance, [18,8]) or dependence analysis techniques (see for instance [21,20]). Testing effort can also be reduced with run-time verification techniques that can partially substitute traditional testing (see for instance [7]).

In this paper, we propose an approach to reduce testing costs for CBSs. The approach, called Behavior Capture and Test (BCT), combines run-time monitoring, run-time verification and testing. We use run-time monitoring to gather data about single executions. The collected data are then used to automatically derive behavioral properties of the components and the system. The inferred properties are used for run-time verification, while single executions are used as regression test cases. BCT can be used to automatically verify the integration of components that have been used in the same or in other systems, which is a very frequent case in component-base development.

2 Behavior Capture and Test

The BCT approach collects behavioral information on components and interactions by monitoring the execution of CBSs, summarizes their behavior by selecting notable behaviors and distilling invariants, and checks for compatibility of changes by verifying invariants and executing notable behaviors. BCT is based on five main phases:

- *generating and installing behavior recorders.* BCT generates and installs two small software modules for each monitored component: the stimuli recorder and the interaction recorder, to suitably instrument the target system.
- *recording executions.* The stimuli and interaction recorders capture run-time interactions between the system and the monitored components.
- *distilling invariants.* The behavior of the target components is distilled into I/O and interaction invariants that summarize relations between system requests and component results, and interaction patterns between the component and the system, respectively.
- *filtering behaviors.* The monitoring of executions produces too many behaviors, many of which not particularly interesting. BCT selects a subset of notable executions to be stored as regression test cases.

- *verifying invariants and executing tests.* BCT uses I/O and interaction invariants to verify the behavior of the target system at run-time. BCT also executes notable behaviors as regression test cases for verifying new versions of the same component or system.

Notable executions, I/O invariants and interaction invariants can be used to verify the quality of CBSs in several interesting cases:

(i) when a component `A` that is part of a running system `S` is added to a new system `S'`

(ii) when a new component `B` replaces a component `A` in a system `S`

(iii) when a component `B` that is part of a system `S'` replaces a component `A` in a system `S`.

Case (i) occurs when a system is extended by adding new components that are taken from a library of components already in use in other systems. In this case, notable executions and invariants distilled when components are used in existing systems can be used to verify the compatibility of the components in the new systems. In particular, notable executions can be used for testing the components as part of the new system, while invariants can be used as oracles to identify deviations in the expected behavior. Invariants can be used to generate monitors that verify consistency of the behavior of the components when they are executed in different systems. Whenever an invariant is violated, the monitor generates a warning that corresponds either to a failure, or to a behavior of the component not previously exploited.

Case (ii) occurs when a component in use is substituted with a newer version. In this case, notable executions and invariants computed for the "old" component `A` executed within the old version `S` can be used for testing and monitoring the "new" component `B`. Notable executions represent regression test cases as in the former case. Invariants can be used to generate monitors to verify that the behavior of the "new" version `B` is compatible with the "old" version `A`. A violation of an invariant can reveal either a failure of the new version or a correct behavior of the new version that was never seen with the old version.

Case (iii) occurs when a component in use in other systems replaces a component in use in system of interest for adding and or modifying the implemented functionalities. In this case, the notable executions computed for the replaced and the replacing components, albeit in different execution contexts, can be used to test the replacing component in the new context. The invariants computed for both replaced and replacing components can be matched to identify possible incompatibilities and can be used to generate monitors for the replacing component.

The main contributions of the work are the following:

- the combination of run-time monitoring, run-time verification and testing to reuse information about components' behavior to verify the behavior of component-based systems,
- the use of reflection and syntactic analysis to automatically identify inspectors that provide non-intrusive access to the state of Java objects, thus supporting automatic deployment of monitors of components' behavior even in absence of source code,
- the definition of a mechanism for "flattening" objects, to producing data that can be processed with Daikon for automatically deducing invariants that describe components' interactions that involve complex objects,
- the application of the computed invariants to identify behavioral differences between different versions of a software system, thus revealing possible inconsistencies and defects,
- preliminary experimental data that indicate the validity of the proposed approach.

3 Run-Time Monitoring

The first two phases of BCT (generating behavior recorders and recording executions) allow for automatically monitoring the run-time behavior of the system.

Run-time monitoring can be implemented with several techniques characterized by different degree of intrusiveness and complexity: by instrumentation of the source code [22], by modifying the run-time support [23], by developing built-in monitor facilities [3], by wrapping the program under test.

