Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 141 (2005) 171–197

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Adaptive Verification using Forced Simulation

Roopak Sinha[1]

University of Auckland
Auckland, New Zealand

Partha S. Roop[2]

University of Auckland
Auckland, New Zealand

Bakhadyr Khoussainov[3]

University of Auckland
Auckland, New Zealand

Abstract

Simulation (a pre-order) over Kripke structures is a well known formal verification technique. Simulation guarantees that all behaviours of an abstracted structure (a property or function, F) are contained in a larger structure (a model M). A model, however, may not always simulate a property due to the presence of design errors. In this case, the model is debugged manually. In this paper, we propose a weaker simulation over structures called *forced simulation* for automated debugging. Forced simulation is applied when normal simulation fails. Forced simulation between a model (M) and a function (F) guarantees the existence of a modifier, D, to adapt M so that the composition of M and D is *observationally equivalent* to F. Observational equivalence over structures called weak bisimulation is developed in this paper. It is also established that when two structures are weakly bisimilar all CTL^* properties holding over one also holds over the other structure. Forced simulation based algorithm has been used to adapt many designs which had failed certain properties during conventional verification.

Keywords: Formal verification, adaptive, bisimulation.

[1] Email: rsin077@ec.auckland.ac.nz

[2] Email:p.roop@auckland.ac.nz

[3] Email:b.khoussainov@auckland.ac.nz

doi:10.1016/j.entcs.2005.02.047

1 Introduction

Formal verification techniques are being widely applied in the design, development and validation of reactive systems such as digital hardware and hardware-software embedded systems [18,11], [4,14,17]. Formal methods use precise syntax and semantics for defining specifications and models of systems so that rigorous verification of properties such as correctness, reliability and security is made possible. Model checking [4] is an automatic formal verification technique to check if a model of a design (M) satisfies a given property (a function F) using reachability analysis over the state space of M. Model checking has become very appealing for automated verification due to initial state space reduction techniques like symbolic model checking [2] and more recently approaches using SAT solvers for bounded model checking [1]. The detection of a property failing over a model is a cause for concern as a model checker usually only produces some counter examples and the designer has to manually debug the model prior to attempting another cycle of model checking. Hence, approaches to automated debugging is of considerable interest to the community [6,7].

Model checking failures that lead to design debugging may be due to several reasons:

(i) Specification or property inconsistency: A system design may have errors introduced due to typographical mistakes (for example writing if $(p \geq 0)$ instead of if$(p \leq 0)$) that are very hard to detect using automated techniques.

(ii) Modelling inconsistency: A system may be inconsistently modelled resulting in model checking failure. The formal model extracted from a given system may not accurately indicate all behaviours of the given system resulting in such inconsistencies.

(iii) Consistent but buggy model: There may be errors due to incorrect interpretation of requirements by the designer. Such errors may either result in a design completely inconsistent with the specifications or may introduce redundant paths or states generalizing a more exact specification.

Automatic debugging of the first type of errors is extremely difficult, if not impossible. Approaches for debugging the second type of errors have been recently developed. Adaptive model checking [6] addresses the issue of a model checker providing *false negatives* due to modelling inconsistencies. Another approach debugs the design of a concurrent system using temporal logic formulas to control the stepping between system states [7]. In this paper we are primarily concerned with the third type of errors which are introduced due to the presence of redundant paths and states in a model. We illustrate this problem using the following example.

1.1 Motivating Example and Related Work

Consider a two process mutual exclusion solution (Figure 1) using *semaphores* [3] represented as a Kripke structure [5]. Kripke structures are standard models used in model checking defined as follows:

Definition 1.1: A Kripke structure [9] is a state machine represented in a tuple of the form $M < AP, S, s_0, \Sigma, \delta, \lambda >$ where:

- AP is a set of atomic propositions.
- S is a finite set of states.
- s_0 is the unique start state.
- Σ is a finite set of events or signals that occur in the environment of the system.
- $\delta \subseteq S \times \Sigma \times S$ is the transition relation.
- $\lambda : S \rightarrow 2^{AP}$ is the state labelling function that labels each state with the atomic propositions (in AP) which it satisfies.

For the mutual exclusion example, $AP = \{semaphore_{val} = 0(available)$, $semaphore_{val} = 1(taken)$, $p1_{state} = i(idle)$, $p1_{state} = c(critical)$, $p1_{state} = e(entering)$, $p1_{state} = x(exiting)$, $p2_{state} = i(idle)$, $p2_{state} = c(critical)$, $p2_{state} = e(entering)$, $p2_{state} = x(exiting)\}$. Also, $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}\}$ and $\Sigma = \{p_1, p_2\}$. δ is the transition relation which contains all possible transitions of the system in tuples of the form (s_0, p_1, s_1). λ is the labelling function that provides specific labelling on individual states. For example, $\lambda(s_4) = \{semaphore_{val} = 0, p1_{state} = e, p2_{state} = e\}$ (in short $\{0, e, e\}$).

The semaphore example in Figure 1 contains 12 states, each labelled by three variables $semaphore_{val}$, $p1_{state}$, $p2_{state}$. The variable $semaphore_{val}$ represents whether the semaphore is available (0) or taken (1). The other two variables show the state of the two processes and can be either i(*idle*), e(*entering*), c(*critical*) or x(*exiting*). A process is *idle* when it does not need to enter its critical section, *entering* when it registers its need for the semaphore, *critical* when it enters its critical section and *exiting* when it is leaving its critical section. A transition from the current state to a successor is made when the input that triggers that transition is furnished by the operating environment of the system. The operating environment can be viewed as a process-selector which at every instance decides which process (process 1 or process 2) is allowed to evolve. For example, in the initial state s_0, $semaphore_{val} = 0$, $p1_{state}$=*idle* and $p2_{state}$=*idle*. The transition from state s_0 to s_1 is triggered when the process-selector chooses process $p1$ to evolve. Consequently, the variable $p1_{state}$ changes from i(*idle*) to e(*entering*). The other variables retain their previous values. If no process is chosen to be active in the current instance, the system stays in its current state. A self-loop on every state is not shown to keep a concise description of the system.Let this be an example model M.

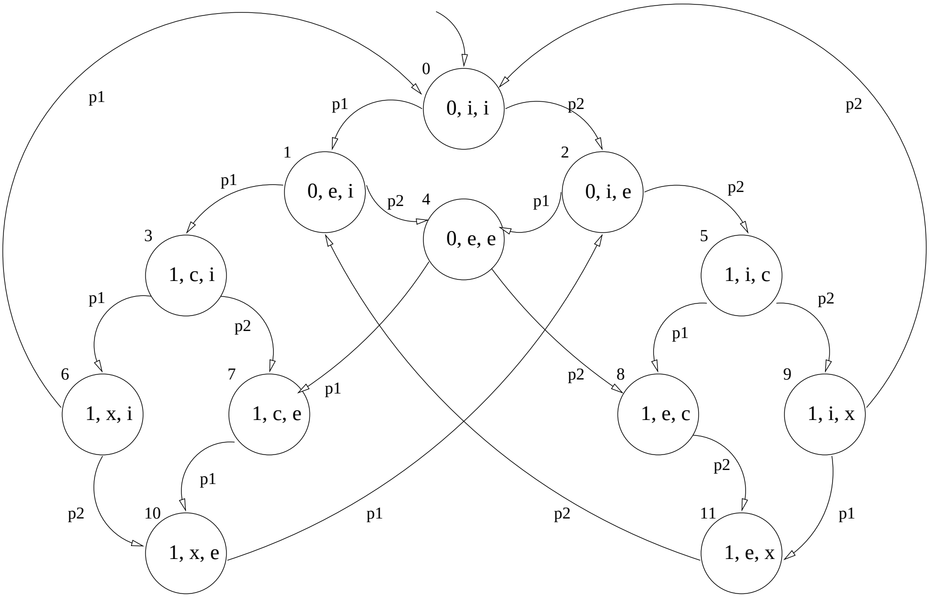

Fig. 1. Kripke representation of a semaphore-based two-process system

A Kripke structure F representing the desired specification of the semaphore system is shown in Figure 2. In this specification, it is required that the first process that makes a request for the semaphore (changes its state to *entering*), is the one that gets to finish executing its critical section first. In the specification, if process 1 evolves from its initial *idle* state (in s_0) to *entering* (in s_1), then it is expected that process 1 is allowed to enter into its critical state (s_3). After this, both processes are allowed to evolve till one of them again changes its state to *entering*. This can be viewed as a fairness constraint which ensures that starvation is prevented. The given system M in Figure 1 does not satisfy this requirement. For example, by reaching state s_1 from the initial state s_0, process 1 has already requested for the semaphore by changing its state to *entering*. However, s_1 can reach state s_4 when the process-selector selects process 2, which in-turn makes a request for the semaphore by changing its state to *entering*. State s_4 may then make a transition to state s_8 if the process-selector chooses process 2 to evolve next. In this case, process 2 gets to execute its critical section before process 1, even though process 1 had requested the semaphore first. This sequence represents the violation of the desired property as the process that first changes its state to *entering* does not get to enter and consequently exit its critical section first.

