Computing Complete Test Graphs for Hierarchical Systems

Deepak D'Souza and Madhu Gopinathan
Computer Science and Automation
Indian Institute of Science
Bangalore, India
{deepakd,gmadhu}@csa.iisc.ernet.in

Abstract

Conformance testing focuses on checking whether an implementation under test (IUT) behaves according to its specification. Typically, testers are interested in performing targeted tests that exercise certain features of the IUT. This intention is formalized as a test purpose. The tester needs a "strategy" to reach the goal specified by the test purpose. Also, for a particular test case, the strategy should tell the tester whether the IUT has passed, failed, or deviated from the test purpose. In [9] Jéron and Morel show how to compute, for a given finite state machine specification and a test purpose automaton, a complete test graph (CTG) which represents all test strategies. In this paper, we consider the case when the specification is a hierarchical state machine and show how to compute a hierarchical CTG which preserves the hierarchical structure of the specification. We also propose an algorithm for an online test oracle which avoids a space overhead associated with the CTG.

1 Introduction

This paper deals with the problem of conformance testing of reactive systems, which are systems that maintain an ongoing interaction with their environment. Here we are given a specification of the system which is to be used as a reference model to check whether the outputs produced by the implementation under test (IUT) in response to inputs from the environment are correct. Conceptually, we could look at conformance testing as a game between the Tester and the IUT [13, 4] in which the Tester makes input moves and the IUT makes output moves, in alternation.

Since exhaustive testing is impossible, testers utilize their knowledge of the system and select important and/or problematic features for testing (typically guided by system usage scenarios). Consider a use case based test generation methodology [7]. End users of the system can go down many paths as they execute the functionality specified in the use case. To generate test cases for a given use case, the tester needs to consider the basic flow and all the other alternate flows (including basic flow and alternate flow of sub-usecases). Some of the flows will result in the end user's goal being achieved while some others end with it being abandoned. Instead of constructing detailed test cases, the tester can alternatively give a *test purpose*, which can be thought of as a specification of a set of test cases [10]. The test purpose can be represented as an automaton which specifies the input-output sequences along which the tester would like to test the IUT's behaviour. The test purpose thus serves to narrow down the set of interactions the tester would like to explore.

In [9] Jéron and Morel consider the problem of conformance testing with respect to such a test purpose. They introduce the notion of a *complete test graph* or CTG, which may be described as a deterministic transition system which tells the tester at each step whether the test so far has passed, failed, proved inconclusive (in the sense that the interaction has deviated from the test purpose without violating the specification), or should be continued. The CTG is "complete" as it represents all possible test cases, in the sense that every test case can be obtained as a pruning of an unfolding of the CTG. Thus, using some coverage criteria, a CTG can be used to automatically generate test cases that adhere to a given test purpose. This underscores the role of a test purpose, as in the face of evolving specifications and IUT's the tester does not have to repeatedly modify his test cases.

In this paper we consider the problem of constructing a CTG for a *hierarchical* system specification. Hierarchy is a useful concept in the description of large systems. In hierarchical finite state machines, the nodes of an FSM can be ordinary states or FSMs themselves (boxes). This is a useful structuring mechanism that allows the user of the specification to look at an appropriate level of detail. Also, this

*This work is partially supported by grants from TRDDC, Pune, and MHRD, Govt. of India.

enables a machine to be specified once and reused in different contexts. This nesting capability is commonly used in modeling languages such as UML (hierarchical scenario diagrams called Interaction Overview diagrams in UML2, hierarchical state machines in UML-RT etc.) as well as in scenario-based testing tools such as Lucent's UBET. It is reported in [6] that hierarchical state diagram of TCAS (Traffic Collision Avoidance System) has approximately 140 states whereas the complete reachability graph is estimated to contain at least 10^{40} states. It is not known how much of this reduction can be attributed to hierarchy as opposed to concurrency. However, we know that hierarchy and concurrency are additive in reducing the size of the specification [2].

By a CTG for a hierarchical specification (with respect to a test purpose) we simply mean a CTG for the "flattened" specification. However, we are interested in representing such a CTG hierarchically, preferably in a way which mirrors the hierarchy of the specification. The advantages of such a hierarchical CTG are twofold. Such a CTG provides a succinct representation of a test graph. For hierarchical models with large state spaces, it may not always be possible to store a flat CTG in memory. Secondly, a hierarchical CTG provides us different notions of coverage for generating test cases. For example, we can use a notion of transition coverage where a transition in a module needs to be covered in at least one context, as against covering every edge of each module in every possible context, which could result in a large number of test cases.

We show that it is indeed possible to compute a hierarchical CTG for a given hierarchical specification and test purpose. The technique we use is essentially an adaptation of the idea of "splitting" used by Alur and Yannakakis [3] for model checking CTL formulas on hierarchical Kripke structures. We also propose an algorithm for an "online" test oracle that can be used by the tester as a guide to whether the current test sequence of inputs and outputs passes, fails, or deviates from the test purpose. While the hierarchical CTG itself can be used to construct such an oracle, the algorithm we give is more space efficient, at the cost of time proportional to the nesting depth of the hierarchical system at each step.

Turning now to related work, as mentioned already, this paper builds on work of [9, 8] and [3]. A graph similar to a Complete Test Graph is used in [4] where it is called a *transient* test graph. Lengauer and Wanke have studied hierarchical graphs used for succinct design description in CAD systems and give efficient solutions for connectivity problems [11]. Mosk-Aoyama and Yannakakis show that the problem of computing a minimum number of test cases such that every transition of every module is included in at least one test case, is NP-hard when the nesting depth is greater than or equal to two [12].

