Black-Box Composition: a Dynamic Approach

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

ABSTRACT

A core issue in component-based software engineering is represented by automatic component adaptation and composition. At behavioural level, an adaptor is an appropriate environment for two or more software components to correctly interoperate with respect to a certain desired behavioural property. This, however, requires for the behavioural interfaces of all components to be well-specified, which is not always the case in industrial practice. What happens if these components are black-boxes: incomplete specification, no provided models and no source code to extract interfaces from? How could we integrate them? Our approach interleaves online monitoring, verification-driven execution and model refinement in order to infer models of the black-box components, provide early access to a part of the system functionality whenever possible and synthesize permissive adaptors. We present both a centralized and a distributed technique, the last one directed towards the exploration and control of remote components.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Reliability, Verification, Design

Keywords

black-box components, model refinement, adaptation

1. INTRODUCTION

Important advances have been done in the area of automatic composition of software systems from off-the-shelf components. Since the seminal work of Yellin and Strom [23], formal specifications of component behaviour have been used to obtain adaptors able to ensure correct interoperability at behavioural level. Such an adaptor is a specific component-in-the-middle that coordinates the interactions in the system under the purpose of achieving desired functionality.

While early approaches, such as [21], focus on ensuring non-deadlocking interaction, later solutions like [2] or [5] result in adaptors that coordinate the system into satisfying desired temporal logic properties. Temporal properties and component behaviour are modelled by means of the same formalism (finite state machines, Büchi automata or labeled transition systems, etc.) and the composition aim is for the adapted system behaviour to simulate the specified goal.

Behavioural system adaptation can also be easily reduced to a control problem [18], if we see the adaptor as a controller over the system plant, enabling transitions by message forwarding and disabling them by message consumption.

However, formally specified component behaviour is still far from becoming a mainstream industrial practice. Therefore, situations when incompletely specified, black-box components have to be correctly assembled to create a larger system are not at all uncommon. As none of the classic adaptor synthesis solutions can be applied, building a correct and reliable system becomes a challenging task.

Our aim is to provide an automatic solution for the composition of software systems out of black-box components. We actively explore component behaviour at runtime by controlling its inputs and monitoring its outputs, while concomitantly building and refining a tentative model of it. Model refinement is performed with focus on the desired property, i.e., we are not interested in inferring component behaviour that is not relevant to the system specification.

In extension to our previous work [16], the behaviour exploration and refinement method presented here addresses both safety and liveness properties in infinite runs. Also, adding to the centralized exploration from [16], we propose a distributed solution for remote components. Early access to a part of the desired system functionality is provided by means of a restrictive adaptor that is obtained and incrementally improved during the behaviour exploration phase.

2. METHOD OVERVIEW

We want to compose a system out of a set of components for which a behavioural model has not been provided. We assume they are deterministic, asynchronous components, whose interactions are of the message send/receive type, by means of bounded buffers. The set of messages that a component can send or receive is assumed to be known, and both send and receive events are assumed observable. Also, all components can be reset anytime to an initial state.

The desired system has a temporal specification given as a Büchi automaton over a set of send/receive events. The goal must be enforced by a controller over the system plant.

Classic control theory [18] provides solutions to synthesize a controller over a plant using the controllable events of the plant to disable any uncontrollable events that might violate the specification. In the case of a plant consisting of a set of black-box components, controllable and uncontrollable behaviour cannot be precisely known. However, a

controller synthesized using an underapproximation of the real controllable and an overapproximation of the real uncontrollable behaviour of the plant, can control the plant.

In order to build a controller over the system plant, we need behavioural models for all components. As these components are black-boxes, their behaviour can only be acknowledged by runtime observation – thus underapproximated. Still, from a number of executions, a model can be built that would underapproximate only the controllable behaviour, while containing all its uncontrollable behaviour.

For such a model to be useful to the control problem, not only should observed executions cover as many controllable sequences of events as possible, but they also should be correct with respect to the desired system specification. Thus, we must generate executions that respect the specification, while covering as many controllable sequences as possible.

We aim to infer these models as Büchi automata, so that they will allow infinite accepting runs. Our black-box components are assumed deterministic, i.e., were they to have a precise model, no more than one transition from a specific state would be triggered by the same event.

In order to always overapproximate any unobserved uncontrollable behaviour, while also avoiding an expensive determinization step, we allow for nondeterminism in the models we infer. We say we enable a controllable transition whenever we forward to the component the message it needs to receive for the transition to be triggered. For each component, model inference takes place as follows. If absolutely no information is available on component behaviour, we start with a most general model (one state, all events triggering self-loops). Execution sequences observed at runtime are confronted to this model, and the model is refined: expected controllable events that cannot be enabled lead to state splits, observed events serve to prune out non-deterministic transitions whenever possible, equivalent states are merged.

However, due to model nondeterminism, a current internal component state might correspond to several model states. To differentiate between these states we need to observe execution sequences with a common prefix up to the disputed point, but containing transitions specific to only one possible state from that point on, i.e., differentiating transitions.

