Falsifying safety properties through games on over-approximating models

Nathaniel Charlton[1] and Michael Huth[2]

Department of Computing
Imperial College London

Abstract

Abstractions of programs are traditionally over-approximations and have proved to be useful for the verification of safety properties. They are presently perceived as being useless for the falsification of safety properties, i.e. showing that program execution definitely reaches a "bad" state. Alternative techniques, such as the computation of under-approximating must transitions, have addressed this shortcoming in the past. We show that over-approximating models can indeed falsify safety properties by relying on and exploiting the seriality and partial determinism of programs: programs don't just stop for no reason, and most program statements have deterministic semantics. Our method is based on solving a two-person attractor game derived from over-approximating models and makes no assumptions about the abstraction domain used. An example demonstrates the successful use of our approach, and highlights the role played by seriality and our handling of nondeterminism. Finally, we show that our method can encode must transitions, if supplied, by a simple modification of the ownership of nodes in the attractor game derived from the over-approximating model.

Keywords: software verification, games, falsification

1 Introduction

For over thirty years, over-approximating models have been used for verifying safety properties of programs. Intuitively an over-approximating model has all the behaviours of the original program, and possibly many more; this is expressed by conditions such as trace-inclusion and simulation. Verification of safety properties is based on the following observation: because an over-approximating model has at least all the behaviours of the original program, any "bad" behaviours (i.e. those that violate the desired safety property) present in the program are also present in the model. Therefore, if the model contains no bad behaviours, the program does not either. Systems such as SLAM [1], BLAST [10] and our own system HECTOR [5] have been developed which automatically extract over-approximating models from programs using abstraction.

[1] Email: `nathaniel.charlton@imperial.ac.uk`

[2] Email: `michael.huth@imperial.ac.uk`

However, the above scheme does not allow the *falsification* of safety properties, because bad behaviours found in the model need not be present in the original program – they may be "artifacts" introduced by the over-approximation process, and therefore "not feasible" in the original program. Approaches to the falsification of safety properties have focused on showing that abstract counterexamples are indeed feasible, for example by:

(i) searching for a corresponding concrete counterexample (e.g. [14]),

(ii) proving the feasibility of the abstract counterexample path by satisfiability checking (e.g. [1]), or

(iii) adding under-approximation or calculation of "must-transitions" to the model (e.g. [8,9]).

Here we present a method of falsifying safety properties which uses only over-approximating models. In particular, our method doesn't perform any of the tasks i, ii, iii above. Instead, our method is based on playing a two-player game over the transition graph of the over-approximating model, and exploits two properties of programs: seriality (execution of a program does not just "stop" for no reason) and what we call *partial determinism*. By this we mean that most program statements are deterministic, so any nondeterminism in the program or its instrumentation is confined to a small number of identifiable locations.

The rest of the paper is structured as follows: Section 2 sets out the notions of programs, models and safety properties we use. Section 3 shows how to use two-player games to falsify safety properties without performing any of the usual tasks listed above. Section 4 presents an illuminating example of falsification. Section 5 shows how our approach supports the easy incorporation of must information when it is available. Section 6 concludes and discusses related work. An appendix contains all omitted proofs, to be read at the discretion of referees.

2 Background

Programs and their semantics.

We begin by setting out the kinds of programs, models and safety properties we deal with in this paper. It will be seen that our setup is very general.

In this paper, we work with programs expressed as control flow graphs (CFGs). Figure 1 shows two simple such programs, drawn with unfilled nodes. Formally, each control flow graph is encoded by giving a set of locations *Locs*, which includes an element *start*, and a function $E : \mathit{Locs} \to \mathit{Edges}$ mapping each location l to the single (hyper)edge leaving it. The allowed forms of edges are:

- Conditional (hyper)edges: $\text{if}(\Phi) : l_1 : l_2$. These transfer control to location l_1 if the condition Φ holds, and to l_2 otherwise.
- Edges for ordinary statements: $\text{f} : l$. These execute the statement f and transfer control to location l.
- Choice (hyper)edges: $\text{choice} : l_1 : l_2$. These represent nondeterminstic choice, and (in a sense) transfer control to *both* l_1 and to l_2.

(Here Φ is a guard condition, and $[\![\Phi]\!] \subseteq \mathit{State}$ denotes the set of states in which

Φ holds. In this paper we assume nothing about guard conditions Φ or $[\![-]\!]$.)

To give semantics to our programs, we assume a set *State* of program states, including an initial state s^{init} in which execution begins. We also assume that for each ordinary statement f, an associated transfer function $f : State \to State$ is given.

Note that we work at a high level of abstraction, assuming nothing about the nature of the state space *State* and the transfer functions f, so that for instance our results apply equally to languages with heaps as to those without.

As is customary, we put a transition system semantics onto programs. A program's transition system has state space $Locs \times State$, whose elements we call configurations. The transition relation $\to \; \subseteq \; State \times State$ is given by the following (named) rules:

ord

$$\frac{E(l) = \mathrm{f} : l'}{(l, s) \to (l', f(s))}$$

choice-1

$$\frac{E(l) = \text{choice} : l_1 : l_2}{(l, s) \to (l_1, s)}$$

choice-2

$$\frac{E(l) = \text{choice} : l_1 : l_2}{(l, s) \to (l_2, s)}$$

if-true

$$\frac{E(l) = \text{if}(\Phi) : l_1 : l_2 \qquad s \in [\![\Phi]\!]}{(l, s) \to (l_1, s)}$$

if-false

$$\frac{E(l) = \text{if}(\Phi) : l_1 : l_2 \qquad s \notin [\![\Phi]\!]}{(l, s) \to (l_2, s)}$$

We say that a configuration (l, s) is *reachable* if there exists a sequence $(l_1, s_1) \to \cdots \to (l_k, s_k)$ such that $(l_1, s_1) = (start, s^{init})$ and $(l_k, s_k) = (l, s)$.