This paper is organised as follows. We begin with preliminary definitions and a description of the testing problem we consider. We then introduce hierarchical transition systems and address the problem of computing a hierarchical CTG. We give details of the precise lemmas required to prove the correctness of our constructions. Proofs not included here due to lack of time can be found in the technical report [5].

2 Complete Test Graphs

In this section we introduce the notion of a complete test graph, and illustrate the TGV approach [9, 8] for computing such a graph for a non-hierarchical specification and a given test purpose. We begin with some necessary preliminary notions.

For an alphabet A, we use A^* to denote the set of finite words over A, and ϵ to denote the empty word. For $x \in A^*$ we write $|x|$ to denote the length of x. The concatenation of words x and y will be denoted by $x \cdot y$ (or simply xy), and we extend this notation to subsets of A^* in the natural way. We write $x \leq y$ to mean x is a prefix of y (i.e. there exists z such that $xz = y$).

A *labeled transition system* over an alphabet A is a structure of the form $\mathcal{T} = (Q, s, \longrightarrow)$, where Q is a finite set of states, $s \in Q$ is the start state, and $\longrightarrow \subseteq Q \times A \times Q$ is the A-labeled transition relation. We write $p \xrightarrow{a} q$ to stand for $(p, a, q) \in \longrightarrow$, and use $p \xrightarrow{x} {}^*q$ to denote the fact that we have a path labeled x from p to q in the underlying graph of the transition system $\mathcal{T}$. More precisely we define $\xrightarrow{x}{}^*$ inductively by saying $p \xrightarrow{\epsilon}{}^* p$ for all $p \in Q$, and $p \xrightarrow{xa}{}^* q$ whenever there exists $r \in Q$ such that $p \xrightarrow{x}{}^* r$ and $r \xrightarrow{a} q$. The language *accepted* (or *generated*) by the transition system $\mathcal{T}$ is defined to be $L(\mathcal{T}) = \{x \in A^* \mid s \xrightarrow{x}{}^* q \text{ for some } q \in Q\}$. We say the transition system $\mathcal{T}$ is *deterministic* if for each $p \in Q$ and $a \in A$, there is at most one $q \in Q$ such that $p \xrightarrow{a} q$.

A *finite state automaton* (FSA) over an alphabet A is of the form $\mathcal{A} = (Q, s, \longrightarrow, F)$ where $(Q, s, \longrightarrow)$ forms a transition system and $F \subseteq Q$ is a set of final states. The language accepted by $\mathcal{A}$ is defined to be $L(\mathcal{A}) = \{x \in A^* \mid s \xrightarrow{x}{}^* q \text{ for some } q \in F\}$.

It will be convenient for us to work with transition systems that generate behaviours that are sequences of alternating inputs and outputs. This assumption plays no technical role in our results, but simply makes it easier to understand the application to testing. More formally, we have a partition of the alphabet A into inputs A_I and outputs A_O, and require a transition system $\mathcal{T} = (Q, s, \longrightarrow)$ over A to satisfy the following conditions: the set of states Q is partitioned into input states Q_I and output states Q_O, s is required to belong to Q_I, and for every transition $p \xrightarrow{a} q$ in $\mathcal{T}$, $p \in Q_I$ implies $a \in A_I$ and $q \in Q_O$, and symmetrically, $p \in Q_O$ implies $a \in A_O$ and $q \in Q_I$. For an automaton $\mathcal{A}$, we will further assume its final states are all in Q_I, so that $L(\mathcal{A}) \subseteq (A_I \cdot A_O)^*$. We say a transition system is *input complete* (resp. *output complete*) if every input state (resp. output state) has transitions on every input (resp. output) letter. The transition system is *complete* if it is both input and output complete. A state q is called a *sink* state if for all inputs/outputs a, $q \xrightarrow{a} q$. For notational convenience, we allow sink states to be both input and output states.

In conformance testing, we are given a system specification in the form of a (possibly non-deterministic) transition system $\mathcal{S}$, and an "implementation under test" or IUT. A tester can interact with the IUT by alternately giving it inputs and receiving outputs from it. The IUT *conforms* to the specification if it is not possible for a tester to generate an interaction with it which violates the specification in the sense that the interaction does not belong to the set of input-output sequences generated by the specification. In the setting we consider, the tester is further given a *test purpose* in the form of a automaton. The function of the test purpose automaton is to narrow the set of interactions that the tester may have to explore. An interaction with the IUT now "passes" the test if the interaction is accepted by the test purpose automaton and produces no violation of the specification. An interaction "fails" the test if it produces a violation of the specification without "deviating" from the test purpose. Another possibility may also arise: an interaction may prove "inconclusive" with respect to the test, in that it "deviates" from the test purpose without violating the specification.

A *test case* in the terminology of [9] is a kind of "strategy" for the tester, which can be represented as a deterministic transition system, such that the input states have exactly one outgoing transition (its label representing an input to the IUT), and output states have transitions on all possible outputs. The transition system has sink states labeled mutually exclusively by "pass", "fail", and "inconclusive", which represent the verdict for any interaction with the IUT which is allowed by the test case.

A complete test graph (CTG) (for a given specification and test purpose) is a structure which represents all possible test cases in the sense that every test case can be obtained as a pruning of an unfolding of the test graph. A CTG may be described as a deterministic transition system which tells the tester at each step whether the test so far has passed, failed, proved inconclusive, or should be continued.

