11-2012

More anti-chain based refinement checking

Ting WANG

Songzheng SONG

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

Jin Song DONG

See next page for additional authors

Citation

WANG, Ting; SONG, Songzheng; SUN, Jun; LIU, Yang; DONG, Jin Song; WANG, Xinyu; and LI, Shanping. More anti-chain based refinement checking. (2012). *Proceedings of the 14th International Conference on Formal Engineering Methods, , ICFEM 2012, Kyoto, Japan, November 12-16*. 364-380.
Available at: https://ink.library.smu.edu.sg/sis_research/5022

Author

Ting WANG, Songzheng SONG, Jun SUN, Yang LIU, Jin Song DONG, Xinyu WANG, and Shanping LI

More Anti-chain Based Refinement Checking*

Ting Wang[1], Songzheng Song[2], Jun Sun[3], Yang Liu[2], Jin Song Dong[2],
Xinyu Wang[1,**], and Shanping Li[1]

[1] College of Computer Science and Technology, Zhejiang University
{qdw,wangxinyu,shan}@zju.edu.cn
[2] National University of Singapore
{songsongzheng@,tslliuya@,dongjs@comp.}nus.edu.sg
[3] Singapore University of Technology and Design
sunjun@sutd.edu.sg

Abstract. Refinement checking plays an important role in system verification. It establishes properties of an implementation by showing a refinement relationship between the implementation and a specification. Recently, it has been shown that anti-chain based approaches increase the efficiency of trace refinement checking significantly. In this work, we study the problem of adopting anti-chain for stable failures refinement checking, failures-divergence refinement checking and probabilistic refine checking (i.e., a probabilistic implementation against a non-probabilistic specification). We show that the first two problems can be significantly improved, because the state space of the product model may be reduced dramatically. Though applying anti-chain for probabilistic refinement checking is more complicated, we manage to show improvements in some cases. We have integrated these techniques into the PAT model checking framework. Experiments are conducted to demonstrate the efficiency of our approach.

1 Introduction

Model checking has established itself as an effective technique for system verification. It works by exhaustively searching through the state space in order to show that an implementation model, in certain modeling language, satisfies a property. Properties are often specified using temporal logic formulae such as CTL or LTL, in other words, a language different from the modeling language. An alternative approach is called refinement checking. Different from temporal-logic based model checking, refinement checking shows a refinement relationship between two models in the same language, one modeling an implementation and one modeling a specification. If the specification satisfies certain property and the refinement relationship is strong enough to preserve the property, we imply that the property is satisfied by the implementation. A variety of refinement relationships have been defined, which preserve different classes of properties. For instance, safety can be verified by showing a trace refinement relationship. Combination of safety and liveness is verified by showing a stable failures

* This research is sponsored in part by NSFC Program (No.61103032) and 973 Program (No.2009CB320701) of China.

** Corresponding author.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 364–380, 2012.

refinement relationship if the system is divergence-free or otherwise by showing a failures-divergence refinement relationship. The readers are refer to [12] for a discussion on the expressiveness of different refinement.

Refinement checking has been traditionally used to verify CSP [11]. The success of the FDR refinement checker [1], which supports fully automatic checking of the abovementioned refinement relationships, evidences the usefulness of refinement checking. Recently, *Sun et al.* extended the idea of automated trace refinement checking to probabilistic systems [14], which we refer to as probabilistic refinement checking in this work[1]. The idea is that, given a probabilistic implementation model (which has the semantics of a Markov Decision Process) and a non-probabilistic specification model, probabilistic refinement checking calculates the probability of the implementation exhibiting traces of the specification model. This is useful as, for instance, if the specification model captures desired system behaviors, the result is the probability of the implementation behaving 'well'.

Due to the non-determinism in the specification, refinement checking often relies on the classic subset construction approach. The subset construction is used to build a deterministic finite-state automaton (DFA) from the specification, which is in general a non-deterministic finite-state automaton (NFA). Next, refinement checking works by computing the synchronous product of the implementation and transforming the problem into a reachability analysis problem in the product. In the worst case, the resultant DFA could have exponentially more states than the original NFA. As a result, refinement checking suffers from state space explosion. Recently, Wulf *et al.* proposed an approach (for solving the language universality problem and trace refinement checking) named anti-chain [16]. It has been shown that this approach outperforms the previous ones significantly. The key point of anti-chain based approaches is that the complete subset construction and computing the complete state space of the product are avoided. Given that the existing approaches for checking other refinement relationships are all based on the subset construction, it is only naturally to investigate whether anti-chain can be used for better performance as it did for trace refinement checking.

In this work, we study three kinds of refinement checking, in particular, stable failures refinement, failures-divergence refinement and probabilistic refinement checking. The problem is non-trivial as we need to formally prove that anti-chain works with stable failures semantics and failures-divergence semantics. Furthermore, it is complicated for probabilistic refinement checking as omitting parts of the product would affect the probability. We make the following technical contribution. Firstly, we show that anti-chain can be readily used to improve stable failures refinement and failures-divergence refinement. Secondly, we show that anti-chain can be used to improve probabilistic refinement checking in some particular cases, using an iterative probability calculation method. Lastly, we implement the technique in the PAT model checker [13] and show improvement over existing approaches (significant for stable failures refinement and failures-divergence refinement).

Related Works. This work is related to research on anti-chain based model checking. Wulf *et al.* proposed the anti-chain based approach for checking the language universality and trace refinement of NFA [16]. It has been shown that the anti-chain based

[1] Probabilistic refinement has been used by different researchers to mean different things.

approach may outperform the standard ones by several orders of magnitude. Their following works show that significant improvements can also be brought to the model checking problem of LTL by using anti-chain based algorithms [8, 17]. Later Abdulla *et al.* improved the approach through exploiting a simulation relation on the states of NFA [2]. Remotely related are anti-chain based methods for solving other problems, e.g., the LTL realizability and synthesis problem [7, 10] and the universality and language inclusion problem of tree automata [2, 6]. In our work, we focus on stable failures refinement, failures-divergence refinement and probabilistic refinement checking.

Organization. Section 2 reviews trace refinement and anti-chain based trace refinement checking. Section 3 presents algorithms for anti-chain based stable failures refinement checking and failures-divergence refinement checking. Section 4 shows that anti-chain can be used to improve probabilistic refinement checking (with a non-probabilistic specification model). Lastly, Section 5 concludes the paper.

2 Background

In this section, we review previous work on anti-chain based trace refinement checking.

2.1 Trace Refinement

Let Σ be a set of event names; τ denote an invisible event; and Σ_τ denote $\Sigma \cup \{\tau\}$.

