Asynchronous Learning for Service Composition

Casandra Holotescu

Politehnica University of Timişoara
Dept. of Computer and Software Engineering
casandra@cs.upt.ro

Abstract. Correctness of system compositions is automatically ensured by using formal behavioural models of services. However, such models are not always provided. We present a model inference technique for black-box asynchronous services, that interleaves behavioural exploration and incremental model refinement. To save learning effort, only behaviour relevant to the desired system specification is explored. Compared to existing inference techniques that assume only controllable behaviour, our method addresses also uncontrollable events. Experimental results show that obtained models can be successfully used for a safe composition.

1 Introduction

The paradigm of service oriented applications relies on assembling third party web services together, so that the final system will satisfy some temporal property. This is done by building an adaptor/mediator – a service in-the-middle which properly coordinates the interactions between the other services towards the satisfaction of the desired composition goal. Several automatic, model-checking based techniques have been developed for mediator synthesis, such as [2,6,18]. All these techniques use specified formal models of the service behaviour. Unfortunately, providing formal specifications for third party services is still far from common industrial practice. One might often need to integrate a third party service with only partial, lightweight specifications. In this case, composition correctness can only be ensured by a preliminary model inference step.

For asynchronous services, which interact by sending/receiving messages, correct composition is also a control problem [21], where the controllable events are the receive events, while send events are uncontrollable. While a black-box synchronous, and thus, completely controllable service can have its model inferred by a variant of the Angluin algorithm [1], just as it is the case for black-box components in the works of Peled [12], Shahbaz [22], or Berg [3], things are different in the presence of uncontrollability. Angluin's algorithm assumes regular inference takes place as a series of membership and equivalence queries, answered to by an oracle [1], which for a black box component/service is the component/service itself. In the presence of uncontrollable events, however, such queries can no longer be generated as simple input sequences, or series of input sequences, since an uncontrollable transition can occur anytime.

Our technique aims at composing systems that contain deterministic black-box services. To the best of our knowledge, it is the first model inference technique for asynchronous black-boxes, and, together with [22], one of the few that consider black-box model inference for compositional purposes.

Considering a desired temporal property, we delegate the task of coordinating the system to an intelligent adaptor, which monitors the services and controls their interactions by intercepting and forwarding their messages, thus actively exploring the behaviour relevant to the desired property.

For each black-box, we start with a most general model, which is incrementally refined each time it proves inconsistent with the run-time behaviour of the service, e.g. when a forwarded message is not accepted, etc. To save learning effort, only behaviour conforming to the system specification is explored. The final model obtained is a safe approximation of the black box behaviour, which can be used for adaptor synthesis. Building upon our previous work [16,17], this paper differs by an improved model refinement method, safety and termination proofs for presented algorithms, and an experimental evaluation of our technique.

2 Method

2.1 Assumptions

A desired system S is to be composed out of a set $\mathcal{W}$ of deterministic services, out of which n are black-boxes. Without loss of generality, we further on consider the case where all services in the system are incompletely specified, and $|\mathcal{W}| = n$.

Each black-box service WS_i is associated to a tentative model, which represents an approximation of its real behaviour. This model is described by a Büchi automaton, $U_i = \langle Q^i, q_0^i, Q_f^i, \Sigma^i, \delta^i \rangle$ where:

- Q^i is the state set and $q_0^i \in Q^i$ the initial state
- Σ^i is the event set of WS_i and $\Sigma = \bigcup \Sigma^i$ is the event set of the system
- $Q_f^i = Q^i$ is the set of accepting states
- $\delta^i : Q^i \times \Sigma^i \to \mathcal{P}(Q^i)$ is the transition function

An event σ in Σ is either a message send: $msg!$, or a receive: $msg?$. We denote by $\Sigma^i(q)$ the set of events σ for which $\delta^i(q, \sigma) \neq \emptyset$, with its two subsets $\Sigma^i_?(q)$ – the subset of controllable events, and $\Sigma^i_!(q)$ – the subset of uncontrollable events.