More formally, let us fix an input-output alphabet A, and let $\mathcal{S}$ be a given specification which we assume to be an input-complete transition system over A. Let $\mathcal{P}$ be a *test purpose automaton*, which is an automaton over A which is complete, and accepts a suffix-closed language. Then a *Complete Test Graph* for $\mathcal{S}$ wrt $\mathcal{P}$ is a complete, deterministic, transition system $C = (Q, s, \longrightarrow)$ over A, together with a labeling function $l : Q \to \{pass, fail, inconc, cont\}$, satisfying:

1. States labeled *pass*, *fail*, or *inconc* are all sink states
2. Let $x \in (A_I \cdot A_O)^*$, and let $s \xrightarrow{x}{}^* q$ (note that q is a unique such state). Then
   - $l(q) = pass$ iff $x \in L(\mathcal{S}) \cap L(\mathcal{P})$.
   - $l(q) = fail$ iff $x \notin L(\mathcal{S})$.
   - $l(q) = inconc$ iff $x \in L(\mathcal{S})$ and there does not exist a $y \in A^*$ such that $xy \in L(\mathcal{P})$.
   - $l(q) = cont$ iff $x \notin L(\mathcal{P})$ and there exists a $y \in A^*$ with $xy \in L(\mathcal{S}) \cap L(\mathcal{P})$.

To illustrate the notion of a CTG, consider the specification of a coffee/tea vending machine shown in Figure 1. The tester plays an input move: *?coin(2)* followed subsequently by a choice of *?coffee* or *?tea*. The machine has a choice of output moves at this stage: either to dispense the drink chosen by the customer (*!coffee* or *!tea*) or return the money (*!coin(2)*). Suppose the tester is interested in testing the coffee dispensing feature. This intention is expressed by the *!coffee* output move leading to the *Accept* state in the test purpose automaton and the *!tea* output move leading to a nonfinal sink state.

We briefly outline the procedure used in the TGV tool [9] to compute a CTG for a given specification $\mathcal{S}$ and test purpose $\mathcal{P}$. We first determinize $\mathcal{S}$ and $\mathcal{P}$ using the standard subset construction. We then take the synchronous product of $\mathcal{S}$ and $\mathcal{P}$, to obtain a transition system $\mathcal{S} \otimes \mathcal{P}$. In this transition system, we label each state with the tag 'L2A' if it can reach an accept state in the product (i.e. a product state in which the $\mathcal{P}$ component is a final state). This labeling is computed using Tarjan's algorithm for detecting strongly connected components. With the 'L2A' labelling, we can now obtain a CTG as follows. We introduce a new state called FAIL and make it a sink state. If a state is an accept state, label it *pass*. If a state is not labeled 'L2A', label it *inconc* and make it a sink state. If a state q is an output state, and if it does not have an outgoing transition on an output letter a, add a transition on a to the FAIL state. Label all remaining states as *cont*.

3 Hierarchical State Machines

We will use a variant of the formalism of hierarchical state machines presented in [3]. A *hierarchical labeled transition system* over an input-output alphabet A is a tuple $\mathcal{K} = \langle \mathcal{K}_1, .., \mathcal{K}_n \rangle$ of modules, where each module $\mathcal{K}_i$ is a structure of the form $(N_i, B_i, s_i, E_i, Y_i, R_i)$, where

- N_i is a finite set of *nodes*.