Thus, to infer models for the components in the system we need to observe execution sequences that either explore the controllability of correct system behaviour or serve to reduce a set of possible current states in a component model.

Our solution obtains these sequences at runtime. We place among the components a "fake" adaptor, that will intercept all sent messages and control the system by forwarding or consuming them, while monitoring the components to see if they accept forwarded messages. Observed events are used to synchronize the current execution with both a path in the system model and a path in the specification automaton.

We conduct the runtime experiments as follows: suppose the current state of a component corresponds in the model to one or more unexplored states q, having several outgoing controllable transitions. If any of these are differentiating transitions, the fake adaptor will enable one of them first, in order to narrow down the set of possible states.

The current state q in the model will always be synchronized with a current state s in the specification automaton.

Suppose a safety property. If for a controllable transition t from q in the model, a transition from s exists on the same event in the specification automaton, then t will be explored at runtime, as it conforms to the safety property. If several such transitions exist, they are explored in a random order.

Considering the safety property, a state q' is marked as forbidden when the specification was observed to be violated at runtime from q', and no controllable event can be disabled in order to prevent this. A state from which a forbidden state cannot be controllably avoided is also marked as forbidden.

If the specification automaton describes a liveness property, a depth-first search is initiated from the current state q, searching for traces of a maximum length b that violate the negation of the liveness property. Outgoing transitions t from q for which such a counterexample has been found will be explored at runtime to check counterexample feasibility.

While model-checking explores simultaneously all outgoing paths from a state, controlled execution can only explore one path at once. Let $T(q)$ is the set of either safe or differentiating transitions, yet unexplored and controllable from a state q. Thus, a transition from $T(q)$ is enabled each time an execution reaches q. Only safe transitions are reexplored.

Whenever an execution violates the specification or reaches a forbidden state by a trace that is not a differentiating trace, or whenever an infinite run is found, the ongoing execution is forced to end by resetting all components in the system.

We say a transition is confirmed when, from its state of origin, its triggering event is observed at runtime. The exploration process stops for a safety property when all controllable transitions in the system are either confirmed or marked as forbidden, and for a liveness property when all cycles conforming to its complement have been explored and no uncontrollable infinite execution has been found.

If a maximum number of states m is reached for all component models and there still is unexplored controllable behaviour in the system, this behaviour is removed, since controllable behaviour can be underapproximated. We then build the needed adaptor as a controller over the system plant. Also, during system exploration, correct and controllable traces found are used to generate a restrictive adaptor that enables the correct partial use of the system.

3. FORMAL DESCRIPTION

Consider a set $\mathcal{S} = \{C_0, C_1, ...C_{n-1}\}$ of n asynchronous black-box components from which the system S is to be composed. Desired system S must comply to a property expressed by means of a Büchi automaton Φ, including both the system specification and the absence of deadlock.

We decompose the specification Φ into a safety and a liveness property that are also expressed as Büchi automata. Φ will represent the synchronous product $\Phi = \Phi_s \| \Phi_l$.

In order for S to comply to the specification, it is necessary for the behaviour of S to be simulated by Φ_s: $S \preceq \Phi_s$.

Each black-box component C_i is associated to a tentative 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, $q_0^i \in Q^i$ the initial state , $Q_f^i = Q^i$ the set of accepting states, Σ^i is the event set of C_i, and $\Sigma = \bigcup \Sigma^i$ the event set of the system, and $\delta^i : Q^i \times \Sigma^i \rightarrow \mathcal{P}(Q^i)$ is the transition function. The event set of the system includes the event set of the property automaton Φ: $\Sigma \supseteq \Sigma^\Phi$. An event σ in Σ is either a message send: $msg!$, or a receive: $msg?$.

Let us 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: it is emitted by component C_i when a message msg arrives in a state q of C_i in which it can be

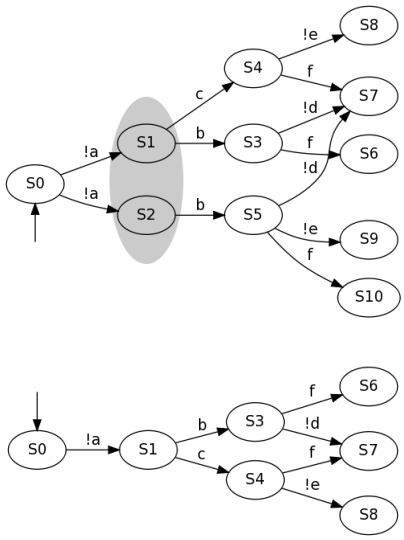

Figure 1: Model nondeterminism: (a) current set of states, (b) pruned model after $a!.c?$ is observed

accepted: $\delta^i(q, msg?) \neq \emptyset$. Event $rst?$ forces the component to return to its initial state, from any state q. This makes receive events externally observable and allows reset.

3.1 Model Refinement

Each black-box component C_i is considered deterministic, i.e., if its behaviour would be precisely modelled by an automaton, from any state q of this automaton, any event σ, controllable or not, can trigger at most one transition.