Considering a string of events $t_k = \sigma_0\sigma_1..\sigma_k$, and Σ^i_* the set of strings formed from events in Σ^i, then the extension δ^i_* to strings of events of the transition function δ^i is the string transition function $\delta^i_* : Q^i \times \Sigma^i_* \to Q^i$ such that $\delta^i_*(t_k) = \bigcup \delta^i(q, \sigma_k)$ for all $q \in \delta^i_*(t_{k-1})$, and $\delta^i_*(t_0) = \delta^i(q_0, \sigma_0)$.

If a trace $t \in \Sigma^i_*$ is observed at runtime, we denote this fact by $obs(t)$. If an event $\sigma \in \Sigma^i$ is observed from a state $q \in Q^i$, we denote by $obs'(\sigma, q)$.

We also assume for every service WS_i that the inner service scheduler is fair with respect to uncontrollable events. We assume a bounded fairness [8], considering a bound θ on the number of state visits for a single state q of the model as a fairness bound. This means that if a state q is reached for at least θ

times, which we denote by $visits(q)$, every uncontrollable event σ enabled in q is observed at least once: $\sigma \in \Sigma^i_!(q) \wedge visits(q) \geq \theta \longrightarrow obs'(\sigma, q)$.

Tentative model U_i approximates the real behaviour of WS_i, which we assume is precisely described by an unknown model R_i, also a Büchi automaton.

The desired system S must comply with a safety property Φ, described by a similar Büchi automaton. For S to comply with Φ, S must simulate Φ.

Let us also assume that $\forall i \leq n-1. ack!, rst? \in \Sigma^i$, where $ack!$, $rst?$ are special, auxiliary events. Event $ack!$ confirms a successful receive. Event $rst?$ forces the service to return to its initial state, from any state q. This assumption makes receive events externally observable and allows for reset. To ensure termination, we consider an upper bound m on the number of states of any service WS_i.

2.2 Behaviour Exploration

Let us denote by $U_\times$ the asynchronous product of the tentative models $U_\times = U_0 \times U_1 \times ... \times U_{n-1}$. As defined in [?], we consider a control point in $U_\times$ as a global state q_{cp} for which at least one of the outgoing controllable event sets $\Sigma^i_?(q_{cp})$ contains 2 or more receive events: $|\Sigma^i_?(q_{cp})| \geq 2$. This basically means that in state q_{cp} we can choose from at least two possible execution paths.

During the exploration process, the property automaton Φ is executed synchronously with the system model $U_\times$. If, from a state q, an uncontrollable event σ' occurs at runtime, and no transition triggered by σ' exists in Φ from the current state, then q is marked as a forbidden state. When the execution reaches a control point q_{cp}, a receive event $\sigma \in \Sigma^i_?(q_{cp})$ is enabled for execution if from the current state of Φ a transition on σ exists, and $\delta(q_{cp}, \sigma)$ does not contain forbidden states. If runtime observations report an inconsistency between U_i and the real behaviour of WS_i, U_i is refined. The execution is forced to end when:

– A forbidden state is reached.
– A cycle that respects the safety property is confirmed:
  - a cycle is found in the system model
  - the execution enters the cycle, and doesn't leave it for a number of λ transitions, during which the property automaton visits at least one accepting state – which ensures an infinite correct run.

The threshold $\lambda = m^n$ is the maximum state space size of the system. As stated in [12], for a black-box component with maximum m states, an accepted trace of the form uv^m, where u and v are traces in δ^i_*, and v^m represents the repetition of v for m times, is enough to prove the existence of a correct infinite run.

2.3 Trace Removal

The basic intention of the trace removal process is to refine the model by removing only the partial trace observed not to be accepted at runtime.

Let us consider a current execution trace $t_k = \sigma_0\sigma_1..\sigma_k$, observed at runtime: $obs(t_k)$. Let t_j, $j \leq k$, be a prefix of t_k such that a state q_j is reached after performing the transitions triggered by t_j, and from the initial state q_0, q_j is uniquely reachable by t_j: $\{q_j\} = \delta^i_*(t_j)$, and, if q_j is part of a cycle in the model, t_j reaches q_j only once. If no such t_j exists, then q_j is the initial state.