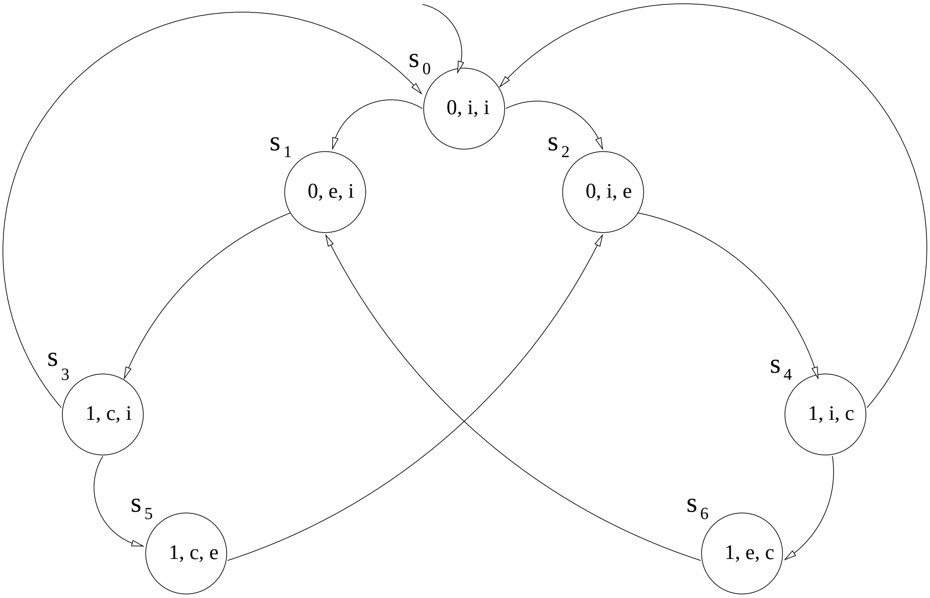

Fig. 2. The desired semaphore system behaviour

Given M and F as above we cannot use existing adaptation techniques [6,7] to debug this model automatically (as there are buggy paths and states in the model not desired in the specification). Moreover, techniques like model checking will find counter examples and debugging of the model will have to be subsequently done manually. The proposed approach attempts to automatically debug models with faulty (buggy) states and paths using a form of *dynamic model checking*. The main idea is the construction of an *modifier process* (another Kripke structure) that dynamically *adapts* the model to remove faulty paths and states.

Given an arbitrary model and specification (M-F) pair, it is important for any adaptation technique to address the following fundamental issues:

(i) Under what conditions can a model be automatically adapted to satisfy a given specification?

(ii) How can it be established that a given adaptation is correct, that is the adapted model will satisfy the given specification?

(iii) How can it be established that the adapted model is indeed consistent from its specification?

In the subsequent sections we present a formal framework for answering these questions. We also present experimental results using an adaptive verification tool called ADVERT (Adaptive Verification of ReacTive Systems). The main contributions of the paper are:

(i) An automated algorithmic framework for debugging verification failures due to buggy states and paths is developed. The proposed algorithm is based on the notion of adaptation of a buggy model using a modifier that dynamically alters the model.

(ii) The debugging algorithm is based on a new simulation relation over Kripke structures called *forced simulation* that is weaker than conventional simulation. Forced simulation between Kripke models of a specification and a model guarantees the existence of an *modifier* (another Kripke structure) to adapt the model dynamically so that inconsistencies with respect to the specification can be removed.

(iii) In order to verify that the proposed approach produces accurate modifiers, a notion of weaker equivalence (observational equivalence) between structures is necessary. This lead to the development of weak bisimulation over Kripke structures, also developed by us. It is established that the composition of the modifier and the model is weakly bisimilar to the desired specification. It is also further established that when two structures are weakly bisimilar they satisfy the same set of CTL^* properties.

(iv) An adaptive verification tool called ADVERT has been developed by extending NuSMV [3]. ADVERT has been used to debug many standard designs having model checking failures.

Simulation and preorders over labelled transition systems (LTS) [12] and Kripke structures [5] are well known. While simulation and refinement have been used for state space reduction and implementation verification, they are not directly applicable to the problem of design debugging as the model in question has buggy paths and states that need to be removed from the model. Recently, forced simulation over labelled transition systems [16] has been developed for dynamically changing a LTS description of a design for automatic reuse. In this paper, the idea of forced simulation over LTSs is substantially altered to develop forced simulation over Kripke models. Automated component reuse using forced simulation [16] proposes an algorithm which matches the states of two given LTSs using the environment inputs that trigger their transitions. However, in the presented approach the states of two given Kripke structures are matched using their state labelling. The aim of the component reuse technique using forced simulation is to make a given *device* weakly bisimilar [12] (functionally equivalent) to a given *design function*. In contrast, the presented approach works in a verification framework and is based on adapting a model to satisfy a given temporal specification. Forced simulation over Kripke models requires a weaker equivalence than strong bisimulation over Kripke models [13] due to the introduction of τ (internal) transitions in the debugged model. This paper develops weak bisimulation over Kripke structures and also proves that this equivalence preserves CTL^* equivalence.

The debugging problem addressed in this paper is somewhat similar to the *controllability* problem [15] within control systems where a *controller* is synthesized to make a *plant* behave as the desired *specification* (e.g, the role of the controller is to adapt the behaviour of the plant dynamically). The task of the controller is to disable certain actions of the plant at specific points. Though this problem has been studied in a general nondeterministic setting, it is not applicable to the debugging task of reactive systems which requires specific types of adaptation. Module checking [10] is a model checking variant developed for property checking of reactive systems. Unlike conventional model checking that ignores the environment, module checking takes the asynchronous environment of a reactive system into consideration. It considers the states of a model as either an *environment state* (a state that requires an environment input) or a *system state* (an internal state requiring no environment input) and performs model checking in the presence of environment states. In the adaptive verification approach developed in the paper, we consider an abstraction of the design that has only environment states and as a result of our proposed debugging some system states are also introduced in the debugged model. Unlike module checking, however, the proposed approach not only considers the environment passively but also introduces an additional environment in the form of a modifier to *force* certain environment events into the model or to block some events in the environment from reaching the model (*called disabling*).

This paper is organized as follows. Section 2 introduces the forced simulation relation, which has been proven to be the necessary and sufficient condition for adaptation. The details of how an automatically generated modifier adapts a given model appears in section 3. Weak bisimulation, an equivalence relation developed to test equivalence between an adapted model and its specification is given in section 4. Section 5 presents the extension of temporal property satisfaction to weak satisfaction, used to show that a model adapted using forced simulation indeed satisfies the same set of temporal properties as the given specification. Section 7 presents theoretical and practical results of design debugging and the final section is devoted to concluding remarks.

2 Forced Simulation

Forced simulation is a simulation relation defined over two Kripke structures, a model (M) and a specification (F) and aims to provide a basis for checking whether M is *adaptable* to meet F. It is defined as follows (In this definition the states of the function (specification) are denoted as s_f and that of the model are denoted as s_m with (s_{f0}, s_{m0}) denoting the start states of the function and model respectively.):

Definition 2.1: For Kripke structures F and M, a relation $B \subseteq S_F \times S_M \times \Sigma_M^*$ is called a forced simulation relation (in short, an f−simulation relation) provided the following hold ($s_f B^\sigma s_m$ is used as a shorthand for $(s_f, s_m, \sigma) \in B$ where length of σ is bounded by $|S_M|$ (number of states in M)):

(i) $s_{f0} B^\sigma s_{m0}$ for some $\sigma \in \Sigma_M^*$.

(ii) $s_f B^{a.\sigma} s_m \Rightarrow (\exists s'_m : s_m \xrightarrow{a} s'_m \wedge s_f B^\sigma s'_m)$ for any $\sigma \in \Sigma_M^*$.

(iii) $s_f B^\epsilon s_m \Rightarrow (\lambda_F(s_f) = (\lambda(s_m)) \wedge (\forall s'_f, \exists s'_m, \exists a.\sigma : s_f \rightarrow s'_f \Rightarrow (s_m \xrightarrow{a} s'_m \wedge s'_f B^\sigma s'_m))$.

The first condition requires that the start states of the two structures be related via some forcing sequence σ. The second condition requires that when any two states (s_f, s_m) are related via some forcing sequence $a.\sigma$ then there must be a transition from s_m to some state s'_m that triggers using environment input $'a'$ and further that s_f, s'_m must be related. This rule is required to successively reduce a forcing sequence until a state that is directly similar to s_f is found. Two states s_f, s_m are directly related when the forcing sequence is empty or ϵ. In that case, the state labelling of these states must match and further every transition out of s_f must be matched by a corresponding transition out of s_m and the resultant states of these transitions must be related via some sequence σ.

Definition 2.2: $F \sqsubseteq_{fsim} M$ provided there exists an f−simulation relation between them.

Consider processes M and F as shown in Figures 1 and 2. $F \sqsubseteq_{fsim} M$ since there exists $R = \{(s_{f0}, s_{m0}),\ (s_{f1}, s_{m1}),\ (s_{f2}, s_{m2}),\ (s_{f3}, s_{m3}),\ (s_{f4}, s_{m5}),\ (s_{f0}, s_{m6}),\ ,(s_{f5}, s_{m7}),\ (s_{f6}, s_{m8}),\ (s_{f0}, s_{m9})\}$, which can be easily shown to be an $f-$simulation relation.

3 Adaptation using a modifier

The approach for automatic debugging is based on controlling the transitions of the model to ensure that the specification is satisfied. The adaptation is performed by an automatically generated *modifier* process which exercises state-based control on the system. A modifier controls a model in the following ways:

- Disabling action: Consider the sequence of states $s_0, s_1, s_4, s_8, s_{11}, s_1$ in M (Figure 1). As described earlier, this path is inconsistent with the given specification F in Figure 2 because state s_1, where process 1 has already changed its state to *entering*, can make a transition to state s_4 where process 2 changes its state to *entering* as well. If state s_1 is forbidden from making a transition to state s_4, the faulty path can be eliminated. This can be achieved if the process-selector signal p_2 is *hidden* from the system when it is in state s_1. This state-based hiding of actions is known as *disabling* and is achieved when the modifier absorbs one or more environment inputs and effectively disables the system to make certain transitions.
- Forcing action: Consider the path s_0, s_1, s_3, s_6, s_0 in M (Figure 1). This path is also inconsistent with the specification due to the presence of state s_6 (p_1 in its exiting state) which has no corresponding equivalent in F. If state s_3 evolves to s_6 by reacting to the process-selector input p_1 and then if s_6 makes a transition to its successor s_0 *without* waiting for the input p_1 (which triggers the transition), it would seem to the operating environment, that the system made a transition from s_3 to s_0 by reacting to the single environment input p_1. The transition from s_6 to s_0 therefore becomes *internal* transition and state s_6 becomes an *unobservable* or *internal* state. This is called *forcing* and is achieved when the modifier *artificially manufactures* an environment input to force the system to make a specific transition without interacting with its environment. Looking at the observable part s_0, s_1, s_3, s_0 of the path s_0, s_1, s_3, s_6, s_0, it can be seen that it is now consistent with the path s_0, s_1, s_3, s_0 in F (Figure 2). Internal or unobservable transitions are similar to CCS τ transitions and originate only from internal or unobservable states.
- A modifier may allow the current state in M to evolve without any disabling or forcing.