However, since we don't have this precise model, the behaviour of C_i is overapproximated by the tentative model U_i, i.e., U_i allows for all possible behaviours of C_i.

If we have no information on the behaviour of C_i, we initialize U_i with the most general model. Its transition function is $\delta^i\left(q_0^i, \sigma\right) = \{q_0^i\}$ for all $\sigma in \Sigma^i$. Model U_i represents an overapproximation of the real component C_i behaviour. The transition function δ^i is a set function, thus it allows for nondeterminism, in order to include all possible transitions.

During model refinement, nondeterminism can be introduced whenever a state is split and we don't know how to distribute its incoming transitions among resulted states. This nondeterminism must be resolved by runtime observations, which confront the model U_i with the real behaviour of C_i. We chose this option over the determinization of U_i, which would have exponentially increased the number of states.

Tentative models U_i are refined based on runtime observations, that will prove traces feasible or unfeasible for C_i.

3.1.1 Refinement cases

A transition triggered by an event σ_k from state q can lead the system into any state $q' \in \delta^i(q, \sigma_k)$ (Figure 1). Let $\Sigma^i(q')$ be the set of outgoing events from such a state q'.

- After σ_k, an event σ_{k+1} occurs. We consider event σ_{k+1} as confirmed from any state q' that would correctly correspond to the internal state of the real component C_i. Thus, all transitions $\delta^i(q, \sigma_k)$ to a state q'' for which $\delta^i(q'', \sigma_{k+1}) = \emptyset$ are removed from U_i, as component C_i is deterministic.

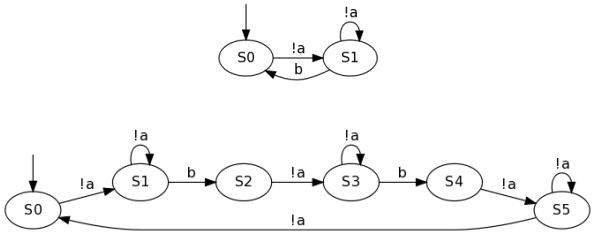

Figure 2: Cycle transformation: (a) before, (b) after split

- After σ_k, a receive event σ_{k+1} is enabled by forwarding the required message, but the message is not accepted.
  - if $\sigma_{k+1} \in \Sigma^i(q')$ for a unique state $q' \in \delta^i(q, \sigma_k)$, then q' becomes a split candidate. The exploration continues with events σ s.t. $\sigma \in \Sigma^i(q'')$ for some $q'' \in \delta^i(q, \sigma_k)$ and $\sigma \notin \Sigma^i(q')$. Only if none of these events σ can be confirmed, q' is split.
  - if $\sigma_{k+1} \in \Sigma^i(q')$, $\forall q' \in Q'$, where $Q' \subseteq \delta^i(q, \sigma_k)$, we mark the states in Q' as split candidates. The exploration continues with receive events that cannot be enabled from any $q' \in Q'$. If any of these is confirmed, the split is canceled. Else, the exploration continues with events $\sigma \in \Sigma^i(q')$ for states $q' \in Q''$, where $Q'' \subset Q'$ and $|Q''|$ is minimal.

Whenever a set of states Q_j end up having similar outgoing transitions, i.e., $\Sigma^i(q_{ja}) = \Sigma^i(q_{jb})$, $\forall ja \neq jb$ and $\delta^i(q_{ja}, \sigma) = \delta^i(q_{jb}, \sigma), \forall \sigma \in \Sigma^i(q_{ja})$, $\forall q_{ja}, q_{jb} \in Q_j$, the states in Q_j are considered equivalent and merged.

3.1.2 State split

A state split can be performed only on a single state. If we have a set of split candidates, we first narrow it down by experimenting with differentiating traces until we have either only one state, or a set of equivalent states to merge.

Consider a trace $\rho = q_0^i \dots q$ in the model U_i, where event σ is not accepted in the last state q although allowed in the model, $\sigma \in \Sigma^i(q)$. Since the last state accepts fewer events than q, it needs to be refined. Let $n' \geq 0$ be the number of transitions $q \xrightarrow{\sigma} \dots$ occurring in ρ from state q. The refinement of U_i must count to n': $\langle q, 0\rangle$, $\langle q, 1\rangle, \dots, \langle q, n'\rangle$, with σ enabled in every state but the last (see Figure 2).

Thus, the refinement of U_i has state space $(Q^i \times \{0\,..\,n'\}) \cup \{\langle q, n'\rangle\} \cup Q^i$, its initial state is $\langle q_0^i, 0\rangle$, and the transition relation δ' is defined by:

- $\delta'(\langle q, k\rangle, \sigma) = \{\langle q'', k+1\rangle \mid q'' \in \delta(q, \sigma)\}$ for $k \leq n'-1$ (transition and count on σ from q),
- $\delta'(\langle q', k\rangle, \sigma') = \{\langle q'', k\rangle \mid q'' \in \delta(q', \sigma')\}$ if $q' \neq q$ or $\sigma' \neq \sigma$ (else keep behavior and counter),
- $\delta'(\langle q, n'\rangle, \sigma') = \delta(q, \sigma')$ for $\sigma' \neq \sigma$ (switch to old state space)
- $\delta'(q', \sigma') = \delta(q', \sigma')$ for all $q' \in Q^i$, $\sigma' \in \Sigma^i$ (transitions of old state space).

