Error-Avoiding Adaptors for Black-Box Software Components

Casandra Holotescu
Department of Computer and Software Engineering
Politehnica University of Timisoara, Romania
casandra@cs.upt.ro

ABSTRACT

A lot of work has been done in the area of building component-based systems with correct-by-construction adaptors. This is accomplished by using preexisting specifications of the component behaviour. But what happens when known components get to interact with incompletely specified, black-box components, and errors occur? How can we avoid these errors without modifying existing/legacy components? We present a method to explore and control such systems. Our approach exploits information in correct and erroneous runs to build a controller that ensures our system will avoid observed errors. We consider the behavioural specifications for our known, legacy component as already documented and we infer partial behaviour information of the unknown component by studying its reactions to various interaction scenarios.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model Checking*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Distributed debugging*

General Terms

Reliability, Verification, Design

Keywords

component-based systems, model refinement, adaptation, maintenance

1. INTRODUCTION

Component-based software engineering aims to improve software productivity by assembling large systems out of reusable components. However, as these components are often developed separately, the risk of mismatching appears at different levels: signature, behaviour, quality of services, etc. This inconvenience is addressed by design-time adaptation.

ASE'10, September 20–24, 2010, Antwerp, Belgium.

In the area of behavioural component adaptation, an adaptor is considered to be a specific component-in-the-middle that would properly coordinate the interactions between components towards the desired functionality. Several approaches for automatically generating correct by construction adaptors for system composition have been proposed, e.g. by Schmidt and Reussner [20], Autili et al. [3] or Canal et al. [7]. While earlier pioneering approaches, such as [21] or [20] though already semiautomatic, have only focused on ensuring a non-deadlocking interaction among components, later research such as [3], [7] has resulted in tools able to automatically synthesize adaptors that mediate the interactions in the component-based system, so that its resulting behaviour satisfies a temporal logic property (described as a Büchi automaton, vector transition system, finite state machine, etc.). If both the system behaviour and the goal property are regarded as automata, the resulting system automaton must be a simulation of the goal.

There is a close relation between component adaptation and control theory [19]. Considering the desired functionality and absence of deadlock as part of the system specification, we may view an adaptor as a controller over the system plant, using message forwarding or consumption as a means to enable and disable behaviour in the plant.

The above approaches assume all component behaviours are known, either prespecified as finite state machines, labeled transition systems or Petri nets, or derived in this form from more intuitive, visual formalisms such as message sequence charts. However, this is not always the case in the real component-based software development environment, and it is not uncommon for a system architect to have to integrate a black-box component. Integrating a component with insufficient behavioural specification is difficult, as none of the existing automatic solutions for adaptor synthesis can be applied, and the resulting system can be quite unsafe. Resolving an observed error in this kind of system, by automatically generating a correctness-enforcing adaptor, is the main aim of our method.

For simplification, consider a component with available behavioural specification, which interacts with another, black-box component, whose behaviour is partially unspecified. The interaction takes place by asynchronous message exchange. The resulting system must satisfy a certain property and also avoid deadlock. At runtime, a certain set of interactions results in an error, i.e., property violation, deadlock or system crash. As the latter two can be subsumed by the former one, we'll only consider temporal property violations from now on. Let us also assume we have a failing run trace.

Our method consists of the following steps:

- Behaviour exploration and refinement: the next steps are repeated until a satisfying refinement is found or a cost limit is reached.
  - Exploration: We introduce a "fake" adaptor among the two components with the purpose of conducting experiments with different execution scenarios. Its actions are directed towards exploring the system state space instead of performing a real, correctness-ensuring adaptation.
  - Refinement: A tentative model of the unspecified component is constructed and incrementally refined, using the information obtained during the previous exploration phase. The component is assumed to have a deterministic behaviour.
- Synthesis: using the refined approximation of the unknown component, we synthesize an adaptor that will forbid the erroneous interaction sequences.

The exploration process takes repeatedly the following steps, as long as there are unexplored controllable choices:

- We launch again the system execution and follow the error trace up to its last controllable and incompletely explored decision point.
- We repeat following actions until execution completes:
  - The fake adaptor initiates a bounded model-checking phase over the system model and searches for the best next steps towards property satisfaction.
  - The best yet unexplored choice is enabled by the fake adaptor in the concrete system execution.
  - Unknown component reaction is observed.
- Execution ends by success/failure and its trace is stored.