Suppose now that after event σ_k, we try to enable controllable event σ_{k+1}, and fail. The trace $t_{k+1} = t_k.\sigma_{k+1}$ is thus missing from the behaviour of real service WS_i, therefore it has to be also removed from the behavioural model U_i, together with all traces prefixed by t_{k+1}, but without removing other possible strings of events. We denote the application of this removal process by $\mathit{remove}(t_{k+1})$.

Then, the refinement of U_i will have a number of $k - j + 1$ new states: $q'_j, q'_{j+1}, .., q'_{k+1}$ and the transition function δ^i is modified as following:

– $\delta'^i(q'_l, \sigma_{l+1}) = \{q'_{l+1}\}$, for $j \le l < k$ (the new string of states corresponding to the unsatisfiable prefix).
  If $j = 0$, q'_0 is the new initial state of the model, otherwise $q'_j = q_j$.
– $\delta'^i(q'_k, \sigma_{k+1}) = \emptyset$ (infeasible transition)
– $\delta'^i(q'_l, \sigma) = \delta^i(q_l, \sigma)$, for $j \le l \le k$, $\sigma \ne \sigma_{l+1}$ (any transition not in prefix returns to initial model)
– $\delta'^i(q, \sigma) = \delta^i(q, \sigma)$, if $q \ne q'_j$ (all other model transitions are preserved)

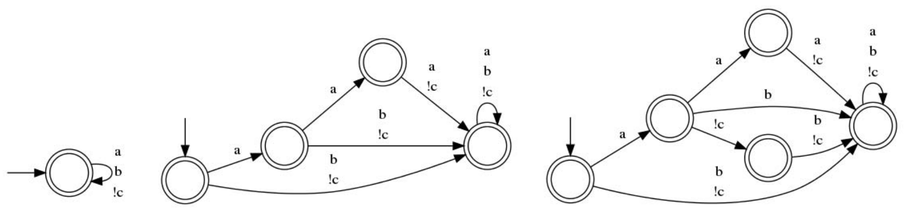


Fig. 1. Trace removal process: 1(a) initial tentative model, 1(b) refined model after removing trace $a?.a?.b?$, 1(c) model after second refinement: removing trace $a?.c!.a?$

After reaching the maximum state number m, the exploration and refinement process still continues until either the maximum number of executions γ, or the fixpoint condition has been reached. Refining the model, in this case, can no longer rely on the presented trace removal algorithm, since no new states can be created. So, in this case, the infeasible transitions on a controllable event σ are simply removed from their state of origin q, which we denote by $\mathit{remove}'(\sigma, q)$.

Expanding Forbidden Behaviour. When an erroneous trace $t_{bad} = \sigma_0\sigma_1..\sigma_k$ is observed, the model is refined and an unrolling of the error trace takes place, to isolate it from other traces with which might share a correct prefix. The purpose is to make it easier for the controller to disable the error trace on the final model.

The transformation is similar to the usual trace removal phase, with the difference that no transitions are removed from $\delta^i_*(t_{bad})$, but instead the new state $q'_p = \delta^i_*(t_{bad})$ is marked as forbidden: $\mathit{bad}(q'_p)$.

Final Pruning. When the exploration process ends, two final refinement actions are applied on a model U_i:

1. any controllable transitions yet unexplored are removed.
2. if a state was visited for a number of times that exceeds the fairness bound, the unobserved uncontrollable transitions are removed from that state

The reason for action 1 is that keeping infeasible controllable transitions in a model used to generate a controller for the system might lead to unsafe behaviour later on, in the composed system, since the adaptor/controller could try to enable an infeasible transition, fail, and thus reach an error state.

Action 2 is based on the fairness assumption, which we use to prune infeasible uncontrollable events, i.e. the uncontrollable events that didn't occurred within the fairness bound. If an uncontrollable event σ hasn't occurred from state q, after the service has been in state q for $nv \geq \theta$ times, then we conclude that σ never occurs from q. Else, all uncontrollable transitions from q rest unchanged.

Thus, each final model overapproximates the real uncontrollable behaviour and underapproximates the controllable one, which allows for a reliable adaptor synthesis an adaptor obtained for these models will also work on the real system.