Internal or unobservable states may be introduced into the adapted system by a modifier during forcing. An internal state is different to an observable state in that it is unobservable to the operating environment and makes an internal τ transition to its successor without waiting for any input from the environment. Such states are labelled by the proposition *intern* to indicate that they are unobservable.

A modifier needs strict control over all transitions of the current state in the model so that each transition can be uniquely forced or disabled if needed. As actions that trigger transitions are used to distinguish between

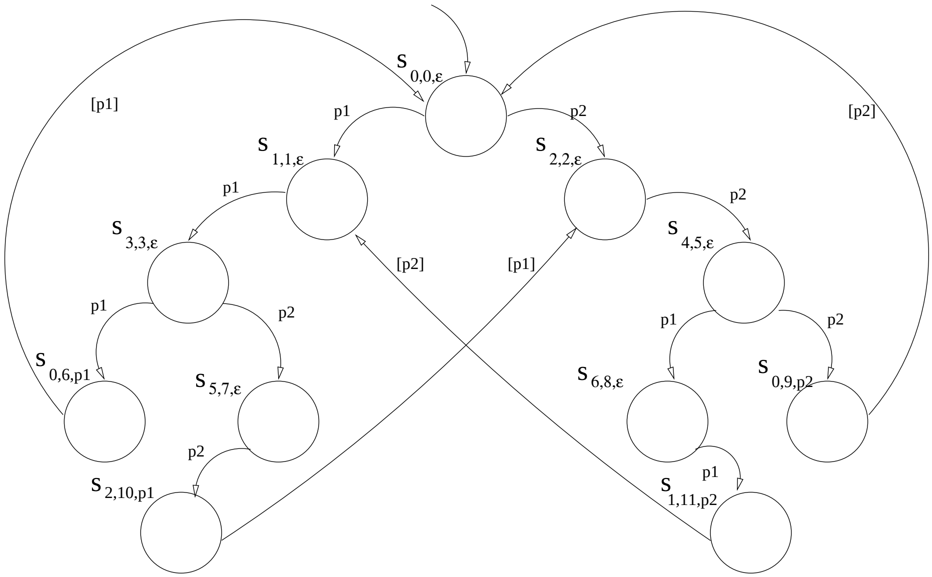

Fig. 3. A well formed modifier for the semaphore example

transitions, no state can have more than one transition triggered by the same action. Therefore, only *deterministic* models can be used under the proposed framework.

Definition 3.1: A Kripke structure $A < AP_A, S_A, s_{a0}, \Sigma_A, \delta_A, \lambda_A >$ is said to be deterministic if and only if for any $s_a \in S_A$, if (s_a, a, s'_a) and $(s_a, a, s''_a) \in \delta_A$ for any $s_a, s'_a, s''_a \in S_M$ and $a \in \Sigma_M$ then $s'_a = s''_a$.

A modifier must be *well formed* which means that it must not allow any system state to accept inputs from the environment if the modifier performs forcing on it. A well-formed modifier for the semaphore example in Figure 1 is shown in Figure 3. A modifier itself does not contain any temporal information and all its states are labelled with the proposition $True$.

3.1 Composition of a modifier and a model

The state-based control exercised by a modifier over a given system is defined using a new $//$ composition operator as follows.

Definition 3.2: Given $D < AP_D, S_D, s_{d0}, \Sigma_D, \delta_D, \lambda_D >$ and $M < AP_M, S_M, s_{m0}, \Sigma_M, \delta_M, \lambda_M >$ as above, $D//M$ is defined to be a process described by the Kripke structure $D//M < AP_{(D//M)}, S_{(D//M)}, (s_{d0}, s_{m0}), \Sigma_{(D//M)}, \delta_{(D//M)}, \lambda_{(D//M)} >$ where:

- $AP_{(D//M)} = AP_M \cup \{intern\}$ ($intern$ is used to label internal or unobservable states)
- $S_{(D//M)} \subseteq S_D \times S_M$

- (s_{d0}, s_{m0}) is the start state.
- $\lambda_{(D//M)}(s_d, s_m) = \lambda_M(s_m)$ for all $s_m \in S_M$ and $s_d \in S_D$, if $Lab(s_d) \neq \{[a]\}$ otherwise $\lambda_{(D//M)}(s_d, s_m) = \{intern\}$. Internal states or states that evolve by forcing are labelled by $intern$.
- $\Sigma_{(D//M)} = \Sigma_M \bigcup \{\tau\}$
- $\delta_{(D//M)}$ is defined as follows:

(i) Forced Move: $D//M$ makes an unobservable τ move, when D ‘forces’ a transition in M.

$$\frac{s_d \xrightarrow{[a]} s_{d1}, s_m \xrightarrow{a} s_{m1}}{(s_d, s_m) \xrightarrow{\tau} (s_{d1}, s_{m1})}$$

In this case, the state (s_d, s_m) is made an unobservable state due to forcing.

(ii) External Move: $D//M$ makes an observable move with both the M and D simultaneously responding to the same environment input.

$$\frac{s_d \xrightarrow{a} s_{d1}, s \xrightarrow{a} s_{m1}}{(s_d, s_m) \xrightarrow{a} (s_{d1}, s_{m1})}$$

In this case, (s_d, s_m) is a state that is observable to the operating environment of $D//M$.

The primary reason for the development of the // operator is to allow the modifier to have tight control over the states of a model. Due to this requirement, other composition operators like the CCS || operator [12] can not be used. The result of composition using the || operator is a model that has no apparent control over M. In this case, the states of either M or D may advance to their respective successors without having to synchronize with the respective states in the other model. The // operator, on the other hand, does not allow any state in either model to advance without synchronizing with its respective state in the other model, providing lock-step control to the modifier. This is similar to lock-step process synchronization [8], however the difference lies in the way a modifier performs forcing, which is not present in any available composition operator. The modifier is not required to have explicit information about which state a system is in. The current state can be determined by keeping track of the environment inputs that the system has received so far.

Given the modifier D in Figure 3 for the semaphore example, $D//M$ is presented in Figure 4. Note that states s_1 and s_2 are disallowed from making a transition using the process-selection inputs p_2 and p_1 respectively. These disabling actions ensure that none of the violating paths s_1, s_2, s_4, s_8 or s_1, s_3, s_4, s_7 are allowed. As explained earlier, the modifier controls the interaction of the system on a state-wise basis, therefore inputs p_2 and p_1 that are disabled in states s_1 and s_2 are allowed in all other states of the system. Similarly, states s_5, s_8, s_9 and s_{10} in $D//M$ (or s_6, s_9, s_{10} and s_{11} in M) are

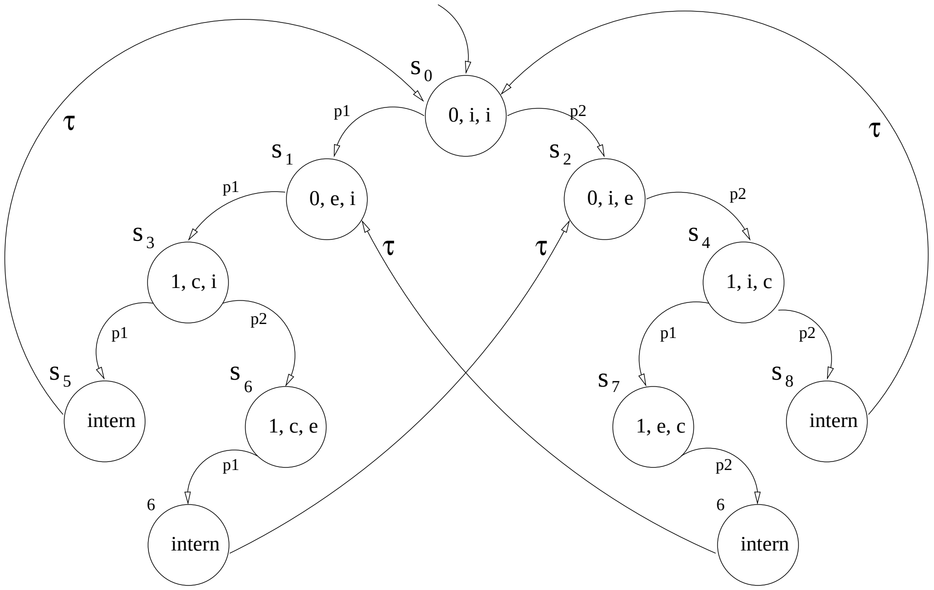

Fig. 4. $D//M$, the sample modifier in Figure 3 driving the semaphore system

forced by the modifier to make transitions to s_0, s_0, s_2 and s_1 respectively. The modifier artificially manufactures the required actions ($p1$,$p2$, $p1$ and $p2$ respectively), making these transitions internal (τ transitions) to the environment. These states therefore become internal and are consequently labelled with the proposition *intern*.

4 Weak bisimulation

Once a model is adapted using forced simulation, there is a need to establish that it is equivalent to the given specification. Traditionally, equivalence between two Kripke structures can be checked using strong bisimulation [13]. However, as described earlier, adaptation using forced simulation may make some states in the adapted model *unobservable*. Unobservable states are invisible to the operating environment of a system and are not required while checking for equivalence of a system with a specification. In other words, there is a need to *extract* only the observable part of a system and use this part to check for equivalence.

To verify equivalence between two Kripke structures which may contain unobservable states or paths, a weaker equivalence relation combining weak bisimulation equivalence over LTS [12] and strong bisimulation for Kripke structures has been formulated. Two Kripke structures are considered weakly bisimilar if their observable behaviours are strongly bisimilar [13].

Before presenting weak bisimulation, some important notations are defined as follows.