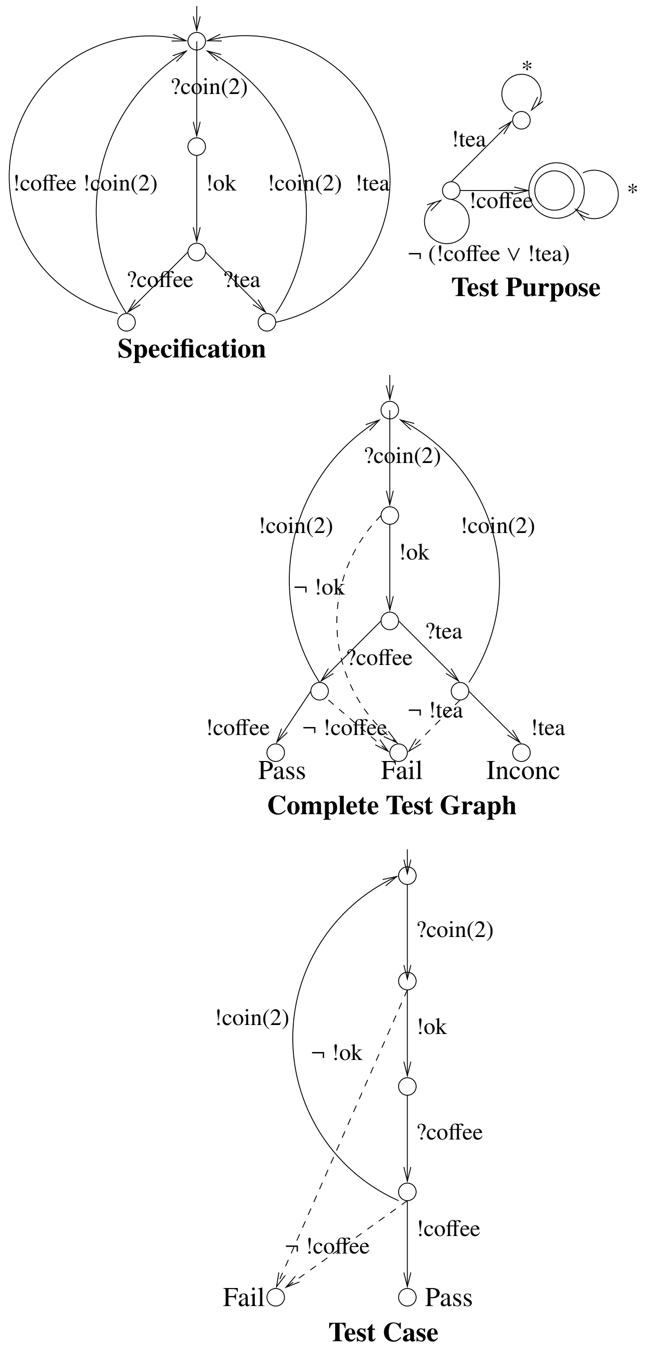

Figure 1. Complete Test Graph Example

- B_i is a finite set of *boxes*.
- $s_i \in N_i$ is the initial node.
- $E_i \subseteq N_i$ is a set of exit nodes.
- $Y_i : B_i \to \{i+1, .., n\}$ is an indexing function that maps each box in B_i to an index greater than i.
- R_i is the transition relation comprising triples of the following form:
  - (u, a, v) where $u, v \in N_i$, and $a \in A$
  - (u, a, b) where $u \in N_i$, $a \in A$, and $b \in B_i$
  - $(b \cdot e, a, v)$ where $b \in B_i$, $e \in E_j$ with $Y_i(b) = j$, $v \in N_i$ and $a \in A$; or
  - $(b \cdot e, a, c)$ where $b, c \in B_i$, $e \in E_j$ with $Y_i(b) = j$, and $a \in A$.

We assume that the N_i's and B_i's are mutually disjoint. We use N to denote $\bigcup_i N_i$, B to denote $\bigcup_i B_i$, and R to denote $\bigcup_i R_i$. The function *module*(u), where $u \in N \cup B$, returns the index of the module to which u belongs. We let $Y(b)$ denote $Y_{module(b)}(b)$.

The hierarchical LTS $\mathcal{K}$ represents a traditional LTS, which we call the "flattened" LTS and denote by $\mathcal{K}^F$. The transition system $\mathcal{K}^F$ is obtained from $\mathcal{K}_1$ by recursively substituting boxes by modules and adding appropriate transitions. Since several boxes can be mapped to the same module, each node in a module can appear in different "contexts" which we represent as a string of boxes.

Formally this can be done as follows. We inductively define for each module $\mathcal{K}_i$, the flattened transition system $\mathcal{K}_i^F$ corresponding to it. $\mathcal{K}_i^F$ is of the form $(Q_i, \longrightarrow_i, t_i)$, where $Q_i \subseteq B^* N$ defined by:

- If $u \in N_i$, then $u \in Q_i$.
- If $b \in B_i$, with $Y_i(b) = j$, and y is a state of $\mathcal{K}_j^F$, then $b \cdot y \in Q_i$.

The initial state t_i of $\mathcal{K}_i^F$ is s_i. The transition relation $\longrightarrow_i$ of $\mathcal{K}_i^F$ is given as follows:

- If $(u, a, v) \in R_i$, then $u \xrightarrow{a}_i v$.
- If $(u, a, b) \in R_i$ with $Y(b) = j$, then $u \xrightarrow{a}_i b \cdot s_j$.
- If $(b \cdot e, a, v)$, then $b \cdot e \xrightarrow{a}_i v$.
- If $(b \cdot e, a, c)$ with $Y(b) = j$, and $Y(c) = k$, then $b \cdot e \xrightarrow{a}_i c \cdot s_k$.
- If $b \in B_i$ with $Y(b) = j$, and $x \xrightarrow{a}_j y$, then $b \cdot x \xrightarrow{a}_i b \cdot y$.

$\mathcal{K}^F$ is now simply $\mathcal{K}_1^F$.

We use the term *nesting depth* of $\mathcal{K}$, denoted by $D(\mathcal{K})$, to denote the maximum length of a state (viewed as a string) in $\mathcal{K}^F$. We use $|\mathcal{K}_i|$ to denote the size of $\mathcal{K}_i$, and it is the sum of $|N_i|$, $|B_i|$ and $|R_i|$. The size of $\mathcal{K}$ is the sum of the sizes of $\mathcal{K}_i$. The size of $\mathcal{K}^F$ can be exponential in the nesting depth, i.e. $O(|\mathcal{K}|^{D(\mathcal{K})})$. We use $E(\mathcal{K})$ to denote the maximum of $|E_i|$, i.e. the maximum size of a set of exit nodes.

As done for non-hierarchical transition systems, we will assume that the nodes of a hierarchical transition system

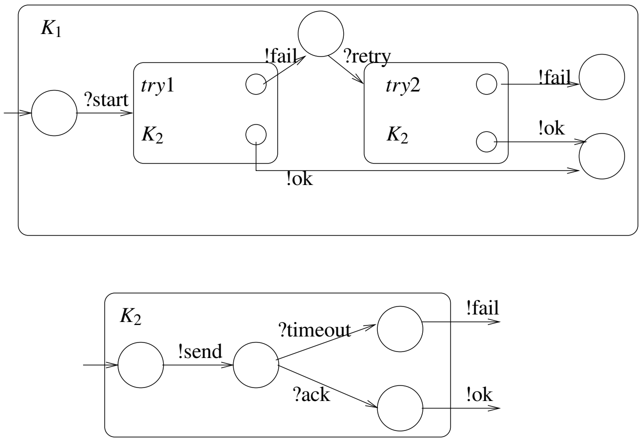

Figure 2. Sample hierarchical specification

are partitioned into input and output states, and that the flattened transition system corresponding to it generates only strings of alternating inputs and outputs.