We stated in the introduction that our development will depend on seriality and determinism, so we establish a lemma for these, which looks fairly innocuous but will be crucial later.

Lemma 2.1 Seriality and partial determinism of $\to$. *The concrete transition relation $\to$ is*

(i) *serial, i.e. for all configurations (l, s), there exists a configuration (l', s') such that $(l, s) \to (l', s')$.*

(ii) *partially deterministic, i.e. for all configurations (l, s) with $E(l)$ not of form choice : $l_1 : l_2$, if $(l, s) \to (l', s')$ and $(l, s) \to (l'', s'')$ then $(l', s') = (l'', s'')$.*

Abstraction domains.

Our abstract models of programs will be built from *abstraction domains.* An abstraction domain in this paper will consist of a set A of *abstract values*, and a *concretisation function* $\gamma : A \to \mathbb{P}(State)$ which gives meaning to the abstract values. This very general formulation is all that is needed in this paper, and so our results apply to arbitrary abstraction domains, though in practise an abstraction domain

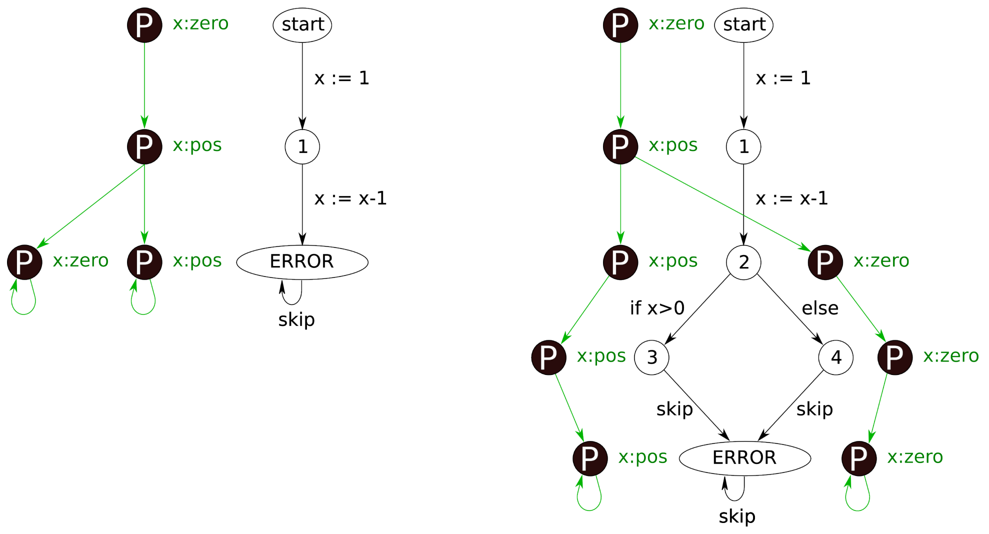

Fig. 1. The unfilled nodes, and their associated transitions, show the control flow graphs of two simple programs. The attached filled nodes, and their associated (green) transitions, depict sound default augmented models for the programs, built using sign analysis. Our method establishes that both programs reach their terminated states, unlike a conventional treatment of must transitions e.g. [8].

comes with more components, such as abstract successor functions (see e.g. [13,6]). Our development here also applies to the *analysis modules* presented in [4,3].

Model checking queries.

In this paper, we will consider a particular type of safety property: our queries will be expressed by giving a set $B \subseteq \mathit{Loc} \times \mathit{State}$ of "bad" configurations, and asking whether any of the bad configurations are reachable in the program. If no $b \in B$ is reachable, then the safety property represented by B is true (which we will abbreviate to "B is true"). On the other hand if some $b \in B$ is reachable, then the safety property represented by B is false (which we will abbreviate to "B is false"). We will not go into the issue of how one abstractly represents such a set B.

Definition 2.2 Abstract (over-approximating) models. An abstract model $M = (N, \xrightarrow{abs})$ for a given program, and built using the abstraction domain (A, γ), consists of a set $N \subseteq \mathit{Locs} \times A$ of abstract nodes, and an abstract transition relation $\xrightarrow{abs} \subseteq N \times N$, satisfying the following healthiness conditions:

H1 If $(l, a) \xrightarrow{abs} (l', a')$ then $E(l)$ must be some edge with l' as a target.

H2 If $(l, a) \xrightarrow{abs} (l', a')$ and $E(l)$ is a choice edge, then $a' = a$.

H1 ensures that transitions in the model only occur between locations that are connected by CFG edges, so that the transition structure of the model falls into line with the structure of the CFG. H2 ensures that choice edges are treated simply as junctions, by not allowing the abstract value to change across a choice edge. □

Figures 1 and 2 (page 9) are examples of such abstract models, drawn using the filled nodes (ignoring the P/F annotations for now). These models are built using a sign analysis, an abstraction domain which tracks only the sign of each variable, i.e. whether it is negative, zero or positive, and discards all other information.

Definition 2.3 Soundness of abstract models. An abstract model M as above is said to be *sound* if the following, standard simulation-type, condition holds:

S1 Let $(l, a) \in N$ and (l, s) be such that $s \in \gamma(a)$. Let $(l, s) \to (l', s')$. Then there exists $(l', a') \in N$ such that $(l, a) \xrightarrow{abs} (l', a')$ and $s' \in \gamma(a')$. □

It can be seen that the models in Figure 1 are sound. Verification of safety properties, in the standard way described in the introduction, can be performed on the basis of S1; we will not dwell on this as we concentrate on falsification here.