Definition 4.1: Given a Kripke structure $M < AP, S, s_0, \Sigma, \delta, \lambda >$ and states $s, s' \in S$ then:

- If for some action $a \in \Sigma$, $(s, a, s') \in \delta$ then:
  · If $\lambda(s) \neq intern$ then $s \underset{E}{\rightarrow} s'$ where E represents an external transition from s to s'.
  · Otherwise, if $\lambda(s) = intern$ then $s \underset{\tau}{\rightarrow} s'$ where τ represents an internal transition from s to s'.
- Consider the set $Trans = \{E, \tau\}$.
  Now, if $\alpha \in Trans^* = A.B.C.D...Z$ is a sequence of possible transitions, then $s \underset{\alpha}{\rightarrow} s'$ implies that $s \underset{A}{\rightarrow} s_1 \underset{B}{\rightarrow} s_2..s_{N-1} \underset{Z}{\rightarrow} s'$.

This definition is used to relabel the transitions of a Kripke structures as either external (E) or internal (τ). For example, given the adapted model $D//M$ in Figure 4, consider the path s_0, s_1, s_3, s_5, s_0. It is known that $s_0 \underset{p1}{\rightarrow} s_1 \underset{p1}{\rightarrow} s_3 \underset{p1}{\rightarrow} s_6 \underset{\tau}{\rightarrow} s_0$. Identifying each transition as either external or internal, and representing each external transition with E and each internal transition as τ, we get $s_0 \underset{E}{\rightarrow} s_1 \underset{E}{\rightarrow} s_3 \underset{E}{\rightarrow} s_6 \underset{\tau}{\rightarrow} s_0$ in the relabelled Kripke structure. We can also say that $s_0 \underset{\alpha}{\rightarrow} s_0$ where $\alpha = E.E.E.\tau$.

Definition 4.2: If $\alpha \in Trans^*$ then $\overset{\wedge}{\alpha} \in \{E\}^*$ is the sequence obtained by deleting all τ occurrences from α. If after deleting all τ occurrences,$\overset{\wedge}{\alpha}$ contains no elements, then $\overset{\wedge}{\alpha} = \epsilon$ ($\overset{\wedge}{\tau^*} = \epsilon$).

In order to check for equivalence between two structures, it is required to extract and compare only their observable paths. This definition describes how only the observable part of a sequence of observable and internal actions can be extracted. Given the path s_0, s_1, s_3, s_5, s_0 in the adapted model $D//M$ in Figure 4, we have seen that $s_0 \underset{\alpha}{\rightarrow} s_0$ where $\alpha = E.E.E.\tau$. Extracting the observable part of α, we get $\overset{\wedge}{\alpha} = E.E.E$. Consider the transition $s_6 \underset{\tau}{\rightarrow} s_0$. In this case, $\alpha = \tau$ and therefore by the above definition, $\hat{\alpha} = \epsilon$.

Figure 5 shows the adapted $D//M$ model with transitions relabelled either with E or τ. All transitions triggered by environment inputs are labelled by E and all internal transitions triggered by the modifier are labelled by τ (as per Definition 4.1).

Definition 4.3: Let $\alpha \in \{E\}^*$. Then $s \underset{\alpha}{\Rightarrow} s'$ if and only if there exists $\alpha' \in Trans^*$ such that $s \underset{\alpha'}{\rightarrow} s' \wedge \alpha = \overset{\wedge}{\alpha'}$.

Consider the initial state s_0 of the adapted model $D//M$ in Figure 5.

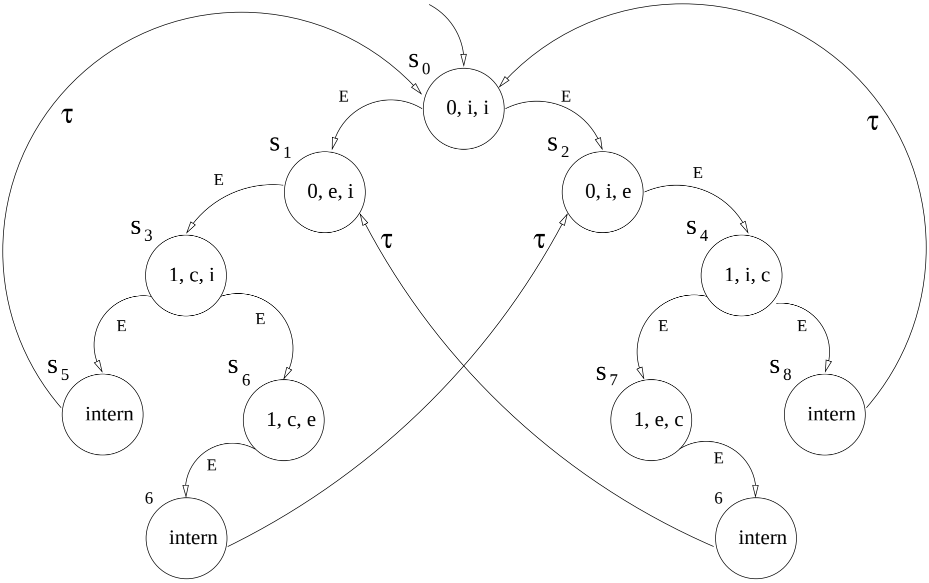

Fig. 5. The adapted model with only external and internal transitions

Given the sequence $\alpha = E.E.E$, s_0 can reach states s_5, s_6, s_7 and s_8 using α. However, states s_5 and s_8 are internal states and therefore make an internal τ transition to their successor s_0. Therefore, state s_0 can reach itself by following the sequence $\alpha' = E.E.E.\tau$. From definition 4.2, $\hat{\alpha}' = E.E.E = \alpha$. Observing from the operating environment of the adapted model, s_0 can reach states s_6, s_7 *and* s_0 if provided with the sequence $\alpha = E.E.E$. The τ transitions are not visible to the environment and therefore not taken into account.

Definition 4.4: Given two Kripke structures $A < AP_A, S_A, s_{a0}, \Sigma_A, \delta_A, \lambda_A >$ and $B < AP_B$, S_B, s_{b0}, $\Sigma_B, \delta_B, \lambda_B >$ over the same set of atomic propositions $AP_A = AP_B$, a relation $B \subseteq S_A \times S_B$ is a weak bisimulation if for any $s_a \in S_A$ and $s_b \in S_B$, $B(s_a, s_b)$ if and only if:

(i) $(\lambda_A(s_a) = \lambda_B(s_b)) \vee (\lambda_A(s_a) = intern) \vee (\lambda_B(s_b) = intern)$

(ii) If $s_a \underset{\alpha}{\longrightarrow} s'_a$ for some $\alpha \in Trans^*$ then $s_b \underset{\hat{\alpha}}{\Rightarrow} s'_b$ and $B(s_{a'}, s_{b'})$

(iii) If $s_b \underset{\alpha}{\longrightarrow} s'_b$ for some $\alpha \in Trans^*$ then $s_a \underset{\hat{\alpha}}{\Rightarrow} s'_a$ and $B(s_{a'}, s_{b'})$

Figure 6 shows the specification F with only external transitions (labelled by E). We can see that s_0 in F and s_0 in the adapted model $D//M$ (Figure 5) are related over a weak bisimulation relation. They satisfy rule 1 as both have the same state labelling $(0, i, i)$. They also satisfy rule 2 and 3. For every state s' that s_0 in F can reach using any sequence α, s_0 in $D//M$ can observably reach a state s'' using $\hat{\alpha}$ such that s' and s'' are also weakly bisimilar (and vice versa).

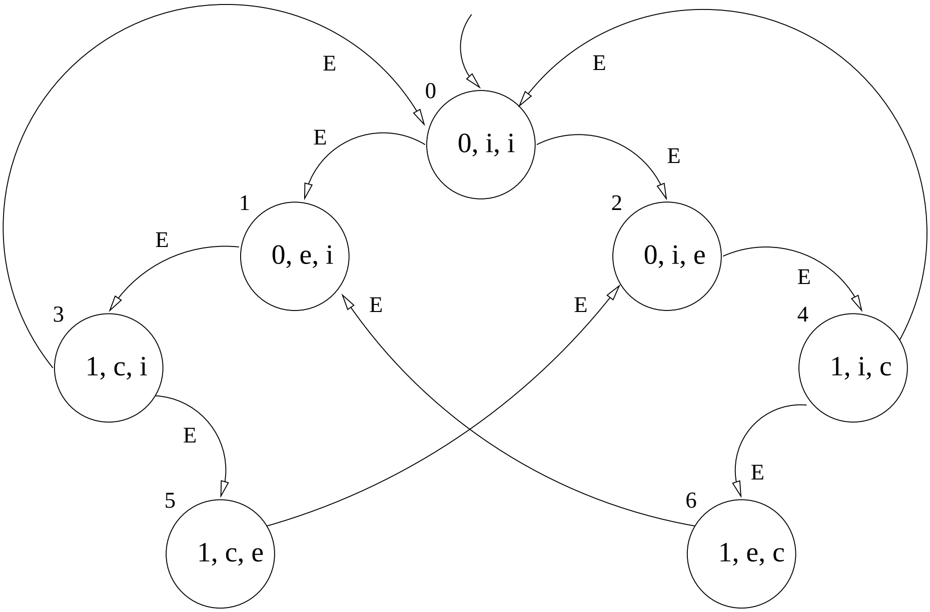

Fig. 6. The specification F

Definition 4.5: Two Kripke structures $A < AP_A,\ S_A,\ s_{a0},\ \Sigma_A,\ \delta_A,\ \lambda_A >$ and $B < AP_B,\ S_B,\ s_{b0},\ \Sigma_B,\ \delta_B,\ \lambda_B >$ are weakly bisimilar if and only if there exists a weak bisimulation relation WB such that $WB(s_{a0}, a_{B_0})$.

Two structures are considered weakly bisimilar if their initial states are related over a weak bisimulation relation. As discussed above, s_0 in F and s_0 in $D//M$ are weakly bisimilar to each other. Therefore, according to the above definition, the adapted semaphore model $D//M$ in Figure 4 is weakly bisimilar to the given specification F in Figure 2.

5 Weak satisfaction

A system adapted using forced simulation may contain unobservable states and paths. To check whether an adapted system satisfies a given temporal property, there is a need to extend property satisfaction to take into account the possible presence of unobservable behaviours in a given system.