We say a hierarchical transition system is deterministic (resp. input-complete, out-complete, or complete) iff its flattened transition system is deterministic (resp. input-complete, output-complete, and complete).

Figure 2 shows a specification of a system that makes two attempts to send a given message. The example is adapted from [3].

4 Computing a Hierarchical CTG

Given a deterministic, input-complete hierarchical specification $\mathcal{S}$, and a test purpose $\mathcal{P}$, a CTG for $\mathcal{S}$ wrt $\mathcal{P}$ is simply a CTG for $\mathcal{S}^F$ wrt $\mathcal{P}$. We are interested in obtaining a CTG which can be represented hierarchically, preferably mirroring the hierarchical structure of $\mathcal{S}$. The aim of this section is to describe the construction of such a CTG.

An important assumption we make is that the given spec $\mathcal{S}$ is *deterministic*. This is because, unlike the non-hierarchical case, there appears to be no simple way to determinize a hierarchical system in a way which preserves some of the original hierarchical structure. A second, less important, assumption we make is that each module has a unique exit node, denoted by e_i, and that this node is reachable from the start node s_i. This is only for conceptual simplicity and the technique can easily be seen to extend to multiple exit nodes.

To compute the hierarchical test graph, we first compute the product of the hierarchical specification $\mathcal{S}$ and the test purpose automaton $\mathcal{P}$, as described in [3]. In the product, all possible ways of pairing the entry node of every module with every state of TP is considered. The product $\mathcal{S} \otimes \mathcal{P}$ is a hierarchical machine with nm structures, where m is the number of states of $\mathcal{P}$. A state (q, t) of $\mathcal{S} \otimes \mathcal{P}$ is an *accept* state if t is an accept state of $\mathcal{P}$. If we can now label nodes in $\mathcal{S} \otimes \mathcal{P}$ with 'L2A' ("leads to accept") information, then as in the non-hierarchical case, we can obtain a CTG by adding a FAIL node to each module, and proceeding similarly.

We should point out that a hierarchical transition system may not always admit an 'L2A' labelling on its nodes. For example in the hierarchical transition system of Fig. 3, state 8 in $\mathcal{K}_2$ leads to accept in context 2, whereas it does not lead to accept in context 3.

Thus we now concentrate on showing that given a hierarchical transition system $\mathcal{K}$ with a set of "accept" nodes T, we can give a hierarchical transition system $\mathcal{H}$ which satisfies (a) its flattened transition system $\mathcal{F}$ is isomorphic to $\mathcal{K}^F$, and preserves reachability to accept states; and (b) it admits a context-independent 'L2A' labelling of its nodes. For the rest of this paper, we fix $\mathcal{K}$ with components as defined in section 3, and a $T \subseteq N$.

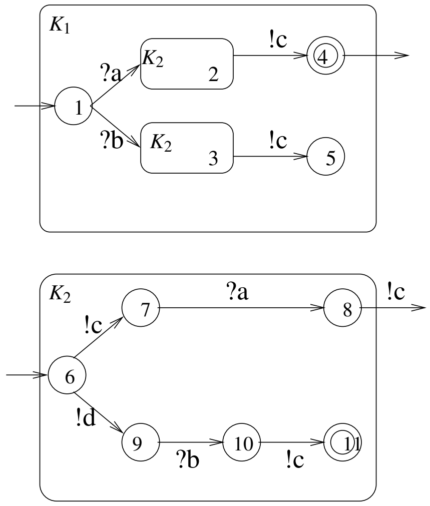

Figure 3. Hierarchical Transition System With Accept Nodes

We now define set $REACH\_IN_i$ which contains nodes/boxes in $\mathcal{K}_i$ that can lead to accept within its flattened module $\mathcal{K}_i^F$. We also define set $REACH\_EXIT_i$ which contains nodes/boxes in $\mathcal{K}_i$ that can reach the exit node e_i. The set $REACH\_EXIT_i$ is useful in determining whether a node or box can lead to accept outside its module.

Definition 1. *For each i, we inductively define $REACH\_IN_i$ as a subset of $N_i \cup B_i$ containing all nodes/boxes w such that w leads to accept in $\mathcal{K}_i$ or w can reach a box b in $\mathcal{K}_i$ with $Y(b) = j$ and $s_j \in REACH\_IN_j$.*

Definition 2. *For each i, $REACH\_EXIT_i$ is a subset of $N_i \cup B_i$ containing all nodes/boxes w which can reach e_i in $\mathcal{K}_i$.*

The algorithm shown in Figure 4 computes the sets $REACH\_IN_i$ and $REACH\_EXIT_i$. Procedure $SCC(in_i)$ is invoked (line 5) to compute the *strongly connected components* of module K_i using Tarjan's algorithm [1]. At this time, $VISITED$ is empty. Therefore, all nodes or boxes in K_i reachable from in_i will be visited. The procedure SCC treats boxes like ordinary nodes except for lines 16-18. Viewing K_i as an ordinary directed graph, it can be used to correctly compute the strongly connected components of K_i. Since all nodes in a component have the same reachability property, if we determine that the root of a component can reach t, then all nodes in that component can also reach t. Also, when K_i is searched, if a box b is encountered, then $Y_i(b) > i$.