Lemma 2.4 Seriality of $\xrightarrow{abs}$. *The abstract may transition relation $\xrightarrow{abs}$ is serial in sound models, in the sense that for all nodes $n = (l, a) \in N$, provided $\gamma(a)$ is nonempty there exists a node n' such that $n \xrightarrow{abs} n'$.*

Proof: Let $s \in \gamma(a)$. By Lemma 2.1 (seriality) there exists (l', s') such that $(l, s) \to (l', s')$. Applying S1 completes the proof. □

3 Our games on over-approximating models

We now show how to use two-player games to falsify safety properties. The intuition of what follows is that, given a safety property B, we are going to play a two-person game, where the positions are the nodes $n \in N$ of an over-approximating model.

We call the players F and P: player F is trying to Falsify B, and player P is trying to Prevent this from happening. A move at position n means choosing n' such that $n \xrightarrow{abs} n'$; node n' becomes the new position.

An extra function ρ determines which player is to move at each position. Player F wins by forcing the game into a position $n = (l, a)$ where for all $s \in \gamma(a)$, we have $(l, s) \in B$, i.e. all concrete configurations represented by n are "bad".

Definition 3.1 Augmented models An augmented model $M^+ = (N, \xrightarrow{abs}, \rho)$ consists of an abstract model $(N, \xrightarrow{abs})$ along with a function $\rho : N \to \{F, P\}$. □

Of course, we cannot just use any old partition of the nodes among the two players. The following definition sets out what we require from such a partition. (In A1 the quantifier pattern is $\forall\exists$, and in A2 it is $\exists\exists$, which is reminiscent of the relations $R^{\forall\exists}$ and $R^{\exists\exists}$ from [7].)

Definition 3.2 Soundness of augmented model. An augmented model $M^+ = (N, \xrightarrow{abs}, \rho)$ is said to be *sound* if $(N, \xrightarrow{abs})$ is sound and the following hold:

A1 Let $\rho((l, a)) = F$ and $s \in \gamma(a)$. Then for all (l', a') such that $(l, a) \xrightarrow{abs} (l', a')$, there exists $s' \in \gamma(a')$ such that $(l, s) \to (l', s')$.

A2 Let $\rho((l, a)) = P$ and $s \in \gamma(a)$. Then there exists (l', a') such that $(l, a) \xrightarrow{abs} (l', a')$ and there exists $s' \in \gamma(a')$ such that $(l, s) \to (l', s')$. □

The models in Figure 1 and Figure 2 (page 9) are sound augmented models, with the ρ function depicted with P/F annotations at each abstract node.

To perform falsification, we introduce a judgement $\mathbf{H}n$ which means that the abstract node (or game position) $n \in N$ is "Hopeless" with respect to the set

B of bad configurations, i.e. that once execution reaches n there is no hope for avoiding forever the set of bad configurations B. The named derivation rules for this judgement are as follows.

h-all-bad

$$\frac{\{(l,s) \mid s \in \gamma(a)\} \subseteq B}{\mathbf{H}(l,a)}$$

h-P-move

$$\frac{\rho(n) = P \qquad \forall n' \in N,\ \text{if } n \xrightarrow{abs} n' \text{ then } \mathbf{H}n'}{\mathbf{H}n}$$

h-F-move

$$\frac{\rho(n) = F \qquad \exists n' \in N \text{ such that } n \xrightarrow{abs} n' \text{ and } \mathbf{H}n'}{\mathbf{H}n}$$

The following theorem shows that what we are calling "hopeless" nodes really do inevitably lead to a bad configuration. Its subsequent corollary justifies falsification using sound augmented models and the judgement $\mathbf{H}$.

Theorem 3.3 *Let M^+ be a sound augmented model, and B a safety property. Let $n = (l,a) \in N$ and $s \in \gamma(a)$. If $\mathbf{H}n$ then there exists $(l_1,s_1) \to \cdots \to (l_k,s_k)$, with $(l_1,s_1) = (l,s)$ and $(l_k,s_k) \in B$.*

Proof: We proceed by structural induction on the derivation of $\mathbf{H}n$.

Base case: The base case is when $\mathbf{H}n$ is derived by a single application of a rule. This can only be the h-all-bad rule. (At first glance it appears there is a possibility of using h-P-move if n has no successors, but since $s \in \gamma(a)$, Lemma 2.4 shows this is impossible.) From the premises of h-all-bad and $s \in \gamma(a)$, we have $(l,s) \in B$. Taking the one-element sequence (l,s) we are done.

Inductive case for h-P-move: From the premises of h-P-move we have $\rho(n) = P$. Applying A2, we see that there exists (l',a') such that $(l,a) \xrightarrow{abs} (l',a')$, and there exists $s' \in \gamma(a')$ such that $(l,s) \to (l,s')$. Now, also from the premises of h-P-move, we see that $\mathbf{H}(l',a')$. Applying the induction hypothesis and renumbering, we obtain $(l_2,s_2),\ldots,(l_k,s_k)$ such that $(l_2,s_2) \to \cdots \to (l_k,s_k)$, with $(l_2,s_2) = (l',s')$ and $(l_k,s_k) \in B$. Set $(l_1,s_1) := (l,s)$ and we are done.

Inductive case for h-F-move: By the premises of h-F-move there exists $n' = (l',a')$ such that $n \xrightarrow{abs} n'$ and $\mathbf{H}n'$. Also from the premises of h-F-move we have $\rho(n) = F$. This means we can apply A1 to obtain an $s' \in \gamma(a')$ such that $(l,s) \to (l,s')$. Applying the induction hypothesis to $n' = (l',s')$ and renumbering, there exists a sequence $(l_2,s_2),\ldots,(l_k,s_k)$ such that $(l_1,s_1) \to \cdots \to (l_k,s_k)$, with $(l_2,s_2) = (l',s')$ and $(l_k,s_k) \in B$. Set $(l_1,s_1) := (l,s)$ and we are done. □

Corollary 3.4 Falsification with sound augmented models. *Let M^+ be a sound augmented model, and let B be a safety property. Let $(start,a) \in N$ such that $s^{init} \in \gamma(a)$. If $\mathbf{H}(start,a)$ then B is false.* □

The above is all well and good, but how do we obtain a sound augmented model for a program? Below we present a way to turn any sound over-approximating model (supporting verification only) into a sound augmented model (supporting falsification also). The construction is very easy, and simply assigns all abstract nodes corresponding to choice edges to player F, and all other nodes to player P.