This problem can be explained as follows. All temporal logic formulas, including those expressed in CTL^* and its subsets like LTL and CTL operate on states and their *direct successors*. However in a system adapted using forced simulation, the direct successor of a state might be unobservable. To ensure that unobservable states are not taken into account, there is a need to weaken property satisfaction using the following approach:

Property satisfaction needs examination of states and paths. Since model debugging introduces internal or unobservable states and paths, there is a need to extract only the observable behaviour of the given model. This is done using the operators $Next_{ob}()$, used for extracting the observable current state, and $Succ_{ob}(s)$, which returns the observable successor of the current state.

The definitions for the $Next_{ob}$ and $Succ_{ob}$ functions are provided as follows.

Definition 5.1: The function $Next_{ob} : S \rightarrow S$ where S is the set of states of a relabelled Kripke structure (definition 4.1), such that:

- $Next_{ob}(s) = s$ if s is observable.
- $Next_{ob}(s) = \{s' | s \underset{\tau^n}{\rightarrow} s' \wedge observable(s')$ where $n \geq 1\}$ if s is unobservable.

Definition 5.2: The function $Succ_{ob} : S \rightarrow S$ where S is the set of states of a relabelled Kripke structure (definition 4.1), such that:

- $Succ_{ob}(s) = \{s' | s \underset{E.\tau^n}{\rightarrow} s' \wedge observable(s')$ where $n \geq 0\}$ if s is observable.
- $Succ_{ob}(s) = \{s'' | (s'' \in Succ_{ob}(s')$ where $(s' \in Next_{ob}(s_a)\}$ if s is unobservable.

Given the adapted model $D//M$ in Figure 5, consider the initial state s_0. This is an observable state. Therefore, $Next_{ob}(s_0) = \{s_0\}$ (definition 5.1) and $Succ_{ob}(s_0) = \{s_1, s_2\}$. It can be seen that for an observable state, $Next_{ob}$ returns the original state and $Succ_{ob}$ returns the observable successors of the given state. Now consider the state s_5 in the adapted model $D//M$. This is an internal state (labelled with *intern*. Therefore, $Next_{ob}(s_5) = \{s_0\}$ and $Succ_{ob}(s_5) = \{s_1, s_2\}$. For an internal state, $Next_{ob}$ returns its observable successor (reached by a series of one or more τ transitions) and $Succ_{ob}$ returns the observable successors of the state returned by $Next_{ob}$. The above definitions indicate that when checking for property satisfaction, an internal state is considered equivalent to its observable successor.

Based on the above definitions, observable path can be defined as follows:

Definition 5.3: An infinite observable path starting from a state s is defined as $\pi_{ob}(s) = s_0, s_1, s_2, ..., s_\infty$ where $s_0 \in Next_{ob}(s)$ and the states s_1 to s_∞ are defined recursively as $s_i = s' | s' \in Succ_{ob}(s_{i-1})$.

Definition 5.4: For any path $\pi(s)$ starting from the state s, its corresponding observable path $\pi_{ob}(s)$ is obtained by removing all unobservable states from it.

The notation $\models_W$ is used to represent weak satisfaction. Weak satisfaction for states, paths and models is defined as follows.

Definition 5.5: For any CTL^* state-formula φ and any state s, $s \models_W \varphi$ if and only if there exists a state s' such that $s' \in Next_{ob}(s')$ and $s' \models \varphi$.

Definition 5.6: For a CTL^* path-formula φ and for any path π with its first state as s, $\pi \models_W \varphi$ if and only if the observable path $\pi_{ob}(s) \models \varphi$.

Definition 5.7: For a CTL^* formula φ and a Kripke structure $M < AP$, S, s_0, Σ, δ, $\lambda >$, $M \models_W \varphi$ if and only if:

- If φ is a state-formula, then $s_0 \models_W \varphi$.
- If φ is a path-formula, then for all possible paths $\pi(s_0)$ with their start state as s_0, $\pi(s_0) \models_W \varphi$.

For a Kripke structure M and a CTL^* formula φ, if $M \models \varphi$ then $M \models_W \varphi$. This assertion shows that weak satisfaction extends the notion of property satisfaction to structures with internal structures and at the same time conserves property satisfaction in structures with no unobservable states.

Weak satisfaction makes forced transitions in the model internal in the composite process. In the current adaptation algorithm, states that are not present in the specification may be forced by the modifier in order to make the model satisfy the desired specification. Obviously, if we extend such adaptation technique to model checking, then forcing needs to be restricted so that bad states (states that may cause critical failure of the property) should not be forced. Hence, forced simulation needs to be modified to include the handling of certain states which are not forceable.

6 Adaptive verification algorithm

An algorithm has been formulated to compute a forced simulation relation given M and F. If successful, the algorithm generates the modifier D. The proposed algorithm is an adaptation of the component matching algorithm based on forced simulation [16]. The algorithm consists of a pre-computation step which computes all reachable states from every state in M and also the shortest paths to those states. The component matching algorithm [16] computes the path from a source state to a destination state as the sequence of environment signals required to reach the destination state from the source state. However, the proposed algorithm also computes another path between two states as a sequence of state labels beginning with the state labelling of the source state, followed by the state labels of all intermediate states and finishing with the labelling of the destination state. This extra information is used to match states based on their labels. All computed states and paths are stored in a set called $RS(M)$. An initial set of blocks is created by pairing each s_f in F with all members of $RS(M)$. One of the blocks is chosen as a refining block and all other blocks are refined based on this. The refinement process stops when a fixed point is reached. The modifier D is then generated from the refined set of blocks following a similar method to the interface generation algorithm in the component matching algorithm [16]. The total complexity of the proposed algorithm is $O(NS_F^2 \times NS_M^2 \times m)$ where NS_F and NS_M are the number of states in F and M respectively and $m = ||\Sigma_M||$.

7 Results

The following theorems prove that forced simulation is the sufficient (theorem 1) and necessary (theorem 2) condition for automated debugging described in the earlier sections. Theorem 3 establishes that if two Kripke structures are bisimilar, they weakly satisfy the same set of properties. As automatic adaptation ensures that $D//M$ is bisimilar to F, theorem 3 guarantees that $D//M$ weakly satisfies the same set of properties satisfied by F (and is therefore equivalent to F).

Theorem 1: Given $F \sqsubseteq_{fsim} M$ there exists D such that $F \approx D//M$, where $\approx$ refers to weak bisimulation equivalence over Kripke structures.

Theorem 2: If there exists a well formed and deterministic interface D such that $F \approx D//M$, then $F \sqsubseteq_{fsim} M$.

Theorem 3: Given two Kripke structures $A < AP_A,\ S_A,\ s_{a0},\ \Sigma_A,\ \delta_A,\ \lambda_A >$ and $B< AP_B,\ S_B,\ s_{b0},\ \Sigma_B,\ \delta_B,\ \lambda_B >$ such that $A \approx B$, then for any CTL^* formula φ:

$(A \models_W \varphi) \Leftrightarrow (B \models_W \varphi)$

where $\models_W$ stands for weak satisfaction.

The proofs of these theorems are provided in Appendix A.

7.1 Implementation Results

An adaptive verification tool called ADVERT has been implemented using the Java programming language and the NuSMV model checker [3]. The steps followed for automated debugging are as follows:

(i) A model M and a specification F are extracted from SMV (symbolic model checker) language files. The extraction algorithm is incorporated in the popular NuSMV model checker [3] written in C programming language. The extraction algorithm performs a breadth-first search starting from the root (or initial) node(s) of a SMV file. The explicit details of each state traversed (labelling, transitions etc) are written to an output text-file. These output files are then used as inputs in the forced simulation algorithm.

(ii) After reading M and F as above, the forced simulation algorithm attempts to establish forced simulation between the two structures and if successful, generates a modifier that can adapt M to satisfy F. The modifier is stored as a text-file. If a forced-simulation relation does not exist between M and F, the algorithm exits with a suitable error message indicating that M is not adaptable.

The above process has been applied to debug several SMV models mainly from the collection of examples on the NuSMV Website [3]. The results of the debugging are presented in table 1. The first column contains the name and size (number of states) of the debugged model M. The next column indicates the size of the specification F. The last column indicates the type of debugging performed (forcing, disabling or both).

In some cases, such as the alternating bit protocol and the batch reactor system, a subset of the actual number of states in the original NuSMV description of the given model was used for debugging. In many cases, such as the mutual exclusion and pipeline examples, models were debugged to satisfy given fairness constraints. In other cases, such as the producer-consumer and priority queue examples, priority-based specifications were used to debug models to give priority to a certain sub-process. Most of the debugging show how a general model can be debugged to satisfy a more specific specification.

8 Conclusions

Existing formal methods can comprehensively detect inconsistencies between a given model and specification. However, the detection of such inconsistencies is followed by manual debugging of the model to satisfy the given specification. This manual debugging may be time consuming and the process of verifying and debugging might be repeated more than once. Therefore, automated design debugging in case of a failure to satisfy critical specifications is of great interest. This paper proposes a formal technique for automated model debugging. Given a model M and a failed specification F, the proposed technique can determine whether M can be automatically adapted to satisfy F. A model can be automatically debugged if it is related to the specification over a weaker simulation relation called *forced simulation* presented in this paper. The existence of forced simulation has been proved to be the necessary and sufficient condition for the proposed debugging. If M is forced similar to F, the debugging algorithm automatically generates a modifier D which is guaranteed to adapt M (in the form of $D//M$) to satisfy F.

The debugging algorithm has been implemented in the Java programming language and several models from the NuSMV collection of examples [3] have been debugged. The current implementation is limited to medium-sized models as it uses explicit representation of models and specifications and executes on the less-efficient Java platform. Work is in progress to extend the proposed

System($\lvert S_M \rvert$)	**Property($\lvert S_F \rvert$)**	**Type of adaptation**
Asynchronous 3-bit counter(7)	Priority(4)	Forcing and Disabling
Microwave(7)	No Reset(6)	Forcing only
Semaphore(12)	Fairness(7)	Forcing and Disabling
Mutual exclusion (16)	Fairness(12)	Disabing only
Producer-consumer(81)	Property(64)	Forcing and Disabling
Data driven pipeline(1000)	Property(200)	Forcing and Disabling
A robotics controller(2400)	Property(512)	Forcing and Disabling
A priority queue model (4144)	Property(128)	Forcing and Disabling
Synchronous arbiter(5120)	Property(2048)	Forcing and Disabling
3 cell distributed mutex(6579)	Property(1024)	Forcing and Disabling
Gigamax cache coherence(8192)	Property(256)	Forcing and Disabling
Batch Reactor system(25378)	Property(8000)	Forcing and Disabling
Alternating bit protocol(65620)	Property(8192)	Forcing and Disabling

Table 1
Implementation Results

algorithm to work with implicit BDD based representation [2].