In general, the source code of components may not be available at integration time, thus CBS must use monitoring techniques that do not require the source code. BCT behavior recorders rely on suitable modifications of the binding mechanisms. Independently developed components are bound either at system start-up or at run-time. BCT behavior recorders are installed at component binding time, as "filters" between the system and the component. Figure 1 shows the run-time configuration of a monitored component. The stimuli recorder captures incoming requests, stores single behaviors in a local repository, and forwards the requests to the monitored component. A single behavior includes the name of the required service, its parameters and the result. Interaction recorders intercept requests from the component to the systems and store the interaction patterns.

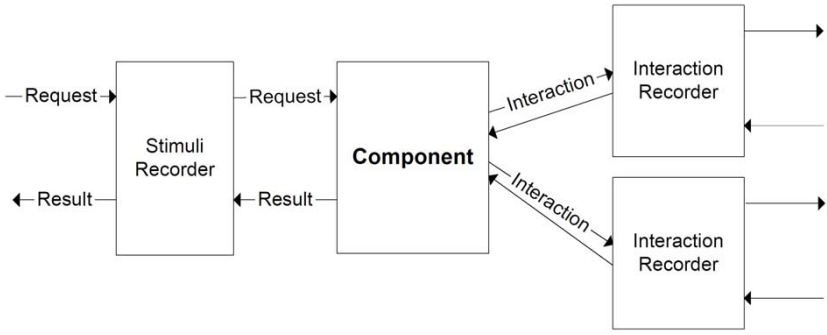

Fig. 1. The Run-time configuration of a monitored component

This approach is not intrusive from the component viewpoint. The intrusiveness from the system viewpoints is limited to performance. The complexity of altering the binding mechanism depends on the component technology: for example, binding based on XML configuration files can be easily modified; changing dynamic binding mechanisms based on service discovery can be complex. We are currently investigating the problem of modifying the dynamic binding mechanism of EJB by taking advantage of services provided by the application server.

The behavior of the recorders is independent from the target component; therefore it is possible to automatically generate recorders from the interface specification of the component. The recorder implements the same interface of the target component towards the system. The single methods of the recorders store the parameters and forward the request to the target component.

4 I/O Invariants

The phase of invariant distilling produces I/O and interaction invariants. I/O invariants capture the relation among parameters and returned results; interaction invariants capture the interaction patterns of the component with the system.

The interface of components can be very complex, especially in the case of object-oriented systems that often hide the internal state of objects. For example, the recorder for the shopping cart component described in Section 6 needs to record the object `CartItem` shown in Figure 2. Capturing the parameter would record the object data type and the object's reference, which in most cases is insufficient for inferring useful invariants.

To effectively record complex parameters, we developed a technique that automatically gathers state information from an object instance without requiring manual instrumentation. Our technique, hereafter "object flattening",

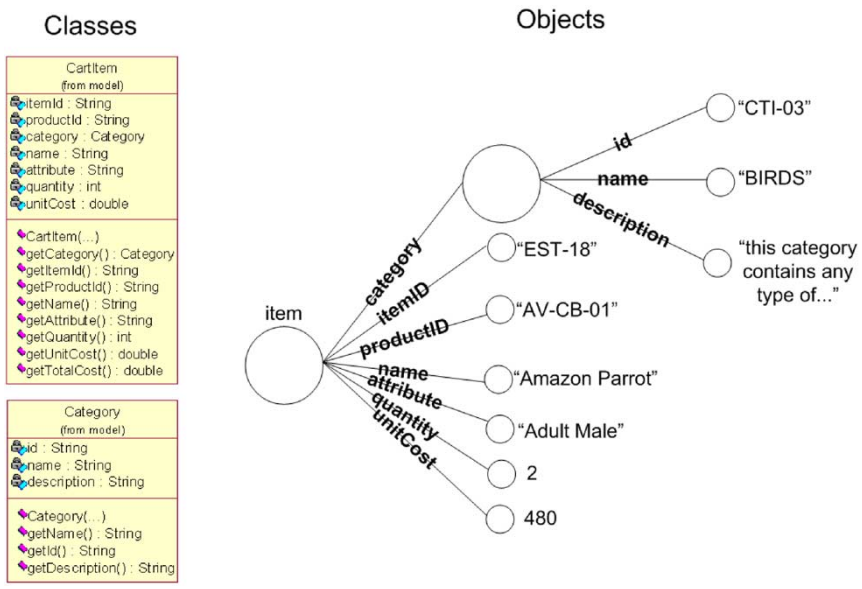

Fig. 2. The class diagrams for `CartItem` and `Category`, and an example of an instance of `CartItem`

works for components implemented with paradigms that support introspection, e.g., Java.

