Modular System Verification by Inference, Testing and Reachability Analysis

Roland Groz[1], Keqin Li[2], Alexandre Petrenko[3], and Muzammil Shahbaz[4]

[1] LIG, Grenoble Computer Science Lab, France
Roland.Groz@imag.fr
[2] SAP Research, France
Keqin.Li@sap.com
[3] CRIM, Canada
Alexandre.Petrenko@crim.ca
[4] France Télécom R&D/Orange Labs, France
Muhammad.MuzammilShahbaz@orange-ftgroup.com

Abstract. Verification of a modular system composed of communicating components is a difficult problem, especially when the models of the components are not available. Conventional testing techniques are not efficient in detecting erroneous interactions of components because such interactions often occur as interleavings of events that are difficult to reproduce in a modular system. The problem of detecting intermittent errors in the absence of models of components is addressed in this paper. A method to infer a controllable approximation of components through testing is elaborated. The inferred finite state models of components are used to detect intermittent errors and other compositional problems in the system through reachability analysis. The models are refined at each analysis step thus making the approach iterative.

1 Introduction

Integration of components is now a major mode of software development. Very often, components coming from outside sources (such as COTS) have to be connected to build a system. In most cases, the components do not come with a formal model, just with executable or in some cases source code. At the same time, the interaction of components may lead to integration bugs that may be hard to find and trace, especially in the absence of any model or other development information. In this paper, we are targeting compositional problems in the behaviours of a system composed of communicating components. Specifically, we aim at identifying intermittent (sporadic) errors occurring in event interleavings that are difficult to reproduce in an integrated system. Generally speaking, the system may eventually produce several event interleavings in response to a given external input sequence, if that sequence is applied several times. This occurs in particular when the execution order of the components changes over time from one experiment to another, typically because of different scheduling, varying load and communication jitters. However, it is unrealistic to expect to be able to enforce all the interleavings during testing. Therefore, the intermittent errors are hard to elicit and to reproduce in functional testing.

K. Suzuki et al. (Eds.): TestCom/FATES 2008, LNCS 5047, pp. 216–233, 2008.
© Springer-Verlag Berlin Heidelberg 2008

On the other hand, all potential interleavings can be checked for potential errors by instrumenting the modular system and executing it in a controlled environment to observe all the possible executions and interactions of all its components, see, e.g., Verisoft [6]. For components without source code, this approach may not be applicable and a model-based approach can be attempted. Indeed, if component models are available, the global model state space can be exhaustively searched (reachability analysis) to look for compositional problems, using, for instance, a model checker. As stated earlier, the models usually do not exist. One possibility would be to reverse engineer them from the code, but reconstruction based on static analysis has a number of limitations. The main alternative is to infer them from executions. However, given the complexity of typical software components, it is unrealistic to assume that components could be modelled with a perfect abstraction in a finite, compact representation. Inferring approximated models of components in a given modular system appears to be more realistic.

In this paper, we are developing an approach to verify a modular system by inferring tunable approximated models of its components through testing and performing reachability analysis to detect intermittent errors and other compositional problems in the system. This approach possesses two main advantages regarding the models that are inferred. First, it allows derivation of models of the components describing the behaviours that can actually be exhibited in the integrated system. Typically, components bundle a number of functions, but it is often the case that only a subset of those functions are used in the system, so inferring them in isolation is hard if not impossible; whereas our approach delivers models which omit behaviours unused in the integrated system. Second, the models are determined with a controllable precision to balance between the level of abstraction and the amount of efforts needed to obtain the models.

We make the following assumptions about a given system:

- The system interacts with a slow environment which submits external inputs only when the system stabilizes.
- Components are black boxes that behave as finite state (Mealy) machines and interact asynchronously. In response to an input, a component can produce several outputs to other components or the environment of the system.
- Each component is deterministic; however, due to possible jitter in communication delays, or scheduling of components, the system might not be.
- The external input actions of the system are known.
- Every output action of the components can be observed in testing.

No additional information about the system, such as the number of states, a priori given positive or negative samples of its behavior or teacher [9], often used in traditional model learning, is available for model inference. The components are modelled using Finite State Machine (FSM) with multiple outputs, or equivalently Input Output Transition System (IOTS) (with restrictions). A modular system is composed of IOTS components that communicate asynchronously through queues modelled by IOTS. We define an approximation of an FSM, called k-quotient, for any given positive integer k, based on state distinguishability. The precision of this approximation can be controlled by the parameter k, which is important, since

identification of a state machine is in general infeasible without knowing the number of its states.

The first contribution of this paper is an algorithm that computes a k-quotient for a component (thus, for a whole system) by testing in the two following steps: behavior exploration bounded by the parameter k and "folding" of the observed behavior by state merging using trace inclusion relation. We then elaborate an approach to infer a k-quotient of a modular system. The k-quotient of the modular system with observable internal actions in the form of an IOTS is used to infer initial models of components by projecting the quotient. Reachability analysis of the models is next performed to identify witness (diagnostic) traces of a composition problem, such as unspecified receptions, livelocks, and races. The identified witness needs to be tested in the real system, to check whether it is an artifact coming from our approximations or the system indeed has this problem. However, since we cannot control the delays occurring in the integrated system, each component is tested in isolation on a projection of the witness trace. A witness refuted by testing the components yields new observations. The models are then refined using new observations and the process iterates until the obtained models are well-formed. The obtained component models are consistent with all observations made on the real system. Once composed, they are at least as accurate as the k-quotient model of the system. At the same time, the obtained models are well-formed, i.e., have no compositional problems that may otherwise occur with sequences of at least k external inputs to the system.

The paper is organized as follows. Section 2 defines a k-quotient of an FSM and presents an algorithm for its inference. Section 3 restates the results of Section 2 for Input Output Transition Systems. Inference of modular systems is elaborated in Section 4. The notion of a slow asynchronous product is defined to formalize the interactions of components in a slow environment and several known compositional problems along with the witness traces are formally defined. The approach is illustrated on a small example. Section 5 discusses the related work. Section 6 concludes the paper.