Definition 1 (LTS). *A labeled transition system (LTS) is a tuple $\mathcal{L} = (S, init, Act, T)$ where S is a set of states; $init \in S$ is an initial state; $Act \subseteq \Sigma_\tau$ is a set of events and $T : S \times Act \times S$ is a labeled transition relation.*

For simplicity, $(s, e, s') \in T$ is sometimes written as $s \xrightarrow{e} s'$. An LTS is deterministic if and only if for all $s \in S$ and $s \in \Sigma_\tau$, if $s \xrightarrow{e} u$ and $s \xrightarrow{e} v$, then $u = v$. We write $enable(s)$ to denote the set $\{e \mid \exists\, s'.\ s \xrightarrow{e} s'\}$. We write $u \rightsquigarrow v$ if there exists a finite sequence of states $\langle s_0, s_1, \cdots, s_n \rangle$ such that $s_i \xrightarrow{\tau} s_{i+1}$ for all i and $u = s_0$ and $v = s_n$. We write $u \overset{e}{\rightsquigarrow} v$ if $u \rightsquigarrow u'$ and $u' \xrightarrow{e} v'$ and $v' \rightsquigarrow v$. A finite sequence of events $\langle e_0, e_1, \cdots, e_n \rangle$ is a trace of $\mathcal{L}$ if and only if there exists a sequence of state $\langle s_0, s_1, \cdots, s_n \rangle$ such that $s_i \overset{e_i}{\rightsquigarrow} s_{i+1}$ for all i and $s_0 = init$. The traces of $\mathcal{L}$ are denoted as $traces(\mathcal{L})$.

Definition 2 (LTS Synchronous Product). *Let $\mathcal{L}_i = (S_i, init_i, Act_i, T_i)$ where $i \in \{1, 2\}$ be two LTSs such that $\tau \notin Act_2$. The synchronous product of $\mathcal{L}_1$ and $\mathcal{L}_2$, written as $\mathcal{L}_1 \times \mathcal{L}_2$, is an LTS $\mathcal{L} = (S, init, Act, T)$ such that $S = S_1 \times S_2$; $init = (init_1, init_2)$; $Act = Act_1 \cup Act_2$; and T is the minimum labeled transition relation satisfying the following conditions.*

- *If $(s_1, \tau, s_1') \in T_1$, $((s_1, s_2), \tau, (s_1', s_2)) \in T$ for all $s_2 \in S_2$;*
- *If $(s_1, e, s_1') \in T_1$ and $(s_2, e, s_2') \in T_2$ and $e \notin \tau$, $((s_1, s_2), e, (s_1', s_2')) \in T$.*

Notice that all events except τ are to be synchronized by the two LTSs.

Definition 3 (Trace Refinement). *Let $\mathcal{L}_i$ where $i \in \{1, 2\}$ be two LTSs. $\mathcal{L}_1$ trace-refines $\mathcal{L}_2$ if and only if $traces(\mathcal{L}_1) \subseteq traces(\mathcal{L}_2)$.*

The standard approach for trace refinement is based on the subset construction. That is, the specification LTS_2 is transformed into trace-equivalent deterministic LTS without τ-transitions through the process of determinization. Let $\mathcal{L} = (S, init, Act, T)$ be an LTS. The determinized LTS of $\mathcal{L}$ is $det(\mathcal{L}) = (S', init', Act', T')$ where $S' \subseteq 2^S$ is a set of sets of states, $init' = \{s \mid init \rightsquigarrow s\}$; $Act' = Act \setminus \{\tau\}$ and T' is a transition relation satisfying the following condition: $(N, e, N') \in T'$ if and only if $N' = \{s' \mid \exists\, s : N.\ s \overset{e}{\rightsquigarrow} s'\}$. Notice that states which can be reached via the same trace are grouped together in $det(\mathcal{L})$.

Given an implementation $\mathcal{L}_1$ and a specification $\mathcal{L}_2$, the standard trace refinement checking is to construct (often on-the-fly) the product $\mathcal{L}_1 \times det(\mathcal{L}_2)$ and then try to construct a state of the product (s_1, s_2) (where s_1 is a state of $\mathcal{L}_1$ and s_2 is a set of states in $\mathcal{L}_2$) such that s_2 is an empty set. Such a 'co-witness' state is called a TR-witness state. In the worst case, this algorithm has a complexity exponential in the number of states of $\mathcal{L}_2$.

2.2 Trace Refinement Checking with Anti-chain

It has been shown that trace refinement checking based on anti-chain offers significantly better performance [16]. Given two LTSs $\mathcal{L}_1$ and $\mathcal{L}_2$, the anti-chain method explores a 'simulation' relation in $\mathcal{L}_1 \times det(\mathcal{L}_2)$. Given any two states (s_1, s_2) and (s_1', s_2') of $\mathcal{L}_1 \times det(\mathcal{L}_2)$, let $(s_1', s_2') \leq (s_1, s_2)$ denote $s_1 = s_1'$ and $s_2 \subseteq s_2'$.

Proposition 1. *If $(s_1', s_2') \leq (s_1, s_2)$ and $(s_1, s_2) \overset{e}{\rightarrow} (u, v)$, then there exists (u', v') such that $(s_1', s_2') \overset{e}{\rightarrow} (u', v')$ and $u' = u$ and $v \subseteq v'$.* □

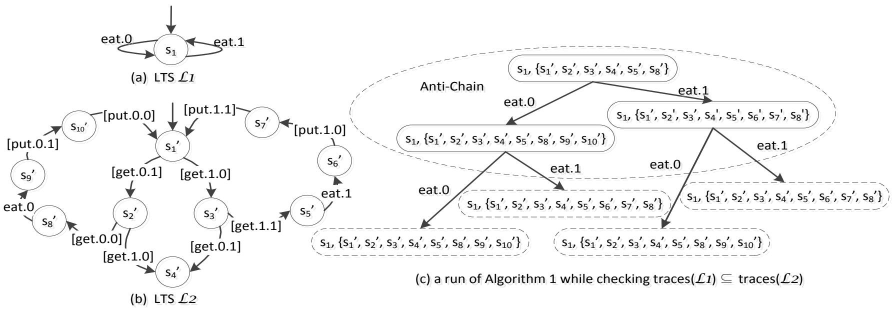
By the above proposition, it can be readily shown that a TR-witness state is reachable from (s_1', s_2') implies that a TR-witness state must be reachable from (s_1, s_2). As a result, if (s_1, s_2) has been explored, we can skip (s_1', s_2').

Formally, an anti-chain is a set A of sets such that $x \not\subseteq y$ and $y \not\subseteq x$ for all $x \in A$ and $y \in A$, i.e., any pair of sets in A are incomparable. An anti-chain supports two operations. One is to check whether it contains a subset of a given set. let x be the given set, we denote $x \Subset A$ if and only if there exists $y \in A$ such that $y \subseteq x$. The other is to add a given set x in A. $A \uplus x$ is defined as $\{y \mid y \in A \wedge x \not\subseteq y\} \cup \{x\}$, i.e., $A \uplus x$ contains x and all sets in A which is not a superset of x. Obviously, an empty set is an anti-chain by definition.

Algorithm 1 shows the anti-chain based algorithm. In an abuse of notation, we write $(s, X) \Subset A$ to denote that the set $(\{s\} \cup X) \Subset A$; and $A \uplus (s, X)$ to denote $A \uplus (\{s\} \cup X)$. The algorithm works as follows. After initialization, the algorithm pops one state $(impl, spec)$ from $working$ and adds it to the set $antichain$, and then generates all successors of the state and adds them to $working$ unless $(impl', spec') \Subset antichain$ is true, till the stack $working$ is empty or a TR-witness state is found. We remark that $antichain$ keeps to be an anti-chain during this algorithm, because line 5 and line 13 guarantee there are no subsets or supersets of the new added state in the updated $antichain$. Soundness of the algorithm can be referred to in [2] [16].