2.4 Adaptor Synthesis

The adaptor for the system is computed from a controller enforcing the property Φ over the system plant. Consider a refined model of a service WS_i to be U'_i. Also, let $U'_\times$ be the asynchronous product of these refined models: $U'_\times = U'_0 \times U'_1 \times ... \times U'_{n-1}$. Thus, $U'_\times$ models the system plant. The computation of the controller, denoted by *Ctrl*, for the specification Φ, relies on the classical result of Ramadge and Wonham: $Ctrl = \mathbf{supcon}(U'_\times, \Phi)$, where **supcon**, described in [21], is a fixpoint procedure. All behaviours of the plant $U'_\times$ that do not violate Φ are allowed by *Ctrl*. Finally, adaptor A is obtained from controller *Ctrl* by mirroring its event set.

2.5 Proofs

We prove below that the obtained adaptor is safe, and that the model learning algorithm eventually reaches its fixpoint termination condition.

Definition 1. *A controller Ctrl that enforces a property Φ over a plant $\mathcal{P}$ is safe if all behaviour of controlled plant $Ctrl \| \mathcal{P}$ satisfies Φ: $\mathcal{L}(Ctrl \| \mathcal{P}) \subseteq \mathcal{L}(\Phi)$. An adaptor A is safe for a system $\mathcal{S}$ and a property Φ iff its corresponding controller Ctrl is safe for the plant $\mathcal{S}$, and property Φ.*

Theorem 1. *Adaptor A, obtained for final models U'_i and property Φ, is safe.*

Proof. Suppose A is not safe $\longrightarrow$ controller *Ctrl* over plant $U_\times$ is not safe. Thus: $\exists t \in \Sigma_*.\, t \in \mathcal{L}(Ctrl \| R_\times) \wedge t \notin \mathcal{L}(\Phi \| R_\times) \longrightarrow t \in \mathcal{L}(R_\times) \setminus \mathcal{L}(\Phi)$ Since $Ctrl = \mathbf{supcon}(U'_\times, \Phi) \longrightarrow \mathcal{L}(Ctrl \| U'_\times) \subseteq \mathcal{L}(\Phi)$, we have $t \in \mathcal{L}(Ctrl \| (R_\times \setminus U'_\times))$.

Let $t = u.\sigma.v$, where $u \in \mathcal{L}(Ctrl||U'_\times)$ is a correct prefix, σ is the error-inducing event, and v represents the rest of the trace. Let $\sigma \in \Sigma^i$. Then: $u \in \mathcal{L}(Ctrl||U'_\times) \longrightarrow u.\sigma \notin \mathcal{L}(Ctrl||U'_\times)$. This implies $\sigma \in \Sigma^i_! \wedge P_i(u).\sigma \notin \mathcal{L}(U'_i)$.

The learning process has started with a most general model, so it means that σ was incorrectly removed from every set $\Sigma^i(q)$, where $q \in \delta^i_*(P_i(u))$.

Case 1. $P_i(u).\sigma$ removed by the trace removal procedure. If $|Q^i| < m$: $remove(t')$, $t' = t_k.\sigma_{k+1} \longrightarrow obs(t_k) \wedge \sigma_{k+1} \in \Sigma^i_?$. But $\sigma \in \Sigma^i_! \longrightarrow$ contradiction: $P_i(u).\sigma$ not removed. Else, if $|Q^i| = m$: $remove'(\sigma', q) \longrightarrow \sigma' \in \Sigma^i_?$. Also, $remove(P_i(u).\sigma) \longrightarrow P_i(u).\sigma = w'.\sigma'.w.\sigma \wedge \sigma' \in \Sigma^i_? \wedge remove'(\sigma', q)$ $\forall q \in \delta^i_*(w')$. But $remove(w'.\sigma') \longrightarrow remove(P_i(u))$. Since $obs(P_i(u))$, contradiction results.