2 Inferring Finite State Machine

2.1 Basic Definitions

In this section, some basic definitions about finite state machines with multiple outputs are given.

A *Finite State Machine with multiple outputs* (FSM) A is a 6-tuple (S, s_0, I, O, E, h), where

- S is a finite set of states with the initial state s_0;
- I and O are finite non-empty disjoint sets of inputs and outputs, respectively;
- E is a finite set of finite sequences of outputs in O (may include the empty sequence ε);
- h is a behavior function $h: S \times I \to 2^{S \times E}$, where $2^{S \times E}$ is the powerset of $S \times E$.

FSM $A = (S, s_0, I, O, E, h)$ is

- *completely specified* (a complete FSM) if $h(s, a) \neq \emptyset$ for all $(s, a) \in S \times I$;

- *partially specified* (a partial FSM) if $h(s, a) = \varnothing$ for some $(s, a) \in S \times I$;
- *deterministic* if $|h(s, a)| \leq 1$ for all $(s, a) \in S \times I$;
- *nondeterministic* if $|h(s, a)| > 1$ for some $(s, a) \in S \times I$;
- *observable* if the automaton $A_\times = (S, s_0, I \times E, \delta)$, where $\delta(s, a\beta) \ni s'$ iff $(s', \beta) \in h(s, a)$, is deterministic.

In this paper, we consider only observable machines; one could use a standard procedure for automata determinization to transform a given FSM into an observable one. We use a, b, c for input symbols, α, β, γ for input (or output) sequences, s, t, p, q for states, and u, v, w for traces.

In FSM $A = (S, s_0, I, O, E, h)$, $(s, a\beta, t)$ is a *transition* if $s, t \in S$ and $(t, \beta) \in h(s, a)$. A *path* from state s_1 to s_{n+1} is a sequence of transitions $(s_1, a_1\beta_1, s_2)(s_2, a_2\beta_2, s_3)\dots(s_n, a_n\beta_n, s_{n+1})$ s.t. $(s_{i+1}, \beta_i) \in h(s_i, a_i)$ $(1 \leq i \leq n)$. The length of the path is n. A sequence $u \in (I \times E)^*$ is called a *trace* of FSM A in state $s_1 \in S$, if there exists a path $(s_1, a_1\beta_1, s_2)(s_2, a_2\beta_2, s_3)\dots(s_n, a_n\beta_n, s_{n+1})$ s.t. $u = a_1\beta_1a_2\beta_2\dots a_n\beta_n$. Note that a trace of A in state s_0 is a word of the automaton $A_\times$.

Let $Tr(s)$ denote the set of all traces of A in state s, while $Tr(A)$ denotes the set of traces of A in the initial state. Given sequence $u \in (I \times E)^*$, the *input projection* of u, denoted $u_{\downarrow I}$, is a sequence obtained from u by erasing symbols in O. Input sequence $\beta \in I^*$ is a *defined* input sequence in state s of A if there exists $u \in Tr(s)$ s.t. $\beta = u_{\downarrow I}$. We use $\Omega(s)$ to denote the set of all defined input sequences for state s.

Given FSM $A = (S, s_0, I, O, E, h)$ and $s, t \in S$, s and t are *equivalent*, $s \cong t$, if $Tr(s) = Tr(t)$. States s and t are *distinguishable*, $s \ncong t$, if there exists $\alpha \in \Omega(s) \cap \Omega(t)$ s.t. $\{u \in Tr(s) \mid u_{\downarrow I} = \alpha\} \neq \{u \in Tr(t) \mid u_{\downarrow I} = \alpha\}$, called an input sequence *distinguishing* s and t.

We use $Tr^k(s)$ to denote the set of traces in $s \in S$ each of which has at most k inputs, i.e., $Tr^k(s) = \{u \mid u \in Tr(s) \wedge |u_{\downarrow I}| \leq k\}$. Two states $s, t \in S$ are *k-equivalent* iff $Tr^k(s) = Tr^k(t)$, otherwise, if $Tr^k(s) \neq Tr^k(t)$, states s and t are *k-distinguishable*, in this case there exists a distinguishing sequence of the length at most k.

Given two complete FSM $L = (S, s_0, I, O, E_L, h_L)$ and $K = (P, p_0, I, O, E_K, h_K)$, L and K are (k-)equivalent, iff s_0 and p_0 are (k-)equivalent.

2.2 Initial *k*-Quotient

Definition 1. Given two complete FSM $L = (S, s_0, I, O, E_L, h_L)$ and $K = (P, p_0, I, O, E_K, h_K)$, K is an *initial k-quotient* of L (or simply k-quotient) if

1. $P \subset 2^S$ s.t. $s_0 \in p_0$ and if $s \in p_1$ and $t \in p_2$ for $p_1, p_2 \in P$ then
   - $p_1 = p_2$, if s and t are k-equivalent or
   - $p_1 \neq p_2$, if s and t are k-distinguishable.
2. For all $p \in P$ there exists $s \in p$, s.t. for all $a \in I$
   - $(s', \beta) \in h_L(s, a)$ implies that there exists $p' \in P$, s.t. $(p', \beta) \in h_K(p, a)$, and $s' \in p'$;
   - $(p', \beta) \in h_K(p, a)$ implies that there exists $s' \in S$, s.t. $(s', \beta) \in h_L(s, a)$, and $s' \in p'$.

Initial k-quotients possess the following properties.

Theorem 1. Given FSM K, an initial k-quotient of a complete FSM L, if all the distinguishable states of L are k-distinguishable, then FSM L and K are equivalent; otherwise, if some distinguishable, but k-equivalent, states of L are reachable from the initial state, then L and K are k-equivalent, but are distinguishable.

A k-quotient of an FSM is, thus, its approximation, whose precision can be varied with the parameter k. We omit the proofs of the theorems in the paper due to space limit.

2.3 Inferring *k*-Quotient of FSM

We want to infer a k-quotient of an FSM A by testing. We assume that A is completely specified and deterministic; moreover, only its input set I is known, while the output set (or at least part of it) will be determined from A's outputs.