The verification driven execution follows each time the initial error trace, exploring different options on the path by enabling and disabling controllable events. This means that also the model refinement, as it employs information obtained during the runtime exploration, is directed towards clarifying the behavioural vicinity of the error trace.

The refinement process ends when the approximated model is refined enough so that all control points on the initial error trace have been completely explored through verification driven execution. We have now experimented with all potential controllable events on the path to error, and we can decide on the correctness ensuring controller synthesis.

2. METHOD

2.1 Assumptions

We consider a real component-based system $S_R = (C_K, C_U)$, where C_K and C_U represent respectively the known and the unknown components.

Let us assume component C_K has a previously specified behaviour, described by a finite state machine: a tuple $K = \langle Q^K, Q_f^K, q_0^K, \Sigma^K, \delta^K \rangle$, where Q^K is the set of states, $Q_f^K \subseteq Q^K$ the set of final states, q_0^K is the initial state, Σ^K a set of events of the message send/receive type: $msg!$ and $msg?$ and $\delta^K : Q^K \times \Sigma^K \to Q^K$ the partial transition function.

We associate the unknown component C_U with a tentative finite state machine $U = \langle Q^U, Q_f^U, q_0^U, \Sigma^U, \delta^U \rangle$. Its event set Σ^U is assumed to be the mirroring of Σ^K, thus $\Sigma^U = \{msg!|msg? \in \Sigma^K\} \cup \{msg?|msg! \in \Sigma^K\}$. As we have no information on this finite state machine, we start by considering it as most general and its transition function becomes $\delta^U : Q^U \times \Sigma^U \to \mathcal{P}(Q^U)$, thus allowing for nondeterminism. We assume that $Q^U = Q_f^U = \{q_0^U\}$ and $\delta^U\left(q_0^U, \sigma\right) = \{q_0^U\}$ is defined for all $\sigma \in \Sigma^U$. It is important to remember that U is not modelling the real behaviour of component C_U, but an overapproximation of it - although C_U is deterministic, by allowing U to be nondeterministic we can make sure that during the refinement process U will always overapproximate the behaviour of C_U.

We consider the interactions between components as asynchronous, by means of bounded buffers. Let $K \times U$ be their asynchronous composition. Each component has an output and an input buffer, both of length l. The input buffer of one component is directly connected to the output buffer of the other, as no adaptor exists between the two components.

All events in the event sets Σ_K and Σ_U, i.e., reading and writing to/from buffers, are assumed observable. Thus, if component C_U receives a message msg, the receive event $\sigma = msg?$ can be externally observed.

The ideal system S to be obtained from the composition of K and U complies with the property φ. The desired property φ is also expressed by means of a finite state machine $\langle Q^\varphi, Q_f^\varphi, q_0^\varphi, \Sigma^\varphi, \delta^\varphi \rangle$, where $\Sigma^\varphi \subseteq \Sigma^K \cup \Sigma^U$. For simplification, the property φ also includes the non-deadlocking requirement, thus $\varphi = \phi \times \rho$, where ϕ is the functional specification of the system and ρ the specification of deadlock avoidance. We consider compliance as simulation: $S \preceq \varphi$.

The real system S_R is found to violate property φ. The resulting error trace is a sequence of events $t = \sigma_1\sigma_2...\sigma_n$, with $\sigma_i \in \Sigma^K \cup \Sigma^U$.

We aim to build a controller that restricts the behaviour of system S_R such that the error observed in t no longer occurs. In a component-based system, such a controller is an adaptor that enables events by message forwarding, and disables them by message consumption, thus controlling the message transfer between component buffers. Therefore, while the receive events $\sigma \in \Sigma_?$ are controllable by the adaptor, send events $\sigma \in \Sigma_!$ are uncontrollable.

2.2 Discovering behaviour

We start by assuming the behaviour of the unknown component as most general. Building a controller for S_R using the overapproximated model of C_U is impractical, as this model contains a large number of virtual, possibly spurious, control sequences. This subsection deals with the issue of discovering new behaviour of component C_U, by exploring its real behaviour at runtime and correspondingly refining the tentative model U.

We implement verification-driven execution as follows. We employ bounded model-checking [5] to verify the desired property φ on the modelled system $K \times U$. At the same time we direct the execution of the real system S_R based on the information provided by model checking. This enables us to verify the model U of component C_U through actual trace execution and, when inconsistencies appear, to appropriately refine it. To perform these operations a "fake" adaptor - an interactive mediating component, is inserted between the two components.