Augmented models constructed in this way capture precisely the "must information" implicitly present in the original over-approximating model: at choice nodes *all* choices are taken, and at other nodes *some* choice must be taken, as execution cannot simply stop. Such models allow, as we shall see, the falsification of some safety properties, by means of solving an attractor game.

Definition 3.5 Let M be a sound model for a program. We define the *default augmented model* for M to be $M^+ = (M, \rho)$ where

$$\rho(l, a) := \begin{cases} F & \text{if } E(l) \text{ has the form choice} : l' : l'' \\ P & \text{otherwise} \end{cases} \qquad \square$$

The augmented models in Figures 1 and 2 are all default augmented models.

Theorem 3.6 *Let M be a sound model. Then the default augmented model M^+ is a sound augmented model.*

Proof of A1: Let $\rho((l, a)) = F$ and $s \in \gamma(a)$. Let also (l', a') be such that $(l, a) \xrightarrow{abs} (l', a')$. Since $\rho((l, a)) = F$, edge $E(l)$ has form $\text{choice} : l_1 : l_2$. By H1, either $l_1 = l'$ or $l_2 = l'$. If $l_1 = l'$ then the choice-1 rule gives us $(l, s) \to (l', s)$; on the other hand if $l_2 = l'$ then the choice-2 rule provides the same conclusion. By H2, we have $a' = a$, whence $s \in \gamma(a')$. Putting $s' := s$ we have found, as required, $s' \in \gamma(a')$ such that $(l, s) \to (l', s')$.

Proof of A2: Let $\rho((l, a)) = P$ and $s \in \gamma(a)$. By Lemma 2.1 (seriality), there exists a configuration (l', s') such that $(l, s) \to (l', s')$. By S1 (soundness) there exists $(l', a') \in N$ such that $(l, a) \xrightarrow{abs} (l', a')$ and $s' \in \gamma(a')$. $\square$

It is in the A2 part of the preceding proof that seriality played its key part.

To decide whether $\mathbf{H}x$ for a particular node x (typically *start*), we simply apply the three rules for $\mathbf{H}$ over and over again, discovering more and more nodes n for which $\mathbf{H}n$, until either we have shown $\mathbf{H}x$, or no more applications of the rules are possible. This can be viewed as computing, in the underlying two-person game, as much of the attractor as is needed to determine whether it includes x.

Example 3.7 Using the augmented model in Figure 1 (left) we can prove that the program reaches the error state, i.e. we can falsify the safety property given by bad states $B := \{ERROR\} \times State$. We begin by using the h-all-bad rule to establish $\mathbf{H}(ERROR, [x : \text{zero}])$ and $\mathbf{H}(ERROR, [x : \text{pos}])$. This reflects the fact that if execution reaches these nodes then clearly the safety property has been broken. Now we consider the node $(1, [x : \text{pos}])$, which is a node of player P. We have shown $\mathbf{H}$ for each of its $\xrightarrow{abs}$-successors, so we can use the h-P-move rule to get $\mathbf{H}(1, [x : \text{pos}])$. This reflects the fact that although the abstraction used (here: sign analysis) cannot tell which way execution goes from the $\mathbf{H}(1, [x : \text{pos}])$, *it must go somewhere*, and wherever it goes, the safety property will be broken. One further application of h-P-move gives us $\mathbf{H}(\mathit{start}, \text{zero})$, whence, by Corollary 3.4, B is false, that is, execution reaches the error state. □

The model in Figure 1 (right) allows a similar proof that *ERROR* is reached, but shows that uncertainty over which path execution takes through the program can be dealt with, as well as uncertainty over the values of the program's variables.

4 In-depth example

Example 4.1 The program in Figure 2 generates an arbitrary natural number n and then computes $n - 1$ in y and the integer square root of n in x, that is, finds x such that $x^2 \leq n < (x + 1)^2$. The program is instrumented with a conditional which checks that the correct square root has been calculated, and transitions to the *ERROR* state if not. However, we have introduced a "mistake": the guard for the square root computation part is the negation of what it should be.

Figure 2 includes a default augmented model constructed by a simple sign analysis, and this is enough to prove the program faulty. Describing the proof in terms of the game, Player F (the Falsifier) is in charge of the choice of which natural number n is generated. If player F plays so as to force a positive n to be generated, this wins the game; player P still has some choice of moves, because the sign analysis could not determine whether y becomes zero or positive, but whichever of these is taken, execution ends up at the *ERROR* node. □

The preceding example only works because we distinguish F and P nodes, and use a different rule for them; if P controlled the choice of n, he could force $n = 0$ and then *ERROR* is not reached. The example also illustrates the style in which we intend to deal with nondeterminism, which is needed to ensure that the program is tested over all inputs. Instead of using atomic nondeterministic statements such as *havoc* (e.g. [12]), we propose to encode them using small control flow graphs consisting of choice edges and deterministic statements, and then analyse these with over-approximation in the same way as the rest of the program. The game structure will take care of making sure that all the possible choices are explored.

When we perform such verifications in HECTOR [5], to which we have added an implementation of this approach, we put each piece of generating code into a non-recursive procedure, which we call a *generator procedure*, which helps structure the instrumentation process. However here we lack the space to discuss how our approach extends to procedures.

We have also used generator procedures with linked data structures, for example to generate all possible linked lists, which we use with models we build from a shape analysis (see [5]). We intend also to experiment with modelling nondeterministic memory allocation in this way.