The basic idea of our inference method is to observe the traces of the unknown FSM A from all k-distinguishable states that can be reached from the initial state, by applying in each state all the input sequences of length k. To represent the observed traces of FSM A, we use a tree FSM.

Definition 2. Given a (prefix closed[1]) set U of observed traces of A over input set I and output set O, the *observation tree* FSM is $(U, \varepsilon, I, O, E_U, h_U)$, where the state set is U, $E_U = \{\beta \in O^* \mid \exists a \in I, \exists u \in U \text{ s.t. } ua\beta \in U\}$, and $h_U(u, a) = \{(ua\beta, \beta) \mid \exists \beta \in O^* \text{ s.t. } ua\beta \in U\}$.

We use U to refer to both, a prefix-closed set of FSM traces and the corresponding tree FSM $(U, \varepsilon, I, O, E_U, h_U)$. In a (tree) FSM U, a state u is a *k-predecessor* of state w, iff u is a proper prefix of w and $|w_{\downarrow I}| - |u_{\downarrow I}| \leq k$.

The inference method includes two steps, behavior exploration resulting in an observation tree and state merging in the tree which yields an FSM. The exploration step terminates when no new state, i.e., k-distinguishable from all the other visited states, can be reached from the initial state. The output of the exploration procedure is an observation tree U. In the next step, we obtain a model M of the FSM A by merging states of U using trace inclusion relation between states.

In the exploration step, we perform a Breadth First Search (BFS) on U, starting with $U = \{\varepsilon\}$, to find a state $u \in U$, which is unmarked and has no k-predecessor marked "prune" (marking is done as explained below), and explore the behavior of the given machine from the state u by applying all the possible input sequences of length k, observing the outputs, and adding the observed traces into U. If the state u is k-equivalent to an already visited state, we mark it "prune".

In the state merging step, we traverse the obtained observation tree U following a BFS and if the trace set of an already visited state is a superset of traces of the current state we merge the two states.

---

[1] Recall that a symbol of an FSM trace is a pair of an input from I and a sequence of outputs from O^*, so every prefix takes an FSM from its initial state into some state.

The following theorem claims that the above method can be used to infer an initial k-quotient of an FSM being tested.

Theorem 2. If the inference method is applied to a deterministic FSM A and yields an FSM M, then FSM M is equivalent to an initial k-quotient of FSM A.

3 Inferring Input/Output Transition System

3.1 Basic Definitions

Certain operations on FSMs, such as composition, are easier to formulate using their transition system counterparts. An *input/output transition system* (IOTS) L is a quintuple $<S, I, O, \lambda, s_0>$, where S is a set of states; I and O are disjoint sets of input and output actions, respectively; $\lambda \subseteq S \times (I \cup O \cup \{\tau\}) \times S$ is the transition relation, with the symbol τ denoting internal actions; and s_0 is the initial state.

$(t, a, s) \in \lambda$ is called a *transition*; (t, a, s) is *input*, *output* or *internal* transition, if $a \in I$, $a \in O$ or $a = \tau$, respectively. Given IOTS L, a *path* from state s_1 to state s_{n+1} is a sequence of transitions $p = (s_1, a_1, s_2)(s_2, a_2, s_3)\ldots(s_n, a_n, s_{n+1})$, s.t. $(s_i, a_i, s_{i+1}) \in \lambda$ for $i = 1, \ldots, n$.

Let ε denote the empty sequence of actions. The *projection operator* $\downarrow A$, which projects sequences of actions onto the set $A \subseteq I \cup O \cup \{\tau\}$, is recursively defined as $\varepsilon\downarrow_A = \varepsilon$, $(va)\downarrow_A = v\downarrow_A a$ if $a \in A$, and $(va)\downarrow_A = v\downarrow_A$ otherwise, where $v \in (I \cup O \cup \{\tau\})^*$ and $a \in I \cup O \cup \{\tau\}$. We also lift the projection operator to IOTS, i.e., given IOTS $L = <S, I, O, \lambda, s_0>$ and set $A \subseteq I \cup O$, the IOTS $L\downarrow_A$ is obtained by first replacing each transition $(t, a, s) \in \lambda$ s.t. $a \notin A$ by internal transition (t, τ, s) and then determinizing the obtained IOTS (with tau-reduction) [18].

A sequence $u \in (I \cup O)^*$ is called a *trace* of IOTS L in state $s_1 \in S$ if there exists a path $(s_1, a_1, s_2)(s_2, a_2, s_3)\ldots(s_n, a_n, s_{n+1})$, s.t. $u = (a_1\ldots a_n)\downarrow_{(I\cup O)}$. Similar to FSM, we use $Tr(T)$ to denote the set of traces in states $T \subseteq S$, while $Tr(L)$ to denote the set of traces of L in the initial state. We use $Tr^k(s)$ to denote the set of traces in $s \in S$, each of which has at most k input actions, i.e., $Tr^k(s) = \{u \mid u \in Tr(s) \wedge |u\downarrow_I| \leq k\}$. *k-equivalent* and *k-distinguishable* states are defined similar to that of FSM.

We use $init(s)$ to denote the set of actions enabled in state s, i.e., $init(s) = \{a \in (I \cup O \cup \{\tau\}) \mid \exists t \in T \text{ s.t. } (s, a, t) \in \lambda\}$. L is *input-enabled* if all input actions are enabled in each state, i.e., $I \subseteq init(s)$ for each $s \in S$; L is *fully specified* if either all or no input actions are enabled in each state, i.e., either $I \subseteq init(s)$ or $I \cap init(s) = \emptyset$ for each $s \in S$. If L is not fully specified, it is *partially specified*. In state s of a fully specified IOTS L, either L does not read input at all if no input is enabled in s, or L's behavior after any input is defined in s. The response of L to any input, therefore, is predictable, and hence we call L "fully specified". L is *deterministic* if it has no internal transitions and λ is a function $S \times (I \cup O) \rightarrow S$. State $s \in S$ is *stable* if no output or internal actions are enabled in s, i.e., $init(s) \cap (O \cup \{\tau\}) = \emptyset$, otherwise it is *unstable*. IOTS L is *conflict-free* if input actions are only enabled in stable states. Intuitively, such a system is allowed to produce all possible outputs in response to input before the environment offers a next input, similar to FSM, where input/output

transitions are atomic. In fact, any FSM, once each of its transitions is unfolded into input transition followed by output transition (which is omitted, if the output is empty, $\varepsilon \in E$), to an intermediate state, yields a conflict-free IOTS.