As our approach aims at finding a controller that enables the system to avoid the observed error, the refinement process is oriented towards obtaining a more precise version of U useful in the controller synthesis. Thus, the exploration is also centered on the error trace t: verification driven execution exploits t, and tries to find successful controllable executions in S_R, from various prefixes of t. The exploration and refinement process stops when all controllable decision points of t have been completely explored.

2.2.1 Verification-driven execution

Let $\overline{L(S)}$ be the prefix-closed language of the ideal system $S = (K \times U) \parallel \varphi$ that complies to the property φ. Let $pre(t)$ be the longest prefix of t that complies to φ, $pre(t) = t_k$ such that $t_k = \sigma_1\sigma_2 \ldots \sigma_k \in \overline{L(S)}$, but $t_{k+1} \notin \overline{L(S)}$. In order to disable t, we have to prevent the event σ from occurring by disabling it, if controllable, or by disabling its previous controllable event in the error trace.

We define a control point in $K \times U$ as a state q for which at least one of the outgoing controllable event sets $\Sigma^K_?(q)$ and $\Sigma^U_?(q)$ contains more than one controllable event: $|\Sigma^K_?(q)| \geq 2 \vee |\Sigma^U_?(q)| \geq 2$. For a controllable event to be disabled, an alternative controllable, receive event $msg?$ has to be enabled in the control point. This can only happen if the expected message msg is already available.

Let now q_{cp} be the last control point in $K \times U$ that precedes the error-inducing event σ in the error trace t.

We want to discover possible new behaviours of the real system S_R starting from the control point q_{cp}. We explore the hypothetical alternative choices from q_{cp} in the system model $K \times U$ by means of bounded model checking, with b the maximum depth bound. The finite state machine describing the property φ will be executed synchronously with the system model $K \times U$.

We launch the run, and force the system to perform the same sequence of interactions executed in the original error trace, until the execution reaches state q_{cp}. Due to the uncontrollability of send events, this might not always be possible. If this happens and the new trace t' is no longer a prefix of t, i.e. $t' \notin Pre(t)$, then the last control point on t' becomes the new q_{cp} for that execution. From the control point q_{cp} of the system $K \times U$ and the corresponding state q^{φ}_{cp} of the system specification, we start a depth-first exploration of the alternative choices offered by q_{cp}.

Let us now focus on the possible traces of maximum length b, generated by enabling an alternative choice σ at the control point q_{cp}. Let $|pre(t_i)|$ be the length of a possible trace t_i, which is enabled from q_{cp} though the controllable event σ_i, such that $pre(t_i)$ satisfies φ.

The trace t_i is considered the best yet unexplored trace from q_{cp} if $|pre(t_i)| \geq |pre(t_j)|$, $\forall j \neq i$. Thus, its corresponding choice σ_i is the best next step from q_{cp} and the fake adaptor enables event σ_i for the on-going execution.

In this way, a decision based on a property φ verification lookahead step is to be taken at each control point met on its way by the execution, thus a controllable event σ being enabled. The mediated execution of the system continues until either the violation of the desired property φ cannot be avoided, or the execution blocks, or it successfully completes.

After one control point q_{cp} has been completely explored, its previous control point on t will follow. The verification-driven execution phase ends when the U model cannot be further refined, since all control points on t in the system model have been explored. The performed system executions are partitioned into two sets: T_c, the set of correct execution traces, and T_e, the set of erroneous traces.

2.2.2 Approximation refinement

Given that the model U overapproximates the behaviour of C_U, we are assured that whenever there is one execution in C_U there is one in U. The converse does not hold in the two following cases.

Case 1. Let us suppose that after an event σ_i in state q_1 of U, the following controllable event σ_{i+1}, assumed by the fake adaptor to happen, fails to do so. That means the component C_U has got into a state in which σ_{i+1} is not enabled anymore. This state, previously thought to be q, must now be replaced with a new copy q'. The transition function becomes: $\delta^U(q_1, \sigma_i) = \{q'\}$, $\delta^U(q', \sigma) = \delta^U(q, \sigma)$, $\forall \sigma \in \Sigma^U \setminus \{\sigma_{i+1}\}$. A self-loop on σ from q will lead in q' to the transitions $\delta^U(q', \sigma) = \{q, q'\}$.