Case 2. $P_i(u).\sigma$ removed in the transition pruning phase: $\sigma \in \Sigma^i_!(q) \wedge remove'(\sigma, q)$ implies $\neg obs(\sigma, q) \wedge visits(q) \geq \theta$. Since WS_i fair by bound θ: $\sigma \in \Sigma^i_!(q) \wedge visits(q) \geq \theta \longrightarrow obs(\sigma, q)$ Therefore, $\sigma \notin \Sigma^i_!(q)$, where $q \in \delta^i_*(P_i(u))$. But $obs(P_i(u).\sigma)$, thus $\sigma \in \Sigma^i_!(q)$, $\forall q \in \delta^i_*(P_i(u)) \longrightarrow$ contradiction.

From *Case* 1 and *Case* 2, we have that partial trace $P_i(u).\sigma \in \Sigma^i_*$ cannot be incorrectly removed from U'_i, therefore adaptor A is safe.

Theorem 2. *The fixpoint termination condition* $\forall i \leq n \wedge \forall q \in Q_i \wedge \forall \sigma \in \Sigma^i_?(q) \longrightarrow\ obs(\sigma, q) \vee bad(q')$, $\forall q' \in \delta^i(\sigma, q)$ *is eventually reached.*

Proof. Suppose the condition is never reached: $\exists i \leq n \wedge \exists q \in Q_i \wedge \exists \sigma \in \Sigma^i_?(q)$ *s.t.* $\neg obs(\sigma, q) \wedge \neg bad(q')$, $\forall q' \in \delta^i(\sigma, q)$. We know that $\sigma \in \Sigma^i_?(q) \longrightarrow \sigma = msg? \longrightarrow \exists j \leq n \wedge \exists \sigma' \in \Sigma^j_!$ *s.t.* $\sigma' = msg!$. Since all available messages are tried on, starting with those yet unconfirmed, then $\sigma \in \Sigma^i_?(q) \wedge \neg obs(\sigma, q) \longrightarrow \neg obs(\sigma', q)$. WS_j is fair under bound θ, therefore $\neg obs(\sigma', q') \wedge \sigma' \in \Sigma^j_!(q') \longrightarrow visits(q') \leq \theta$. Suppose $\{q'\} = \delta^j_*(w)$, where $w \in \Sigma^j_*$. Then, q' is reached for θ times in at most $\theta^{|w|}$ executions. Thus, it results that WS_j is not fair $\longrightarrow$ contradiction. Therefore, the termination condition is eventually reached.

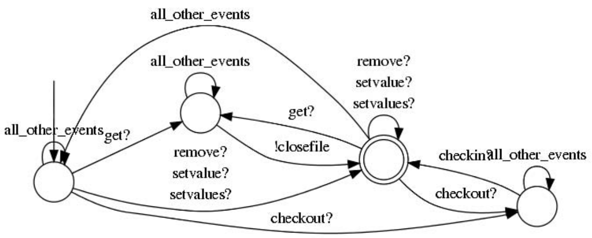
3 Experimental Results and Discussion

We implemented our solution as the BASYL (Black-box Asynchronous Learning) prototype. All experiments were run on an Acer Aspire 3820TG with a 2.26 GHz Intel Core i5 430M processor, 4 GB RAM memory and a 3 MB cache.

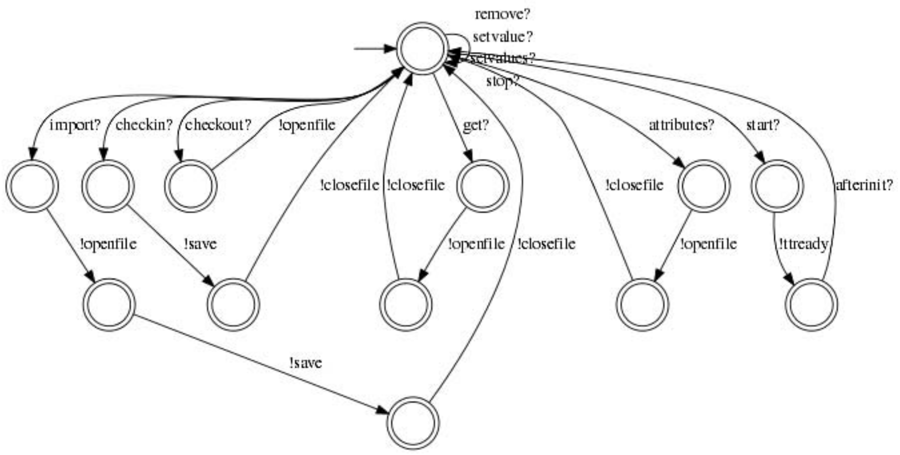
BASYL is written in Java and contains a monitoring component, that observes the events occurring at runtime and advances the service models accordingly, a decision logic component based on a bounded model-checker, that chooses the next controllable transition to be enabled, and a model refiner component, that modifies the approximate models whenever necessary. In order to link the high abstraction level at which BASYL works to a concrete technology, technology specific drivers are needed – for the case study presented below, we implemented such a driver for JMS.