Algorithm 1 Trace Refinement Checking Algorithm with Anti-chain

```
let $working$ be a stack containing a pair $(init_1, \{s \mid init_2 \rightsquigarrow s\})$;
let $antichain := \varnothing$;
while $working \neq \varnothing$ do
    pop $(impl, spec)$ from $working$;
    $antichain := antichain \Cup (impl, spec)$;
    for all $(impl, e, impl') \in T_1$ do
        if $e = \tau$ then
            $spec' := spec$;
        else
            $spec' := \{s' \mid \exists\, s \in spec.\ s \overset{e}{\rightsquigarrow} s'\}$;
        if $spec' = \varnothing$ then
            return false;
        if $(impl', spec') \Subset antichain$ is not true then
            push $(impl', spec')$ into $working$;
return true;
```


eat.0
eat.1
s1
s1'
s2'
s3'
s4'
s5'
s6'
s7'
s8'
s9'
s10'
[put.0.0]
[put.1.1]
[put.1.0]
[put.0.1]
[get.0.1]
[get.1.0]
[get.0.0]
[get.1.1]
eat.0
eat.1
Anti-Chain
s1, {s1', s2', s3', s4', s5', s8'}
s1, {s1', s2', s3', s4', s5', s8', s9', s10'}
s1, {s1', s2', s3', s4', s5', s6', s7', s8'}
s1, {s1', s2', s3', s4', s5', s6', s7', s8'}
s1, {s1', s2', s3', s4', s5', s6', s7', s8'}
s1, {s1', s2', s3', s4', s5', s8', s9', s10'}
s1, {s1', s2', s3', s4', s5', s8', s9', s10'}

(a) LTS $\mathcal{L}1$

(b) LTS $\mathcal{L}2$

(c) a run of Algorithm 1 while checking traces($\mathcal{L}1$) ⊆ traces($\mathcal{L}2$)

Fig. 1. Trace Refinement Checking Algorithm with Anti-Chain

Theorem 1. *[16] Algorithm 1 returns true if and only if $traces(\mathcal{L}_1) \subseteq traces(\mathcal{L}_2)$.* □

Example 1. Figure 1 shows a simple example of dining philosopher [11] to demonstrate how Algorithm 1 works. The problem is summarized as N philosophers sitting around a round table with a single fork between each pair, and each philosopher requiring both neighboring forks to eat. The LTS $\mathcal{L}_2$ in Figure 1 (b) shows the complete state graph of the system with two philosophers. The invisible events is denoted as $[event]$, i.e., $[get.0.1]$ means that the hidden event is philosopher 0 getting the right fork which is represented as 1. Note that checking $traces(\mathcal{L}_1) \subseteq traces(\mathcal{L}_2)$ is to confirm whether every philosopher can eat. From Figure 1 (c) we can see that the search does not continue from the state $(s_1, \{s_1', s_2', s_3', s_4', s_5', s_8', s_9', s_{10}'\})$ because $\{s_1', s_2', s_3', s_4', s_5', s_8'\} \subseteq \{s_1', s_2', s_3', s_4', s_5', s_8', s_9', s_{10}'\}$. In this case, Algorithm 1 generates 3 states which are labeled with $Anti{-}Chain$, while the classical algorithm based on subset construction generates 7 states. □

Algorithm 2 Stable Failures Refinement Checking Algorithm with Anti-chain

```
let $working$ be a stack containing a pair $(init_1, \{s \mid init_2 \rightsquigarrow s\})$;
let $antichain := \varnothing$;
while $working \neq \varnothing$ do
    pop $(impl, spec)$ from $working$;
    $antichain := antichain \Cup (impl, spec)$;
    if $refusals(impl) \not\subseteq refusals(spec)$ then
        return false;
    for all $(impl, e, impl') \in T_1$ do
        if $e = \tau$ then
            $spec' := spec$;
        else
            $spec' := \{s' \mid \exists\, s \in spec.\ s \overset{e}{\rightsquigarrow} s'\}$;
        if $spec' = \varnothing$ then
            return false;
        if $(impl', spec') \Subset antichain$ is not true then
            push $(impl', spec')$ into $working$;
return true;
```

3 Failures/Divergence Refinement Checking with Anti-chain

In this section, we demonstrate that anti-chain can be used to improve stable failures refinement checking and failures-divergence refinement checking.

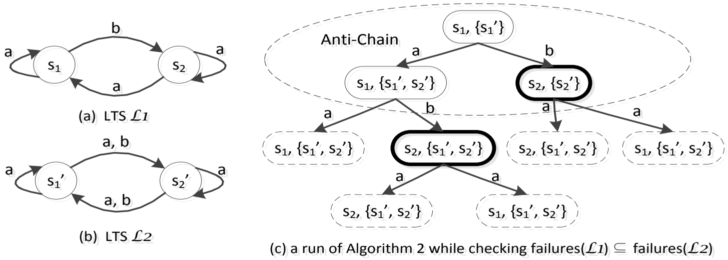
3.1 Stable Failures Refinement Checking

Let $\mathcal{L} = (S, init, Act, T)$ be an LTS. Given a state $s \in S$, s is stable if $\tau \not\in enable(s)$. Given a stable state s, the refusals of s, written as $refusals(s)$, is defined as $\{X \mid \exists\, s'.\ s \rightsquigarrow s' \wedge \tau \not\in enable(s') \wedge X \subseteq \Sigma \setminus enable(s')\}$. The failures of $\mathcal{L}$, written as $failures(\mathcal{L})$, is defined as $\{(tr, X) : \Sigma^* \times 2^{\Sigma} \mid \exists\, s.\ init \overset{tr}{\rightsquigarrow} s \wedge X \in refusals(s)\}$ where $init \overset{tr}{\rightsquigarrow} s$ denotes that there exists a run $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1}\rangle$ such that $s_0 = init$ and $s_{n+1} = s$ and $tr = \langle e_0, e_1, \cdots, e_n\rangle$.

Definition 4 (Stable Failures Refinement). *Let $\mathcal{L}_i$ where $i \in \{1, 2\}$ be two LTSs. $\mathcal{L}_1$ refines $\mathcal{L}_2$ in stable failures semantics if and only if $failures(\mathcal{L}_1) \subseteq failures(\mathcal{L}_2)$.*

The existing stable failures refinement checking algorithm [1] works by searching for a state (x, y) of $\mathcal{L}_1 \times det(\mathcal{L}_2)$ such that $y = \varnothing$ or $refusals(x) \not\subseteq refusals(y)$. Such a state is called a SFR-witness state. In the following, we extend Algorithm 1 for stable failures refinement checking. Given a set states x, we write $refusals(x)$ to denote $\{r \mid \exists\, s \in x.\ r \in refusals(s)\}$. The algorithm is shown in Algorithm 2.