Case 2. If from a state q we have nondeterminism in the model U: $|\delta^U(q, \sigma_i)| \geq 2$ and we observe an execution sequence $\sigma_i.\sigma_{i+1}$ from q, we can use this execution to resolve the nondeterminism, as component C_U is assumed deterministic. Thus, all transitions $\delta^U(q, \sigma_i)$ to a state q' for which $\delta^U(q', \sigma_{i+1}) = \emptyset$ are removed from U.

Thus, by always splitting the current state and/or reducing the number of outgoing transitions from a state, the C_U component model U is incrementally refined during the exploratory training phase. The refinement stops either when state number limit λ has been reached, or all the control points on t have been explored. We obtain a model U' that is a more precise approximation of the real component C_U.

The model U' might still contain, however, nondeterministic transitions triggered by uncontrollable events σ, from states q where σ was neither observed at execution, nor eliminated from $\Sigma^U(q)$. We propose three ways to resolve this situation, and build a final model U''.

- **Optimistic approach**: For all states q, all transitions triggered by uncontrollable events $\sigma \in \Sigma^{U'}_!(q)$ not observed at execution are removed from U. Thus, the transition function becomes $\delta^{U''}(q, \sigma) = \emptyset$ iff $\sigma \in \Sigma^{U'}_!(q)$ and $observed(q, \sigma) = false$. We obtain a deterministic refined model U'', that leads to a small controller, but further runtime risks may appear if σ can actually manifest from q.

- **Pessimistic approach**: We let the final model U'' be U', thereby keeping in this model the nondeterminism due to uncontrollable events unobserved at execution. The controller synthesis problem is harder for nondeterministic models, but this choice ensures that the obtained controller is safe with respect to the possible executions of C_U.

- **Semi-optimistic approach**: For any state q, all uncontrollable events $\sigma \in \Sigma^{U'}_!(q)$ not observed during the verification-driven execution phase, are assumed unobservable. The transition function becomes $\delta^{U''}(q, \sigma) = \{q\}$ iff $\sigma \in \Sigma^{U'}_!(q)$ and $observed(q, \sigma) = false$. We obtain a deterministic refined model U'', that still accepts all $\sigma \in \Sigma^{U'}_!(q)$. The resulting controller, while significantly smaller than the pessimistic one, is still much larger than the optimistic controller. Further

risks may still appear if for some such events σ the actual transition from q is not a self loop.

Our model refinement phase is related to the counterexample guided abstraction refinement (CEGAR) developed by Clarke et al. [8], since the blocking execution can be regarded as a spurious counterexample. The difference is that we do not have an accessible concrete model for the refinement of the overapproximated model, but a black-box component, which we explore at runtime.

2.3 Controller Synthesis

An adaptor, as earlier stated, acts as a controller in a component-based system: it can enable receive events by forwarding the corresponding messages, or disable them by consuming these messages. Considering the set of send events $\Sigma_!$ in a component finite state machine as uncontrollable, and the set of receive events $\Sigma_?$ as controllable, we can solve the control problem in our system by building an adaptor A. The plant to be controlled will be the asynchronous product $K \times U''$ and the specification will be property φ.

In order to ensure that the real system composed from the components C_K and C_U will satisfy φ, as the model U'' still represents an approximation of C_U, two possible variants of controller synthesis can be employed. The first, classical one, allows the construction of a most permissive controller, while the second one, more restrictive, limits the system to the observed correct and controllable behaviour.

2.3.1 Permissive control

Let us note by $Ctrl$ the system controller. The computation of $Ctrl$ as the controller of the plant $K \times U''$, for the specification φ relies on the classical result of Ramadge and Wonham: $Ctrl = \mathbf{supcon}(K \times U'', \varphi)$, where **supcon**, described in [19], is a fixpoint procedure.

This leads us to a most permissive controller $Ctrl$, where all behaviours of the plant $K \times U''$ that do not end up in a violation of property φ are allowed. As shown in [4, 2] when the control problem is modelled as an acceptance game, the same result can be obtained as the most permissive winning strategy by an alternating reachability algorithm. If the specification φ changes to an ATL formula ψ, within a system comprising two agents, one determined by the set of controllable transitions, and the other one by the set of uncontrollable ones, the desired controller is obtained through model checking of the formula $\ll Ctrl \gg \mathbf{G}\psi$. [1]

The adaptor A is then obtained directly from the controller $Ctrl$ by mirroring in the set Σ^A all events from Σ^{Ctrl}.