To validate BASYL, we built an abstract version of one of the case studies presented in [24], using the Glassfish Server 3.1 implementation of Java Message Service for asynchronous message passing – it is worth noting here that using SOAP over JMS, although not a standard solution, is considered more reliable for building web services than SOAP over HTTP. We have chosen the case study of components ThinkTeam and CAD, but used only one CAD instance instead of two, and rewrote the coordination policy accordingly (see figure 2). For controller synthesis, the Supremica tool [19] was used. We assumed ThinkTeam as incompletely specified. Its alphabet contains 15 events, out of which 4 are uncontrollable, and the fairness bound θ was set to 15. We experimented with various maximum model sizes. The real model has 13 states and 22 transitions.

Because we safely approximate the real service behaviour, models that can lead to the synthesis of a safe adaptor can be obtained without reaching the refinement fixpoint. In the case studied, it took BASYL a number of 2 executions for $m = 5$, 90 for $m = 13$ and 290 for $m = 20$, to obtain models for which Supremica could synthesize system controllers (see Table 1). Since these models are not complete, but only safe approximations, the obtained controllers are more restrictive, and usually larger than the controller obtained for the real model.



(a)



(b)

Fig. 2. Case study: 2(a) Property / Coordination policy, 2(b) Think Team real model

Table 1. Controller variation by max. model size and permissiveness

max. model size	min. ex. nr.	1st ctrl. size	1st ctrl tr.	γth ctrl size	γth ctrl tr.
5	2	7	58	28	204
13	90	56	380	84	577
20	290	126	841	133	913
real size = 13	**real model**	**56**	**247**	**56**	**247**

Due to the trace removal procedure, our technique does not lead to minimal models, as opposed to the Angluin algorithm. Therefore, we continue the exploration and refinement process even after the maximum state size is reached, until the fixpoint condition becomes true. However, the model is never guaranteed to be complete, i.e. equivalent to the real service model, due to the final pruning phase, while the Angluin algorithm obtains complete models within the state size limit. But what our algorithm does guarantee is obtaining safe approximate models, for which a generated controller is also safe, while the Angluin algorithm is not fit for learning systems with uncontrollable events, and would not produce safe approximations. Thus, our approach is specifically adapted to the reliable composition of asynchronous safety-critical systems. Also, depending on the system requirements, the system controller can be obtained in two ways:

– the first possible controller: after each model refinement, a controller synthesis phase takes place; if a controller can be obtained for the pruned model, the learning process stops. Thus, the learning process is shorter.
– the most permissive controller: since learned models become more precise with refinement, thus leading to increasingly more permissive controllers, the learning process only stops when the termination condition is reached.

Another limitation of our approach is its time complexity. As stated above, the cost of exploring behaviour by execution is greater than the cost of model checking, since paths are not explored simultaneously, but sequentially. Still, due to the assumption of fairness, each path can be eventually explored, if feasible.

If all states are control points, to obtain a trace of length λ a number of $O(|\Sigma|^\lambda)$ decisions are to be taken, which implies $O(|\Sigma|^\lambda)$ actual executions for an exhaustive, controllable exploration. This is, however, the large cost of black-box exploration. The black-box checking algorithm of Peled in [12] has a time complexity of $O(l^3|\Sigma|^l + l^3|\Sigma|^{m-1} + l^2pm)$ – where l is the size of an equivalent minimal automaton and p the size of the property automaton – to prove the correctness of the component, without addressing the controllability issue.