Lemma 1. *For every state (s_1, s_2) of $\mathcal{L}_1 \times det(\mathcal{L}_2)$, for all (s_1, s_2') in the product, if $s_2' \subseteq s_2$, then a SFR-witness state is reachable from (s_1, s_2) implies a SFR-witness state is reachable from (s_1, s_2').*



(a) LTS $\mathcal{L}1$

(b) LTS $\mathcal{L}2$

(c) a run of Algorithm 2 while checking failures($\mathcal{L}1$) ⊆ failures($\mathcal{L}2$)

Fig. 2. Stable Failures Refinement Checking Algorithm with Anti-Chain

Proof: By induction. The base case is that (s_1, s_2) is a SFR-witness state, then $s_2 = \varnothing$ or $\mathit{refusals}(s_1) \nsubseteq \mathit{refusals}(s_2)$. Because $s_2' \subseteq s_2$ by assumption, $\mathit{refusals}(s_2') \subseteq \mathit{refusals}(s_2)$. Then we have $s_2' = \varnothing$, or $\mathit{refusals}(s_1) \nsubseteq \mathit{refusals}(s_2')$. Thus, (s_1, s_2') is a SFR-witness state. Next, we prove the induction step. Assume that (s_1, s_2) satisfies the condition, i.e., for any state (s_1, s_2') such that $s_2' \subseteq s_2$, a SFR-witness state is reachable from (s_1, s_2) implies a SFR-witness state is reachable from (s_1, s_2'). Let (x, y) be a state of the product such that $(x, y) \overset{e}{\rightarrow} (s_1, s_2)$. For all (x, y') such that $y' \subseteq y$, we can get $(x, y') \overset{e}{\rightarrow} (s_1, s_2')$ such that $s_2' \subseteq s_2$. Therefore, the induction step holds by induction hypothesis. Thus, the lemma is true. □

Theorem 2. *Algorithm 2 returns true if and only if* $\mathit{failures}(\mathcal{L}_1) \subseteq \mathit{failures}(\mathcal{L}_2)$.

Proof For a state S of $\mathcal{L}_1 \times \mathit{det}(\mathcal{L}_2)$, define $\mathit{Dist}(S) \in N \cup \{\infty\}$ as the length of the shortest SFR-witness trace from S (if a SFR-witness state is not reachable from S, $\mathit{Dist}(S) = \infty$). For a set of states States, if $\mathit{States} = \varnothing$, $\mathit{Dist}(\mathit{States}) = \infty$, otherwise, $\mathit{Dist}(\mathit{States}) = \mathit{min}_{S \in \mathit{States}}\, \mathit{Dist}(S)$. The predicate $\mathit{SFR}(\mathit{States})$ is true if and only if all the states in States are not SFR-witness states. Then the correctness of Algorithm 2 can be proved using the two invariants below. The invariants can be proved in a very similar way to [2].

1. $\neg \mathit{SFR}(\mathit{antichain} \cup \mathit{working}) \Rightarrow \neg \mathit{SFR}(\{(i, \{s \mid \mathit{init}_2 \rightsquigarrow s\}) \mid i \in \mathit{init}_1\})$.
2. $\neg \mathit{SFR}(\{(i, \{s \mid \mathit{init}_2 \rightsquigarrow s\}) \mid i \in \mathit{init}_1\}) \Rightarrow \mathit{Dist}(\mathit{antichain}) > \mathit{Dist}(\mathit{working})$.

Because the number of state is finite and all states are only visited once, Algorithm 2 eventually terminates. Algorithm 2 returns false only if the state spec' is an empty set on line 13, or $(\mathit{impl}, \mathit{spec})$ satisfies the condition $\mathit{refusals}(\mathit{impl}) \nsubseteq \mathit{refusals}(\mathit{spec})$ on line 6. The former case has been proved in Algorithm 1. In the latter case, $(\mathit{impl}, \mathit{spec})$ is a SFR-witness state, and hence $\mathit{SFR}(\mathit{antichain} \cup \mathit{working})$ is false. By invariant 1, $\mathcal{L}_1$ cannot refine $\mathcal{L}_2$ in stable failures semantics. Algorithm 2 returns true only when $\mathit{working}$ is empty, which implies that $\mathit{Dist}(\mathit{antichain}) > \mathit{Dist}(\mathit{working})$ is not true. By invariant 2, $\mathcal{L}_1$ refines $\mathcal{L}_2$ in stable failures semantics. □

Example 2. If Algorithm 2 is applied to the example presented in Figure 1, the reduction remains the same as for trace refinement checking (from 7 states to 3 states). We show another example with some counterexamples, as shown in Figure 2. Notice that the refusal set of s_2 is $\{b\}$, and s_1, s_1', s_2' do not refuse any event. Then the two states with bold circles are SFR-witness states. Since $\{s_1'\} \subseteq \{s_1', s_2'\}$, the search does not continue from the state $(s_1, \{s_1', s_2'\})$. We can see that a SFR-witness state which is reachable from $(s_1, \{s_1', s_2'\})$ is also reachable from $(s_1, \{s_1'\})$ after the pruning of states. In this case, Algorithm 2 generates 3 states which are labelled with $Anti{-}Chain$, while the classical algorithm may generate 4 states before finding a SFR-witness state. Moreover, Algorithm 2 may find a shorter witness trace than the classical algorithm. □

3.2 Failures-Divergence Refinement Checking

In the following, we show how to adopt anti-chain for failures-divergence refinement checking. Let $\mathcal{L} = (S, init, Act, T)$ be an LTS. Given a state s, s diverges if and only if s can performance an infinite number of τ-transitions. A trace tr is divergent, written as $div(tr)$, if and only if there exists a prefix pre of tr or tr itself such that $init \overset{pre}{\rightsquigarrow} s$ and s diverges. We write $divergences(\mathcal{L})$ to be $\{tr \mid div(tr)\}$.

Definition 5 (Failures-Divergence Refinement). *Let $\mathcal{L}_i$ where $i \in \{1, 2\}$ be two LTSs. $\mathcal{L}_1$ refines $\mathcal{L}_2$ in failures-divergence semantics if and only if $divergences(\mathcal{L}_1) \subseteq divergences(\mathcal{L}_2)$ and $failures(\mathcal{L}_1) \subseteq failures(\mathcal{L}_2)$.*

In the following, we extend Algorithm 2 for failures-divergence refinement checking. The algorithm is shown in Algorithm 3. Given a set of state x, we say that x diverges if there exists $s \in x$ such that s diverges. Like in the existing failures-divergence refinement checking algorithm [1], the idea is to search for a FDR-witness state (x, y) of $\mathcal{L}_1 \times det(\mathcal{L}_2)$ such that $y = \varnothing$ or $refusals(x) \not\subseteq refusals(y)$ or x diverges but not y.

Lemma 2. *For every state (s_1, s_2) of $\mathcal{L}_1 \times det(\mathcal{L}_2)$, for all (s_1, s_2') in the product, if $s_2' \subseteq s_2$, then a FDR-witness state is reachable from (s_1, s_2) implies a FDR-witness state is reachable from (s_1, s_2').*

Proof: By induction. The base case is that (s_1, s_2) is a FDR-witness state, then $s_2 = \varnothing$ or $refusals(s_1) \not\subseteq refusals(s_2)$ or s_1 diverges and s_2 does not. Because $s_2' \subseteq s_2$ by assumption, $refusals(s_2') \subseteq refusals(s_2)$ and if s_2' diverges, so does s_2. Thus, (s_1, s_2') is a SFR-witness state since $s_2 = \varnothing$ implies $s_2' = \varnothing$; $refusals(s_1) \not\subseteq refusals(s_2)$ implies $refusals(s_1) \not\subseteq refusals(s_2')$; and s_2 not divergent implies that s_2' does not diverge either. Next, we prove the induction step. Assume that (s_1, s_2) satisfies the condition. Let (x, y) be a state of the product such that $(x, y) \overset{e}{\rightarrow} (s_1, s_2)$. For all (x, y') such that $y' \subseteq y$, we can get $(x, y') \overset{e}{\rightarrow} (s_1, s_2')$ such that $s_2' \subseteq s_2$. Therefore, the induction step holds by induction hypothesis. Thus, the lemma is true. □

Theorem 3. *Algorithm 3 returns true if and only if $divergences(\mathcal{L}_1) \subseteq divergences(\mathcal{L}_2)$ and $failures(\mathcal{L}_1) \subseteq failures(\mathcal{L}_2)$.*

Proof Define $Dist(S) \in N \cup \{\infty\}$ as the length of the shortest FDR-witness trace from a state S of $\mathcal{L}_1 \times det(\mathcal{L}_2)$(if a FDR-witness state is not reachable from S, $Dist(S) = \infty$). Given a set of states $States$, if $States = \varnothing$, $Dist(States) = \infty$, otherwise, $Dist(States) = min_{S \in States}\, Dist(S)$. The predicate $FDR(States)$ is true if and only if all the states in $States$ are not FDR-witness states. The correctness of Algorithm 3 can be proved similarly as for Algorithm 2, using the following two invariants.