In a conflict-free IOTS, there is no nondeterminism related to concurrency of inputs and outputs, but there may still be nondeterministic choices of outputs. A conflict-free IOTS L is output deterministic if it produces a single output sequence in response to any input sequence. Formally, $L = \langle S, I, O, \lambda, s_0 \rangle$ is *output-deterministic* iff $\alpha_{\downarrow I} = \beta_{\downarrow I}$ implies $\alpha = \beta$ for any $\alpha, \beta \in Tr(s_0)$, otherwise it is *output-nondeterministic*. Thus, an IOTS can be output-nondeterministic and, at the same time, deterministic.

State $s \in S$ is a *deadlock* if no action is enabled in it, i.e., $init(s) = \varnothing$. State $s \in S$ is a *livelock* if there is a cycling path of output or internal transitions that includes s; if the path includes only internal transitions then livelock is *internal*, otherwise it is an *output* livelock. IOTS L is *deadlock-free* or *livelock-free*, if there is no deadlock or livelock state reachable from a starting state, respectively. L is *input-progressive* if it is deadlock-free and livelock-free. If L is input-progressive, input actions are defined in each stable state, and it enters a stable state executing fewer than $|S|$ transitions after any input.

3.2 Inferring *k*-Quotient of IOTS

As mentioned earlier, livelock-free, deterministic, and conflict-free finite IOTS can be considered as another representation of observable FSM. This indicates that the inference method for FSM can be used for this kind of IOTS. We assume that an unknown IOTS A is finite, fully specified, deterministic, output-deterministic, and conflict-free, moreover, its input action set I is known. Notice that by requiring IOTS be output-deterministic we make the results of Section 2 directly applicable to IOTS. Since a deadlock state cannot observationally be distinguished from a stable state where all inputs cause looping transitions, we also assume that the IOTS A has no deadlock.

The observation tree is defined as follows: given a (prefix closed) set U of observed traces of A over input action set I and output action set O, the observation tree is an IOTS $\langle U, I, O, \lambda_T, \varepsilon \rangle$, where the state set is U, and $(\beta, a, \beta a) \in \lambda_T$ iff $\beta a \in U$. A state β of U is stable if $\beta a \notin U$ for all $a \in O$. We use U to refer to both, a prefix-closed set of traces and the IOTS $\langle U, I, O, \lambda_T, \varepsilon \rangle$. Based on the observation tree, we can determine k-equivalence of stable states.

In a (tree) IOTS U, a stable state u is a *k-predecessor* of stable state w, iff u is a proper prefix of w and $|w_{\downarrow I}| - |u_{\downarrow I}| \leq k$.

The inference method for IOTS also includes two steps, behavior exploration and state merging.

In the exploration step, we perform a BFS on stable states of U, starting with $U = \{\varepsilon\}$, to find a stable state $u \in U$, which is unmarked and has no k-predecessor marked “prune”, and explore the behavior of the given IOTS from the state u by applying all the possible input sequences of length k, observing the outputs, and adding the observed traces into U. If the state u is k-equivalent to an already visited stable state, we mark it “prune”. Applying an input action to IOTS A, we wait for output actions. By our assumption, the IOTS A has no deadlock; at the same time, it may still have

output livelocks. To detect output livelock in a black box, we set a bound for the maximum length of output sequences the IOTS can produce in response to a single input action. This bound cannot exceed the number of states in the IOTS. If the length of observed output sequence exceeds the given bound, exploration terminates and output livelock is declared.

In the state merging step, we traverse the stable states of the obtained observation tree U following a BFS and if the trace set of an already visited stable state is a superset of traces of the current stable state we merge the two states.

Since livelock-free, deterministic and conflict-free IOTS is, in fact, another representation of observable FSM, Theorem 2 still applies to the above procedure once initial k-quotient of FSM is replaced by its IOTS counterpart, which definition we omit here for simplicity.

4 Inference of Modular Systems

4.1 Basic Definitions

In this paper, we use queued communications between system's components. Modeling queues, we distinguish the same action at the two ends of a queue by using the relabeling $'$ operator [1]. The operator is defined on input actions: for $a \in I$, $(a)' = a'$, and $(a')' = a$. It is lifted to the sets of input actions, traces, and IOTS: for an action set I, $I' = \{a' \mid a \in I\}$; for traces, $'$ is recursively defined as $\varepsilon' = \varepsilon$ and $(ua)' = u'a'$ for trace u if a is an input action, otherwise $(ua)' = u'a$; L' is obtained from L by relabeling each action $a \in I$ to a'. A (unbounded) *queue with input set I*, is an IOTS $<I^*, I, I', \lambda_I, \varepsilon>$, denoted Q_I, where the state set I^* and transition relation $\lambda_I = \{(u, a, ua) \mid u, ua \in I^*\} \cup \{(av, a', v) \mid av, v \in I^*\}$. The only stable state of a queue is its initial state.

We consider a modular system consisting of components communicating asynchronously, where each component reads inputs from its input queue and writes outputs to other components' input queues. We assume in this paper that the components are conflict-free, as well as input-progressive; moreover, without losing generality, we also assume that they are deterministic. Each component is not input-enabled, but the composition of the component with its input queue is input-enabled. It means that even if the component does not read an input in some state, the input is not lost, kept in the input queue, and can be consumed in subsequent state reached by internal or output transitions, in other words, we do not need to assume that inputs can be blocked or refused, as in other work [2].