When exploring behaviour at runtime, we have to reset services each time we want to explore another path, and restart the run from the initial state. To reach again the original control point is difficult, due to the uncontrollable transitions in the system, which may steer the execution astray. To perform an efficient exploration, we need to reduce the number of necessary executions. This is why we explore only the paths that conform to the specification.

The number of runs needed to observe a trace depends on its controllability. If θ is the fairness bound, a trace t with p uncontrollable events needs up to θ^p runs to be observed. This is why we avoid classic querying: a query for membership of a trace t has a cost exponential in the number of its uncontrollable events.

4 Related Work

A classical reference in this area is the black-box checking technique of Peled et al. [12], that uses the Angluin algorithm [1] for model inference. An approximated model is proposed, verified, and compared to the real behaviour. Found differences are used to generate a new model, while counterexamples are validated by testing. However, their model only has input events, which highly simplifies the learning process, and their model underapproximates real behaviour. Also, in our case, verification, testing and model refinement are strongly interleaved, which allows us to obtain an useful model early, while their work has distinct such phases, aiming for an early confirmed counterexample.

Recent advances on dynamic model mining underline the critical significance of this domain. The works of Berg et al. [3], or the GK-tail algorithm developed by Lorenzoli et al. [14] focus on extracting extended finite state machines from execution traces. GK-tail uses inferred invariants and positive execution samples to extract EFSMs, while the method of Berg et al. uses an adaptation of the Angluin algorithm [1], thus querying for trace membership and model equivalence. In contrast, our approach learns the model on-the-fly, using both positive and negative samples, while always maintaining a safe approximation of the real service behaviour, instead of querying for equivalence.

Derived from the Angluin algorithm is also RALT, the work of Shahbaz [22], which can infer parametrized Mealy machines by adding invariant learning to regular inference, and studies a component behaviour first in isolation and then while in interaction with the rest of the system. Other automata learning tools are LearnLib [20], which also learns Mealy machines, and Libalf [5], which learns deterministic and even non-deterministic automata – both are algorithm libraries, containing various optimizations and extensions of the classic Angluin algorithm. However, none of these approaches deals with the issue of uncontrollable events, which would make membership and equivalence queries difficult.

In [23], Suman et al. describe a method to extract state models for black-box components under the form of finite state machines with guard conditions. It considers that a state is defined by the method invocations it accepts, and it discovers potential new states by invoking all active methods from the current state. The work of Dallmeier et al. in [7] relies on test case generation to systematically extend the execution space and thus obtain a better behavioural model by dynamic specification mining. Also, in [9], Ghezzi et al. infer behavioural models of black-box components that embody data abstractions. These techniques work with synchronous method calls, while our approach addresses asynchronous message exchange, which involves the issue of uncontrollable events. Also, our technique learns the model not in isolation, but with respect to both the desired property and feasible interactions with the other services in the system.

In the area of web services, the work of Bertolino et al. [4] and Cavallaro et al. [13] is related to ours. While [4] uses the WSDL interface to extract possible dependencies between the I/O data, and then validates them through testing, thus obtaining the behavioural protocol, the work in [13] takes this technique further, and develops a method for synthesizing web service adaptors that enable a correct interoperation when an old service is replaced by a new one. However, the web services in [4,13] are stateless, while our approach addresses stateful black-box services. While their work starts from a data-dependency perspective, ours explores the language of the service and its controllability, regarding data at a higher level of abstraction, as passed messages. Therefore, we consider the two approaches complementary.

Grabe et al. [10] develop a technique for testing asynchronous black-box components, by using an executable interface specification. Their method addresses the difference between interacting under component control or under environment control, by describing the component behaviour in an assumption-commitment style. This work is closely related to ours, as we also test asynchronous black-boxes, and we also use a variant of an executable specification when we synchronize the property automaton with monitored services. However, their method doesn't infer a model for the tested component.

Another related method is [15] by Păsăreanu et al., which relies on environmental assumption generation when verifying a software component against a property. Their approach is based on the work of de Alfaro and Henzinger [11] stating that two components are compatible if there exists an environment that enables them to correctly work together. This divide-and-conquer technique analyzes components separately to obtain for each the weakest environment needed for the property to hold. By building a system controller, our approach also creates such an environment. However, while in [15] the analyzed component is well specified, our approach addresses systems that contain black-box services, whose behaviour and controllability must be understood before building an adaptor.