Lemma 3. *Let $u \in N_i$. Then u leads to accept in $\mathcal{K}_i^F$ iff $u \in REACH\_IN_i$.*

Proof. We prove Lemma 3 by backward induction on i.

Base Case ($i = n$): K_n has no boxes. Therefore, K_n^F is the same as K_n. By definition 1, u leads to accept in $\mathcal{K}_n^F$ iff $u \in REACH\_IN_n$.

Inductive Step ($i < n$): Suppose u leads to accept in $\mathcal{K}_i^F$. In the first case, the accept state is in $\mathcal{K}_i$. By definition 1, $u \in REACH\_IN_i$. In the second case, the accept state lies inside some box b with $Y(b) = j$ and $i < j$ in $\mathcal{K}_i$ and u can reach b. By inductive hypothesis, $s_j \in REACH\_IN_j$ and therefore by definition 1, $u \in REACH\_IN_i$. Conversely, suppose that $u \in REACH\_IN_i$. Then u leads to accept either in $\mathcal{K}_i$ or u can reach a box b with an initial node that leads to accept. In either case, u leads to accept in $\mathcal{K}_i^F$. □

Lemma 4. *Let $b \in B_i$ with $Y_i(b) = j$. Then $b \cdot e_j$ leads to accept along a path that exits $\mathcal{K}_j$ in $\mathcal{K}_i^F$ iff $b \in REACH\_IN_i$.*

Proof. ($\Rightarrow$): In the first case, the accept state is a node of K_i. By definition 1, $b \in REACH\_IN_i$. In the second case, b can reach an accept state that lies inside another box c with $Y(c) = k$. By lemma 3, $s_k \in REACH\_IN_k$ and therefore by definition 1, $b \in REACH\_IN_i$.

($\Leftarrow$): In the first case, the accept state is a node of K_i. In the second case, b can reach an accept state that lies inside another box. In both cases, $b \cdot e_j$ leads to accept along a path that exits $\mathcal{K}_j$ in $\mathcal{K}_i^F$. □

We can now define a hierarchical state machine $\mathcal{H}$ which admits a local "L2A" labeling.

Definition 5. *We define $\mathcal{H}$ as follows. $\mathcal{H} = \langle \mathcal{K}_{1,0}, \mathcal{K}_{1,1}, .., \mathcal{K}_{n,0}, \mathcal{K}_{n,1} \rangle$ where each module $\mathcal{K}_{i,0}$ has the components $(N_i \times \{0\}, B_i \times \{0\}, (s_i, 0), (e_i, 0), Y_{i,0}, R_{i,0})$. The indexing function $Y_{i,0}$ and the transition relation $R(i, 0)$ are defined as follows. Let $b \in B_i$ with $Y(b) = j$.*

$$Y_{i,0}((b,0)) = \begin{cases} 2j & \text{if } b \in REACH\_IN_i \\ 2j-1 & \text{otherwise} \end{cases}$$
$$R_{i,0} = \{((u,0), a, (v,0)) \mid (u,a,v) \in R_i\}$$

Each module $\mathcal{K}_{i,1}$ has the components $(N_i \times \{1\}, B_i \times \{1\}, (s_i, 1), (e_i, 1), Y_{i,1}, R_{i,1})$.

$$Y_{i,1}((b,1)) = \begin{cases} 2j & \text{if } b \in REACH\_IN_i \\ & \text{or } b \in REACH\_EXIT_i \\ 2j-1 & \text{otherwise} \end{cases}$$
$$R_{i,1} = \{((u,1), a, (v,1)) \mid (u,a,v) \in R_i\}$$

Definition 6. *For each i, $f_i : \mathcal{K}_i^F \rightarrow \mathcal{K}_{i,0}^F$ and $g_i : \mathcal{K}_i^F \rightarrow \mathcal{K}_{i,1}^F$ are defined inductively as follows ($Y(b) = j$):*

$$f_i(u) = (u, 0)$$
$$f_i(bxu) = \begin{cases} (b,0)\, g_j(xu) & \text{if } b \in REACH\_IN_i \\ (b,0)\, f_j(xu) & \text{otherwise} \end{cases}$$
$$g_i(u) = (u, 1)$$
$$g_i(bxu) = \begin{cases} (b,1)\, g_j(xu) & \text{if } b \in REACH\_IN_i \\ & \text{or } b \in REACH\_EXIT_i \\ (b,1)\, f_j(xu) & \text{otherwise} \end{cases}$$

It is not difficult to argue that:

Lemma 7. *For each i, f_i and g_i are isomorphisms from $\mathcal{K}_i^F$ to $\mathcal{K}_{i,0}^F$ and $\mathcal{K}_i^F$ to $\mathcal{K}_{i,1}^F$ respectively. Also, f_i and g_i preserve reachability to accept states in $\mathcal{K}_{i,0}^F$ and $\mathcal{K}_{i,1}^F$ respectively.*

□

Definition 8. *We define a function l2a as follows:*

$$l2a((u,l)) = \begin{cases} true & \text{if } u \in REACH\_IN_i \\ & \text{or } (u \in REACH\_EXIT_i \wedge (l = 1)) \\ false & \text{otherwise} \end{cases}$$