5 Incorporation of must information

The PhD thesis [7] proposes the use of *mixed transition systems* (MTSs) as models which can both verify and falsify properties of programs. This is achieved by using

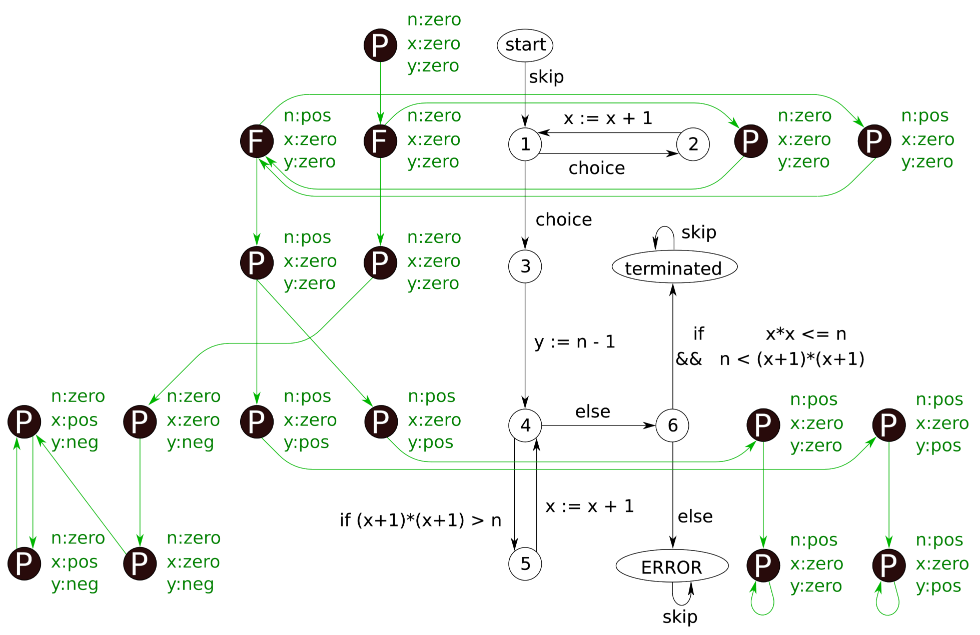

Fig. 2. An example of a faulty program, and a default augmented model which proves that the error state is reached, thanks to our differing treatment of F and P nodes. See Example 4.1 for details.

two transition relations: a “may” transition relation, which over-approximates and is like our $\xrightarrow{abs}$, and a “must” transition relation $\xrightarrow{must}$ which under-approximates.

In this section we show that our augmented models can neatly capture all the must information that is present in a MTS, while: keeping the same node structure, remaining sound for both verification and falsification, and still only needing one transition relation. This is achieved essentially by changing the player in charge of particular nodes, and works because of the way we have carefully isolated nondeterminism into the choice statement, which is used to build generator procedures. We begin by defining MTSs.

Definition 5.1 Mixed Transition Systems (MTSs). A Mixed Transition System $M^\dagger = (M, \xrightarrow{must})$ for a given program consists of an abstract model M along with another transition relation $\xrightarrow{must} \subseteq N \times N$ satisfying the following healthiness conditions:

M1 If $(l, a) \xrightarrow{must} (l', a')$ then $E(l)$ must be some edge with l' as a target.

M2 If $(l, a) \xrightarrow{must} (l', a')$ and $E(l)$ is a choice edge, then $a' = a$. □

As in [7], our definition relaxes the requirement in [11] that all must transitions are also may transitions, but it also adds M1 and M2 as natural constraints for our program abstractions.

Definition 5.2 Soundness of MTSs. An MTS $M^\dagger$ as above is said to be *sound* if M is sound and the following condition holds:

S2 Let $(l, a) \in N$ and (l, s) be such that $s \in \gamma(a)$. Let $(l, a) \xrightarrow{must} (l', a')$. Then there exists s' such that $(l, s) \to (l', s')$ and $s' \in \gamma(a')$. □

The follow theorem shows how MTSs can be used to falsify safety properties.

Theorem 5.3 Falsification with MTSs. *Let $M^\dagger = (N, \xrightarrow{abs}, \xrightarrow{must})$ be a sound MTS for a program P. Consider a safety property expressed by a set B of bad configurations. To falsify B it is sufficient to find a sequence $n_1, \dots, n_k \in N$ with*

(i) $n_1 \xrightarrow{must} n_2 \xrightarrow{must} \cdots \xrightarrow{must} n_k$

(ii) $n_1 = (start, a_1)$ *with* $s^{init} \in \gamma(a_1)$, *and*

(iii) $n_k = (l_k, a_k)$, *with* $\{l_k\} \times \gamma(a_k) \subseteq B$. □

Next, as promised, we show how to simply construct an augmented model which neatly captures all the must information from an MTS, while keeping the same node structure as the underlying over-approximating model, remaining sound for both verification and falsification, and still only needing one transition relation.

The construction is simple, differing from the default augmented model in that, at any abstract node which has a must transition leaving it, we put player F in charge, and replace the outgoing $\xrightarrow{abs}$ edges with the provided outgoing $\xrightarrow{must}$ edges. This is sound due to the partial determinism of the concrete semantics $\rightarrow$.

Definition 5.4 Given an MTS $M^\dagger = (M, \xrightarrow{must})$ as above, we define the *augmented model incorporating* $\xrightarrow{must}$ for M to be $M^+[\xrightarrow{must}] = (N, \xrightarrow{abs}^+, \rho)$ where (listing cases in order of priority):

$$(l, a) \xrightarrow{abs}^+ ('l, a') \qquad \Leftrightarrow \qquad (l', a') \in T(l, a)$$

$$T(l,a) := \begin{cases} \{(l',a') \mid (l,a) \xrightarrow{abs} (l',a')\} & \text{if } E(l) \text{ has the form choice} : l_1 : l_2 \\ \{(l',a') \mid (l,a) \xrightarrow{must} (l',a')\} & \text{if there exists } n' \in N \text{ such that } n \xrightarrow{must} n' \\ \{(l',a') \mid (l,a) \xrightarrow{abs} (l',a')\} & \text{otherwise} \end{cases}$$

$$\rho(l,a) := \begin{cases} F & \text{if } E(l) \text{ has the form choice} : l_1 : l_2 \\ F & \text{if there exists } n' \in N \text{ such that } n \xrightarrow{must} n' \\ P & \text{otherwise} \end{cases}$$

□

The following theorem shows that, after incorporating must information, the augmented model is still sound for both verification and falsification.