Algorithm 3 Failures-Divergence Refinement Checking Algorithm with Anti-chain

```
let $working$ be a stack containing a pair $(init_1, \{s \mid init_2 \rightsquigarrow s\})$;
let $antichain := \varnothing$;
while $working \neq \varnothing$ do
    pop $(impl, spec)$ from $working$;
    $antichain := antichain \Cup (impl, spec)$;
    if $impl$ diverges then
        if $spec$ does not diverge then
            return false;
    else
        if $refusals(impl) \not\subseteq refusals(spec)$ then
            return false;
        for all $(impl, e, impl') \in T_1$ do
            if $e = \tau$ then
                $spec' := spec$;
            else
                $spec' := \{s' \mid \exists\, s \in spec.\ s \overset{e}{\rightsquigarrow} s'\}$;
            if $spec' = \varnothing$ then
                return false;
            if $(impl', spec') \Subset antichain$ is not true then
                push $(impl', spec')$ into $working$;
return true;
```

1. $\neg FDR(antichain \cup working) \Rightarrow \neg FDR(\{(i, \{s \mid init_2 \rightsquigarrow s\}) \mid i \in init_1\})$.
2. $\neg FDR(\{(i, \{s \mid init_2 \rightsquigarrow s\}) \mid i \in init_1\}) \Rightarrow Dist(antichain) > Dist(working)$.

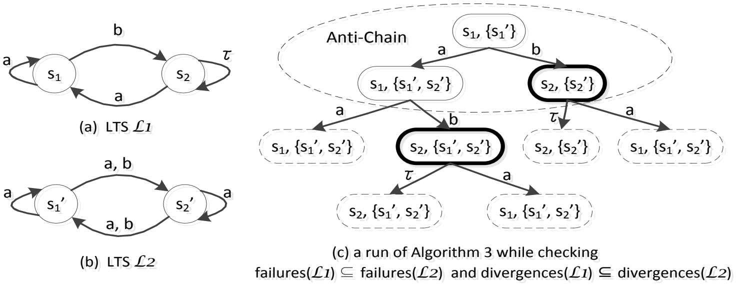
□

Example 3. We use the example shown in Figure 3 to demonstrate how algorithm 3 works. The state s_2 in LTS $\mathcal{L}_1$ has a self-loop labeled with τ. The two states with bold circles are FDR-witness states now. Since $\{s'_1\} \subseteq \{s'_1, s'_2\}$, the search does not continue from the state $(s_1, \{s'_1, s'_2\})$. We can see that a FDR-witness state which is reachable from $(s_1, \{s'_1, s'_2\})$ is also reachable from $(s_1, \{s'_1\})$ after pruning the states. □

3.3 Implementation and Evaluation

The proposed algorithms have been adopted in the Process Analysis Toolkit (PAT) [13]. PAT is designed for systematic validation of distributed/concurrent systems using state-of-the-art model checking techniques. In the following, we evaluate the performance of the algorithms using a range of real-life parameterized systems. All the systems are embedded in the PAT package and available online. The data is obtained with Intel(R) Core(TM) i7-2640M CPU at 2.80GHz and 8GB RAM.

The pairs of LTSs (one as implementation and one as specification) are generated from different systems or same systems with different parameters. The systems include a multi-valued register simulation system with one reader or multiple readers [4], an implementation of concurrent stack with or without linearization point [15], a mailbox system [3], a system of scalable nonzero indicator [9] and the dining philosopher



(a) LTS $\mathcal{L}1$

(b) LTS $\mathcal{L}2$

(c) a run of Algorithm 3 while checking failures($\mathcal{L}1$) $\subseteq$ failures($\mathcal{L}2$) and divergences($\mathcal{L}1$) $\subseteq$ divergences($\mathcal{L}2$)

Fig. 3. Failures-Divergence Refinement Checking Algorithm with Anti-Chain

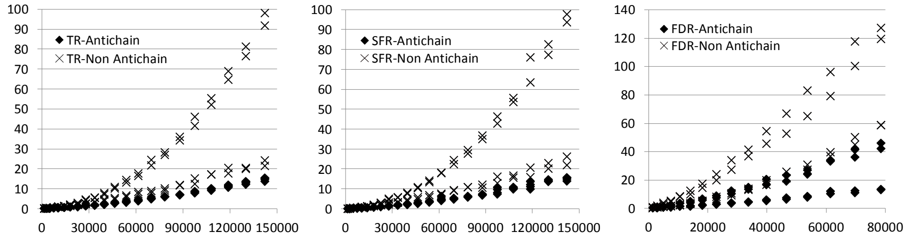


Fig. 4. Refinement Checking Results of Concurrent Stack Implementation

problem [11]. In total about 300 pairs of LTS were generated to compare the anti-chain algorithms and the classical ones for all three kinds of refinement checking.

Figure 4 shows the statistics of a typical example, i.e., the concurrent stack, from which we can see significant performance improvement. In the figure, the horizontal axis is the sum of the sizes of the two LTSs for refinement checking, and the vertical axis is the execution time (in seconds) of the corresponding algorithm. Each point shows the checking time for a pair of LTSs. We can see that for all three kinds of refinement checking, anti-chain based algorithms offer significantly better performance. The complete experimental results, with the refinement checking assertions always being valid, are summarized in Table 1. Notice that if the pair of LTSs are equivalent in terms of traces or failures or failures-divergence, we can perform the refinement checking in both directions. This is shown in the table using two columns $\subseteq$ and $\supseteq$. A few cases are marked as '—' as the result is false, which are discussed later. Furthermore, '*unknown*' means either out of memory or running for more than 30 minutes. It can be observed that the speedup differs for different systems. In most cases, the anti-chain approach has a much better performance than the classical one, e.g., in the concurrent stack linearization point implementation, it is 30.28 times faster for stable failures refinement checking and 12.16 times faster for failures-divergence refinement checking. Moreover, the larger the system is, the larger the speedup is. In some cases, anti-chain can not reduce the number of states at all simply because the specifications are deterministic. In some cases (e.g., SNZI), although anti-chain reduces the number of states, the benefit is not significant enough to overcome the computational overhead of the anti-chain operations defined in section 2.2.

