An Abstract Schema for Equivalence-Checking Games

Li Tan

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11790 USA
tanli@cs.sunysb.edu

Abstract. Equivalence games have been shown as an efficient way to diagnose design systems. Nevertheless, like other diagnostic routines, equivalence games utilize the information already computed by equivalence checker during verification. Therefore, these diagnostic routines tightly gear to the data structure of checker being used, and their ability of migrating to a different checker is not always guaranteed. Moreover, different equivalence relations demand different game schemas, which makes it tedious to implement equivalence games. We solve the first problem by utilizing a generalized version of partition refinement tree (PRT) as an abstract of proof structures. With a little bookkeeping, a partition refinement-based checker is able to supply PRT as the evidence to support its result. The diagnostic routines built on PRTs are independent of equivalence checkers being used. PRTs may also be used to certify the equivalence-checking result.
To solve the second problem, we introduce a *semantics hierarchy*. Implementation following this hierarchy enjoys greater code sharing among different games. The prototype of this schema, including PRT-friendly algorithms and the architecture of semantics hierarchy, has been implemented on the Concurrency Workbench.

1 Introduction

The ability of generating diagnostic information is an important feature of a verification tool. In the case of equivalence checking, a tool usually returns a textual-based property satisfied by one process but not by the other. This could be a Hennessy-Milner logic formula in (weak-) bisimulation [HM85], a trace in language equivalence, or a failure in testing-based equivalence [Mai87]. Nevertheless, this information lacks of intuition, and is often inadequate for spotting errors in design systems.

Equivalence game [Sti87] has been shown to be an efficient way to help the user understand why or why not two processes are related. In equivalence game, user plays against computer to challenge the verification result. Computer convinces user about the correctness of result by showing that he has a strategy to win each and every play, no matter how user reacts. Nevertheless, there are certain difficulties in designing and implementing game semantics: first, like other diagnostic routine, the construction of winning strategies utilize the information already computed during equivalence checking. Building games on a checker requires case-by-case study on the proof structure of individual checker, i.e., the data structure by which a checker reaches its result; second, each equivalence relation demands a unique definition of game. Implementing each game semantics separately turns out to be very tedious work.

We start to solve the first problem by abstracting in a "standard" form the proof structures of checkers, so diagnostic routines built on these abstract proof structures can migrant between checkers effortlessly. The abstract proof structure we are looking for turns out to be a generalized version of partition refinement tree (forest) (PRT). PRT serves as the evidence to support the result of equivalence checking. With some bookkeeping effort, a partition refinement-based algorithm can produce PRTs without compromising its complexity. PRT may also be used to certify the equivalence-checking result. Our study shows that PRT provides sufficient information for producing many diagnostics, including Hennessy-Miller logic formula in (weak-) bisimulation, trace in language equivalence, and failure suite in testing-based equivalence. Particularly in this paper we consider how to build equivalence-checking games on PRTs.

Different equivalence semantics demand different game schemas. Implementing them separately turns out to be a very tedious work. We introduce *semantic hierarchy* to promote code sharing between games, and make it easy to add a new game. The semantics hierarchy consists of three levels: at the top is an *abstract game*, which includes the common functionalities shared by all the (property-checking) games; in the middle is *equivalence game module*(*EQM*), which subsumes all the equivalence games; and on the bottom is semantics plugins (*plugins*), which include those functions unique for each individual game. The implementation following this hierarchy greatly improves the code sharing among games at appropriate levels. For example, one of *EQM*'s functions is to build *abstract* winning strategy by tracing PRTs. Semantics plug-ins will interpret this abstract winning strategy as the winning strategy for the targeting equivalence game. Therefore, each game don't have to implement its own function to handle PRTs. In most cases, to introduce a new equivalence game one only needs to supply relatively small *plugins*.

The introduction of PRTs and semantics hierarchy are two features making this schema distinct. Our experiments on CWB-NC has showed the combination of two dramatically reduced the implementation cost while adding flexibility and other nice features, including uniform interfaces and centralized bookkeeping.

This paper chooses bisimulation game and language equivalence game as examples to show how they can fit to this schema. The discussion on more complicate case of failure- and testing- equivalence games is left to the full version of this paper, available at `www.cs.sunysb.edu/~ tanli/bisgame.ps`. The rest of paper is organized as follows: section 2 prepares notations and definitions; section 3 introduces an abstract version of partition refinement tree as the standardized interface data structure between checkers and upper-level diagnostic routines. Section 4 introduces *semantics hierarchy.* Finally, section 5 discusses a prototype implemented on Concurrency Workbench of the New Century.