2.3.2 Restrictive control

If, out of cost reasons, the behaviour exploration process has been stopped early, the resulting model U'' might still exhibit some false controllability, i.e., transitions $\delta^{U''}(q, \sigma)$, where $\sigma \in \Sigma_?^{U''}$, that do not manifest at runtime. This may lead to execution errors in the controlled system. For many systems, this can be accepted: the run is stopped, the model U'' re-refined to a new model U''', and another adaptor A' is synthesized. However, for systems with strong safety constraints, a more restrictive solution is needed, with the price of sacrificing part of the allowed behaviour.

During the exploration of system S_R behaviour, two sets of execution traces have been obtained: the set T_c containing all correct traces, and T_e containing all erroneous execution traces. Let $T_{cctrl} \subseteq T_c$ be the set of all correct and controllable execution traces. If the behaviour exploration of S_R has been wide enough, there are sufficient correct controllable traces in T_{cctrl} for us to limit the controlled system behaviour to them without being excessively restricting.

Let C be the finite state machine resulting from merging all the traces in T_{cctrl}. The desired restrictive controller for the system is then $CSafe = C$, while the adaptor $ASafe$ is obtained from $CSafe$ by mirroring the event set.

3. RELATED WORK

Part of our approach is related to the explaining of counterexamples in [15], as we also explore the vicinity of an error trace. However, as it uses model checking for exploration, their work applies only to fully specified programs, while we cannot anticipate the behaviour of the unspecified component and we need to actually run the system.

Another related method is the one by Giannakopoulou et al., which relies on environmental assumption generation when verifying a software component against a certain desired property [14]. Their approach is based on the work of de Alfaro and Henzinger [12] stating that two components are compatible if there exists an environment that enables them to correctly work together. The technique obtains a formal characterisation of the weakest environment needed for a certain property to hold. By building a correctness-enforcing controller, our approach actually creates an environment in which a desired property would hold. However, while in [14] the analyzed component is well specified, our approach addresses systems that also contain black-box components, whose behaviour and controllability must be understood before building an adaptor.

The use of verification-driven execution relates our method to the work of Harel et al. in the domain of smart play-out [11, 10]. Smart play-out is a lookahead technique that employs model-checking to execute and analyze Live Sequence Charts. The play-out technique is mainly used to actually execute specifications from a GUI, during the software design process, in order to better understand the application requirements. Both approaches use verification-driven execution to improve knowledge, but while we automatically infer the behaviour of an existing, unknown component with the purpose of controlling it, smart play-out experiments with execution scenarios to find the best design options.

Peled et al. have developed a model checking technique for programs with missing or inaccurate models [16, 13]. This approach is closely related to ours, since an approximated model is proposed, verified, and compared to the real behaviour using black-box testing. Found differences are used to generate a new model, while counterexamples are validated by testing. However, in contrast to our approach, their employed model only has input events, which highly simplifies the learning process, and, since the aim is to find feasible counterexamples, their model rather underapproximates real behaviour. When refining the model, an intermediary learning phase ensures its consistency, a problem which our algorithm avoids by overapproximating the system behaviour. Also, in our case, the verification, testing and model refinement phases are strongly interleaved, which allows us to obtain a good model earlier, while their approach has distinct phases, since an early confirmed counterexample would save further learning effort. More than just verifying the black-box, our approach explores correct

controllable behaviour in the vicinity of an error trace with the purpose of generating a correctness-enforcing controller.

By starting its analysis from an observed error trace, our approach is also related to Zeller's delta debugging [9], which has been applied to the component-based area to isolate sets of interactions relevant to the fault. In this case, the delta debugging algorithm works by recording all the interactions in one erroneous and one correct system execution, and then systematically reproducing parts of the failing and successful scenarios using a specialised tool, JINSI [6]. An important difference with respect to our approach is that JINSI has a diagnosis-oriented nature, and it doesn't focus on providing a fix for the system. Also, while our solution generates new execution traces, thus inferring new knowledge about the system, JINSI minimizes the information in two preexisting logs - thus the two techniques are complementary.