Table 1. Testing on Refinement Checking Assertions which are valid

System	Size	Trace (Speedup)		Stable Failures (Speedup)		Failures-Divergence (Speedup)	
		$\subseteq$	$\supseteq$	$\subseteq$	$\supseteq$	$\subseteq$	$\supseteq$
Multi-valued Register	0-10000	2.21	1.42	2.32	1.63	3.48	1.46
Simulation with	10000-100000	4.54	2.14	4.45	2.09	6.61	1.73
1 Reader and 1 Writer	100000-700000	6.74	2.88	6.64	2.92	unknown	2.59
Multi-valued Register	0-10000	2.17	1.49	1.99	1.45	3.33	1.43
Simulation with	10000-100000	3.32	2.05	3.32	2.11	3.55	1.70
Multiple Readers	100000-700000	6.45	2.68	6.14	2.72	unknown	3.16
Concurrent	0-10000	1.48	1.85	1.63	1.72	2.26	1.60
Stack	10000-100000	1.70	3.72	1.71	3.71	2.71	3.22
Implementation	100000-200000	1.62	5.90	1.56	6.44	2.85	4.96
Concurrent Stack	0-10000	0.75	5.33	—	5.99	—	2.70
Linearization Point	10000-30000	0.84	13.94	—	14.11	—	5.00
Implementation	30000-60000	0.91	30.37	—	30.28	—	12.16
Mailbox	0-700000	1.13	1.54	1.11	1.61	1.39	1.01
SNZI	0-50000	0.89	2.45	0.93	2.42	1.03	1.14
Dining Philosopher	0-50000	0.99	1.14	—	1.02	—	1.17

Table 2. Testing on Refinement Checking Assertions which are invalid

System	Size	Trace With AC(s)	Trace W/o AC(s)	Stable Failures With AC(s)	Stable Failures W/o AC(s)	Failures-Divergence With AC(s)	Failures-Divergence W/o AC(s)
Multi-valued	3175	0.10	0.42	0.09	0.43	0.44	2.88
Register	24655	1.52	12.12	1.71	12.02	9.78	146.95
Simulation with	117288	3.92	136.37	3.57	138.50	48.52	985.24
Multiple Readers	194455	22.11	294.71	21.51	299.21	180.91	unknown

In presence of counterexamples, anti-chain based algorithms may find the counterexample more quickly. The verification results with the register example (with multiple readers), shown in Table 2 (‘Anti-Chain’ and ‘Without’ are denoted as AC and W/o for short), evidences that anti-chain finds the counterexample more quickly in all three kinds of refinement checking. Nonetheless, we remark that because the algorithms are on-the-fly, whether the counterexample is found earlier depends on the searching order and sometimes anti-chain based algorithms may be slower if a ‘wrong’ order is taken.

4 Probabilistic Refinement Checking with Anti-chain

In this section, we show that anti-chain can be used to improve a particular kind of probabilistic refinement checking, i.e., the implementation is given as an MDP and the specification is given as an NFA.

4.1 MDP and Probabilistic Model Checking

Given a set of states S, a distribution is a function $\mu : S \rightarrow [0,1]$ such that $\varSigma_{s \in S}\, \mu(s) = 1$. Let $\mathit{Distr}(S)$ be the set of all distributions over S. A Markov Chain is a tuple $\mathcal{M} = (S, \mathit{init}, \mathit{Act}, \mathit{Pr})$ where S is a countable set of states; $\mathit{init} \in S$ is an initial state; Act is a set of events; and $\mathit{Pr} : S \times \mathit{Act} \times S \rightarrow [0,1]$ is a labeled transition probability function such that for all state $s \in S$, $\exists\, e \in \mathit{Act}, \varSigma_{s' \in S} \mathit{Pr}(s, e, s') = 1$, and $\forall\, e' \neq$

$e, \Sigma_{s' \in S} Pr(s, e, s') = 0$. Notice that Markov Chains are deterministic as there is only one event (and one distribution) at each state.

A sequence of alternating states and events $\pi = \langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1} \rangle$ is a path of $\mathcal{M}$ if $Pr(s_i, e_i, s_{i+1}) > 0$ for all i. The probability of executing π from s_0, written as $Pr(\mathcal{M}, \pi)$, is $Pr(s_0, e_0, s_1) \times Pr(s_1, e_1, s_2) \times \cdots \times Pr(s_n, e_n, s_{n+1})$. It is often also interesting to find out the probability of reaching a certain set of states (e.g., what is the probability of reaching the state of system failure?). Given a set of target states G, the probability of reaching any state in G from a starting state s_0, written as $Pr(\mathcal{M}, s_0, G)$, is the accumulated probability of all paths from s_0 to any state in G, which can be calculated systematically [5]. Given a path π, we define $trace(\pi)$ to be the sequence of visible events in π. We write $Pr(\mathcal{M}, s_0, tr)$ to denote the probability of exhibiting a trace tr from state s_0, which is the accumulated probability of all paths π from s_0 such that $trace(\pi) = tr$. Given a set of traces Tr, the probability of $\mathcal{M}$ exhibiting any trace in Tr from state s_0 is the accumulated probability $\Sigma_{tr \in Tr} Pr(\mathcal{M}, s_0, tr)$.

Different from Markov Chains, an MDP can express both probabilistic choices and non-determinism. An MDP is a tuple $\mathcal{D} = (S, init, Act, Pr)$ where S is a set of system states; $init \in S$ is the initial system configuration[2]; Act is a set of actions; $Pr : (S \times Act) \rightarrow Distr(S)$ is a transition probability function such that for all states $s \in S$ and $a \in Act$: $\Sigma_{s' \in S} Pr(s, a, s') \in \{0, 1\}$. Notice that there could be multiple events at any state. A transition of the system is written as $s \overset{e}{\rightarrow} \mu$ where μ is a distribution. A path of $\mathcal{M}$ is a sequence of alternating states, events and distributions $\pi = \langle s_0, e_0, \mu_0, s_1, e_1, \mu_1, \cdots \rangle$ such that $s_0 = init$ and $s_i \overset{e_i}{\rightarrow} \mu_i$ and $\mu_i(s_{i+1}) > 0$ for all i. Given a path π, we define $trace(\pi)$ to be the sequence of visible events in π.

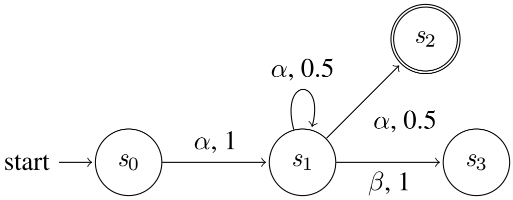
Intuitively speaking, given a system configuration, firstly an event and a distribution is selected non-deterministically by the *scheduler*, and then one of successor states is reached according to the probability distribution. A scheduler is a function deciding which event and distribution to choose based on the execution history. With a scheduler δ, we effectively obtain a Markov Chain from $\mathcal{D}$, written as $\mathcal{D}_\delta$. Note that with different scheduling, the probability of reaching a state or exhibiting a trace may be different. The measurement of interest is thus the maximum and minimum probability. Given a set of target states G and an MDP $\mathcal{D}$, the maximum probability of reaching any state in G from state s_0 is defined as $\mathcal{P}_{max}(\mathcal{D}, s_0, G) = \sup_\delta Pr(\mathcal{D}_\delta, s_0, G)$. Note that the supremum ranges over all, potentially infinitely many, schedulers. Accordingly, the minimum is written as $\mathcal{P}_{min}(\mathcal{D}, s_0, G)$. Similarly, we define the maximum probability of exhibiting a trace in a set Tr by $\mathcal{D}$ from s_0.

$$\mathcal{P}_{max}(\mathcal{D}, s_0, Tr) = \sup_\delta (\Sigma_{tr \in Tr} Pr(\mathcal{D}_\delta, s_0, tr))$$

Accordingly, the minimum is written as $\mathcal{P}_{min}(\mathcal{D}_\delta, s_0, Tr)$.