5 Conclusion

We presented a method to automatically compose a system with black-box asynchronous services. BASYL infers safe behavioural models for the black-box services by incrementally refining a tentative model. To learn only behaviour useful to the composition goal, the system is controlled by an intelligent adaptor, which guides the run towards the satisfaction of the system specification. The main contribution of our method is enabling a safe composition for systems with asynchronous black box services, by inferring safe approximations of the real service behaviour. Experiments performed show that BASYL can obtain models precise enough for controller synthesis, and can obtain them early in the exploration process.

Acknowledgments. We are grateful to Marius Minea and Ioana Bercea for consistent and useful feedback. This work was partially supported by the European FP7-ICT-2009-5 project no. 257876 SPaCIoS ("Secure Provision and Consumption in the Internet of Services"), by the Politehnica University of Timişoara and by the strategic grant POSDRU 6/1.5/S/13 (2008) of the Ministry of Labour, Family and Social Protection, Romania, co-financed by the European Social Fund: Investing in People.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inform. and Computation (1987)
2. Berardi, D., et al.: Automatic service composition via simulation. Int. J. of Foundations of Computer Science (2009)
3. Berg, T., Jonsson, B., Raffelt, H.: Regular Inference for State Machines with Parameters. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 107–121. Springer, Heidelberg (2006)
4. Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M.: Automatic Synthesis of Behaviour Protocols for Composable Web-Services. In: ESEC/FSE 2009 (2009)
5. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: `libalf`: The Automata Learning Framework. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 360–364. Springer, Heidelberg (2010)
6. Calvanese, D., et al.: Automatic Service Composition and Synthesis: the Roman Model. IEEE Data Eng. Bull. (2008)
7. Dallmeier, V., et al.: Generating Test Cases for Specification Mining. In: ISSTA 2010 (2010)
8. Dershowitz, N., Jayasimha, D.N., Park, S.: Bounded Fairness. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 304–317. Springer, Heidelberg (2004)
9. Ghezzi, C., Mocci, A., Monga, M.: Synthesizing Intentional Behavior Models by Graph Transformation. In: ICSE 2009 (2009)
10. Grabe, I., Kyas, M., Steffen, M., Torjusen, A.B.: Executable Interface Specifications for Testing Asynchronous Creol Components. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 324–339. Springer, Heidelberg (2010)
11. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC/FSE-9 2001 (2001)
12. Peled, D., Vardi, M.Y.: Black box checking. In: FORTE/PSTV 1999 (1999)
13. Cavallaro, L., et al.: Synthesizing adapters for conversational web-services from their WSDL interface. In: SEAMS 2010 (2010)
14. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic Generation of Software Behavioral Models. In: ICSE 2008 (2008)
15. Păsăreanu, C., Giannakopoulou, D., et al.: Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. In: FMSD 2008 (2008)
16. Holotescu, C.: Controlling the Unknown. In: FoVeOOS 2010 (2010)
17. Holotescu, C.: Black-Box Composition: a Dynamic Approach. In: SAVCBS 2010 (2010)
18. Marconi, A., et al.: Automated Composition of Web Services: the ASTRO Approach. IEEE Data Eng. Bull. (2008)
19. Åkesson, K., Fabian, M., et al.: Supremica: an integrated environment for verification, synthesis and simulation of discrete event systems. In: WODES 2006 (2006)
20. Raffelt, H., Steffen, B., Margaria, T.: Dynamic Testing Via Automata Learning. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 136–152. Springer, Heidelberg (2008)
21. Ramadge, P., Wonham, W.: The control of discrete event systems. Proc. of the IEEE 77(1) (1989)
22. Shahbaz, M.: Reverse Engineering Enhanced State Models of Black Box Software Components to support Integration Testing. Ph.D Thesis (2008)
23. Suman, R.R., et al.: Extracting State Models for Black-Box Software Components. J. Obj. Tech. (2010)
24. Tivoli, M.: An architectural approach to the automatic composition and adaptation of software components. Ph.D Thesis (2005)