ClearView, developed by Perkins et al. [18], is a tool for automatically patching errors in application software. It learns invariants from observed correct executions, detects failing executions and monitors them to find violations of the former invariants. A set of candidate repair patches are generated, then several instances of the patched applications are observed to select the best patch. Both ClearView and our approach work towards ensuring software robustness: while ClearView only extracts its information out of observing correct executions, our method performs an experimental, directed search of the possible system behaviour space, thus being able to reach corner cases that are usually ignored in normal executions. Also, the ClearView technique is prone to bad invariant-error correlations: only some patches are efficient. By contrast, though not complete, our approach is sound: found error traces, if controllable, are always disabled from further occurrence by the generated controller.

4. CONCLUSIONS AND FUTURE WORK

We have presented an approach that enables the control of a system containing an unknown component, towards the satisfaction of a desired global property. Starting from a most general model of the unspecified component, our technique explores its runtime behaviour and uses the information acquired to refine the model. The aim is to ensure the avoidance of erroneous executions by building a permissive controller. However, if the system safety requirements demand it, a more restrictive controller can be computed.

The main contributions of our method are the behaviour discovery algorithm, the application of controller synthesis to black-box component integration and the three solutions for resolving nondeterminism in the unknown component model. Our approach is non-intrusive, thus being well-adapted for legacy software. It addresses the rather frequent problem of integrating insufficiently known components, and resolves observed errors, while improving the component knowledge. Also, the obtained sets of erroneous and correct traces can become maintenance documentation.

We currently work to implement our method and develop a specialised framework for it. For initial experiments, we have used Supremica tool [17] for controller synthesis, while the bounded model-checking and model refinement modules will be locally implemented. We will validate our prototype on a set of Enterprise JavaBeans component systems using Java Message Service for asynchronous messaging.

Future extensions of this work include generating regression tests for components in the corrected system, since later component versions should exhibit similar behaviour for the adaptor to be effective, and synthesizing distributed, local error-avoiding controllers for remote components.

Acknowledgments

We are grateful to Marius Minea and Mihai Balint for substantial feedback. This work is supported by the European project AVANTSSAR and by the grant POSDRU (2008).

5. REFERENCES

[1] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *J. ACM*, 2002.

[2] A. Arnold, A. Vincent, I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theor. Comput. Sci.*, 2003.

[3] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. Synthesis: A tool for automatically assembling correct and distributed component-based systems. *ICSE* 2007.

[4] J. Bernet, D. Janin, I. Walukiewicz. Perm. strategies: from parity games to safety games. *ITA*, 2002.

[5] A. Biere, A. Cimatti, E. M. Clarke et al. Bounded model checking. *Advances in Computers*, 2003.

[6] M. Burger and A. Zeller. Replaying and isolating failing multi-object interactions. *WODA*, 2008.

[7] C. Canal, P. Poizat, and G. Salaün. Model-based adaptation of behavioral mismatching components. *IEEE Trans. Softw. Engg.*, 2008.

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 2003.

[9] H. Cleve and A. Zeller. Locating causes of program failures. *ICSE*, 2005.

[10] D. Harel and S. Maoz. *Concurrency, Compositionality, and Correctness*, chapter On the Power of Play-Out for Scenario-Based Programs, LNCS. Springer, 2010.

[11] D. Harel, H. Kugler and A. Pnueli. Smart play-out. *OOPSLA* 2003. demo paper.

[12] L. de Alfaro and T. A. Henzinger. Interface automata. *ESEC/FSE-9*, 2001.

[13] D. Peled, Moshe Y. Vardi. Black box checking. *FORTE/PSTV*, Kluwer, 1999.

[14] D. Giannakopoulou, C. Păsăreanu, and H. Barringer. Component verification with automatically generated assumptions. *ASE*, 2005.

[15] A. Groce and W. Visser. What went wrong: Explaining counterexamples. *SPIN*, 2003.

[16] D. Peled. Model checking and testing combined. *ICALP*, Springer, 2003.

[17] K. Åkesson, M. Fabian et al. Supremica: an integrated environment for verification, synthesis and simulation of discrete event systems. *WODES*, 2006.

[18] J. H. Perkins et al. Automatically patching errors in deployed software. *SOSP*, 2009.

[19] P. Ramadge and W. Wonham. The control of discrete event systems. *Proc. of the IEEE*, Jan. 1989.

[20] H. W. Schmidt and R. H. Reussner. Generating adapters for concurrent component protocol synchronisation. *FMOODS*, 2002.

[21] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Syst.*, 1997.