Example 4. The following shows a simple example MDP.

[2] This is a simplified definition. In general, there can be an initial distribution.



where s_0 is the initial state and s_2 is a target state. For simplicity, we omit the self-loop of s_2 and s_3. s_1 has two distributions, following two actions α and β. If s_1 nondeterministically chooses α, then it has equal probability to transfer to s_2 or stay in s_1; and if β is chosen, it will transfer to s_3 with probability 1. □

Definition 6 (Refinement Probability). *Let $\mathcal{D} = (S, init, Act, Pr)$ be an MDP; $\mathcal{L}$ be an LTS. The maximum probability of $\mathcal{D}$ trace-refining $\mathcal{L}$ is $\mathcal{P}_{max}(\mathcal{D}, init, traces(\mathcal{L}))$. The minimum is $\mathcal{P}_{min}(\mathcal{D}, init, traces(\mathcal{L}))$.*

Intuitively, the probability of $\mathcal{D}$ refines $\mathcal{L}$ is the probability of $\mathcal{D}$ exhibiting a trace of $\mathcal{L}$. As we mention earlier, the probability may vary due to different scheduling.

Definition 7 (MDP and LTS Synchronous Product). *Let $\mathcal{D} = (S_d, init_d, Act_d, Pr_d)$ be an MDP; $\mathcal{L} = (S_l, init_l, Act_l, T)$ be an LTS such that $\tau \notin Act_l$. The synchronous product of $\mathcal{D}$ and $\mathcal{L}$, written as $\mathcal{D} \times \mathcal{L}$, is an MDP $(S, init, Act, Pr)$ such that $S = S_d \times S_l$; $init = (init_d, init_l)$; $Act = Act_d \cup Act_l$; and Pr is defined as follows.*

- *If $(s_1, \tau, \mu) \in Pr_d$, then $((s_1, s_2), \tau, \mu') \in Pr$ for all $s_2 \in S_l$ such that for all $s_1' \in S_d$, $\mu'((s_1', s_2)) = \mu(s_1')$;*
- *If $(s_1, e, \mu) \in Pr_d$ and $(s_2, e, s_2') \in T$, then $((s_1, s_2), e, \mu') \in Pr$ such that for all $s_1' \in S_d$, $\mu'((s_1', s_2')) = \mu(s_1')$.*

A state (s_1, s_2) of the product $\mathcal{D} \times \mathcal{L}$ is a TR-witness state if and only if $s_2 = \varnothing$. In [14], we show that the refinement probability can be calculated systematically by (1) building the deterministic LTS $det(\mathcal{L})$; (2) computing the synchronous product of $\mathcal{D}$ and $det(\mathcal{L})$; (3) calculating the maximum/minimum probability of reaching any TR-witness state.

Theorem 4. *Let $\mathcal{D} = (S_d, init_d, Act_d, Pr_d)$ be an MDP; $\mathcal{L} = (S_l, init_l, Act_l, T)$. Let G be the set of TR-witness states of $\mathcal{D} \times det(\mathcal{L})$.*

- *$\mathcal{P}_{max}(\mathcal{D}, init_d, traces(\mathcal{L})) = \mathcal{P}_{max}(\mathcal{D} \times det(\mathcal{L}), (init_d, init_l), G)$*
- *$\mathcal{P}_{min}(\mathcal{D}, init_d, traces(\mathcal{L})) = \mathcal{P}_{min}(\mathcal{D} \times det(\mathcal{L}), (init_d, init_l), G)$* □

Based on the above theorem, the probabilistic refinement checking problem is reduced to a probabilistic reachability problem, which can be solved by two standard methods. One is by solving a linear program. That is, we firstly associate a variable x_s to each state s in $\mathcal{P}$ to represent the probability of reaching any target state from s; then we construct a linear program which constraints the value of every x_s using a set of linear inequalities, based on the probability transition function; and lastly, we solve the linear program to get the maximum/minimum value of each x_s. Notice that the solution of state $(init_d, init_l)$ is the refinement probability. The other is to iteratively approximate the probability through graph traversing. Notice that for systems having large state space, it is impractical to store the entire linear program and solve it directly, therefore the iterative calculation approach is more widely used in probabilistic verification. As a result, in this paper we just focus on this approach.

Example 5. In the following, we show how the iterative calculation method works using the simple example shown in Example 4. That is, the maximal probability from initial state s_0 to accepting state s_2 is calculated step by step. Assume p_i^k is the maximal probability of s_i after the k-th iteration. Starting from the target state s_2, in k-th iteration we update the probability of states which could reach s_2 in exact k steps. Obviously, $p_0^0 = p_1^0 = 0$. As p_2^k = 1 and p_3^k = 0 for any k, k is ignored in these two states. In the 1st iteration, only p_1 can be updated, and p_1^1 = max$\{0.5 \times p_1^0 + 0.5 \times p_2^0, 1 \times p_3^0\} = max\{0.5, 0\} = 0.5$.; in the 2nd iteration, both p_0 and p_1 can be updated. It is trivial to show p_0^2 = p_1^1 = 0.5, and p_1^2 = max$\{0.5 \times p_1^1 + 0.5 \times p_2^1, 1 \times p_3^1\} = max\{0.75, 0\} = 0.75$. Iteratively, p_0 and p_1 in the long run can be calculated. A user-defined threshold is usually necessary to terminate the calculation, according to the desired precision. □

4.2 Anti-chain Based Approach

Now we introduce how anti-chain can be used to speed up the iterative calculation approach, by first introducing a lemma.

Lemma 3. *Let $\mathcal{D} = (S_d, init_d, Act_d, Pr_d)$ be an MDP; $\mathcal{L} = (S_l, init_l, Act_l, T)$. Let $\mathcal{P}$ be $\mathcal{D} \times det(\mathcal{L})$. Let G be the set of TR-witness states of $\mathcal{P}$. For all state (u_1, v_1) and (u_2, v_2) of $\mathcal{P}$ s.t. $(u_2, v_2) \leq (u_1, v_1)$, $Pr_{max}(\mathcal{P}, (u_1, v_1), G) \geq Pr_{max}(\mathcal{P}, (u_2, v_2), G)$ and $Pr_{min}(\mathcal{P}, (u_1, v_1), G) \geq Pr_{min}(\mathcal{P}, (u_2, v_2), G)$.*

Proof The above can be proved with an induction. The base case is that (u_2, v_2) is in G. By definition, (u_1, v_1) must be in G and therefore the lemma holds. Next, we show the induction step. Assume that (u_2', v_2') satisfies the lemma above. For every distribution μ_2 from (u_2, v_2), by Definition 7, there must exist a distribution μ_1 from (u_1, v_1) and for every state (u_2', v_2'), there exists (u_1', v_1') such that $\mu_2((u_2', v_2')) = \mu_1((u_1', v_1'))$ and $(u_2', v_2') \leq (u_1', v_1')$. By induction hypothesis, we have $Pr_{max}(\mathcal{P}, (u_1', v_1'), G) \geq Pr_{max}(\mathcal{P}, (u_2', v_2'), G)$ and $Pr_{min}(\mathcal{P}, (u_1', v_1'), G) \geq Pr_{min}(\mathcal{P}, (u_2', v_2'), G)$. Thus we have $Pr_{max}(\mathcal{P}, (u_1, v_1), G) \geq Pr_{max}(\mathcal{P}, (u_2, v_2), G)$ and $Pr_{min}(\mathcal{P}, (u_1, v_1), G) \geq Pr_{min}(\mathcal{P}, (u_2, v_2), G)$. Therefore, we conclude that the lemma holds. □

Compared to probabilistic reachability calculation for a general MDP, the above lemma gives us additional information, which can be potentially useful in speeding up the calculation. In the following, we discuss how we can make use of the information so as to improve the probabilistic refinement checking using the iterative calculation method.