2 Preliminaries

2.1 Transition System and Equivalence Relations

We model the processes as labelled transition systems (*LTSs*). A labelled transition system is a triple $\langle S, A, \rightarrow \rangle$, where S is the set of states, A is the a set of actions containing visible actions $A - \{\tau\}$ and an internal action τ, and $\rightarrow \subseteq S \times A \times S$ is the transition relation. We shall write $s \xrightarrow{a} s'$ in lieu of $\langle s, a, s' \rangle \in \rightarrow$. We write $s \xrightarrow{a}$ if there exists a s' such that $s \xrightarrow{a} s'$, and $s \not\xrightarrow{a}$ if such s' doesn't exist. We denote $s \xrightarrow{a} \bullet$ for the set $\{s' \mid s \xrightarrow{a} s'\}$. We lift the notation of $\rightarrow$ to sets of states straightforwardly: $P \xrightarrow{a} P'$ iff $P' = \{s' \mid \exists s \in P.s \xrightarrow{a} s'\}$. A state s is stable if $s \not\xrightarrow{\tau}$. We refer to $s \xrightarrow{\bullet} = \{a \mid s \xrightarrow{a}\}$ as the set of initial actions of s. The notion of deterministic and non-deterministic transition systems is defined straightforwardly based on the behavior of their outgoing transitions.

Weak (Observational) transition relation $\Rightarrow$ is defined as below. Let $\beta \in (A - \{\tau\})^*$ be a sequence of visible actions, then,

1. $\overset{\epsilon}{\Rightarrow} = \xrightarrow{\tau^*}$, where $\xrightarrow{\tau^*}$ is the transitive and reflexive closure of $\xrightarrow{\tau}$.
2. $\overset{a\beta'}{\Rightarrow} = \overset{\epsilon}{\Rightarrow} \circ \xrightarrow{a} \circ \overset{\beta'}{\Rightarrow}$. where $\circ$ denotes relational composition.

We extend the notion of $\Rightarrow$ to sets of states: $P \overset{\beta}{\Rightarrow} P'$ iff $P' = \{s' \mid s \overset{\beta}{\Rightarrow} s'\}$. We refer to P^ϵ as P's ϵ-closure $\{s' \mid s \overset{\epsilon}{\Rightarrow} s'\}$. β is a trace of s_0 if there exists s' such that $s_0 \overset{\beta}{\Rightarrow} s'$. The language of $s_0 \in S$, written as $L(s_0)$, is defined as all the traces of s_0.

We now define semantics equivalences that relate processes (states in a transition system) on the basis of their behavior. In what follows, we fix the labelled transition transition $T = \langle S, A, \rightarrow \rangle$.

Definition 1. *Given a relation $\Pi \subseteq S \times S$, $\sim_\Pi$ is a Π-bisimulation iff $\sim_\Pi \subseteq \Pi$ and $s_0 \sim_\Pi s_1$ implies the following,*

1. $s_0 \xrightarrow{a} s_0' \Leftrightarrow \exists s_1'.s_1 \xrightarrow{a} s_1' \wedge s_0' \sim_\Pi s_1'$.
2. $s_1 \xrightarrow{a} s_1' \Leftrightarrow \exists s_0'.s_0 \xrightarrow{a} s_0' \wedge s_1' \sim_\Pi s_0'$.

Note that if $\Pi = S \times S$ then a Π-bisimulation is a bisimulation in the usual sense [Mil89].

Definition 2. *Let s_0 and s_1 be two states of T. $s_0 =_{may} s_1$ iff $L(s_0) = L(s_1)$.*

$=_{may}$ is also called *language equivalence relation.*

2.2 Computing Other Equivalence Relations as Bisimulation

[CH93] showed that many other equivalence relations, including language and testing equivalences, may be reduced to Π-bisimulation. Their idea is to first transform the original transition system to a new one which they call *reference transition system*. Computing an equivalence relation on original system is equivalent to compute a Π-bisimulation on *reference transition system*. [CH93] also suggests a family of reference transition systems, called *acceptance graph*, for language equivalence, failure equivalence, and testing equivalence. An *acceptance graph* is a deterministic transition system whose states are pairs of boolean value and set of states. In addition, each state q is labelled with a set of sets of actions $q.acc$. An *acceptance graph* for testing T on language equivalence, written $D(T) = \langle S_r, A_r, \rightarrow\rangle$, may be defined as below,

1. $S_r = \{\langle Q, false\rangle \mid Q = Q^\epsilon\}$
2. For each $t \in S_r$, $t.acc = \emptyset$
3. For $t_1 = \langle Q_1, b_1\rangle$ and $t_2 = \langle Q_2, b_2\rangle$, $t_1 \xrightarrow{a} t_2$ exactly when the following properties hold.
   (a) $a \neq \tau$
   (b) $(Q_1 \overset{a}{\Rightarrow} \bullet)^\epsilon = Q_2$

In other words, $D(T)$ is just the deterministic version of T. We denote $t.closed = b$ for the boolean value associated with the node $t = \langle Q, b\rangle$. $D(T)$ is a very simple acceptance graph in the sense that both $t.closed$ and $t.acc$ are trivial: they are always $false$ and $\emptyset$. We keep these two fields for the reason of compatibility. Testing and failure equivalence games need these two fields for maintaining divergence information.

The mapping function g associating each states in original system with a state in reference system can be defined as $g(p) = \langle\langle\{p\}^\epsilon\rangle, false\rangle$. Let $L(p)$ and $L(g(p))$ be the language accepted by p in T and that accepted by $g(p)$ in $D(T)$. Since $D(T)$ is merely the deterministic version of T on the relation $\Rightarrow$, we have $L(p) = L(g(p))$.