Lemma 9. *Let $x'b'(e,l)$ be a state in $\mathcal{H}^F$ ($Y(b) = j$). Then $x'b'(e,l)$ leads to accept via an edge that exits $\mathcal{K}_{j,l}$ (i.e. $x'b'(e,l) \xrightarrow{a} y'u' \longrightarrow^* Accept$ and $x'b' \not\leq y'$) iff $l = 1$.*

Proof. Base Case ($|x'| = 0$): b' is a box $(b, 0)$ in $K_{1,0}$ that can reach accept via an edge that exits it. By lemma 7, b can reach accept in K^F. By lemma 3, $b \in REACH\_IN_1$ which implies that b is mapped to an even index (definition 5). Therefore, by definition 6, the exit node of b is (e,1). Conversely, if the exit node of b' is $(e, 1)$, by definition 5, $b \in REACH\_IN_1$. By lemma 3, b leads to accept in $\mathcal{K}^F$. By lemma 7, b' will lead to accept in $\mathcal{H}^F$.

Inductive Step The inductive step can be done in a similar manner. □

Input: Hierarchical Machine K, Target Region T
Output: $REACH\_IN_i, REACH\_EXIT_i$ ($\forall i = 1..n$)

```
for i = n downto 1 do
    count := 1
    SCCSTACK := φ
    VISITED := φ
    SCC(in_i)
    REACH_IN_i := {u ∈ N_i ∪ B_i | u.reach_in = true}
    REACH_EXIT_i := {u ∈ N_i ∪ B_i | u.reach_exit = true}

procedure SCC(u)
VISITED := VISITED ∪ {u}
u.lowlink := u.dfnumber := count
count := count + 1
i := module(u)
if u ∈ N_i then
    u.reach_in := (u ∈ T)
    u.reach_inside_box := false
else
    j := Y_i(u)
    u.reach_in := false
    u.reach_inside_box := (s_j ∈ REACH_IN_j) {u.reach_inside_box is true iff its initial node can reach a target inside
    its module}
u.reach_exit := (u = e_i)
push u on SCCSTACK
for all (u, a, v) ∈ R_i do
    if v ∉ visited then
        SCC(v)
        u.lowlink := MIN(u.lowlink, v.lowlink)
        u.reach_in := u.reach_in ∨ v.reach_in ∨ v.reach_inside_box
        u.reach_exit := u.reach_exit ∨ v.reach_exit
    else
        if v.dfnumber < u.dfnumber then {(u, a, v) is a back edge or cross edge}
            if v is on SCCSTACK then
                u.lowlink := MIN(v.dfnumber, u.lowlink)
            else {v is a node in a different component and (u, a, v) is a cross edge}
                u.reach_in := u.reach_in ∨ v.reach_in ∨ v.reach_inside_box
                u.reach_exit := u.reach_exit ∨ v.reach_exit
{Exploration from u is over now}
if u.lowlink = u.dfnumber then {u is the root of a new SCC}
    current := new Component()
    make u the root of current
    repeat
        pop x from STACK
        put x in current
        x.reach_in := u.reach_in
        x.reach_exit := x.reach_exit
    until x = u
    {Set up edges of the component graph}
end SCC
```

Figure 4. Compute Reach Sets

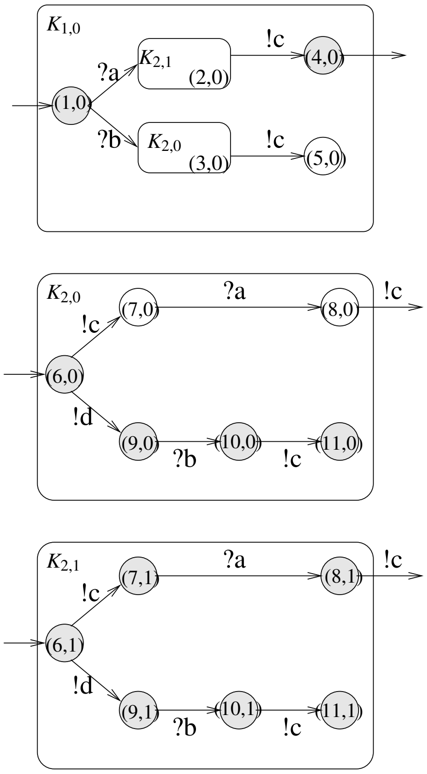

Figure 5. Shaded nodes lead to accept

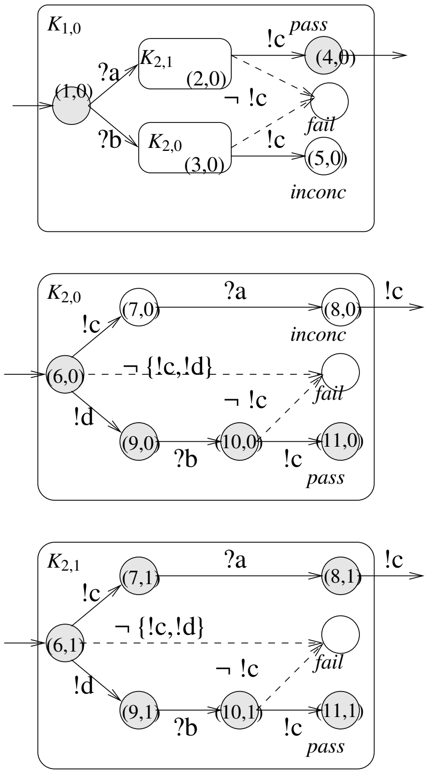

Figure 6. Hierarchical CTG

The following theorem now follows easily using lemma 9 and the definition of the *l2a* function.