Let $C = \{C_1, C_2, \ldots, C_n\}$ be a set of component IOTS's, where $C_i = <S_i, I_i, O_i, \lambda_i, s_{0i}>$, s_{0i} is a stable initial state, s.t. $I_i \cap I_j = \varnothing$ and $O_i \cap O_j = \varnothing$ for $i \neq j$. Let I and O denote the union of all input action sets and output action sets, respectively. Two components C_1 and C_2 *communicate* if $I_1 \cap O_2 \neq \varnothing$ or $O_1 \cap I_2 \neq \varnothing$. For each component C_i with the input set I_i, there is an unbounded input queue $Q_{I_i} = <I_i^*, I_i, I_i', \lambda_{I_i}, \varepsilon_i>$, thus, each component consumes inputs from its input queue and produces an external output or internal output, the later is stored in input queue of other component. The set $I_{\text{ext}} = I \backslash O$ contains *external inputs*; components constitute a *closed*

system, if I_{ext} is empty, otherwise an *open* system. Let $O_{\text{ext}} = O \backslash I$ be the set of *external outputs* of the system. In this paper, we consider only an open system $C = \{C_1, C_2, \ldots, C_n\}$ with at least one external output, s.t. each component communicates in the system, more precisely, we assume that for each component C_i it holds that if $a \in I_i$ then either $a \in I_{\text{ext}}$ or $a \in O_j$ and if $a \in O_i$ then $a \in O_{\text{ext}}$ or $a \in I_j$ for some C_j.

A behavior of an open system may vary, depending on the speed of its environment. We distinguish two types of the environment, fast and slow. A *fast* environment can supply external inputs at any state of a system with which it communicates. A *slow* environment does so only when the system is in a stable global state.

The behavior of the system operating in the fast environment is described by the IOTS $C_1' \parallel C_2' \parallel \ldots C_n' \parallel Q_{I_1} \parallel Q_{I_2} \parallel \ldots \parallel Q_{I_n}$, where $\parallel$ is the standard LTS parallel composition operator, $C_i' = <S_i, I_i', O_i, \lambda_i, s_{0i}>$, for $C_i = <S_i, I_i, O_i, \lambda_i, s_{0i}>$, and $Q_{I_i} = <I_i^*, I_i, I_i', \lambda_{I_i}, \varepsilon_i>$. To describe the behavior in case of the slow environment, we modify the $\parallel$ operator to allow external inputs only in stable global states.

Definition 3. Given a system of communicating components $C = \{C_1, C_2, \ldots, C_n\}$, where $C_i = <S_i, I_i, O_i, \lambda_i, s_{0i}>$ and the set of queues over input alphabets of the components $Q = \{Q_{I_1}, \ldots, Q_{I_n}\}$, where $Q_{I_i} = <I_i^*, I_i, I_i', \lambda_{I_i}, \varepsilon_i>$, the *slow asynchronous product* of C, denoted $\Sigma = \Pi_{i=1}^{n} C_i$, is the IOTS $<R, I_{\text{ext}}, O, \lambda, s_{01}\ldots s_{0n}\varepsilon_1\ldots\varepsilon_n>$, where $I_{\text{ext}} = I \backslash O$, $I = I_1 \cup \ldots \cup I_n$ and $O = I_1' \cup \ldots \cup I_n' \cup O_1 \cup \ldots \cup O_n$; the set of states $R \subseteq S_1 \times \ldots \times S_n \times I_1^* \times \ldots \times I_n^*$ and the transition relation λ are the smallest sets obtained by applying the following inference rules:

- $s_{01}\ldots s_{0n}\varepsilon_1\ldots\varepsilon_n \in R$;
- if $a \in I_{\text{ext}} \cap I_i$, $(s_1\ldots s_n\varepsilon_1\ldots\varepsilon_n) \in R$ s.t. states $s_1, \ldots, s_n$ are stable, then $(s_1\ldots s_n\varepsilon_1\ldots\varepsilon_n, a, s_1\ldots s_n b_1\ldots b_n) \in \lambda$ and $(s_1\ldots s_n b_1\ldots b_n) \in R$ s.t. $b_i = a$, and $b_j = \varepsilon_j$ for $j \neq i$ (external input is buffered in the queue of the component, which has it as input);
- if $a \in I_i$, $(s_1\ldots s_n b_1\ldots b_n) \in R$ s.t. $a \in init(s_i)$, $b_i = av$, then $(s_1\ldots s_n b_1\ldots b_n, a', s_1'\ldots s_n' c_1\ldots c_n) \in \lambda$ and $(s_1'\ldots s_n' c_1\ldots c_n) \in R$, s.t. $(s_i, a, s_i') \in \lambda_i$, and $c_i = v$; $s_j' = s_j$ and $c_j = b_j$ for $j \neq i$ (input is consumed from a queue, and input transition is executed);
- if $a \in O_i$, $(s_1\ldots s_n b_1\ldots b_n) \in R$ s.t. $a \in init(s_i)$, then $(s_1\ldots s_n b_1\ldots b_n, a, s_1'\ldots s_n' c_1\ldots c_n) \in \lambda$ and $(s_1'\ldots s_n' c_1\ldots c_n) \in R$, s.t. $(s_i, a, s_i') \in \lambda_i$, $s_j' = s_j$ for all $j \neq i$, and if $a \in I_j$ then $c_j = b_j a$, otherwise, $c_j = b_j$ (output transition is executed, and output is buffered in the queue of the component, for which it is an input).

Notice that all components' inputs, save external ones, as well as outputs, become (observable) outputs of the composition, as is usually the case for input-output automata composition [3]. The notion of slow asynchronous product defined here is similar to what has already been implemented for SDL tools such as Object Geode.