Theorem 1. *[CH93] $s_0 =_{may} s_1$ if and only if $g(s_0) \sim_\Pi g(s_1)$, where $g(s)$ is the state related to s in $D(T)$, and $\Pi = \{\langle t_1, t_2\rangle \mid t_1.acc = t_2.acc\}$.*

Practically, if we only want to know whether $s_0 =_{may} s_1$, we only need to construct the part of $D(T)$ reachable from $g(s_0)$ or $g(s_1)$.

3 Partition Refinement Tree

A binary tree D is prefix-closed subset of $\{0, 1\}^*$, where λ, the empty sequence, is the root of the tree. An edge is a tuple $\langle d, d.i\rangle \in E$ such that $d, d.i \in Q$. d is an internal node if d is the non-trivial prefix of some $d' \in D$, and it is a leaf otherwise. A binary tree D is complete if both $d.0$ and $d.1$ are in D for every internal node d. A forest of binary trees is the set of binary trees. We shall write $k\beta$ for the node β in k-th subtree.

In what follows, we fix the transition system $T = \langle S, A, \rightarrow \rangle$. We refer to $P \subseteq S$ as *stable* to $P' \subseteq S$ on action $a \in A$ iff either $Q = P$ or $Q = \emptyset$, where $Q = \{s \in P \mid \exists s' \in P'. s \xrightarrow{a} s'\}$. $P \subseteq S$ is *stable* to P' iff P is stable to P' on any action $a \in A$.

Definition 3. *A partition refinement tree (PRT) $\gamma = \langle D, \ell, \ell_s, \ell_a, T \rangle$ is a labelled complete binary tree D defined with respect to a LTS $T = \langle S, A, \rightarrow \rangle$. $\ell : D \rightarrow 2^S - \emptyset$ labels each node with a non-empty subset of S. $\ell_a : D \rightarrow A$ and $\ell_s : D \rightarrow D$ associates each* internal node *with an action and a node, respectively. In addition, ℓ_a and ℓ_s satisfies the following properties,*

1. *$\ell(\epsilon) = S$.*
2. *For each internal node d, $\ell(d.0) = \{s \in \ell(d) \mid \exists s' \in \ell_s(d). s \xrightarrow{\ell_a} s'\}$, and $\ell(d.1) = \ell(d) - \ell(d.0)$.*
3. *There is a well-founded order ω on D such that, for any $d \in D$, $d < d.i$ and $d \geq \ell_s(d)$ on ω.*

A node d is stable *to node d' and action a iff $\ell(d)$ is stable to $\ell(d')$ and a. A partition refinement tree is* stable *if, for any leaves q_1 and q_2, $\ell(q_1)$ is stable to $\ell(q_2)$ on any action.*

We refer to $\ell_s(q)$ and $\ell_a(q)$ as the *splitter* and *splitting action* of q. With the clear context we may avoid the formal discrimination of a node q and the set $\ell(q)$ it represents. Intuitively, when q splits with respect to a splitter $\ell_s(q)$ and a splitting action $\ell_a(q)$, its left child $\ell(q.0)$ contains those states of q which can make $\ell_a(q)$ transition to some states of $\ell_s(q)$, and the right child $q.1$ contains the rest of states in q which are not able to do so.

Most of properties of PRTs are clear to those reader who are familiar with the partition refinement-based algorithms (cf. [KS83]), except (3). The well-founded order ensures that the final partition represented by leaves is not "finer" than the partition induced by bisimulation. We need to introduce some definitions before we give the formal proof.

A stable partition on S is a partition Γ such that a block $P \in \Gamma$ is stable to any block $P' \in \Gamma$. A partition Γ is coarser than Γ', written $\Gamma' \sqsubseteq \Gamma$, iff for each $\mathbf{P} \in \Gamma$, there exists a $P' \in \Gamma'$ such that $P' \subseteq P$.

Lemma 1. *[PT87]*

1. *There is only one coarsest stable partition for any transition system, and*
2. *The coarsest stable partition is just the partition induced by bisimulation.*

Theorem 2. *Given a stable partition refinement tree $\gamma = \langle D, \ell, \ell_s, \ell_a, T \rangle$, $\Gamma_\gamma = \{\ell(q) \mid q \text{ is a leaf}\}$ is the partition induced by the bisimulation relation $\sim$.*

Proof. Let Γ is the coarsest stable partition. We need to show $\Gamma_\gamma = \Gamma$. Clearly from the definition of PRTs $\Gamma_\gamma \sqsubseteq \Gamma$. We only need to show that $\Gamma \sqsubseteq \Gamma_\gamma$.

We choose to prove a stronger condition: γ respects partition Γ, i.e., each node of γ is an union of some blocks in Γ. We use the contradiction to show this.

Let w be the well-founded order on D implied by the definition. Clearly the first node on w, the root, satisfies this. Assume that $d.k$ is the first node on w which violates the condition. We consider the case $k = 0$, while the case $k = 1$ can be proven similarly.