Theorem 10. *Let $x'v'$ be a state in $\mathcal{H}^F$. Then $x'v'$ leads to accept in $\mathcal{H}^F$ iff $l2a(v') = true$.* □

Fig 5 shows the hierarchical LTS $\mathcal{H}$ corresponding to the hierarchical LTS in figure 3, along with the local 'L2A' labelling.

Fig. 6 shows the hierarchical CTG for a system and test purpose whose product is the hierarchical LTS in Fig. 3. The nodes are labeled by *pass*, *fail*, and *inconc* (all other nodes are assumed to be labeled *cont*). A node is labeled *pass* if it is an accept state, *inconc* if it does not lead to accept, and transitions to *fail* are added from output states on all output letters for which an outgoing transition does not exist.

5 Constructing a Test Oracle

In this section we consider the problem of constructing an "online" test oracle (once again with respect to a specification $\mathcal{S}$ and test purpose $\mathcal{P}$) that given an input-output interaction of the tester with the IUT, returns a *pass*, *fail*, *inconc*, *cont* verdict.

We can of course use the CTG constructed in the previous section to do this, by moving deterministically in the CTG each time the tester/IUT provides us with an input/output. However, though each step takes a constant amount of time, we will need to keep track of a context in $\mathcal{H}$, which could be as much as $E(\mathcal{K}) \cdot D(\mathcal{K})$ bits more compared to a state of $\mathcal{K}$ (here $\mathcal{K}$ is $\mathcal{S} \otimes \mathcal{P}$). Instead we can keep track of a context in $\mathcal{K}$ itself, and use $O(D(\mathcal{K}))$ amount of time in each step to give an appropriate verdict to the tester. This algorithm is given in figure 7. Note that the sets $REACH\_IN_i$ and $REACH\_EXIT_i$ are precomputed in time linear in $|\mathcal{K}|$.

To argue the correctness of the algorithm, we first observe the following claim:

Lemma 11. *The function L2A-Oracle returns **true** on an input xw iff w is a node and xw leads to accept in $\mathcal{K}^F$, or w is a box with $Y(w) = j$ and $xw \cdot e_j$ leads to accept in $\mathcal{K}^F$.*

Proof. By induction on $|x|$. □

This lets us conclude the correctness of the algorithm:

Theorem 12. *Let xu be a state in $\mathcal{K}^F$. Then L2A-Oracle(xu) returns **true** iff xu leads to accept in $\mathcal{K}^F$.* □

Input: Node or box w of $N \cup B$ and a context x.
Output: Returns **true** iff w is a node and xw leads to accept in $\mathcal{K}^F$, or w is a box with $Y(w) = j$ and $xw \cdot e_j$ leads to accept in $\mathcal{K}^F$.

function boolean L2A-Oracle($x \cdot w$)

$i := module(w)$
if $w \in REACH\_IN_i$ **then**
    return **true**
else if $x = \epsilon$ **then** $\{w \in N_1 \bigcup B_1\}$
    return **false**
else if $w \in REACH\_EXIT_i$ **then**
    return L2A-Oracle(x);
else
    return **false**

end L2A-Oracle

Figure 7. A Test Oracle

6 Conclusion

In this paper, we have shown how to compute the complete test graph for deterministic hierarchical specifications efficiently. This avoids flattening which could result in a much larger graph. We also describe how one could build online test oracles. These test oracles could be useful even in the case of nondeterministic specifications.

The complete test graph can serve as a basis for test case generation algorithms that achieve the desired level of coverage. In the future, we plan to study coverage algorithms for generating test cases from hierarchical complete test graphs.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[2] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *ICALP*, volume 1644 of *Lecture Notes in Computer Science*, pages 169–178. Springer, 1999.

[3] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3):273–303, 2001.

[4] A. Blass, Y. Gurevich, L. Nachmanson, and M. Veanes. Play to Test. Technical Report MSR-TR-2005-04, Microsoft Research, Redmond, WA, January 2005.

[5] D. D'Souza and M. Gopinathan. Computing Complete Test Graphs for Hierarchical Systems. Technical Report CSA-IISc-TR-2006-2, Dept. of CSA, IISc, Bangalore, India, April 2006.

[6] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Trans. Software Eng.*, 22(6):363–377, 1996.

[7] J. Heumann. Generating test cases from use cases. *The Rational Edge*, June 2001.

[8] C. Jard and T. Jéron. Tgv: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)*, 6, october 2004.

[9] T. Jéron and P. Morel. Test generation derived from model-checking. In N. Halbwachs and D. Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 108–121. Springer, 1999.

[10] Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat, and M.-L. Potet. Test purposes: Adapting the notion of specification to testing. In *ASE*, pages 127–134. IEEE Computer Society, 2001.

[11] T. Lengauer and E. Wanke. Efficient solution of connectivity problems on hierarchically defined graphs. *SIAM J. Comput.*, 17(6):1063–1080, 1988.

[12] D. Mosk-Aoyama and M. Yannakakis. Testing hierarchical systems. In *SODA*, pages 1126–1135. SIAM, 2005.

[13] M. Yannakakis. Testing, optimizaton, and games. In J. Díaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *ICALP*, volume 3142 of *Lecture Notes in Computer Science*, pages 28–45. Springer, 2004.