The first step is building the product MDP meanwhile finding the target states, which is shown in Algorithm 4. The implementation and specification are defined in Definition 7. Different from the non-probabilistic cases, *the state space cannot be reduced in the probabilistic models*; instead, we define a function sub of the product state s satisfying $s.sub = \{t \mid t \in S \wedge s \leq t\}$, where S is the state space of the product MDP. Then the refinement checking is reduced to probabilistic reachability of a set of target states, denoted by $Target$. During the iterative calculation, whenever the probability of

Algorithm 4 Building MDP in Probabilistic Refinement Checking with Anti-chain

```
let $working$ be a stack containing a pair $(init_d, \{s \mid init_l \rightsquigarrow s\})$;
let $visited := \{(init_d, \{s \mid init_l \rightsquigarrow s\})\}$;  let $Target = \varnothing$; ;
while $working \neq \varnothing$  do
    pop $(impl, spec)$ from $working$;
    for all $(impl, e, \mu) \in Pr_d$ do
        if $e = \tau$ then
            $spec' := spec$;
        else
            $spec' := \{s' \mid \exists\, s \in spec.\ s \overset{e}{\rightsquigarrow} s'\}$;
        for all $impl' \in S_d$  do
            if $\mu(impl') > 0 \wedge (impl', spec') \notin visited$ then
                push $(impl', spec')$ into $working$;
                $visited := visited \cup (impl', spec')$;
                if $spec' = \varnothing$ then
                    $Target := Target \cup (impl', spec')$;
                for all $(impl', spec'') \in visited$ do
                    if $(impl', spec'') \leq (impl', spec')$ then
                        $(impl', spec'').sub.Add(impl', spec')$;
                    else if $(impl', spec') \leq (impl', spec'')$ then
                        $(impl', spec').sub.Add(impl', spec'')$;
return  true;
```

state s is updated, e.g., to p, according to lemma 3, **all states in $s.sub$ whose probability is less than p could be set to p directly.** This could speed up each iteration and potentially improve probabilistic refinement checking.

Now we evaluate whether the above method is indeed beneficial. The proposed probabilistic refinement checking algorithm has also been implemented in PAT. We evaluate it using a modified system based on the implementation of a distributed concurrent stack example [15]. Probabilistic choices are used to model a concurrent stack model composed by *two processes*, so as to capture the situation in which the communication between different processes fails from time to time. Failures do exist in real world cases and the experiments results are summarized in Table 3.

Table 3. Experiments: Probabilistic Concurrent Stack Implementation

System	Size	Verification Time (s)			#States Involved in Iterations		
		W/o AC	With AC	Gain	W/o AC	With AC	Gain
K = 2	20600	2.74	2.21	19.3%	4.2M	3M	28.6%
K = 3	45584	15.98	12.04	24.6%	18.6M	11.7M	37.1%
K = 4	86704	48.72	37.50	22.6%	55.5M	36.2M	34.8%
K = 5	117408	123.9	80.83	34.9%	130.7M	76.3M	41.6%
K = 6	231440	271.2	182.6	32.7%	272.1M	160.7M	40.9%
K = 7	342544	511.1	340.3	33.5%	515.2M	298.8M	42.0%

We compare the efficiency of the implementation with and without (W/o) Anti-chain (AC) using several cases. *K* means length of the stack; *Size* indicates the number of states in the whole system; *#States Involved in Iterations* represents the total number of states involved in the iterative calculation. For example, a state s updates its probability in two iterations, then *#States* should increase two. From the experiments, we can see that the anti-chain approach could reduce the total number of states accumulated during the calculation, through dynamically updating states' probability based on the subset relation sub. This speeds up the verification around 29%. We remark that the gains here are not as significant as the non-probabilistic cases, because the state space cannot be reduced; however, in some cases, it does shorten the verification time.

5 Conclusion

In this work, we proposed to adopt anti-chain approach to improve stable failures refinement, failures-divergence refinement and probabilistic refinement checking. These algorithms have been implemented in model checking framework PAT, and some experiments based on benchmark systems demonstrated the dramatic improvement of the verification efficiency of our method. To our best of knowledge, we are the first to investigate anti-chain approaches for these refinement checking.

As for future work, we are trying to extend anti-chain based refinement checking approach in real-time system; meanwhile, we are exploring the refinement relation between probabilistic models, which may also benefit from anti-chain based method.

References

1. Roscoe, A.W.: Model-checking CSP, ch. 21. Prentice-Hall (1994)
2. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When Simulation Meets Antichains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 158–174. Springer, Heidelberg (2010)
3. Aguilera, M.K., Gafni, E., Lamport, L.: The Mailbox Problem. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 1–15. Springer, Heidelberg (2008)
4. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics, 2nd edn. The Oxford University Press (2004)
5. Baier, C., Katoen, J.: Principles of Model Checking. The MIT Press (2008)
6. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 57–67. Springer, Heidelberg (2008)
7. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Algorithms for Omega-Regular Games with Imperfect Information. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 287–302. Springer, Heidelberg (2006)
8. Doyen, L., Raskin, J.F.: Antichains for the automata-based approach to model checking. Logical Methods in Computer Science 5(1:5), 1–20 (2009)
9. Ellen, F., Lev, Y., Luchangco, V., Moir, M.: SNZI: Scalable nonzero indicators. In: PODC, pp. 13–22. ACM (2007)
10. Filiot, E., Jin, N., Raskin, J.-F.: An Antichain Algorithm for LTL Realizability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 263–277. Springer, Heidelberg (2009)

11. Hoare, C.: Communicating Sequential Processes. Prentice-Hall (1985)
12. Roscoe, A.W.: On the expressive power of CSP refinement. Formal Aspects of Computing 17(2), 93–112 (2005)
13. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
14. Sun, J., Song, S., Liu, Y.: Model Checking Hierarchical Probabilistic Systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 388–403. Springer, Heidelberg (2010)
15. Treiber, R.K.: Systems programming: Coping with parallelism. Technical report, IBM Almaden Research Center (1986)
16. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: A New Algorithm for Checking Universality of Finite Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)
17. De Wulf, M., Doyen, L., Maquet, N., Raskin, J.-F.: Antichains: Alternative Algorithms for LTL Satisfiability and Model-Checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 63–77. Springer, Heidelberg (2008)