The assumption that $d.0$ is not the union of some classes in Γ implies the existence of s_i and s_j such that $s_i \in \ell(d.0)$, $s_j \notin \ell(d.0)$, and $\{s_i, s_j\} \subseteq P$ for some $P \in \Gamma$. Since d always appear before $d.i$ on w, s_i and s_j must be both in $\ell(q)$. Therefore, $s_j \in \ell(d.1)$. Moreover, By lemma 1, $\{s_i, s_j\} \subseteq P$ implies $s_i \sim s_j$.

By (2) in definition 3, there is a state $s'_i \in \ell(\ell_s(d))$ such that $s_i \xrightarrow{\ell_a(d)} s'_i$. Nevertheless, $s_i \sim s_j$ implies the existence of a s'_j such that $s_j \xrightarrow{\ell_a(d)} s'_j$ and $s'_i \sim s'_j$. Since d's splitter $\ell_s(d)$ cannot appear after d on w, $\ell_s(d)$ shall appear before $d.k$. Hence $\ell_s(d)$ should respect Γ by the assumption. It follows that s'_i and s'_j should be both in $\ell(\ell_s(d))$ because $s'_i \in \ell(\ell_s(d))$ and $s'_i \sim s'_j$. Therefore, $s_j \in \ell(d)$ can make an action $\ell_a(d)$ to a state in $\ell(\ell_s(d))$, hence $s_j \in \ell(d.0)$, which contradicts to the fact that $s_j \notin \ell(d.0)$. Thus, we proved that γ should respect Γ.

A partition refinement forest for a Π-bisimulation can be defined similarly. Each tree in the forest is a partition refinement tree, with one modification: the root of each tree now represents a unique block in the partition induced by Π instead of the set of all the states. One may prove in a similar manner that the leaves of a stable partition refinement forest implies the partition induced by $\sim_\Pi$.

Generating Partition Refinement Trees and Forests An execution of a partition refinement-based checker usually implies a partition refinement tree (forest). All we need to do is to construct a PRT from the information already existing in the internal data structures of checkers. To avoid too much details, we consider only a naive partition refinement-based algorithm. More sophistic refinement-based algorithms such as the three-way splitting algorithm [PT87], and Kernel-Auxiliary partition algorithm [TC01], may produce the refinement tree (forest) with similar modification.

Procedure $bisim(s_0, s_1, T = \langle S, A, \rightarrow \rangle)$
  $\Gamma = \{S\}$
  while Γ is not stable **do**
    Choose $P, P' \in \Gamma$ and an action a such that
    P is not stable w.r.t. to P and a.
    Replace P in Γ by $P_1 = \{s \in P \mid \exists s'.s \xrightarrow{a} s'\}$ and $P_2 = P - P_1$.
    if s_0 and s_1 in different blocks **then**
      return **false**
  return true.

We construct a PRT $\gamma = \langle D, \ell, \ell_s, \ell_a \rangle$ during the execution of $bisim(s_0, s_1)$. The main iteration maintains the following invariant for Γ, *For each block P in Γ, there is a leaf d in the current configuration of γ such that $\ell(d) = P$*

Initially, γ has only one node ϵ with the label $\ell(\epsilon) = S$. Assume that P has been split by P' and a. Let d and d' be the current leaves related to P and P' by the above invariant. We create two children for d such that $\ell(d.0) = P_1$ and $\ell(d.1) = P_2$. Meanwhile, we also mark the splitter and splitting action for d as $\ell_s(d) = d'$ and $\ell_a(d) = a$.

Clearly the resulting $\mathsf{PRT}\gamma$ satisfies (1) and (2) of definition 3. To see (3), we may order the nodes by its splitting time. One may prove that this will yield a desired well-founded order.

In this example, the extra cost to construct a PRT won't exceed the time complexity of original algorithm. We found that the modification also won't increase the complexity of other more sophisticate algorithms. The space required to store ℓ_s and ℓ_a is linear to the size of tree. The space for storing ℓ is $O(|S|)$ because we only need to record $\ell(q)$ for each leaf q, and the states represented by an internal node can be constructed as the union of all the leaves in its subtree. Therefore, the extra space cost is linear to the size of transition system, which can be hidden by the complexity of checkers.

If two states s_0 and s_1 are not related, then PRT generated above may not necessarily be stable. However, if $s_0 \sim s_1$, then s_0 and s_1 should be in the same leaf , and the resulting PRT must be stable. Such refinement tree, for example, can be constructed by above procedure. This observation as well as lemma 2 gives the soundness and completeness of PRTs.

Theorem 3. *Given a labelled transition system $T = \langle S, A, \rightarrow\rangle$ and two states $s_0, s_1 \in S$, $s_1 \sim s_2$ if and only if there is a stable partition refinement tree γ for T such that s_0 and s_1 are in the same leaf.*

Verifying Partition Refinement Trees and Forests By theorem 3, we can verify the equivalence- checking result by verifying the validity of PRTs submitted by checkers. An independent verifier will check property (1), (2), and (3) on a candidate PRT γ. In addition, in the case that s_0 and s_1 are related, verifier also need to check the stability of γ. Most of mechanism of this verifier are clear, except checking (3). To check (3), we construct a graph $\beta = \langle Q, E\rangle$ from γ in which $\langle q, q'\rangle \in E$ if $\langle q, q'\rangle$ is an edge in γ, or, $q \neq q'$ and q is q''s splitter. Clearly (3) holds if and only if there is non-trivial loop [1] on β. Moreover, if such loop doesn't exist, then any topological order on β will yield a desired well-founded order. Therefore, checking (3) on γ can be done in $O(|\gamma|)$.