Theorem 5.5 *Let $M^\dagger = (M, \xrightarrow{must})$ be a sound MTS. Then $M^+[\xrightarrow{must}]$ is a sound augmented model.*

Proof: Sketch only due to space constraints. First prove S1, i.e. that $(N, \xrightarrow{abs}^+)$ actually forms a sound model. This is where Lemma 2.1 (partial determinism) comes into play: it is needed, along with S2 (soundness of the must relation) at the nodes where we have replaced $\xrightarrow{abs}$-edges with $\xrightarrow{must}$-edges. Then prove A1 and A2. The argument is similar to that for Theorem 3.6 (soundness of the plain M^+, that is, with nothing incorporated), except that S2 (soundness of the must relation) is needed to prove A1 at nodes where we have replaced $\xrightarrow{abs}$-edges with $\xrightarrow{must}$-edges.□

The following theorem confirms that, as promised, $M^+[\xrightarrow{must}]$ really does capture all the must information from the MTS $(M, \xrightarrow{must})$.

Theorem 5.6 *Let P be a program and $M^\dagger = (M, \xrightarrow{must})$ a sound MTS for P. Consider any safety property expressed by a set B of bad configurations. If B is falsified by $(M, \xrightarrow{must})$ (using Corollary 5.3) then B is also falsified by the augmented model $M^+[\xrightarrow{must}]$ (using Theorem 3.3).*

Proof: Sketch only due to space constraints. Consider the sequence $n_1, \ldots, n_k \in N$ demanded by the premises of Corollary 5.3. Proceed by induction on k. In the inductive step, apply the induction hypothesis to the suffix $n_2, \ldots, n_k$ to obtain $\mathbf{H}n_2$ and from this derive $\mathbf{H}n_1$. □

We end this section with an unresolved question. When incorporating must information, we have shown that we obtain all the falsification power *of the MTS from which the must edges come*; we also know that the augmented model remains sound for both verification and falsification. But it remains to be seen what happens to the verification and falsification power *of the default augmented model* when additional must edges are incorporated; we can contrive situations where this increases (as we would hope), decreases or remains unchanged, but do not have a feeling for what will happen in practice.

6 Conclusions and related work

In this paper, we used a two-player game to show that models which only over-approximate can nevertheless be used to falsify safety properties, that is, without using any under-approximation, feasibility checking or concrete counterexample search. To make this work, we focused on two properties of programs that are not accounted for in a conventional treatment of must transitions (e.g. [8]) namely seriality and partial determinism. Through Example 4.1 we demonstrated how and why our method works. Finally, we showed that if some must transitions *are* available, they can be incorporated into our approach very easily. We proved that by doing this, we obtain in a simple way all the falsification power of the must transition approach, and yet our models remain sound for both verification and falsification, retain the same node structure and still require only a single transition relation.

Related work

The present paper explores what *generalised model checking* [2], which effectively "case splits" on unknown propositions, means in the particular context of checking safety properties of programs. The existing works closest to ours, as far as we are aware, are [9] and [15], which also build models which can both verify and falsify properties.

In [9], which is specific to predicate abstraction domains, seriality is exploited but only for conditional statements (as in Figure 1 (right)), and not for ordinary statements (needed for Figure 1 (left)). For ordinary statements, [9] uses must transitions to weaker *tri-vector* states. The “must hyper-transitions” used in [15] also capture seriality, though this is not the motivation given in [15] for introducing them; rather, they are proposed as a way to make abstraction refinement monotonic. Both [9] and [15] require the use of two separate transition relations, whereas we need only one. Here we handle only safety properties expressed by giving a set of bad configurations, whereas [9,15] handle the much more expressive temporal logic CTL, and additionally address automatic abstraction refinement which we do not. We emphasise the expected role of generator procedures, rather than atomic statements such as *havoc*, in producing more falsifications. Our method subsumes the “choose-free-paths” technique from [14].

References

[1] Ball, T. and S. K. Rajamani, *Automatically validating temporal safety properties of interfaces*, in: *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, LNCS 2057 (2001), pp. 103–122.

[2] Bruns, G. and P. Godefroid, *Generalized model checking: Reasoning about partial state spaces*, Lecture Notes in Computer Science 1877 (2000), pp. 168+.

[3] Charlton, N., *Program verification with interacting analysis plugins*, Formal Aspects of Computing (2006), springer Verlag, DOI: 10.1007/s00165-007-0029-4.

[4] Charlton, N., *Verification of Java programs with interacting analysis plugins*, Electronic Notes in Theoretical Computer Science 145, Proceedings of the 5th International Workshop on Automated Verification of Critical Systems (AVoCS 2005) (2006), pp. 131–150.

[5] Charlton, N. and M. Huth, *HECTOR: software model checking with cooperating analysis plugins*, in: *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, LNCS 4590 (2007).

[6] Cousot, P. and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1977), pp. 238–252.

[7] Dams, D., “Abstract Interpretation and Partition Refinement for Model Checking,” Ph.D. thesis, Eindhoven University of Technology (1996).

[8] Godefroid, P., M. Huth and R. Jagadeesan, *Abstraction-based model checking using modal transition systems*, in: *CONCUR 2001 - concurrency theory: 12th international conference, Aalborg, Denmark, 20 - 25 August 2001*, Lecture Notes in Computer Science 2154, 2001, pp. 426–440.