References

[1] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). Symbolic Model Checking without BDDs. Technical report, Carnegie Mellon University.

[2] Burch, J. R. et al. (1990). Symbolic model checking: 10^{20} states and beyond. In *Fifth Annual IEEE Symposium on Logic in Computer Science*.

[3] Cavada, R., Cimatt, A., Olivetti, E., Pistore, M., and Roveri, M. (2003). *NuSMV 2.1 User Manual*.

[4] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. In *ACM Transactions on Programming Languages and Systems*, volume 8, pages 244–263.

[5] Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. The MIT Press.

[6] Groce, A., Peled, D., and Yannakakis, M. (2002). Adaptive model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 357–370.

[7] Gunter, E. and Peled, D. (2002). Temporal debugging for concurrent systems. In *Tools and Algorithms for the Construction and Analysis of Systems : 8th International Conference, TACAS 2002*, page 431.

[8] Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

[9] Hughes, G. E. and Creswell, M. J. (1977). *Introduction to Modal Logic.* Methuen.

[10] Kupferman, O. and Vardi, M. (1996). Module checking [model checking of open systems]. In *Computer Aided Verification. 8th International Conference, CAV '96*, pages 75–86, Berlin, Germany. Springer-Verlag.

[11] Manna, Z. and Pnueli, A. (1983). How to cook a temporal proof system for your pet language. In *10th annual symposium on Principles of Programming Languages*, pages 141–154.

[12] Milner, R. (1989). *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, N.J.

[13] Park, D. (1981). Concurrency and automata on infinite sequences. In *5th Conference on Theoretical Computer Science.*

[14] Peled, D. A. (2001). *Software Reliability Methods.* Springer.

[15] Ramadge, P. J. G. and Wonham, W. M. (1989). The control of discrete event systems. *Proceedings of the IEEE*, 77:81–98.

[16] Roop, P. S. and Sowmya, A. (2001). Forced simulation: A technique for automating component reuse in embedded systems. In *ACM Transactions on Design Automation of Electronic Systems.*

[17] Sowmya, A. and Ramesh, S. (1998). Extending statecharts with temporal logic. *IEEE Tr. on Software Engineering*, 24(3):216–231.

[18] Vardi, M. Y. (1987). Verification of concurrent programs: The automata theoretic framework. In *2nd IEEE Symp. on Logic in Computer Science.*

Appendix A: Forced Simulation theorems

Theorem 1. Given $F \sqsubseteq_{fsim} M$ there exists D such that $F \approx (D//M)$, where $\approx$ refers to weak bisimulation equivalence over extended Kripke structures.

Proof.

The proof of the theorem involves the generation of a modifier given $F \sqsubseteq_{fsim} M$.

Given $F \sqsubseteq_{fsim} M$, there exists an f−simulation Y between F and M. For simplicity, let Y be a minimal f−simulation relation such that for any pair $s_f \in S_f, s_m \in S_M$, there is at most one $\sigma \in \Sigma^*$ with $(s_f, s_m, \sigma) \in Y$. The proof can be carried out, however, even for a non minimal Y.

Let D be $< S_D, (s_{f0}, s_{m0}, \sigma_0), \delta_D, \lambda_D, \Sigma_D >$ where:

(i) $S_D = Y$ is the set of states of D,

(ii) $(s_{f0}, s_{m0}, \sigma_0) \in Y$ is the start state of D,

(iii) $\lambda_D(s_d) = \{\text{True}\}$

(iv) $\Sigma_D = \{[a] | a \in \Sigma_M\} \bigcup \Sigma_M$ is the set of events,

(v) δ_D, the transition relation is defined by the following rules:

- if $(s_f, s_m, a.\sigma) \in Y$ and $s_m \xrightarrow{a} s'_m$ then $(s_f, s_m, a.\sigma) \xrightarrow{[a]} (s_f, s'_m, \sigma)$ or $\delta_D((s_f, s_m, a.\sigma), [a], (s_f, s'_m, \sigma))$.
  In this case $Lab(s_d) = \{[a]\}$ ($Lab(s)$ returns the set of actions the state s can react to make a transition).
- if $(s_f, s_m, \epsilon) \in Y$ and $(s'_f, s'_m, \sigma') \in Y$ and $s_f \rightarrow s'_f$ and $s_m \xrightarrow{a} s'_m$ then $(s_f, s_m, \epsilon) \xrightarrow{a} (s'_f, s'_m, \sigma')$ or $\delta_D((s_f, s_m, \epsilon), [a], (s'_f, s'_m, \sigma'))$.
  In this case $Lab(s_d) = \{a | (a \in \Sigma_M) \wedge (\delta_D(s_d, a, s'_d)$ for some $s'_d \in S_D)\}$

The following set of simple observations and lemmata follow directly from the definition of D.

Observation 1: For every $s_f \in S_F$ there exists $(s_f, s_m, \sigma) \in S_D$ for some $s_m \in S_M$ and $\sigma \in \Sigma_M^*$.

Observation 2: Suppose $s \in S_D$ is $(s_f, s_m, a.\sigma)$. Then $Lab(s) = \{[a]\}$.

Observation 3: Suppose $s \in S_D$ is (s_f, s_m, ϵ). Then $Lab(s) = \{a | (a \in \Sigma_M) \wedge (\delta_D(s, a, s')$ for some $s' \in S_D)\}$

Observation 4: D is well formed.

Observation 5: D is deterministic.

Based upon the observations, we can prove a set of lemmata useful for the proof of the theorem.

Consider the composition of D and M, $(D//M) = < S_{(D//M)}, (s_{d0}, s_{m0}), \delta_{(D//M)}, \lambda_{(K_D//M)}, \Sigma_{(D//M)} >$. Then the following lemmata state some properties of $(D//M)$.

Lemma 1:

For any state $s = ((s_f, s_m, \sigma), s'_m) \in S_{(D//M)}$, $s_m = s'_m$.

For the following lemmata let $s, s' \in S_{(D//M)}$ such that s is $((s_f, s_m, \sigma), s_m)$ and $s' = ((s'_f, s'_m, \sigma'), s'_m)$.

Lemma 2:

$s \xrightarrow{\tau} s'$ if and only if $s_f = s'_f$, $\sigma = a.\sigma'$ and $s_m \xrightarrow{a} s'_m$ for some $a \in \Sigma_M$.

This lemma establishes the fact that in $(D//M)$, any internal state has an observable successor, which does not need to necessarily be its immediate successor. In other words, every internal state reaches an external state after a number of τ actions.

Corollary 1: Let $\sigma \neq \epsilon$. Then there exists s''_m such that:

$s_m \xrightarrow{\sigma} s''_m$ and $s(\xrightarrow{\tau^*})((s_f, s''_m, \epsilon), s''_m)$. Note that $\xrightarrow{\sigma}$ is the usual transitive closure over $\xrightarrow{a}$ and $(\xrightarrow{\tau^*})$ is the reflexive and transitive closure over $\xrightarrow{\tau}$.

Lemma 3:

$s \xrightarrow{a} s'$ if and only if $s_f \rightarrow s'_f$ and $\sigma = \epsilon$.

Lemma 4:

$(D//M)$ is deterministic.

Now, to prove the main theorem, a relation $\mathcal{X}$ over states of F and $(D//M)$ is defined as follows:

For any $s_f \in S_F$ and $((s'_f, s_m, \sigma), s_m) \in S_{(D//M)}$,

$s_f \mathcal{X}$ $((s'_f, s_m, \sigma), s_m)$ if and only if $s_f = s'_f$.

Lemma 5:

$\mathcal{X}$ is a weak bisimulation over F and $(D//M)$.

Proof:

To prove that $\mathcal{X}$ is a weak bisimulation, the following must be established:

1: $s_{f0} \mathcal{X}$ $((s_{f0}, s_{m0}, \sigma_0), s_{m0})$.

Given any $s_f \in S_F$ and $s = ((s_f, s_m, \sigma), s_m) \in S_{(D//M)}$ for some $s_m \in S_M$ and $\sigma \in \Sigma_M^*$, if $s_f \mathcal{X} s$ then

2: If $s_f \underset{\alpha}{\rightarrow} s'_f$, then there exists $s' = ((s'_f, s'_m, \sigma'), s'_m)$ such that $s \underset{\hat{\alpha}}{\Rightarrow} s'$ and $s'_f \mathcal{X}$ s'.

3: If $s \underset{\alpha}{\rightarrow} s' = ((s'_f, s'_m, \sigma'), s'_m) \in S_{(D//M)}$, there exists s'_f such that $s_f \underset{\hat{\alpha}}{\Rightarrow} s'_f$ and s'_f $\mathcal{X}$ s'.

Proof:

1 follows directly from the definition of $\mathcal{X}$.

2 can be proved as follows:

Assume $s_f \rightarrow s'_f$ or $s_f \underset{E}{\rightarrow} s'_f$ where $\lambda_F(s_f) = P_0$ and $\lambda_F(s'_f) = P_1$. In this case $\alpha = E$ (a single external transition sequence).

For any $s = ((s_f, s_m, \sigma), s_m) \in S_{(D//M)}$, there are two possibilities:

Sol. 1 If $\sigma = \epsilon$.

Then $\lambda_{(D//M)}(s) = \lambda_F(s_f) = \lambda_M(s_m) = P_0$

By lemma 3, $s \xrightarrow{a} s'((s'_f, s'_m, \sigma'), s'_m)$ for some action $a \in \Sigma_{(D//M)}$ or $s \underset{E}{\rightarrow} s'$

Again, there are two possibilities:

(i) If $\sigma' = \epsilon$.