A special case appears when from the initial state $q_0^i \in Q^i$, a receive event σ is enabled and the message fails to be accepted. Here, q_0^i splits into $\langle q_0^i, 0\rangle$ and q_0^i, the new initial state becomes $\langle q_0^i, 0\rangle$ and transitions modify as following:

- $\delta'(\langle q_0^i, 0\rangle, \sigma') = \delta(q_0^i, \sigma') \cup \{q_0^i\}$ for $\sigma' \neq \sigma$
- $\delta'(q', \sigma') = \delta(q', \sigma')$, for all $q' \in Q^i$, $\sigma' \in \Sigma^i$

The refinement of a component model U_i stops when the behaviour exploration is complete or a maximal state number m is reached and no state merging is possible.

Our model refinement phase is related to the counterexample guided abstraction refinement (CEGAR) developed by Clarke et al. [6], since the blocking execution can be regarded as a spurious counterexample, however, here the unfeasibility of the counterexample cannot obtained from an accessible concrete model, but only explored at runtime.

3.2 Centralized Exploration

The desired property Φ can be decomposed into safety and liveness properties: $\Phi = \Phi_s \| \Phi_l$. We first explore the system with respect to the safety property, and only afterwards, within the confirmed safe behaviour, we analyze the satisfaction of the liveness property.

3.2.1 Safety Properties

Let $U_\times$ be the asynchronous product of the tentative models $U_\times = U_0 \times U_1 \times ... \times U_{n-1}$. As defined in [16], 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: $|\Sigma^i_?(q_{cp})| \geq 2$.

We put the real components together and start the system execution with a fake adaptor in the middle, that intercepts component messages and forwards or consumes them in order to control the execution, while also monitoring the component reactions. The property automaton Φ_s will be executed synchronously with the system model $U_\times$.

If, from a state q, an uncontrollable event σ' occurs at runtime, for which no transition triggered by σ' exists in Φ_s from the current state, then q is marked as a forbidden state. The same happens when for all controllable transitions from q, no corresponding transition exists in Φ_s or all controllable transitions from q end up in 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 by the fake adaptor if from the current state of Φ_s a transition triggered by σ exists, and $\delta(q_{cp}, \sigma)$ does not contain forbidden states.

Whenever the runtime observations report one of the described model refinement cases, the component model U_i for which the observation was reported will be 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 cycle does not violate the safety property
  - if the execution is steered towards the cycle, it doesn't leave it after a number of θ transitions.

The mentioned transition threshold $\theta = m \cdot n$, where m is the maximum number of states admitted for a component, and n is the total number of components in the system, θ representing the maximum number of states in the system.

We base this result upon the consideration made by Peled in [12], that for a black-box component with a maximum number m of states, an accepted trace of the form uv^m is enough to prove the existence of an infinite run.

Below we give a pseudocode description of the behaviour exploration algorithm for safety properties.

```
1: def fixpoint =
2: ∀t. t.msg ∈ Σ? → confirmed(t) ∨ t.next ∈ Bad
3:
4: function good(Ui, i, t) [confirmed transitions]
5: confirm(t)
6: prune(Ui,t, last(i)) [prunes out nondeterminism]
7: last(i) := t
8: if cycle.in then
9:    cycle := true
10:   transitions := 0 [reset counter when new cycle]
11: else if cycle then
12:   if cycle.out then
13:      cycle:=false
14:   inc(transitions) [only count when in a cycle]
15: return Ui
```

In **fixpoint** we define the ideal terminating condition for behavioural exploration: all controllable transitions in the system model are either confirmed, i.e., have been observed at execution, or found to lead towards forbidden states.

Function **good** is applied when an event is observed in a component C_i: it marks transition t as confirmed, prunes out nondeterminism, and memorizes t in $last(i)$. Transitions inside a cycle are counted, and when their number reaches $m{\cdot}n$ an infinite execution has been found. For simplification, we have left out the case of nested cycles, where each distinct cycle has its own counter.

The procedure **execution** conducts the experiments during one execution. Incoming messages are read in the buffer, dropping old messages whenever the buffer is full. The current system state is marked as *Bad*, i.e., forbidden, if the transition t triggered by an observed event leads to a property violation (model U_i and specification Φ do not synchronize on t) or if t leads to a forbidden state. If from the current system state q, a controllable outgoing transition t is unexplored and conforms to Φ, it is enabled, i.e., the requested message is forwarded to the component. If the message acceptance is acknowledged, the transition is confirmed, else the model is refined by state splitting. If all transitions from q end in a forbidden state, then q is forbidden. If, however, all controllable transitions from q end in a forbidden state, and $\Sigma(q) \neq \Sigma_?(q)$, then at least one such transition t has to be confirmed by runtime execution in order to mark q as forbidden. An already confirmed transition is only explored if it conforms to the specification.