In general, the state of an object can be examined using suitable methods that do not alter the state itself. Such methods are called *inspectors*. Object flattening uses introspection to retrieve the signatures of methods of an object, and to automatically select inspectors from the list. The selected inspectors are executed to gather state information at run-time. The approach is recursively applied to complex objects.

We heuristically identify inspectors by matching a set of syntactic rules with the signature of each method. When a method satisfies one of these rules, it is classified as inspector. These rules are based on conventions normally used when writing code, for example a method with no parameters, with a return value, and with a name starting with `get` is classified as inspector. The rules can be applied automatically to the objects without requiring their source code or specifications. The set of rules that we are currently experimenting is given in [10].

By applying the Object flattening techniques on the example in Figure 2, it is possible to automatically gather the whole state information. The output generated from our tool is shown in Figure 3.

Our preliminary studies highlight the effectiveness of these rules: so far, we never erroneously select an intrusive method as inspector. We are currently

```
CartItem.getItemId = "EST-18"
CartItem.getProducId = "AV-CB-01"
CartItem.getName = "Amazon Parrot"
CartItem.getAttribute = "Adult Male"
CartItem.getQuantity = 2
CartItem.getUnitCost = 480
CartItem.getTotalCost = 960
CartItem.getCategory.getName = "BIRDS"
CartItem.getCategory.getId = "CTI-03"
CartItem.getCategory.getDescription = "this category contains any type of ..."
```

Fig. 3. Output generated by Object Flattener applied to an instance of `CartItem`

investigating the completeness of the rules, which is measured as the amount of state that is accessible with non-intrusive methods, but is not inspected by the inspectors automatically selected with the rules.

Being the technique neither safe nor complete, we allow users to extend the object flatteners with plug-ins that modify the set of inspectors by removing methods identified as inspectors by the rules, or adding methods that are not automatically selected by the rules. The object flatteners can be configured by defining the maximum depth of recursive examination of nested objects and a condition for preventing the examination of large data structures like arrays, when not interesting for analysis.

Object flatteners collect information that can be elaborated with invariant detection techniques to compute I/O invariants. In our experiments, we use Daikon [4] to automatically infer invariants that represent properties of the services offered by the components. Daikon starts with a large set of invariants compatible with scalar and structured parameters of a service. The initial set of invariants is pruned by removing all invariants that are violated by single behaviors, thus remaining invariants describe significant relations among considered parameters. Figure 4 shows I/O invariants automatically derived for the parameter `CartItem` shown in Figure 2.

```
size(CartItem.getItemId) = 6
size(CartItem.getProducId) = 6
min(size(CartItem.getName)) = 5
max(size(CartItem.getName)) = 35
min(size(CartItem.getCategory.getName)) = 4
max(size(CartItem.getCategory.getName)) = 15
size(CartItem.getCategory.getId) = 6
CartItem.getQuantity >= 1
CartItem.getUnitCost *  CartItem.getQuantity = CartItem.getTotalCost
```

Fig. 4. I/O invariants automatically generated for `CartItem`

5 Interaction Invariants

Interaction invariants capture the interactions of the monitored component with other components. For example, to return an updated catalog, a com-

ponent `Catalog` may need to interact with a component `Catalog-DAO` that provides connectivity to a database.

We distill interaction invariants as regular expressions over an alphabet Σ that contains a symbol for each service used by the monitored component. An interaction pattern can be modeled with a string over Σ. For instance, the execution of a service `s` that issues the sequence of requests `s1` to component `c1`, `s2` to `c2` twice, and `s3` to `c3`, can be modeled with the string $w = c1.s1\ c2.s2\ c2.s2\ c3.s3$.

Regular expressions can be derived in many ways [14,1,9] that reflect different assumptions about the modeled system. When distilling interaction invariants, we assume that interaction patterns present regularities similar to execution flows. In particular, we assume that interactions between two or more components can be decomposed in sets of conceptually self-contained sequences of requests (hereafter "behavioral patterns"). For instance, a component `A` can first access a database, then the service of a library, and finally a print service. This interaction pattern can be decomposed in three behavioral patterns, corresponding to the requests exchanged for accessing to each single service. We thus need a grammar inference algorithm that refers to behavioral patterns. Memory is a precious resource. To limit memory usage, the inference engine should limit the storage of the examined behaviors. Finally, behavioral patterns are produced incrementally, and thus the algorithm must work incrementally, by using only positive samples and without requiring additional resources.