[9] Gurfinkel, A. and M. Chechik, *Why waste a perfectly good abstraction?*, in: *TACAS*, 2006, pp. 212–226.

[10] Henzinger, T. A., R. Jhala, R. Majumdar and G. Sutre, *Lazy abstraction*, in: *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, ACM Press, 2002, pp. pp. 58–70.

[11] Larsen, K. G. and B. Thomsen, *A modal process logic*, in: *LICS*, 1988, pp. 203–210.

[12] Leino, K. R. M. and F. Logozzo, *Loop invariants on demand*, in: *APLAS*, 2005, pp. 119–134.

[13] Nielson, F., H. R. Nielson and C. Hankin, “Principles of Program Analysis,” Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[14] Pasareanu, C. S., M. B. Dwyer and W. Visser, *Finding feasible counter-examples when model checking abstracted Java programs*, in: *TACAS*, 2001, pp. 284–298.

[15] Shoham, S. and O. Grumberg, *Monotonic abstraction-refinement for CTL*, in: *TACAS*, 2004, pp. 546–560.

A Proofs omitted from main text for space reasons

Proof of Lemma 2.1: Consider an arbitrary configuration (l, s). Then $E(l)$ has three possible forms:

(i) $E(l) = \mathrm{f} : l_1$. For seriality, put $(l', s') := (l_1, f(s))$ and the ord rule gives $(l, s) \to (l', s')$. For partial determinism, simply note that no other rule is applicable.

(ii) $E(l) = \mathrm{if}(\Phi) : l_1 : l_2$. Either $s \in [\![\Phi]\!]$ or $s \notin [\![\Phi]\!]$. If $s \in [\![\Phi]\!]$ then for seriality, put $(l', s') := (l_1, s)$ and the if-true rule gives $(l, s) \to (l', s')$; for partial determinism note that no other rule is applicable. If $s \notin [\![\Phi]\!]$ then for seriality, put $(l', s') := (l_2, s)$ and the if-false rule gives $(l, s) \to (l', s')$; for partial determinism note that no other rule is applicable.

(iii) $E(l) = \mathrm{choice} : l_1 : l_2$. For seriality, put $(l', s') := (l_1, s)$ and the choice-1 rule gives $(l, s) \to (l', s')$. For partial determinism there is nothing to check. □

We prove Theorem 5.3 with the aid of the following lemma:

Lemma A.1 *Let $M^\dagger$ be a sound MTS. Let there exist a sequence $n_1, \ldots, n_k \in N$ (where each n_i is (l_i, a_i)) such that $n_1 \xrightarrow{must} n_2 \xrightarrow{must} \cdots \xrightarrow{must} n_k$. Let $s \in \gamma(a_1)$. Then there exist $s_1, \ldots, s_k \in State$ such that $s_1 = s$, $s_k \in \gamma(a_k)$ and $(l_1, s_1) \to \ldots \to (l_k, s_k)$.*

Proof: We proceed by induction on k. The base case when $k = 1$ is trivial. For the inductive case, $k > 1$, let there exist a sequence $n_1, \ldots, n_k \in N$ (where each n_i is (l_i, a_i)) such that $n_1 \xrightarrow{must} n_2 \xrightarrow{must} \cdots \xrightarrow{must} n_k$. Let $s \in \gamma(a_1)$.

Applying the induction hypothesis to the prefix $n_1, \ldots, n_{k-1}$, there exist $s_1, \ldots, s_{k-1}$ such that $s_1 = s$, $s_{k-1} \in \gamma(a_{k-1})$ and $(l_1, s_1) \to \ldots \to (l_{k-1}, s_{k-1})$.

Applying S2 to the transition $n_{k-1} \xrightarrow{must} n_k$, there exists (l_k, s') such that $(l_{k-1}, s_{k-1}) \to (l_k, s')$ and $s' \in \gamma(a_k)$. Putting $s_k := s'$ we are done. □

Proof of Theorem 5.3: Let $M^\dagger = (N, \xrightarrow{abs}, \xrightarrow{must})$ be a sound MTS for a program P. Let B be a set B of bad configurations. Suppose there exists a sequence $n_1, \ldots, n_k \in N$ with

(i) $n_1 \xrightarrow{must} n_2 \xrightarrow{must} \cdots \xrightarrow{must} n_k$

(ii) $n_1 = (start, a_1)$ with $s^{init} \in \gamma(a_1)$, and

(iii) $n_k = (l_k, a_k)$, with $\{l_k\} \times \gamma(a_k) \subseteq B$.

Due to i and ii, we can apply the previous lemma (Lemma A.1) to get a sequence of states $s^{init}, s_2, \ldots, s_k \in State$ such that $(start, s^{sign}) \to (l_2, s_2) \to \ldots \to (l_k, s_k)$ and $s_k \in \gamma(a_k)$. From iii we have $(l_k, s_k) \in B$, i.e. we've found an execution sequence starting at $(start, s^{init})$ and leading to the bad state (l_k, s_k). □

Proof of Theorem 5.5:

Proof of S1: First we must check that $(N, \xrightarrow{abs}^{+})$ is actually a sound model, i.e. that it satisfies S1. (This part of the proof depends on the determinism of ordinary program statements.)

Let $(l, a) \in N$ and (l, s) be such that $s \in \gamma(a)$. Let $(l, s) \rightarrow (l', s')$.

If $E(l)$ has the form $\boxed{\text{choice} : l_1 : l_2}$ or has no must transitions leaving it, then (l, s) has the same successors under $\xrightarrow{abs}^{+}$ as it does under $\xrightarrow{abs}$, and the conclusion follows from the S1 property of $(N, \xrightarrow{abs})$.

So suppose $E(l)$ doesn't have the form $\boxed{\text{choice} : l_1 : l_2}$, and there exists $n'' = (l'', a'') \in N$ such that $n \xrightarrow{must} n''$. By definition of $\xrightarrow{abs}^{+}$ we have $n \xrightarrow{abs}^{+} n''$. By S2, there exists s'' such that $(l, s) \rightarrow (l'', s'')$ and $s'' \in \gamma(a'')$.