The execution ends when $m \cdot n$ transitions have been counted inside a cycle, or a forbidden state was reached. The exploration process finishes when either the **fixpoint** condition is true, or all component models have reached m states. In the last case, unexplored controllable transitions are removed from component models. If a controller can be obtained for this underapproximation of the controllable behaviour, it will ensure correct interoperability in the system.

3.2.2 Liveness Properties

Let $\overline{\Phi_l}$ be the complement automaton of Φ_l. If Φ_l expresses a liveness property, $\overline{\Phi_l}$ will express a safety property.

The behaviour exploration algorithm for the liveness specification is similar with the one given for the safety specification, with two significant differences: the negated specifi-

```
1: procedure execution [directing one execution]
2: buffer := ∅ [initialize]
3: cycle:=false
4: transitions:=0
5: for all i = 0, n − 1 do
6:   last(i) := none [no last transition]
7:   reset C_i
8: repeat
9:   for all i = 0, n − 1 do
10:    if incoming(C_i) ≠ ∅ ∧ (buffer.size ≤ l − 1) then
11:      t:= incoming(C_i) [accepts sent messages]
12:      buffer := buffer ∪ {t.msg}
13:      U_i:=good(U_i, i, t) [confirm send]
14:      if buffer.size = l then
15:        buffer := buffer \ buffer.old [full buffer]
16:      if sync(U_i, Φ, t)=Err ∨ sys.state.next ∈ Bad
         then
17:        Bad:= Bad ∪ sys.state.current [found bad state]
18:        stop
19:
20:  for all i = 0, n − 1 do
21:    q := sys.state.current
22:    Ctrl(i):= Σ^i_?(q) [explore controllable events]
23:    for all t ∈ Ctrl(i) ∧ t.msg ∈ buffer do
24:      if not(explored(t)) then
25:        if sync(U_i, Φ,t)=Ok ∧ sys.state.next ∉ Bad
           then
26:          enable(t) [try a safe transition]
27:          if ack then
28:            U_i:= good(U_i, i, t)
29:          else {enable fails, refine model}
30:            U_i:=split(q, t, U_i)
31:        else if Σ(q) \ Ctrl(i) = ∅ then
32:          Bad:= Bad ∪ {q} [bad state]
33:          break
34:        else if Σ_?(q) \ Ctrl(i) = ∅ then
35:          enable(t)
36:          if ack then
37:            U_i:=good(i, t)
38:            Bad:= Bad ∪ {q} [bad state]
39:            stop
40:          else {enable fails}
41:            U_i := split(q, t, U_i)
42:      else if sync(U_i, Φ,t)=Ok ∧ sys.state.next ∉ Bad
         then
43:        enable(t)
44:        if ack then
45:          U_i:=good(i, t)
46:        else
47:          U_i := split(q, t, U_i)
48: until transitions ≥ θ [infinite cycle found]
49: return
```

```
1: [behaviour exploration main]
2: for all i = 0, n − 1 do
3:   initialize(U_i)
4: Bad :=∅
5: repeat
6:   execution
7: until fixpoint ∨ (∀i ≤ n − 1). |Q_i| ≥ m
```

cation, $\overline{\Phi_l}$, is used, and the decision on the transitions to be explored from a control point is taken differently.

When the execution reaches a control point q_{cp}, a depth-first search phase will return the outgoing transitions from which controllable traces that violate $\overline{\Phi_l}$ can be found in the system model for a length of b transitions, where b is the maximum depth bound.

Only the outgoing transitions on traces that violate $\overline{\Phi_l}$ will be enabled for execution. An execution ends when:

- $\overline{\Phi_l}$ is violated and if the execution reaches a cycle that complies to $\overline{\Phi_l}$, it can also be controllably steered away after less than θ transitions
- if a cycle is reached where $\overline{\Phi_l}$ cannot be violated in θ transitions, we reset the components and conclude that the system S does not satisfy Φ_l, unless entering the cycle can be controllably avoided.

If the safety property is trivial, i.e., the model hasn't been refined before, exploration stops when all controllable transitions are either confirmed, or do not violate property $\overline{\Phi_l}$.

If, however, the model has already been refined by runtime exploration with respect to the safety property Φ_s, the exploration process stops when either all cycles that conform to $\overline{\Phi_l}$ have been explored and controllably exited after less than θ transitions, or one such cycle is entered by means of an uncontrollable transition and proved to lead to infinite executions in $\overline{\Phi_l}$.

3.3 Distributed Exploration

3.3.1 Motivation

Our system S contains n components, each with a maximum number of m states. During one directed execution, $2 \cdot n_t$ messages are exchanged between the components and the adaptor, where n_t is the number of executed transitions, due to the fact that each message is sent and received twice (by the component and by the adaptor). During cycle exploration, the number n_t of transitions can reach a maximum of $\theta = m \cdot n$, resulting in $O(m \cdot n)$ exchanged messages.