Then $\lambda_{(D//M)}(s') = \lambda_F(s'_f) = \lambda_M(s'_m) = P_1$

Therefore, $s \underset{E}{\rightarrow} s'$ or $s \underset{\hat{\alpha}}{\Rightarrow} s'$

Also, $s'_f \ \mathcal{X} s'$ by definition of $\mathcal{X}$.

(ii) If $\sigma' \neq \epsilon$. By corollary 1, there exists $s''((s'_f, s''_m, \epsilon), s''_m)$ such that $s'(\overset{\tau^*}{\longrightarrow})s''$ or $s' \underset{\tau^*}{\rightarrow} s''$.

Also, $(\lambda_{(D//M)}(s') = \{intern\})$ and $(\lambda_{(D//M)}(s'') = P_1)$.

So $s' \underset{\alpha'}{\rightarrow} s''$ where $\alpha' = \tau^*$.

Also, $s \underset{E}{\rightarrow} s'$

Therefore $s \underset{\alpha''}{\rightarrow} s''$ where $\alpha'' = E.\tau^*$.

In this case $\hat{\alpha''} = \alpha$ and therefore $s \underset{\hat{\alpha}}{\Rightarrow} s''$

Also, $s'_f \ \mathcal{X} s''$ by definition of $\mathcal{X}$.

Sol. 2 If $\sigma \neq \epsilon$.

By corollary 1, there exists $s''((s_f, s''_m, \epsilon), s''_m)$ such that $s(\overset{\tau^*}{\longrightarrow})s''$ or $s \underset{\tau^*}{\rightarrow} s''$. Also, $\lambda_{(D//M)}(s) = \{intern\}$ and $\lambda_{(D//M)}(s'') = P_0$

So $s \underset{\alpha'}{\rightarrow} s''$ where $\alpha' = \tau^*$. (1)

From Sol. 1 above, we can prove that $s'' \underset{\hat{\alpha}}{\Rightarrow} s'$ where $s' = ((s'_f, s'_m, \epsilon), s'_m)$. (2)

Combining (1) and (2), we get

$s \underset{\alpha'}{\rightarrow} s'' \underset{\hat{\alpha}}{\Rightarrow} s'$

As $\alpha' = \tau^*$ (or one or more internal transitions), we get

$s \underset{\hat{\alpha}}{\Rightarrow} s'$

Also, $s'_f \ \mathcal{X} s'$ by definition of $\mathcal{X}$.

The same proof can be extended to accommodate $\alpha = E^*$ (a sequence of more than one external transitions) by applying Sol 1 and 2 recursively.

3 can be proved as follows: Let $s \underset{\alpha}{\rightarrow} s'$ where

$s = ((s_f, s_m, \sigma), s_m)$ and $s = ((s'_f, s'_m, \sigma'), s'_m) \in S_{(D//M)}$ and α represents a single transition. There are two distinct possibilities:

(i) $\sigma \neq \epsilon$.

Then $\lambda_{(D//M)}(s) = \{intern\}$ and $\alpha = \tau$. Therefore, $s \underset{\tau}{\rightarrow} s'$ (internal transition forced by the interface) or

$s \xrightarrow{\tau} s'$.

Therefore $s_f = s'_f$

$\hat{\tau} = \epsilon$

We know that $s_f \underset{\epsilon}{\Rightarrow} s_f$

Also, $s_f \ \mathcal{X} s'$ by definition of $\mathcal{X}$.

(ii) $\sigma = \epsilon$.

$\lambda_{(D//M)}(s) \neq \{intern\}$ and $s \underset{E}{\rightarrow} s'$ (externally observable transition) or

$s \xrightarrow{a} s'$ for some a in $\Sigma_{(D//M)}$.

Using Lemma 3, as $s \xrightarrow{a} s'$ and $\sigma = \epsilon$, $s_f \rightarrow s'_f$.

Also, $s'_f \ \mathcal{X} s'$ by definition of $\mathcal{X}$.

Theorem 1 establishes that forced simulation is a sufficient condition for forced model checking. The next theorem shows the existence of forced simulation as a necessary condition for D.

□

Theorem 2. *If there exists a well formed and deterministic interface D such that $F \approx (D//M)$, then $F \sqsubseteq_{fsim} M$.*

Proof.

Assume that $F \approx (D//M)$. Then, there exists a weak bisimulation relation $\mathcal{S}$ over F and $(D//M)$. Given $\mathcal{S}$ a new relation $Y \subseteq S_F \times S_M \times \Sigma_M^*$ is constructed.

By assumption D is well-formed and deterministic. Also M is deterministic. It is easy to show that then $(D//M)$ is deterministic. Further, for any state s in $(D//M)$,

Let Y be the smallest relation such that (s_f, s_m, σ) is in Y if and only if there exists $s_d \in D$ and $(s_d, s_m) \in S_{(D//M)}$ with $(s_f, (s_d, s_m)) \in \mathcal{S}$ and one of the following holding:

(i) $\sigma = \epsilon$ and $\lambda_M(s_d, s_m) = \lambda_F(s_f) \neq \{intern\}$ (no forcing, disabling might be present).

(ii) $s_d \overset{[\sigma]}{\to} s'_d$ is not a forcing symbol where $[\sigma]$ is the sequence of forcing symbols appearing in σ. (forcing)

It is quite easy to establish that Y as defined is a forced simulation relation.

So far, a new simulation relation between F and M is proposed and Theorem 1 and Theorem 2 together establish that forced simulation is a necessary and sufficient condition for the existence of a correct D which can adapt M to satisfy F.

□

Theorem 3. Given two extended Kripke structures $A < AP_A,\ S_A,\ s_{a0},\ \Sigma_A,\ \delta_A,\ \lambda_A >$ and $B < AP_B,\ S_B,\ s_{b0},\ \Sigma_B,\ \delta_B,\ \lambda_B >$ such that $A \approx B$, then for any CTL^* formula φ:

$(A \models_W \varphi) \Leftrightarrow (B \models_W \varphi)$

where $\models_W$ stands for weak satisfaction.

Restriction 1: The proof of this theorem is based on the restriction that an unobservable state may always lead to one or more observable states.

This restriction is an important consideration as for temporal logics like CTL^*, every state must have an observable successor. If an internal state has an infinite path starting from it where no state in this path is observable, the given model will keep undergoing an infinite number of τ transitions and will not interact with its environment any more, entailing that it has terminated.

Before embarking on the proof of theorem 3, the following Lemmas that assist the proof are presented. The following Lemmas work on arbitrary states s_a and s_b from the state-spaces of the models A and B respectively.

Lemma 6:

If $s_a \approx s_b$ then for all s'_a where $s'_a \in Next_{ob}(s_a)$, there is a state $s'_b \in Next_{ob}(s_b)$ such that $s'_a \approx s'_b$.

Proof:

By definition of $Next_{ob}$, for any $s'_a \in Next_{ob}(s_a)$, $s_a \underset{\alpha}{\to} s'_a$ where there are the following two possibilities:

(i) s_a is an internal state: $\alpha = \tau^i$ where $i \geq 1$ and s'_a is an observable state.

In this case, $\hat{\alpha} = \epsilon$ and $s_a \underset{\epsilon}{\Rightarrow} s'_a$

(ii) s_a is an observable state: $\alpha = \epsilon$ and $s'_a = s_a$. In this case, $\hat{\alpha} = \alpha = \epsilon$ and $s_a \underset{\epsilon}{\Rightarrow} s'_a$

As described above, regardless of whether s_a is an observable state (or otherwise), $s_a \underset{\epsilon}{\Rightarrow} s'_a$ and that s'_a is an observable state. By definition of bisimulation (definition 4.5) on page 14, as $s_a \approx s_b$, there must be a state s'_b such that $s_b \underset{\epsilon}{\Rightarrow} s'_b$ and $s'_a \approx s'_b$.

It is now important to prove that s'_b is (or leads to) a state in $Next_{ob}(s_b)$.

Consider such a state s'_b such that $s_b \underset{\epsilon}{\Rightarrow} s'_b$ and $s_a \approx s'_b$. There are two distinct possibilities:

- s'_b is observable: By definition of $Next_{ob}$, $s_b \in Next_{ob}(s_b)$.

- s'_b is unobservable: By restriction 1 stated above, s'_b will always lead to at least one observable state s''_b such that $s'_b \underset{\tau^j}{\rightarrow} s''_b$ where $j \geq 1$. Therefore, $s'_b \underset{\epsilon}{\Rightarrow} s''_b$. It is quite clear that as $s_b \underset{\epsilon}{\Rightarrow} s'_b \underset{\epsilon}{\Rightarrow} s''_b$, $s''_b \in Next_{ob}(s_b)$ (by definition of $Next_{ob}$).

  Also, it is known that for the observable state s'_a, $s'_a \underset{\epsilon}{\Rightarrow} s'_a$. Again by definition of bisimulation, $s'_a \approx s''_b$ and both are observable states.

The same proof can be used to demonstrate that for any $s'_b \in Next_{ob}(s_b)$, there is a state $s'_a \in Next_{ob}(s_a)$ such that $s'_a \approx s'_b$.

Lemma 7:

If $s_a \approx s_b$ where both s_a and s_b are observable states, then for each $s'_a \in Succ_{ob}(s_a)$, there is a state $s'_b \in Succ_{ob}(s_b)$ such that $s'_a \approx s'_b$.

Proof

Given: both s_a and s_b are observable states.

Consider a state $s'_a \in Succ_{ob}(s_a)$. By definition of $Succ_{ob}$, $s_a \underset{\alpha}{\rightarrow} s'_a$ where $\alpha = E.\tau^i$ and $i \geq 0$. In this case, $\hat{\alpha} = \mathrm{E}$.

By definition of bisimulation (definition 4.5 on page 14), if $s_a \approx s_b$, then if $s_a \underset{\alpha}{\rightarrow} s'_a$ for some $\alpha \in Trans^*$ then $s_b \underset{\hat{\alpha}}{\Rightarrow} s''_b$ and $s'_a \approx s'_b$.