By Lemma 2.1 (partial determinism), $l' = l''$ and $s' = s''$. Hence, putting $a' := a''$, we have found as required $(l', a') \in N$ such that $(l, a) \xrightarrow{abs}^{+} (l', a')$ and $s' \in \gamma(a')$.

Proof of A1: Let $\rho((l, a)) = F$ and $s \in \gamma(a)$. Let (l', a') be such that $(l, a) \xrightarrow{abs}^{+} (l', a')$. There are two situations in which we can have $\rho((l, a)) = F$.

The first situation is when $E(l)$ has the form $\boxed{\text{choice} : l_1 : l_2}$. By H1, either $l_1 = l'$ or $l_2 = l'$. If $l_1 = l'$ then the choice-1 rule gives us $(l, s) \rightarrow (l', s)$; on the other hand if $l_2 = l'$ then the choice-2 rule provides the same conclusion. By H2, we have $a' = a$, whence $s \in \gamma(a')$. Putting $s' := s$ we have found, as required, $s' \in \gamma(a')$ such that $(l, s) \rightarrow (l', s')$.

The second situation is when $E(l)$ doesn't have the form $\boxed{\text{choice} : l_1 : l_2}$, and there exists $n' = (l', a') \in N$ such that $n \xrightarrow{must} n'$. By definition of $\xrightarrow{abs}^{+}$ we have $(l, a) \xrightarrow{abs} (l', a')$ (because here we chose the must edges). By S2, there exists s' such that $(l, s) \rightarrow (l', s')$ and $s' \in \gamma(a')$ and we are done.

Proof of A2: Let $\rho((l, a)) = P$ and $s \in \gamma(a)$. By Lemma 2.1 (seriality), there exists a configuration (l', s') such that $(l, s) \rightarrow (l', s')$. By S1, there exists $(l', a') \in N$ such that $(l, a) \xrightarrow{abs} (l', a')$ and $s' \in \gamma(a')$. To finish, note that by definition of $\xrightarrow{abs}^{+}$ and the fact that $(l, a) \xrightarrow{abs} (l', a')$, we have $(l, a) \xrightarrow{abs}^{+} (l', a')$ (because here we have chosen the ordinary abstract edges). □

We prove Theorem 5.6 with the aid of the following lemma:

Lemma A.2 *Let $M^{\dagger} = (N, \xrightarrow{abs}, \xrightarrow{must})$ be a sound MTS. Let $B \subseteq Loc \times State$ and let there exist a sequence $n_1, \ldots, n_k \in N$ (where each n_i is (l_i, a_i)) such that $n_1 \xrightarrow{must} n_2 \xrightarrow{must} \cdots \xrightarrow{must} n_k$ and $\gamma(a_1)$ is nonempty. Let $\{l_k\} \times \gamma(a_k) \subseteq B$. Then $\mathbf{H}n_1$ with respect to $M^{+}[\xrightarrow{must}]$ (the augmented model incorporating $\xrightarrow{must}$).*

Proof: Write $M^{+}[\xrightarrow{must}]$ as $(N, \xrightarrow{abs}^{+}, \rho)$. We proceed by induction on k. For the base case, when $k = 1$, the condition $\{l_k\} \times \gamma(a_k) \subseteq B$ is exactly what is needed to invoke the h-all-bad rule to obtain $\mathbf{H}n_1$.

For the inductive case, $k > 1$, let $n_1, \ldots, n_k \in N$ (where each n_i is (l_i, a_i)) be such that $n_1 \xrightarrow{must} n_2 \xrightarrow{must} \cdots \xrightarrow{must} n_k$ and $\{l_k\} \times \gamma(a_k) \subseteq B$ and $\gamma(a_1)$ is nonempty.

$\gamma(a_1)$ is nonempty so there exists $s_1 \in \gamma(a_1)$. By S2, there exists some s_2 in $\gamma(a_2)$ such that $(l_1, s_1) \to (l_2, s_2)$. Therefore $\gamma(a_2)$ is also nonempty, and we can apply the induction hypothesis to the suffix $n_2, \ldots, n_k \in N$, obtaining $\mathbf{H}n_2$. Because there is a must transition leaving n_1 (i.e. the one to n_2) it follows from the definition of $M^+[\xrightarrow{must}]$ that $\rho(n) = F$. We can use the h-F-move rule to complete the proof if we can show $n_1 \xrightarrow{abs}^+ n_2$.

There are two cases to check.

In the first case, $E(l_1)$ has form $\;choice : l' : l''\;$. It follows from M2 that $a_1 = a_2$. From $(l_1, s_1) \to (l_2, s_2)$, using S1, there exists $(l_2, a') \in N$ such that $(l_1, a_1) \xrightarrow{abs} (l_2, a')$. By H2, $a' = a_1$, and we already know $a_1 = a_2$. Thus we have $(l_1, a_1) \xrightarrow{abs} (l_2, a_2)$, i.e. $n_1 \xrightarrow{abs} n_2$. Finally, in the definition of $\xrightarrow{abs}^+$ we choose the ordinary abstract edges at n_1, so we have $n_1 \xrightarrow{abs}^+ n_2$.

In the second case, $E(l_1)$ has some other form, and in the definition of $\xrightarrow{abs}^+$ we choose the must edges at n_1. Thus the required $n_1 \xrightarrow{abs}^+ n_2$ follows from $n_1 \xrightarrow{must} n_2$. □

Proof of Theorem 5.6: This follows easily from the previous lemma (Lemma A.2); just note that the premises of Theorem 5.3 demand that $s^{init} \in \gamma(a_1)$, and this is what assures the nonemptiness of $\gamma(a_1)$ needed to invoke the lemma. □