If our components are placed remotely, the procedure of monitoring and controlling an execution will involve $O(m \cdot n)$ remote message exchanges, all taking place through the centralized controller. Also, remote communication is prone to issues such as message loss, large delays, etc., which can make remote behaviour exploration very inefficient if centralized. Thus, the solution should be localized.

3.3.2 Overview

Let us now assume that we have local fake adaptors, one for each remote component, and that each local fake adaptor broadcasts sent messages over the network, to the other adaptors, in a non-redundant way, i.e., won't broadcast the same message twice. Once a message is received, it can be reused many times, as the adaptor will forward it locally to the component whenever needed, at no communication cost. Thus, assuming all messages are broadcast to all components, the number of remotely exchanged messages during the whole behaviour exploration will be $O(n \cdot |\Sigma_!|)$.

Assume that all n components in the system S are remote. This results in n local fake adaptors FA_i, $i = \overline{0, n-1}$. Each FA_i monitors and controls only the local component C_i, and communicates with the other adaptors by broadcasting the

messages sent by C_i and receiving messages sent by the other components C_j, $j \neq i$.

When it receives a message msg, FA_i will only keep it if $msg? \in \Sigma^i_?$. Also, as soon as component C_i sends a message msg, FA_i intercepts it, but, in order to save bandwidth, it will only broadcast it to all FA_j, iff msg hasn't been broadcast yet during the exploration process.

The behaviour exploration and refinement are performed by the local adaptor on the monitored and controlled component similarly to the centralized case.

However, an important difference between the centralized and the distributed approach resides in the way components interact during the exploration process: if in the centralized approach the real components are explored together, in the distributed case each component is explored individually, together with only the tentative models of the other components. While allowing for parallelism in the exploration process, this method has the disadvantage that, as the tentative models overapproximate the real behaviour of the components, the local component might be explored against execution scenarios that are actually infeasible for the real system. In order to minimize this issue, the local copies of the tentative models should be up to date.

Whenever the local model U_i of component C_i is refined, the refinement increment is broadcast, so that all FA_j can update the local copy of U_i, and use it to refine U_j.

The parallel exploration of component behaviour ends when each local component has a model in which all controllable transitions are either confirmed, or leading to forbidden states, and all the transitions from the local copies of other component models, that are part of at least one correct local executions, are also confirmed transitions.

3.4 Adaptor synthesis

The final aim of the technique is to correctly compose the system S from the components $C_i, i \leq n-1$. For the adaptor synthesis phase, the composed property expressed by the Büchi automaton Φ will be considered.

Consider a refined model of a component C_i to be U'_i. Also, let $U'_\times$ be the asynchronous product of these refined models: $U'_\times = U'_0 \times U'_1 \times ... \times U'_{n-1}$.

3.4.1 Early adaptation

As mentioned in our previous work, having obtained during the behaviour exploration phase a set T_{cctrl} of correct and controllable traces, we can easily merge these traces into an automaton C. We then obtain adaptor A by mirroring C, i.e., transform all its send transitions into receive ones and conversely: $\Sigma^A = \{msg!|msg? \in \Sigma^C\} \cup \{msg?|msg! \in \Sigma^C\}$.

We can compute A only using executions obtained during the exploration of the liveness property, after the safety property exploration has completed. The adaptor A will be incrementally enriched when new correct and controllable traces are discovered. Thus, we can provide an early, safe access to a part of the desired system functionality.

3.4.2 Final adaptation

After both the behaviour exploration and model refinement end, a permissive adaptor can be computed starting from a controller over the plant $U'_\times$.

However, models U'_i might still contain non-deterministic transitions triggered by uncontrollable events σ, from states q where σ was neither observed at execution, nor eliminated from $\Sigma^i(q)$. In our previous work [16], we proposed three ways to resolve this situation, and build a final model U''.

- **Optimistic approach**: If we consider components C_i to be fair, then, for all states q, all transitions triggered by uncontrollable events not observed at execution are considered not to exist and can be removed from U'_i. We obtain a deterministic refined model, that leads to a small controller.

- **Pessimistic approach**: We know C_i as unfair. Therefore, its final model will be U'_i, unchanged, thereby the nondeterminism due to uncontrollable events unobserved at execution remains. The controller synthesis problem is harder for nondeterministic models, but the obtained controller is safe.

- **Semi-optimistic approach**: We have no information regarding the fairness of C_i, but we believe it is fair. For any state q, all uncontrollable events $\sigma \in \Sigma^i_!(q)$ not observed during the verification-driven execution phase, are assumed unobservable, and only the self-loops triggered by such events σ are kept, all other transitions being removed. We obtain a deterministic refined model U''_i, that still accepts all $\sigma \in \Sigma^i_!(q)$. Computing the resulting controller, while much easier than in the case of the pessimistic one, is still more difficult than for the optimistic controller. Further risks may still appear if some such σ do manifest and the actual transition from q is not a self-loop.