4 Equivalence Games

To define and implement an equivalence game, we introduce a three-level hierarchy by abstract levels of game semantics. At the top is an abstract game, which

[1] A loop is non-trivial iff it has more than one nodes

considers a generalized property-checking game. In the middle is a generalized equivalence game, which captures the common properties of all the equivalence games. On the bottom are individual games as instances of the generalized equivalence game. Also in this section, we will show how to efficient construct the winning strategy from PRTs.

4.1 Game Semantics Hierarchy

Abstract Game The purpose of a (property-checking) game is to determine whether or not property Φ is satisfied by a transition system. There are two players in a game: **I**, which believes that Φ holds, and **II**, which assumes the opposite. We use $A, B \cdots$ to range over $\{\mathbf{I}, \mathbf{II}\}$.

The definitions of a game shall specify the following aspects of game.

- *Configurations Q and initial configuration* $c_0 \in Q$. A play $P = c_0 \cdots \in Q^*$ is a sequence of configurations.
- *Rules.* The rules for valid next moves may be modelled as a function $\delta : Q^* \rightarrow 2^Q$. A play $P = c_0 c_1 \cdots$ is valid iff $c_{i+1} \in \delta(c_i)$. The rules should also declare the next player based on current configuration. This can be modelled as a function $p : Q \rightarrow \{\mathbf{I}, \mathbf{II}\}$. Player $p(c_i)$ shall choose the next move from $\delta(c_i)$.
- *Winning criteria.* The winning criteria judges who wins the game. It can be written as a partial function $w : 2^{(Q^*)} \rightarrow \{\mathbf{I}, \mathbf{II}\}$. We require that the *winning criteria* is complete, that is, any play shall eventually have a winner. In what follows, we shall consider only non-trivial plays, that is, the play is valid and terminates whenever someone already wins the game.

Another important component of game is the *strategies* of players. Intuitively a strategy of player is a decision function by which a player chooses his next move. Formally a strategy for player A is a partial function σ_A such that $\sigma_A(c_i) \subseteq \delta(c_i)$ where $\sigma_A(c_i)$ is defined and $p(c_i) = A$. A strategy is *deterministic* if $\sigma_A(c_i)$ gives only one choice. A play $P = c_0 c_1 \cdots$ is under a strategy σ_A if $c_{i+1} \in \sigma_A(c_i)$ whenever $p(c_i) = A$. A strategy is complete if for any play P under the strategy σ_A, $\sigma_A(c_i)$ is defined whenever $p(c_i) = A$. A strategy σ_A is a winning strategy iff any play under σ_A will be won by A. It follows that at most one player has a winning strategy at any game.

When a game is used as a diagnostic routine, it have two physical players: computer and user. **I** and **II** will be referred as logical players, which can be seen as the roles the computer and user choose to play. Computer want to convince user about the correctness of property-checking result. Therefore, it always chooses to act as the player in favor of property-checking result. If the checker reports that Φ does hold for T, computer will choose to act as **II**, otherwise it acts as **I**. User takes the opposite role. We require that property-checking game is a fair game, i.e., if Φ is indeed true, then, **II** has a winning strategy, otherwise, **I** has a winning strategy. Therefore, computer always has a winning strategy in a fair game, if the property-checking result is correct.

There may exist more than one game schema for a property-checking semantics. Beyond the logic design, a good definition of game should also be understandable and helpful for diagnose design system.

A Generalized Equivalence Game The second level of semantics hierarchy is a generalized equivalence game, which can be seen as an instance of abstract game. In an equivalence game, The property in question is whether two processes (two states in a transition system) are related by some equivalence relation $=$. Configurations have the form either $F(q_0, q_1)$ or $S(q_0, q_1, i, b, q')$, where q_0, q_1, q' are the set of states, and $b \in B$ is a symbolic action. We refer to q_j as the side j, and B as the set of symbolic actions. A valid play P is an alternating sequence of F-configuration and S-configuration. Each round starts with a move from a F-configuration to a S-configuration, followed by a move from a S-configuration to F-configuration. The initial configuration is a F-configuration $F(q_0, q_1)$ such that $s_0 \in q_0$ and $s_1 \in q_1$.