Definition 4. A system C of communicating components in slow environment has

- *unspecified reception*, if in the product Σ, there exists a state $(s_1\ldots s_n b_1\ldots b_n)$ s.t. for some component C_i, a is a prefix of b_i and $a \in I_i$, but $a \notin init(s_i)$;
- *compositional livelock*, if Σ has a livelock state;

- *divergence*, if in Σ, there exists a path from state $(s_1...s_n b_1...b_n)$ to state $(s_1...s_n c_1...c_n)$ s.t. each b_i is a prefix of c_i, and there exists $d \neq \varepsilon$, s.t. $b_i dd$ is a prefix of c_i.
- *races*, if there exist traces $\alpha, \beta \in Tr(\Sigma)$ s.t. $\alpha{\downarrow}_{I_{\text{ext}}} = \beta{\downarrow}_{I_{\text{ext}}}$ and $\alpha{\downarrow}_{O_{\text{ext}}} \neq \beta{\downarrow}_{O_{\text{ext}}}$.

The system with at least one of the above properties is said to have a *compositional* problem, the system with no compositional problem is *well-formed*. Notice that unspecified receptions cause compositional deadlocks, divergence causes buffer overflow, and races are a witness of a nondeterministic behavior of a system composed of deterministic components.

Given a system of communicating components, the slow asynchronous product can be constructed and, thus, its well-formedness can be checked using a classical reachability analysis (RA) procedure which we will not discuss further in this paper. We simply assume that given a system $C = \{C_1, C_2, \ldots, C_n\}$, the procedure either confirms that the system is well-formed or outputs the following *witness* (diagnostic) traces for:

- unspecified reception of $a \in I_i$, a trace βa, s.t. β takes the product into a state, where the action a is not enabled in the corresponding state of some component C_i whose input queue contains just a;
- compositional livelock and divergence, a trace $\alpha\beta$, s.t. α takes the product into a state of a cycle or path, respectively, labeled by the sequence β;
- races, two traces α and β with a common external input projection which leads to races.

4.2 The Approach

To infer models of communicating components, one can determine a k-quotient of each component in isolation using the method of Section 3. This approach requires that an appropriate value of the parameter k be chosen individually for each component. Intuitively, a component with more states would require a bigger value than components with fewer states, however, it is unclear how one can determine these parameters without taking into account the components' interactions, except for (rare) cases when the maximal number of states is a priori known for each component. A trial-and-error approach can be followed, and even if it succeeds, it may result in "redundant" models of components which describe functionalities unused in a given system.

Another approach which we follow in this paper is to infer first a k-quotient of the slow asynchronous product (thus, assuming that internal outputs are observable for testing) and determine models of components by projecting the product onto the alphabets of each component and refining them if needed. If the given system has "a single message in transit", (see, e.g., [16]) then the slow asynchronous product is output-deterministic and the inference method of Section 3 directly applies. If, however, several actions can concurrently be executed in the system, the product becomes output-nondeterministic. This means that the system could produce several

output interleavings in response to a given external input sequence, if that sequence is applied several times. The problems coming from such causes are often hard to elicit and to reproduce in functional testing; they often appear as a side effect in stress testing, but are harder to analyze in that stage. The crux of our approach is precisely to be able to identify such intermittent problems that occur only under specific circumstances in integrated systems.

Therefore, we assume that during the exploration the system behaves as an output-deterministic IOTS and use the inference method of Section 3. We do not rely on "all weather", aka "complete testing", assumption [4]. First, such an assumption is very costly because each input sequence must be tested a potentially huge number of times (esp. for long sequences); second, it is often unrealistic to assume that all interleavings will be observed in a given test configuration, and if configurations must be changed for each occurrence of an input sequence, this is even more costly in test execution time.

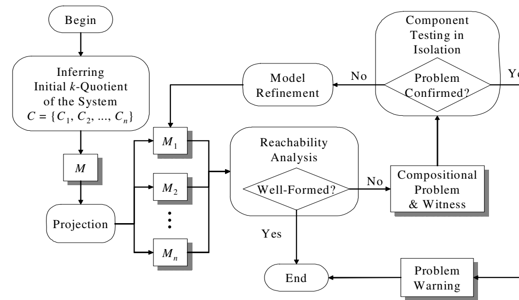

Fig. 1. The verification approach

To identify and check all possible interleavings, we use the RA procedure. Even if the components themselves constitute a well-formed system the inferred models do not necessarily do so. Each obtained model is only an approximation of the actual behavior of a real component and may possess a behavior absent in the component, so the inferred models may exhibit compositional problems. RA can discover them by exploring all the possible execution interleavings.

The models exhibit a compositional problem in two cases: either the problem exists in the real system or the inferred models are not adequate models of the behavior of the system exposed during the exploration step and need to be further refined. To confirm that a compositional problem detected in RA exists in the real system, one can check whether each projection of a witness trace, which was not observed in the exploration step, belongs to the set of traces of the corresponding component by executing the projected trace against the component in isolation. The actions of the witness trace determine the order of testing the components involved in the execution, which terminates as soon as some component produces a trace different from the expected one. In this case, the compositional problem is refuted, and the newly

obtained trace is used to refine the model of that component. The iterative process terminates when either a compositional problem is confirmed to exist in the real system or a well-formed system of models is obtained. Notice the real system to have compositional livelock or divergence should exhibit a cyclic behavior in all the involved components when they are tested using the projected witness traces. In black-box testing, we can only confirm this by repeatedly applying input sequence of a trace to each component. The verification approach is summarized in Figure 1.

While the main steps of this procedure are intuitive and clear from the previous discussions, the model refinement needs some explanation.

Let U be a global observation tree obtained by exploring the behavior of a given system of communicating components $C = \{C_1, C_2, \ldots, C_n\}$. Moreover, let M be an IOTS obtained by merging states of the global observation tree and $M_1, M_2, \ldots, M_n$ be the IOTS models of the components obtained by projecting M onto their alphabets. Assume that for some witness trace, the component C_i in response to the input projection of the witness trace produces a trace α which is not in IOTS M_i. To refine the latter, we add this trace to the local observation tree U_i (obtained by projecting the global observation tree U), thus, $U_i := U_i \cup \{\alpha\}$ and merge states in the updated tree using the above given procedure to obtain a refined model M_i'. The refined model is used to check again the well-formedness of the current models.

4.3 Example

We illustrate the approach for inference of communicating components using the example shown in Figure 2. We infer a k-quotient of the product of the given system for $k = 1$. The global observation tree U obtained after applying the external input x is shown in Figure 3 (we use the actual components in Figure 2 to determine the reaction of the system).