The computation of the controller, noted 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 [18], is a fixpoint procedure. Thus, all behaviours of the plant $U'_\times$ that do not violate Φ. are allowed.

A permissive adaptor A is obtained from the controller $Ctrl$ by mirroring its event set. It allows the user a safe, correct and complete access to the system S functionality.

4. DISCUSSION

The main challenge when it comes to black-box component integration is to make behaviour exploration feasible for real-world software components. Executing the components is vital to obtain information on their behaviour in the absence of a precise model. However, the cost of exploring behaviour by execution is greater than the cost of model checking, since paths cannot be explored simultaneously. When exploring behaviour at runtime, we cannot simply backtrack to the previous control point in order to try another path. Instead, we have to reset the components each time, and restart the run from the initial state. To reach again the original control point is difficult, due to the uncontrollability in the system, which can take the execution astray. This is very expensive for deep paths.

However, this is the unavoidable cost of exploring asynchronous black-box components at runtime. If all states are control points, to obtain a trace of length $m \cdot n$ – required to determine an infinite run – a number of $O(|\Sigma|^{mn})$ decisions are to be taken, which implies $O(|\Sigma|^{mn})$ actual executions for an exhaustive, controllable exploration, with no lookahead. 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 problem of trace controllability.

Generating a controller based on an inaccurate, approximated model is useless and unsafe, if the approximated model contains spurious controllable transitions, or lacks uncontrollable transitions that actually occur at runtime. We need to explore the runtime behaviour of the components in order to be able to control the system, because the controllable behaviour of the plant is of no use if it is not confirmed.

In order to perform an efficient exploration, we need to reduce the number of necessary executions as much as possible. This is why, when choosing a transition to enable from several possible, we make use of a bounded model-checking based step, which lets us select for exploration only the paths that conform to the specification up to a bound b. If confirmed, the controllable transitions on these paths are the ones to be used by the controller, later on, to enforce the desired behaviour on the plant. This lookahead step has an exponential complexity: $O(|\Sigma|^b)$, but the bound b can be chosen as convenient, and a directed execution has the advantage of always inferring useful information.

However, in order to infer models useful for system composition, we do not have to exhaustively explore the components, but only reach a fixpoint in the runtime exploration and model-refinement interleaving, where all controllable behaviour, correct with respect to the specification, is either confirmed, or forbidden. An upper bound on the number of traces to be experimented with, considering n_{cp} the number of control points in an hypothetical, precise model of the system, is $|\Sigma_?^\Phi|^{n_{cp}}$, which, considering also the lookahead step, leads to a complexity $O(n_{cp}|\Sigma|^b \cdot |\Sigma_?^\Phi|^{n_{cp}})$. If, worst case, $n_{cp} = m$, the complexity becomes $O(m|\Sigma|^b \cdot |\Sigma_?^\Phi|^m)$.

The total number of runs needed to observe these traces, however, depends on their controllability. Still, even if we fail to reach the fixpoint before reaching the maximum number of states for component models, we can use only the confirmed controllable behaviour to generate a controller.

5. RELATED WORK

A closely related approach is the black-box checking technique developed by Peled et al. for programs with missing or inaccurate models [12]. It employs the Angluin algorithm [1] for model inference. 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 model only has input events, which highly simplifies the learning process, and, since the aim is to find feasible counterexamples, their model 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, verification, testing and model refinement are strongly interleaved, which allows us to obtain a good model earlier, while their work has distinct 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. [10], 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 regular inference method of Berg et al. uses an adaptation of the Angluin algorithm [1], thus assuming the possibility of 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 component behaviour, instead of querying for equivalence. Also, our samples are not random traces, but are generated using a lookahead technique that aims towards a compositionality goal.

In [20], 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. These potential states can be merged or confirmed, existing cycles can be detected within a certain bound, etc. Somehow similar, 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. The execution space is extended by adding/removing method calls from the current state, and thus enriching an existing test suite. Also, in [8], Ghezzi et al. present the SPY approach, which is able to infer a behavioural model of black-box components that embody data abstractions, by using both invariants computed on input/output data relations and graph transformations, while also assumming a "uniform" component behaviour. These techniques work with synchronous method calls, while our approach addresses asynchronous message exchange, which is more difficult as it involves issues such as uncontrollable events, etc. Also, our technique is composition-oriented and works on discovering specific behaviour required by the desired global property, thus, the model is refined not in isolation, but with respect to the property and to its possible interactions with the other components in the system.

In the domain of conversational web services, we found the work of Bertolino et al. [3] on behaviour protocol synthesis, and Cavallaro et al. [13] on service adaptation to be related to ours. While [3] describes a technique that uses the WSDL interface to extract possible operation dependencies between the I/O data, and then validates these dependencies through testing, thus obtaining the behavioural web service protocol, the work in [13] takes this technique further, and develops a method for synthesizing web service adaptors starting from WSDL descriptions, which enables a correct interoperation when an old service is replaced by an equivalent, new one. However, the web services in [3, 13] are stateless, while our approach addresses stateful black-box components. Thus, while their work starts from a data-dependency perspective, ours explores the language of the component and its controllability, while regarding data from a higher level of abstraction, as passed and received messages. Therefore, we consider the two approaches complementary.