The next player is determined by the form of current configuration. If it is a F-configuration, then **I** will be the next player, otherwise **II** will be the next player. The notion of valid moves relies on two transition relations different for each individual game: the nature transition $T_f : Q \to B \times Q$ and the matching transition $T_s : Q \times B \to 2^Q$, where $Q = 2^S$. Given T_f and T_s, the next move is determined as below. For a F-configuration $F(q_0^k, q_1^k)$, **I** will pick one side $i \in \{0, 1\}$, then choose a nature transition for q_i^k. The resulting S-configuration $S(q_0^k, q_1^k, i, b, q')$ satisfies that $\langle b, q' \rangle \in T_f(q_i^k)$. For $S(q_0^k, q_1^k, i, b, q')$, **II** matches **I**'s choice by choosing a matching transition for q_{1-i}^k. The resulting F-configuration $F(q_0^{k+1}, q_1^{k+1})$ satisfies $q_i^{k+1} = q'$ and $q_{1-i}^{k+1} \in T_s(q_{1-i}^k, b)$. The reason why we need two transition functions is that for some equivalence games the definition of transition in the first step is different from that in the second step. For example, in the weak bisimulation game the former refers to original transitions, while the latter refers to observational transitions.

The winning criteria includes those rules common to all the equivalence games, and the semantics-oriented rules different for each individual game. The latter is usually given in the term of an equivalence relation Π. The winning criteria checks only F-configuration. Let $P = F(q_0^0, q_1^0)S(\cdots)\cdots F(q_0^k, q_1^k)$ be a play, **I** wins the game if,

1. There exists an action b and a side i such that $\exists q'.(\langle b, q' \rangle \in tr_f(q_i^k))$ and $tr_s(q_{1-i}^k, b) = \emptyset$, i.e., Side i may fire an action b for which the other side cannot match; Or,
2. $\langle q_0^k, q_1^k \rangle \notin \Pi$.

II wins if **I** doesn't win and,

1. $tr_f(q_0^k) = tr_f(q_1^k) = \emptyset$, i.e., none of sides may fire an action; or,
2. The play is infinite. Since we are considering only the finite transition systems, this is equivalent to check whether a F-configuration $F(q_0^n, q_1^n)$ has been repeated.

Semantics Plug-ins The lowest level of semantics hierarchy defines the part of semantics unique to each individual game. This includes the definition of initial configuration, the set of symbolic actions and two transition relations mentioned before: T_f and T_s. We will consider strong-bisimulation game and language (trace) equivalence game. The more complicate case of testing preordering game are left for the full version of this paper.

Strong-bisimulation game. The property in question is the strong bisimulation relation $\sim$. In this game, each side contains only one single state. The initial configuration is $F(\{s_0\}, \{s_1\})$. The set of symbolic actions is just A, the set of actions of T. Both nature and matching transitions refer to the transition relations of T. Formally, we have $T_f(\{s\}) = \{\langle b, \{s'\}\rangle \mid s \xrightarrow{b} s'\}$ and $T_s(b, \{s\}) = \{\{s'\} \mid s \xrightarrow{b} s'\}$. There is no special semantics-oriented winning criteria for this game. Therefore, Π is just the universal transition. **I** wins a play $P = F(\{s_0\}, \{s_1\}) \cdots F(\{s_0^k\}, \{s_1^k\})$ if $\exists a.\ (s_i^k \xrightarrow{a}) \wedge (s_{1-i}^k \not\xrightarrow{a})$. **II** wins the game if $F(\{s_0^k\}, \{s_1^k\})$ has occurred before.

The complete definition of strong-bisimulation game can be assembled from its three-level definition. Nevertheless, it can be interpreted in the plain English easily as below. The game involves two players and two processes (i.e., sides). A play is a sequence of rounds. In each round **I** always goes first. He pick up a process and make an action on this process. **II** will choose the other process to show it too can fire the same action. If initial actions of two processes, then **I** wins. If a configuration has been repeated, then, **II** wins. To access the fairness of this game, one may refer to [Sti96].

Language (trace) equivalence game. The property in question is whether two processes have the same set of traces. The initial configuration $F(\{s_0\}^\epsilon, \{s_1\}^\epsilon)$ starts with the ϵ- closure of s_0 and s_1. The nature and matching transitions refer to an extended definition of weak transitions in term of sets of states. That is, $t_f(q) = \{\langle a, q'\rangle \mid q \stackrel{a}{\Rightarrow} q' \wedge q' \neq \emptyset\}$ and $t_s(b, q) = \{\langle b, q'\rangle \mid q \stackrel{a}{\Rightarrow} q' \wedge q' \neq \emptyset\}$. There is no need for semantics-oriented winning criteria in this game. i.e., Π is just the universal transition.

Apparently both t_f and t_s are deterministic. The proceeding of a play is controlled solely by **I**. **II** just passively matches the action chosen by **I** because the deterministic matching transition leaves no choice to **II**. Now we give the proof skeleton for the fairness: if **I** wins the play $P = F(\{s_0\}^\epsilon, \{s_1\}^\epsilon) S(\cdots) \cdots S(\cdots)$ $F(q_0^k, q_1^k)$ because q_i^k can make action a_{k+1} while q_{1-i}^k cannot, then, clearly there is a trace $\beta = a_0 a_1 \cdots a_{n+1}$ accepted by s_i but not by s_{1-i}. Therefore, **I** won't win the game if two sides have the same set of traces. The winning strategy for **II** is very simple: it just passively follows whatever action chosen by **I**. **II** will win the play because the winning criteria is complete. In the case that two states don't have the same set of traces, the computer as **I** just follows the trace not shared by both states. Clearly this will yield a winning strategy for the computer.