None of the algorithms proposed in literature meets all the requirements, thus we developed a new incremental algorithm based on the concept of inaccurate inference. The algorithm initializes the regular expression ER to a string `w` that corresponds to the first sequence of observed interactions for the execution of a service s.

Further executions of the same service s produce new strings. If the new string w' belongs to the language generated by the current regular expression ER for service s ($w' \in \mathcal{L}(ER)$), ER is not changed. Otherwise, the regular expression ER is extended to a new expression ER' such that:

(i) $\mathcal{L}(\mathcal{ER}) \subseteq \mathcal{L}(\mathcal{ER}')$

(ii) $w' \in \mathcal{L}(\mathcal{ER}')$

The property (i) assures that the new expression captures all behaviors included so far, while the property (ii) guarantees the inclusion of the new behavior in the invariant.

We compute ER' from ER by extending the Finite State Automata (FSA) corresponding to ER to include the new string w'. Here we illustrate the

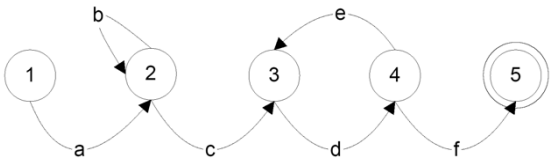

Fig. 5. The FSA for a regular expression ER

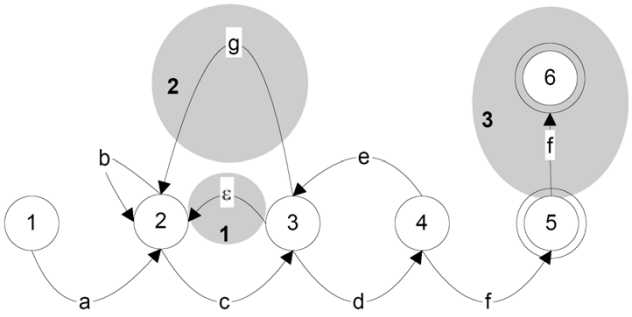

Fig. 6. The FSA for the new regular expression ER'

basic principles with an example. Let us consider for example the FSA A_{ER} of Figure 5 and a new string $w' = abbdedegcdff$.

We first identify the longest prefix of w' recognized by A_{ER}: in the example *abb*. We then consider incrementally the substrings of length l_{MAX} following the identified prefix (l_{MAX} is the maximum value for the length of a searched substring), and we try to identify a sub-automaton of A_{ER} that recognizes one of such substrings. In the example, with $l_{MAX} = 4$, we immediately identify the subautomaton $3, 4$ that recognizes the substring *dede* that immediately follows the prefix. We then extend the automaton to recognize the prefix and the identified substring. Region 1 of Figure 6 shows the new edge ϵ added to recognize the substring *abbdede*. We then proceed incrementally with the tail of the string until we obtain a new automaton. The final automaton is shown in Figure 6; region 2 and 3 has been added to recognize the tail of w'.

The regular expression produced by the algorithm is an inaccurate approximation, but preliminary experiments show its validity. Here we only presented the underlying ideas of the algorithm. Further details can be found in [10].

6 Early Validation

We validated BCT on the Sun Java pet store, an on-line pet shop [19] based on EJB technology. We experimented with two versions of the pet store characterized by different versions of components *shopping cart*, *registration*

manager, and *catalog*. Version `A` of the *shopping cart* is the original Sun implementation that sets to 1 the number of each purchased item, regardless of the amount present in the cart. Purchases in version `B` shopping cart increment by one the quantity of items in the cart. Version `A` of the *registration manager* requires the e-mail address for the input form to be valid, while version `B` leaves the e-mail optional. Version `A` of the *Catalog* interacts with the `service locator` to locate the database, while version `B` interacts directly with the database.

We first computed the invariants for version `A` by running a set of randomly generated test cases. Table 6 show the I/O invariants for services `add` of component *shopping cart* and `perform` of component *registration manager*, and the interaction invariant for service `getItem` of component *Catalog*.