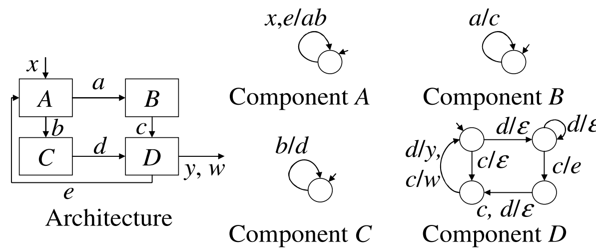

Fig. 2. A Modular System

The final state needs to be explored, so the input sequence xx is now applied to the system, the input x creates in the global observation tree the same trace, so we conclude that the stable states ε and $xx'aa'bcb'c'dd'y$ are 1-equivalent. The behavior exploration procedure terminates, and both stable states are merged, to obtain the IOTS M which is a 1-quotient of the product, it is shown in Figure 3 with the dashed transition labeled y. Next, we determine models by projecting the IOTS M. M_2 and M_3 are trace included by B and C respectively. M_1 and M_4 are shown in Figures 4 and 5.

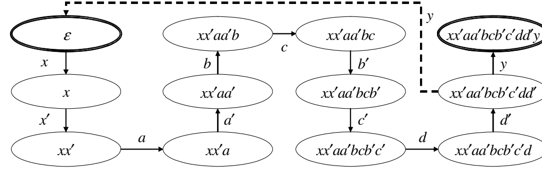

Fig. 3. The global observation tree U after applying x; stable states are depicted in bold

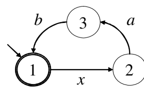

Fig. 4. M_1

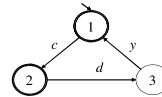

Fig. 5. M_4

Now we perform RA of the system composed of M_1, M_2, M_3 and M_4 to check whether it is well-formed and find that there exists an unspecified reception. The witness trace is $xx'aa'bb'd$. The components M_1 and M_3 are involved in the execution of this trace, their traces are $x'ab$ and $b'd$, respectively; the component M_4 has unspecified reception of d. The traces $x'ab$ and $b'd$ are already in the global observation tree, so only the component D has to be tested in isolation by applying the input d. We observe that the component has no output in response to this input, so the obtained trace d is added to its observation tree, as shown in the first row of Table 1. The table contains local observations trees and models of the components A and D as well as a witness trace for each version of the models. The table illustrates the iterative process of refining the two models based on detected unspecified receptions. In these figures, "?" and "!" are used to represent input and output respectively, as usual.

Using models in the last row of the table, we detect a livelock, the witness traces are xx', leading to livelock and $aa'bb'dd'cc'ee'$ which labels it. The corresponding projections of $xx'aa'bb'dd'cc'ee'$ are $x'abde'$, $a'c$, $b'd$, and $d'c'e$. To confirm this livelock in the real system, we proceed as follows. Assuming that in each component a trace executed consecutively three times indicates that the component cycles, we add to the above projections two more instances of the trace labeling a cycle of the component, obtaining $x'(abe')^3$, $(a'c)^3$, $(b'd)^3$, and $(d'c'e)^3$.

None of these traces is in the local observation trees, so we have to apply *xeee*, *aaa*, *bbb*, *dcdcdc* to the components A, B, C, and D, respectively. We use again the real components to obtain the following traces: *xabeabeabeab*, *acacac*, *bdbdbd*, and *dcedcwdce*. Only component D produces a new trace *dcedcwdce*, the current model of D expects *dcedcedce* instead, so we add the trace *dcedcwdce* to the observation tree and refine the model of D to obtain the IOTS shown in Table 2 which illustrates final model refinements.

The RA of the obtained models results now in the witness traces for races: $xx'aa'cc'bb'dd'y$ and $xx'aa'bb'dd'cc'ee'aa'bb'dd'cc'w$. In fact, the external input sequence x yields several traces in the product which differ in their output projections, y and w. We project the two traces to each component, and find that all the obtained traces are already in the corresponding local observation trees. The compositional problem is confirmed. With this report the procedure terminates. Even though the inferred models exhibit compositional problem, they are trace included in the actual models.

Table 1. Iterative model refinement

U_1	M_1	U_4	M_4	Witness traces
				$xx'aa'bb'dd'c$
				$xx'aa'bb'dd'cc'e$
				$xx'aa'cc'bb'dd'y$ $xx'aa'cc'bb'dd'yx$
				$xx'aa'cc'bb'dd'y$ $xx'aa'bb'dd'cc'e$
				$xx'aa'bb'dd'cc'ee'aa'c$

Table 1. (*continued*)

				$xx'aa'bb'dd'cc'ee'aa'cc'bb'dd'yx$
				$xx'aa'bb'dd'cc'ee'aa'bb'dd'cc'e$
				xx' and $aa'bb'dd'cc'ee'$

Table 2. Final Iterations

U_4	M_4	Witness traces
		$xx'aa'bb'dd'cc'ee'aa'cc'bb'd$
		$xx'aa'cc'bb'dd'y$ and $xx'aa'bb'dd'cc'ee'aa'bb'dd'cc'w$

5 Related Work

Finite State Machine learning is widely addressed, specifically in the grammatical inference works, e.g., [9], [8] etc. Angluin presents a polynomial time algorithm [5] to infer regular language as deterministic finite automaton. She uses an equivalence oracle which provides a (minimal) counterexample. In our work, the state *k*-equivalence relation can be viewed as an approximated equivalence oracle. Learning and testing through model checking approach is used to infer a grey box system [15]. However, the upper bound on the number of states in the system is required to test conformance of the conjectured model to the actual system. On the other hand, the notion of *k*-quotient provides a means for inferring a variable size approximation without upper bound on the number of states.