4.2 Constructing Winning Strategies Efficiently

To make the games defined above usable as a diagnostic routine, one needs to find an efficient way to construct a winning strategy for computer. Instead of building an algorithm exclusively for this purpose, we "recycle" the information already existing in proof structures of equivalence checkers. This information will be represented by partition refine trees (forests). We start with bisimulation game to show how to construct the winning strategy from PRTs.

Assume γ be the partition refinement tree for the problem $s_0 \sim s_1$. Recall that one side contains only one state in this game, we shall speak a side while actually refer to the state in this side. There are two cases, depending on whether two states are related.

1. $s_0 \not\sim s_1$. Therefore, s_0 and s_1 are in the different leaves of γ. We will form a winning strategy W_I for **I**. A play under W_I keeps the following invariant,

Two sides are always in different leaves of γ.

The initial configuration holds the invariant trivially. Now, let $F(q_0^k \equiv \{s_0^k\}, q_1^k \equiv \{s_1^k\})$ be the starting configuration of round k. By the invariant, s_0^k and s_1^k are in two different leaves, say d_0 and d_1. Let d be the closest common ancestor of d_0 and d_1. W.l.o.g., assume s_i^k be the state in the left child and s_{1-i}^k be the state in the right child. Computer will pick up the next configuration $S(q_0^k, q_1^k, i, b, q' \equiv \{s_i^{k+1}\})$ such that $b = \ell_a(d)$ is the splitting action of d and s_i^{k+1} is a b-derivative of s_i^k in d, i.e., $s_i^{k+1} \in (\ell_s(d) \cap s_i^k \xrightarrow{b} \bullet)$. By (2) of definition 3, such choice is always feasible. Moreover, no matter which s_{1-i}^k user will choose, it cannot be in the same leaf as s_i^{k+1}, as guaranteed by (2).

Now we show that **II** cannot win under W_I. First, the play cannot reach a configuration in which both side have no transitions, as **II** always can choose a side and make a transition. Second, the repetition of F-configuration under W_I implies that there is no well-founded order on the nodes of γ, therefore such repetition cannot exist. By the fairness of game, **I** has a winning strategy if **II** doesn't.

2. $s_0 \not\sim s_1$. Therefore, s_0 and s_1 are in the same leaf of γ, and γ is stable. The winning strategy W_{II} maintains the following invariant,

In any round, two sides are in the same leaf of γ

Assume that $F(s_0^k, s_1^k)$ is the starting configuration of round k, and **I** chooses b-derivates s_i^{k+1} as the successor of s_i^k. Let b_i be the leaf b_i which s_i^{k+1} is in. **II** chooses a b-derivative s_{1-i}^{k+1} as the successor of s_{1-i}^k such that s_{1-i}^{k+1} is in b_i. Following the argument similar to the above, one can show that this yields a winning strategy for **II**.

For other games, the winning strategies can be constructed by tracing those PRTs on the reference transition systems. We extend the notion of side to a tuple $p_i^k \equiv \langle q_i^k, s_{ri}^k \rangle$, where q_i^k is the side in original form, and s_{ri}^k is a state in the reference system related to q_i^k. We define h as the mapping function which associates the states (sides) in games with the states in its reference system. The technique of tracing PRTs in these games is similar to the one we used for bisimulation game: if two processes are not related, the winning strategy maintains the invariant that the pair of states related to current configuration are in different leaves. Otherwise, the pair of states shall be in the same leaf.

As we have identified that the function of tracing PRTs is common for all the equivalence games, it can be implemented as a part of equivalence game module.

It generates an abstract winning strategy, in which configuration is given in the term of states in the reference system. Semantics plugins will interpret these states as sides in the targeting game. The interpretation may be implemented in the form of a callback function $\langle q', b\rangle = T_c(p \equiv \langle q, s_r\rangle, s'_r, a)$, indicating, if s_r in the reference system makes action a to s'_r, side q shall make a symbolic action b to q'.

The definition of h and T_c is straightforward for the language equivalence game: for any side $\langle q, s\rangle$ in the extended form, q and s refer to the same set of states. That is, $h(q) = \langle q, closed\rangle$ for a side q in the game and $\langle q', closed\rangle$ in the reference system $D(T)$, and $T_c(p \equiv \langle q, s_r\rangle, s'_r \equiv \langle q', closed\rangle, a) = \langle a, q'\rangle$.

Let T_r and γ be the reference transition system and the PRTs for T_r. The detail of forming a winning strategy is below, depending on whether two processes are related.