Component	**Service / I/O invariants**
ShoppingCart `A`	add / item.getUnitCost*item.getQuantity=item.getTotalCost add / item.getQuantity = 1
Registration Manager `A`	perform / CustomerEvent.getContactInfo.getEmail ≠ `NULL` perform / CustomerEvent.getContactInfo.getAddress. getCountry == "Italy"
	Interaction invariants
Catalog `A`	getItem / ServiceLocator.ServiceLocator.new SeriveLocator.getLocalHome (CatalogEJB.create CatalogEJB.getItem+ϵ)

Table 1
I/O invariants computed for the modified components

We then updated version `A` by replacing components *shopping cart*, *registration manager*, and *catalog* with their versions `B`, and we monitored the execution of version `B` with respect to the available invariants. Table 6 shows the results of the monitored execution for the invariants of Table 6.

These results highlight the different aspects that can be detected by monitoring invariants: an invariant can be violated when a requirement of the system is modified, when a fault generates some effects, or when the system is used in an unexpected way. Monitors can signal the effects to the users who can diagnosis each case, identifying faults or needs for additional tests.

Inv.	Result	Comment
inv 1	Satisfied	Version `B` does not alter the behavior wrt this invariant.
inv 2	Violated	Version `B` treats differently the number of items in the cart, satisfying a different requirement.
inv 3	Violated	Version `B` does not require e-mail address for confirming orders.
inv 4	Violated	The invariant signals a different use of attribute `Country` thus revealing insufficient test of version `A`.
inv 5	Violated	the new component produces the expression (*Catalog-DAO.CatalogDAOFactory.getDAO Catalog-DAO.CatalogDAO.getItem*) that indicates the presence of a new interaction pattern.

Table 2
Evaluation of the invariants on version `B`.

7 Related Work

BCT originally combines many technologies for verifying component-based systems. The authors are not aware of any other work that combines similar technologies for the same goal, but BTC uses ideas and techniques proposed by previous work.

BCT shares the idea of using data collected from the field for testing and verifying software systems with other research: *perpetual testing*, *remote testing* and *field-data*. Pavlopoulou and Young's *Perpetual Testing* extends testing, and in particular the statement coverage adequacy criterion to normal software operation [15]. ANTS' *Remote Testing* relies on actual usage of selected users to test the navigability of web sites [17]. Orso et al.'s Gamma gathers *field-data* from deployed software, to support both impact analysis and regression testing [13]. Differently from other approaches, BCT uses data from the field for automatically testing and verifying component-based software systems.

BCT uses the *invariant detection* technique implemented in Daikon [4] for inferring I/O invariants. Daikon can compute invariants for both scalar and structured variables. BCT provides a technique for applying Daikon to object-oriented software.

Linearization of object's attributes proposed by Ernst et al. provides support for analyzing complex data structures, but requires code instrumentation [5]. BCT's object flattening uses reflection and does not require instrumentation of the single classes.

McCamant and Ernst propose a technique for formally proving the safety of components' updates, by using invariants computed with Daikon [11]. BCT does not focus on formal verification, but uses invariants to generate monitors

for checking for compatibility of observed behaviors with respect to expected ones.

Nimmer and Ernst enhance invariants dynamically detected by Daikon with static verification performed by ESC/Java [2] to confirm properties proposed by the tool [12]. BCT works in a context where source code may not be available, and thus static analysis of source code is not always applicable.

Raz et al. use invariants computed by Daikon and by statistic algorithms to synthesize the behavior of data feed systems [16]. This approach is similar to BCT but it is based on a distributed setting where invariants are both computed and stored at the client side. BCT uses invariants with a very different perspective: to track evolution of both systems and components and are used to verify correctness of the actual implementation.

Hangal's DIDUCE tool instruments the source code to derive invariants that are continuously verified and updated at run-time [6]. The technique focuses on debugging problems and tries to minimize the consumed resources. Lightweight computation of invariants is obtained at the cost of limited expressiveness power of inferred invariants. BCT works in a framework that privileges accuracy over resource consumption.

8 Conclusions

This paper presented BCT, a new approach for testing CBSs. BCT computes invariants from the use of components in the field and uses such invariants to automatically check for compatibility of new components that are added to update and enhance evolving systems. The BCT technology requires little user intervention, applies to complex object-oriented software, and produces valuable information on discrepancies between new and old systems that can reveal subtle faults.