The observation tree FSM used in this paper is, in fact, an input-output version of a prefix tree machine [13] used in various techniques for learning an automaton from positive examples. However, we do not require samples of the behavior given for inference, the samples (traces) are obtained by testing. At the same time, as in a common paradigm in learning from positive examples, the observation tree FSM is also iteratively merged. The rule used for state merging in the proposed approach is language containment (input-output trace inclusion). The exploration step ensures that the observation tree FSM contains *k*-equivalent states; these states have the same traces for *k* consecutive inputs. As in [14], the parameter *k* allows one to control the precision and complexity of the synthesized machine. However, differently from that work, the inferred *k*-quotient is a deterministic FSM.

There is much less work published on the inference of modular systems. The work [17] relies on observed traces to construct automata models of communicating components which are then model-checked using user defined properties. An object-flattening technique [11] is used to collect system behavior and then invariants are calculated on the behaviors to check against the new version of the system. This work is more related to regression testing. Moreover, the system behavior is observed while the system is running as in [17]. In this paper, we rely on testing communicating system by stimulating it through external inputs and then use the observations to obtain tunable approximated models. The verification of black box communicating system on the architectural level is also addressed recently [7]. Similar to the previous approaches, the system is monitored at runtime by instrumenting the middleware, no testing strategy is used. On the contrary, we infer models by testing and use them to check the system for compositional problems. Our past work [10] and [12] in this domain also concerned inference of components as finite state machines through testing. In fact, we were implicitly inferring a 1-quotient of each component in isolation without defining *k*-quotient. Moreover, the previous approaches focused on learning components separately, one component at a time; whereas in this paper, we proposed to test the integrated system avoiding thus unnecessary testing efforts to learn models only related to the composition.

6 Conclusion

In this paper, we offered a solution to the problem of modular system verification by blending together techniques of inference, testing, and reachability.

We first suggested the notion of an approximation of a Mealy machine, called k-quotient, where all k-equivalent states of the given machine are represented as a single state of the quotient, provided that at least one of them is on a path from the initial state which includes only pairwise k-distinguishable states. The precision of this approximation can be varied with the parameter k. We then proposed an approach to infer models of a modular system which starts with exhaustive (limited by some test length) testing of the integrated system and iterates between RA of intermediate models and pinpointed testing of components in isolation. A k-quotient of the modular system with observable internal actions in the form of an IOTS is used to infer initial models of components by projecting the quotient. RA of the models is next used to identify composition problems, such as unspecified receptions, livelocks, divergences, and races. A witness (diagnostic) trace is then used to test concerned components in isolation either to confirm that a problem exists in the real system or to obtain new observations. The models are then refined using new observations until the obtained models are well-formed.

The proposed approach relies on application of all external input sequences of length k, however, the parameter allows one to find a compromise between complexity of testing of the integrated system and precision of the resulting models. Moreover, the use of all input sequences of given length is completely avoided in testing a component in isolation, since only single diagnostic test is executed in each iteration. Another advantage of the approach is that inferred models capture the functionalities of components used in the given system; unused behaviors of components are not modeled.

As a future work, it would be interesting to investigate whether instead of testing a number of components in isolation based on a witness trace one would test just a subsystem consisting of these components to reduce the number of iterations needed to infer well-formed models. There are also a number of options for treating witness traces to update the global observation tree once the individual models are refined. This may help converge faster and shorten the RA process. It is known that reachability analysis can provide more than one witness traces, evidencing multiple problems in one step. The treatment of multiple traces at a time could be another improvement in the approach.

References

1. Huo, J., Petrenko, A.: Covering Transitions of Concurrent Systems through Queues. In: ISSRE, pp. 335–345 (2005)
2. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)
3. Lynch, N., Tuttle, M.: An Introduction to Input/output Automata. CWI-Quarterly 2(3), 219–246 (1989)
4. Luo, G., Bochmann, G.v., Petrenko, A.: Test Selection Based on Communicating Nondeterministic Finite State Machines Using a Generalized Wp-Method. IEEE Transactions on Software Engineering (February 1994)
5. Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Information and Computation 2, 87–106 (1987)

6. Godefroid, P.: Model Checking for Programming Languages Using VeriSoft. In: POPL, pp. 174–186 (1997)
7. Bertolino, A., Muccini, H., Polini, A.: Architectural Verification of Black-box Component-based Systems. In: Guelfi, N., Buchs, D. (eds.) RISE 2006. LNCS, vol. 4401, pp. 98–113. Springer, Heidelberg (2007)
8. Balcazar, J.L., Diaz, J., Gavalda, R.: Algorithms for learning finite automata from queries: A unified view. In: AALC, pp. 53–72 (1997)
9. Kearns, M.J., Vazirani, U.V.: An introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
10. Li, K., Groz, R., Shahbaz, M.: Integration Testing of Components Guided by Incremental State Machine Learning. In: TAIC PART, pp. 231–247 (2006)
11. Mariani, L., Pezzè, M.: Behavior Capture and Test: Automated Analysis of Component Integration. In: ICECCS, pp. 292–301 (2005)
12. Shahbaz, M., Li, K., Groz, R.: Learning and Integration of Parameterized Components Through Testing. In: TestCom, pp. 319–334 (2007)
13. Cook, J.E., Wolf, A.L.: Discovering Models of Software Processes from Event-Based Data. ACM Trans. Softw. Eng. Methodol. 7(3), 215–249 (1998)
14. Biermann, A., Feldman, J.: On the Synthesis of Finite State Machines from Samples of their Behavior. IEEE Transactions on Computers 21(6), 592–597 (1972)
15. Elkind, E., Genest, B., Peled, D., Qu, H.: Grey-box Checking. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 420–435. Springer, Heidelberg (2006)
16. Petrenko, A., Yevtushenko, N.: Solving Asynchronous Equations. In: FORTE, pp. 231–247 (1998)
17. Hallal, H.H., Boroday, S., Petrenko, A., Ulrich, A.: A Formal Approach to Testing Properties in Causally Consistent Distributed Traces. Formal Aspects of Computing 18(1), 63–83 (2006)
18. Jéron, T., Morel, P.: Test Generation Derived from Model-Checking. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 108–121. Springer, Heidelberg (1999)