1. s_0 and s_1 are related. Let $F(p_0^k \equiv \langle q_0^k, s_{r0}^k\rangle, p_1^k \equiv \langle q_1^k, s_{r1}^k\rangle)$ be the starting configuration of round k. s_{r0}^k and s_{r1}^k are in the different leafs of γ. *EQM* will choose side i, an action a and an a-derives s_{ri}^{k+1} of s_{ri}^k according to the splitting information of the closest common ancestor, as described before. The next configuration will be $S(p_0^k, p_1^k, i, a, p' \equiv \langle q', s_{ri}^{k+1}\rangle)$ such that $\langle b, q'\rangle = T_c(p_i^k, a, s_{ri}^{k+1})$.
2. s_0 and s_1 aren't related. Let $S(p_0^k \equiv \langle q_0^k, s_{r0}^k\rangle, p_1^k \equiv \langle q_1^k, s_{r1}^k\rangle, i, a, p' \equiv= \langle q_i^{k+1}, s_{ri}^{k+1}\rangle)$ be the configuration after **I** moves in round k, then, *EQM* will choose the successor $s_{r(1-i)}^{k+1}$ of $s_{r(1-i)}^{k+1}$ such that s_{ri}^{k+1} and $s_{r(1-i)}^{k+1}$ are in the same leaf. The next configuration is $F(p_0^{k+1} \equiv \langle q_0^{k+1}, s_{r0}^{k+1}\rangle, p_1^{k+1} \equiv \langle q_1^{k+1}, s_{r1}^{k+1}\rangle)$, where q_{1-i}^{k+1} is the state related to $s_{r(1-i)}^{k+1}$ by T_c. That is, $\langle a, q_{1-i}^{k+1}\rangle = T_c(p_i^{k+1}, a, s_{r(1-i)}^{k+1})$.

The correctness of the winning strategy can be proven similarly as that for bisimulation game, although it may be slightly more complicate because of the use of reference systems. We leave it to readers.

5 Experimental Works

A prototype of this schema has been implemented on on Concurrency Workbench-the New Century (CWB-NC). CWB-NC is a verification toolkit for finite state systems which provides equivalence, preorder and model checking[CS96]. The part of architecture related to the equivalence games and other diagnostic routines are shown in Figure 1.

The input transition system is translated to a reference transition system, which is then checked by equivalence checkers. Currently CWB-NC has implemented Paige-Tarjan algorithm [PT87,Fer90] and Kernel-Auxiliary partition algorithm [TC01]. They both have been equipped with bookkeeping codes to produce PRTs.

The architecture of game routine follows the semantics hierarchy. Abstract game module is in charge of overall controlling: applying the rules, checking the

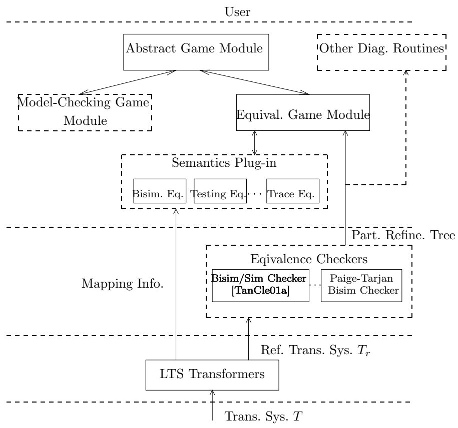

Fig. 1. Game Semantics Hierarchy on CWB-NC

winning condition, and providing common services, which include user interface and bookkeeping, etc. User's strategy is just to display a list of valid transitions and ask him to choose one as his next move. Hence, it is implemented as a part of user interface. The bookkeeping code maintains the history of a play, as required by the winning criteria. It also allow user to take back a few steps. One advantages of having the abstract game module have as many functions as possible is that they can be implemented consistently. For instance, the look and feel of bisimulation game is quite similar to those of language equivalence game because user interface has been implemented as part of abstract game module shared by all the games.

The equivalence game module (*EQM*) define the winning criteria, the rules, and computer strategy. The semantics plug-in supplies the semantics-oriented information, including Π as the special winning criteria, T_f, T_s, and T_c.

The abstract game module has 580 lines of Standard ML code. The equivalence game module (*EQM*) and preorder game module (*PRM*) are encoded as one single module with 547 lines of SML codes. The bisimulation game plug-in, observational equivalence game plug-in, language equivalence game plug-in, and testing equivalence game plug-in have 284, 417, 448, and 604 lines of code, respectively. We estimate that each game would need average 1,500 lines if it were implemented separately. According to our experience, it takes about one hour to write the language equivalence game plug-in. The prototype of this work is available at `ftp://ftp.cs.sunysb.edu/pub/CWB/working/playgame`.

References

[CH93] R. Cleaveland and M. C. B. Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.

[CS96] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[Fer90] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Sicence of Computer Programming*, 13:219–236, 1989/90.

[HM85] M. C. B. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.

[KS83] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proceedings of the 2nd ACM Symposium on the Principles of Distributed Computing*, Montreal, Canada, August 1983.

[Mai87] M. Main. Trace, failure and testing equivalences for communication, concurrency, and time. *International Journal of Parallel Programming*, 16(5):383–400, 1987.

[Mil89] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[PT87] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

[Sti87] C. Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49:311–347, 1987.

[Sti96] C. Stirling. Games and modal μ-calculus. In *Proceedings of the Third International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer-Verlag, 1996.

[TC01] L. Tan and R. Cleaveland. Simulation revisited. In *The 7-th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 480–495. Springer-Verlag, 2001.