Another related method is [15] by Păsăreanu et al., which relies on environmental assumption generation when verifying a software component against a desired 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 in order to obtain for each the weakest environmental assumption needed for the property to hold.

By building a system controller, our approach actually creates such an environment. However, while in [15] the analyzed component is well specified, our approach addresses systems with black-box components, whose behaviour and controllability must be understood before building an adaptor. In [19], the authors use both underapproximation and overapproximation to learn component interfaces, but with respect to predicate abstraction, and in a white-box manner.

The use of verification-driven execution relates our method to smart play-out [9], which 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. Both approaches use verification-driven execution to improve knowledge, but, while we infer existing, unknown behaviour, smart play-out experiments with execution scenarios to find the best design options.

CrystalBall [22] also makes use of model checking for lookahead analysis. Here, nodes in a distributed system run continuously a state exploration algorithm on a snapshot of their neighbourhood in order to detect future inconsistencies. If an error is predicted, the ongoing execution can also be steered away from it. However, in contrast to our approach, CrystalBall addresses well-specified distributed systems. Also, while CrystalBall uses the lookahead and execution steering in a defensive way, our method employs similar techniques aggressively, for runtime exploration.

6. CONCLUSIONS

We have presented a method to automatically compose a system from a set of black-box components. Our technique infers behavioural models for the black-box components by starting from an initial abstraction that is repeatedly confronted with the runtime behaviour of the component and incrementally refined. In order to refine the useful behaviour of the model, the execution of the system is controlled by an intelligent adaptor, which employs a bounded model-checking [4] based lookahead technique to guide the run towards the satisfaction of the system specification.

Our method does not depend on source code availability, but can be combined with static component interface extraction, which is known to produce overapproximated models, in order to appropriately refine these models by dynamic behavioural exploration, and thus provide more precise interfaces. Also, it can be applied to improve the knowledge on maintenance components for which no behaviour model has been provided, or the provided models are outdated.

The main contributions of this paper are the behaviour exploration methods for infinite runs, addressing safety and liveness properties, and inferring models for system composition. Also, this paper makes preliminary considerations on a distributed exploration of remote black-box components – an idea we plan to further develop in our future work.

We are currently working on the implementation of this technique. We will validate our prototype on a set of Enterprise JavaBeans systems using Java Message Service for asynchronous messaging. For initial experiments, we have used the Supremica tool [17] for controller synthesis.

Acknowledgements

We are grateful to Marius Minea for substantial feedback. This work was partially supported by the European FP7-ICT-2007-1 project 216471, AVANTSSAR 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.

7. REFERENCES

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Inform. and Computation* 1987.

[2] M. Autili, L. Mostarda, A. Navarra, and M. Tivoli. Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems. *Journal of Systems and Software* 2008.

[3] A. Bertolino, P. Inverardi, P. Pelliccione, M. Tivoli Automatic Synthesis of Behaviour Protocols for Composable Web-Services. *ESEC/FSE* 2009.

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

[5] C. Canal et al. Model-based adaptation of behavioral mismatching components. *Trans. Soft. Eng.* 2008

[6] E. Clarke et al. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 2003.

[7] V. Dallmeier and A. Zeller. Generating Test Cases for Specification Mining. *ISSTA* 2010

[8] G. Ghezzi et al. Synthesizing Intentional Behavior Models by Graph Transformation. *ICSE* 2009.

[9] D. Harel et al. Smart play-out. *OOPSLA* 2003

[10] T. Berg et al. Regular Inference for State Machines Using Domains with Equality Tests. *FASE* 2008

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

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

[13] L. Cavallaro, E. Di Nitto, P. Pelliccione et al. Synthesizing adapters for conversational web-services from their WSDL interface. *SEAMS* 2010

[14] D. Lorenzoli, L. Mariani, M. Pezzè. Automatic Generation of Software Behavioral Models. *ICSE* 2008

[15] C. Păsăreanu, D. Giannakopoulou et al. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *FMSD* 2008.

[16] C. Holotescu. Controlling the Unknown. *FoVeOOS* 2010

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

[18] P. Ramadge and W. Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77(1), 1989.

[19] R. Singh, D. Giannakopoulou, and C. Păsăreanu. Learning Component Interfaces with May and Must Abstractions. *CAV* 2010.

[20] R. Suman et al. Extracting State Models for Black-Box Software Components. *J. Obj. Tech.* 2010

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

[22] M. Yabandeh, N. Knežević, D. Kostić, V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. *NSDI* 2009

[23] D. Yellin, R. Strom. Protocol specifications and component adaptors *Tr. on Prog. Lang. and Syst.* 1997