Therefore $s_b \underset{E}{\Rightarrow} s'_b$ for some s'_b and $s'_a \approx s'_b$.

It is now shown that s'_b is (or leads to) a state in $Succ_{ob}(s_b)$. As $s_b \underset{\hat{\alpha}}{\Rightarrow} s''_b$ and $s'_a \approx s'_b$, there are two possibilities:

- s'_b is observable: By definition of $Succ_{ob}$, $s_b \in Succ_{ob}(s_b)$.

- s'_b is unobservable: By restriction 1 stated above, s'_b will always lead to atleast one observable state s''_b such that $s'_b \underset{\tau^j}{\rightarrow} s''_b$ where $j \geq 1$. Therefore, $s'_b \underset{\epsilon}{\Rightarrow} s''_b$. It is quite clear that as $s_b \underset{E}{\Rightarrow} s'_b \underset{\epsilon}{\Rightarrow} s''_b$, $s''_b \in Succ_{ob}(s_b)$ (by definition of $Succ_{ob}$).

  Also, it is known that for the observable state s'_a, $s'_a \underset{\epsilon}{\Rightarrow} s'_a$. Again by definition of bisimulation, $s'_a \approx s''_b$ and both are observable states.

The same reasoning can be applied to demonstrate that for every state $s'_b \in Succ_{ob}(s_b)$, there is a state $s'_a \in Succ_{ob}(s_a)$ such that $s'_a \approx s'_b$.

Lemma 8:

If $s_a \approx s_b$, then for every observable path starting from s_a, there is an equivalent observable path starting from s_b and vice versa.

Proof

Let that $s_a \approx s_b$. Let $\pi_{ob}(s_a) = s_{a1}, s_{a2}, s_{a3}, ...$ be an observable path starting from s_a. By definition of observable paths, $s_{a1} \in Next_{ob}(s_f)$, $s_{a2} \in Succ_{ob}(s_{a1})$, $s_{a3} \in Succ_{ob}(s_{a2})$ and so on.

A corresponding observable path $\pi_{ob}(s_b) = s_{b1}$, s_{b2}, s_{b3}, ... starting from s_b is constructed by induction on the structure of $\pi_{ob}(s_a)$.

As $s_a \approx s_b$, it is known that for the state $s_{a1} \in Next_{ob}(s_a)$, there is a state $s_{b1} \in Next_{ob}(s_b)$ such that $s_{a1} \approx s_{b1}$ (by Lemma 6). This state s_{a1} is chosen as the first observable state of $\pi_{ob}(s_b)$.

Similarly, the second state in $\pi_{ob}(s_b)$ can be chosen as the state $s_{b2} \in Succ_{ob}(s_{a1})$ with $s_{a2} \approx s_{b2}$. Lemma 7 guarantees that there exists such a state s_{b2}.

The path $\pi_{ob}(s_b)$ can then be further constructed by choosing states $s_{b(i)} \in Succ_{ob}$ that are bisimilar to $s_{a(i)}$.

Assuming $s_{a(i)} \approx s_{b(i)}$ for some $i \geq 1$. Consider the next state $s_{b(i+1)}$ in $\pi_{ob}(s_b)$. It is known that $s_{a(i+1)} \in Succ_{ob}(s_{a(i)})$. Using Lemma7, there must be an observable successor s'_b of $s_{b(i)}$ such that $s_{a(i+1)} \approx s'_b$. This state s'_b can then be chosen as $s_{b)(i+1)}$ to construct $\pi_{ob}(s_b)$.

Given a path $\pi_{ob}(s_b)$ starting from s_b, the construction of $\pi_{ob}(s_a)$ is similar.

Lemma 9:

Let φ be either a state formula or a path formula. If s_a and s_b are bisimilar states and $\pi_{ob}(s_a)$ and $\pi_{ob}(s_b)$ are their corresponding paths, then:

- If φ is a state formula then if $s_a \models_W \varphi \Leftrightarrow s_b \models_W \varphi$.
- If φ is a path formula then if $pi(s_a) \models_W \varphi \Leftrightarrow pi(s_b) \models_W \varphi$.

Proof
The proof of this lemma is inductive on the structure of φ.
Basis:
If $\varphi = p$ for $p \in AP_A$.
If $s_a \models_W p$ then $Next_{ob}(s_a) \models p$ (by definition 5.5).
Now, $s_b \models_W p$ provided $Next_{ob}(s_a) \models p$.
By lemma 6, $Next_obs_a \approx Next_{ob}(s_b)$.
Hence, $\lambda_A(Next_{ob}(s_a)) = \lambda_B(Next_{ob}(s_b))$ (characteristic of weak bisimulation: definition 4.5 on page 14).
Hence, $Next_{ob}(s_b) \models p$.
Therefore, $s \models_W p$.
Induction: Consider the following cases:
Case 1. $\varphi = \neg\varphi_1$, a state formula.
$s_a \models_W \varphi \Leftrightarrow s_a \not\models_W \varphi_1$
$\Leftrightarrow s_b \not\models_W \varphi_1$ (basis)
$\Leftrightarrow s_b \models_W \varphi$
The same arguments hold if φ is a path formula

Case 2. $\varphi = \varphi_1 \vee \varphi_2$, a state formula.
$s_a \models_W \varphi \Leftrightarrow s_a \models_W \varphi_1$ or $s_a \models_W \varphi_2$.
$\Leftrightarrow s_b \models_W \varphi_1 or s_b \models_W \varphi_2$ (induction hypothesis)
$\Leftrightarrow s_b \models_W \varphi$
The same arguments hold if φ is a path formula.

Case 3. $\varphi = \varphi_1 \wedge \varphi_2$, a state formula. This case is similar to the previous case and the same arguments can be used if φ is a path formula.

Case 4. $\varphi = \mathbf{E}\varphi_1$, a state formula.
$s_a \models_W \varphi$.
This implies that $Next_ob(s_a) \models \varphi$.
Therefore, there is an observable path $\pi_{ob}(s_a)$ starting from s_a such that $\pi_{ob}(s_a) \models \varphi_1$.
By Lemma 8, there is a corresponding path $\pi_{ob}s_b$ starting from s_b.
By induction hypothesis, $\pi_{ob}(s_a) \models_W \varphi_1$ if and only if $\pi_{ob}(s_b) \models_W \varphi_1$.
Therefore, $s_b \models_W \mathbf{E}\varphi_1$.
The same arguments can be used to prove that if $s_b \models_W \varphi$ then $s_a \models_W \varphi$.

Case 5. $\varphi = \mathbf{A}\varphi_1$, a state formula. This case is similar to the previous case and the same arguments can be used.

Case 6. $\varphi = \varphi_1$, where φ is a path formula and φ_1 is a state formula.
Although the φ and φ_1 are of the same lengths, it can be conceived that $\varphi = path(\varphi)$ where path is a special operator that converts a state formula into a path formula.
Therefore, it is simplifying φ by dropping this path operator. If s_a and s_b are the start states of the paths $\pi_{ob}(s'_a)$ and $\pi_{ob}(s'_b)$, where $s'_a = Next_{ob}(s_a)$ and $s'_b = Next_{ob}(s_b)$, ($s'_a \approx s'_b$ by Lemma 6) then,
$\pi_{ob}(s'_a) \models_W \varphi \Leftrightarrow s'_a \models \varphi_1$
$\Leftrightarrow s'_b \models \varphi_1$ (induction hypothesis)
$\Leftrightarrow \pi_{ob}(s'_b) \models_W \varphi$ (

7. $\varphi = \mathbf{X}\varphi_1$, a path formula.
Assuming $\pi(s_a) \models_W \varphi$.
Therefore, the observable path $\pi_{ob}(s_a) \models \varphi$.
Therefore, $pi_{ob}(s'_a) \models \varphi_1$ where $s'_a \in Succ_{ob}(s_a)$.
Also, there is a corresponding path $\pi_{ob}(s_b)$ for $\pi_{ob}(s_a)$ (Lemma 8).
As $\pi_{ob}(s_a)$ for $\pi_{ob}(s_b)$, correspond, so will the paths $\pi_{ob}(s'_a)$ for $\pi_{ob}(s'_b)$ where $s'_b \in Succ_{ob}(s_b)$.

Therefore $\pi_{ob}(s'_b) \models \varphi_1$.
Therefore $\pi_{ob}(s_b) \models \varphi$.
Therefore $\pi(s_b) \models_W \varphi$ (definition 5.6).

8. $\varphi = \varphi_1 U \varphi_2$, a path formula. Assuming $\pi(s_a) \models_W \varphi_1 U \varphi_2$.
Therefore, the observable path $\pi_{ob}(s_a) \models \varphi$. (definition 5.6)
By definition of the until operator, there is a k such that $\pi_{ob}(s_{ak}) \models_W \varphi_2$ and for all $0 \leq j < k, \pi_{ob}(s_{aj}) \models_W f1$.
For the path $\pi_{ob}(s_a)$, there is a corresponding path $\pi_{ob}(s_b)$.
As, $\pi_{ob}(s_a)$ and $\pi_{ob}(s_b)$ correspond, so do $\pi_{ob}(s_{aj})$ and $\pi_{ob}(s_{bj})$. (lemma 8).
Therefore by induction hypothesis, $\pi_{ob}(s_{bk}) \models \varphi_2$ and for all $0 \leq j < k$,$\pi_{ob}(s_{bj})$$\models \varphi_1$.
Therefore $\pi_{ob}(s_b) \models \varphi$
Therefore $\pi(s_b) \models_W \varphi$. (definition 5.6).

9. $\varphi = \varphi_1 R \varphi_2$, a path formula. This case is similar to the previous case and the same arguments can be used.

Proof of Theorem 3

Theorem 3 is the direct consequence of the preceding Lemma. If two Kripke structures A and B are weakly similar, $s_{a0} \approx s_{b0}$ by definition of bisimulation. Therefore by Lemma 9, if π and π' are there corresponding observable paths, then:

- If φ is a state formula then if $s_a \models_W \varphi \Leftrightarrow s_b \models_W \varphi$.
- If φ is a path formula then if $pi \models_W \varphi \Leftrightarrow pi \models_W \varphi$.