Early experimental results obtained by using invariants for detecting mismatches between different versions of the Java Pet Store indicate that BCT can detect a wide range of faults. We plan to increase experimental data on BCT to gain evidence of the benefits provided by this technology. We are currently developing a tool suite for automatically applying BCT to large experimental software, and thus evaluate the difficulty of applying BCT to industrial scale systems, and clarify effectiveness, synergies and complementarities of faults detected by either interaction or I/O invariants.

References

[1] Cicchello, O. and S. C. Kremer, *Inducing grammars from sparse data sets: a survey of algorithms and results*, Journal of Machine Learning Research **4** (2003), pp. 603–632.

[2] Detlefs, D., K. Rustan, M. Leino, G. Nelson and J. Saxe, *Extended static checking*, SRC Research Report 159, Compaq Systems Research Center (1998).

[3] Edwards, S. H., *A framework for practical, automated black-box testing of component-based software*, Journal of Software Testing, Verification and Reliability **11** (2001).

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

[5] Ernst, M. D., W. G. Griswold, Y. Kataoka and D. Notkin, *Dynamically discovering pointer-based program invariants*, Technical Report UW-CSE-99-11-02, University of Washington, Seattle, WA (1999).

[6] Hangal, S. and M. S. Lam, *Tracking down software bugs using automatic anomaly detection*, in: *Proceedings of the 24th International Conference on Software Engineering* (2002), pp. 291–301.

[7] Havelund, K. and G. Roşu, *Synthesizing monitors for safety properties*, in: J.-P. Katoen and P. Stevens, editors, *proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science **2280** (2002), pp. 324–356.

[8] Jones, J. and M. Harrold, *Test-suite reduction and priorization for modified condition/decision coverage*, IEEE Transactions on Software Engineering **29** (2003), pp. 195–209.

[9] Lang, K., B. Pearlmutter and R. Price, *Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm*, in: *proceedings of the Fourth International Colloquium on Grammatical Inference*, 1998.

[10] Mariani, L., *Capturing and synthesizing the behavior of component-based systems*, Technical Report LTA:2004:01, Università di Milano Bicocca (2003).

[11] McCamant, S. and M. D. Ernst, *Predicting problems caused by component upgrades*, in: *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering* (2003), pp. 287–296.

[12] Nimmer, J. W. and M. D. Ernst, *Static verification of dynamically detected program invariants: Integrating daikon and ESC/Java*, in: *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, 2001.

[13] Orso, A., T. Apiwattanapong and M. J. Harrold, *Leveraging field data for impact analysis and regression testing*, in: *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering* (2003), pp. 128–137.

[14] Parekh, R. and V. Honavar, *Grammar inference, automata induction, and language acquisition*, in: Dale, Moisl and Somers, editors, *Handbook of Natural Language Processing*, Marcel Dekker, 2000 .

[15] Pavlopoulou, C. and M. Young, *Residual test coverage monitoring*, in: *proceedings of the 21th International Conference on Software Engineering (ICSE'99)*, 1999, pp. 277–284.

[16] Raz, O., P. Koopman and M. Shaw, *Semantic anomaly detection in online data sources*, in: *Proceedings of the 24th international conference on Software engineering* (2002), pp. 302–312.

[17] Rodriguez, M. G., *Automatic data-gathering agents for remote navigability testing*, IEEE Software **19** (2002), pp. 78–85.

[18] Rothermel, G. and M. Harrold, *A safe, efficient regression test selection technique*, ACM Transactions on Software Engineering and Methodology **6** (1997), pp. 173–210.

[19] Singh, I., B. Stearns, M. Johnson and the Enterprise Team, "Designing Enterprise Application with Java^{TM}2, Enterprise Edition, 2/e," Addison-Wesley, 2002.

[20] Sinha, S., M. Harrold and G. Rothermel, *System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow*, in: ACM, editor, *proceedings of the 1999 International Conference on Software Engineering*, 1999, pp. 432–441.

[21] Stafford, J. A. and A. L. Wolf, *Architecture-level dependence analysis for software systems*, International Journal of Software Engineering and Knowledge Engineering **11** (2001), pp. 431–451.

[22] Templer, K. and C. Jeffery, *A configurable automatic instrumentation tool for ANSI C*, in: *proceedings of the 13th IEEE International Conference on Automated Software Engineering*, 1998, pp. 249–258.

[23] Wolczko, M., *Using a tracing java^{TM} virtual machine to gather data on the behavior of java programs*, Technical Report SML 98-0154, Sun Microsystems Laboratories (1999).